VMM Register Abstraction User Guide

Version C-2009.06
June 2009
VMM RAL Version 1.10 (Synopsys)

Comments?
E-mail your comments about this manual to:
vcs_support@synopsys.com.

Contents

1

Overview of the Register Abstraction Layer

The VMM Register Abstraction Layer (RAL) is a SystemVerilog or OpenVera VMM application package used to automate the creation of a high-level, object-oriented abstraction layer for memory-mapped registers and memories in a design under verification. The abstraction mechanism allows verification environments and tests to be migrated from block to system levels without any modifications. It also allows fields to be moved between physical registers without requiring modifications in the verification environment or tests.

The VMM Register Abstraction Layer can be used in a SystemVerilog VMM-based methodology or in an OpenVera RVM-based methodology. Regardless of the language used, all of the classes implementing the RAL are prefixed with `vmm_`.

The RAL includes predefined testcases that you can use to verify the correct operation of registers and memories in a design under verification. It includes usage assertions to detect incorrect register and memory accesses. A functional coverage model is included to accurately measure how thoroughly the registers and memories have been exercised.

The RAL supports 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. The RAL also supports designs with multiple physical interfaces, as well as registers and memories shared across multiple interfaces.

Figure 1-1 Structure of a RAL-based Environment

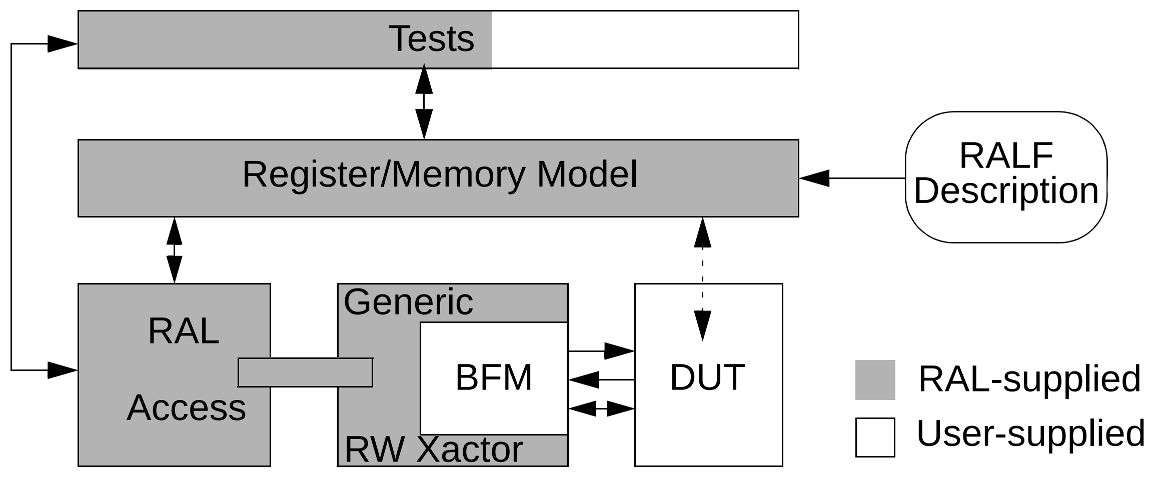


Figure 1-1 shows the structure of a verification environment based on a RAL. Existing VMM environments can be easily retrofitted to include a RAL.

The sections in this User Guide describe how to use the RAL in the order of which each task should be accomplished. Reading and applying this document in a linear fashion will result in a working RAL-based verification environment with automatically verified registers and memories.

Unsupported Features

The following features are still under development and are currently not supported:

- @none offset specification
- Virtual fields and virtual registers

Appendix B, "RAL Classes" identifies those methods in the base classes that are not yet implemented.

2

Usage Model

A RAL model comprises fields grouped into registers. Registers and memories are grouped into blocks, and blocks correspond to individually designed and verified components with their own host processor interfaces, address decoding and memory-mapped registers and memories. Even if a memory is physically implemented externally to the block and it is accessed through the block, as part of the block's address space, then the memory is considered as part of the block RAL model.

Blocks may be grouped into systems. Systems and blocks can be grouped into larger systems. A system may contain multiple instances of the same block or subsystem.

The smallest RAL 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. A system may contain a single block and no subsystems, or several instances of the same block and different subsystems.

For each element in a RAL model—field, register, memory, block or system—there is a class instance that abstracts the read and write operations on that element. Figure 2-1 shows the structure of a design block with two registers, which have two and three fields respectively, and an internal and external memory. Figure 2-2 shows the structure of the corresponding RAL model.

Figure 2-1 Structure of a Design Block

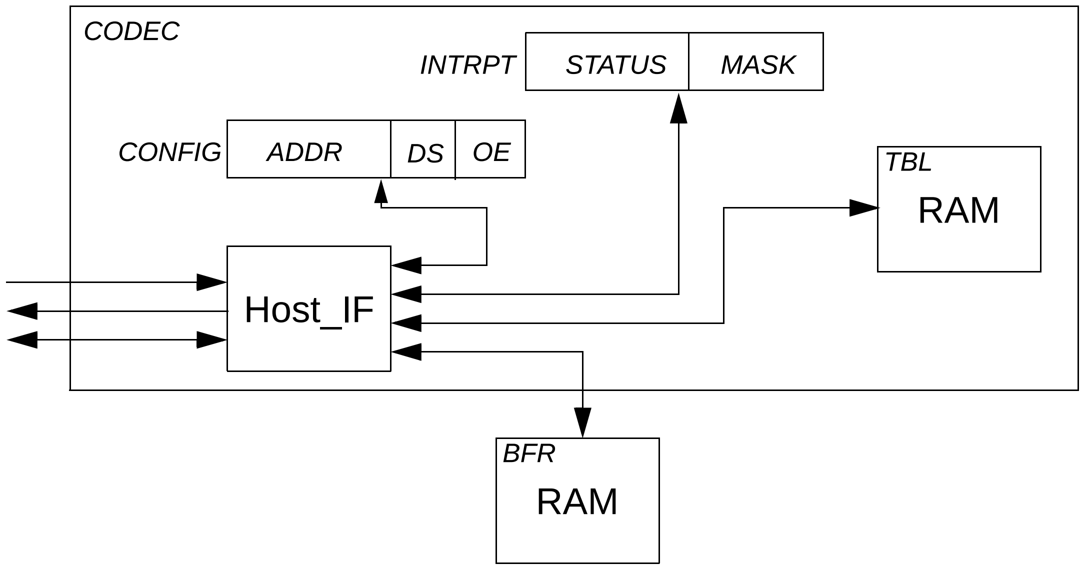
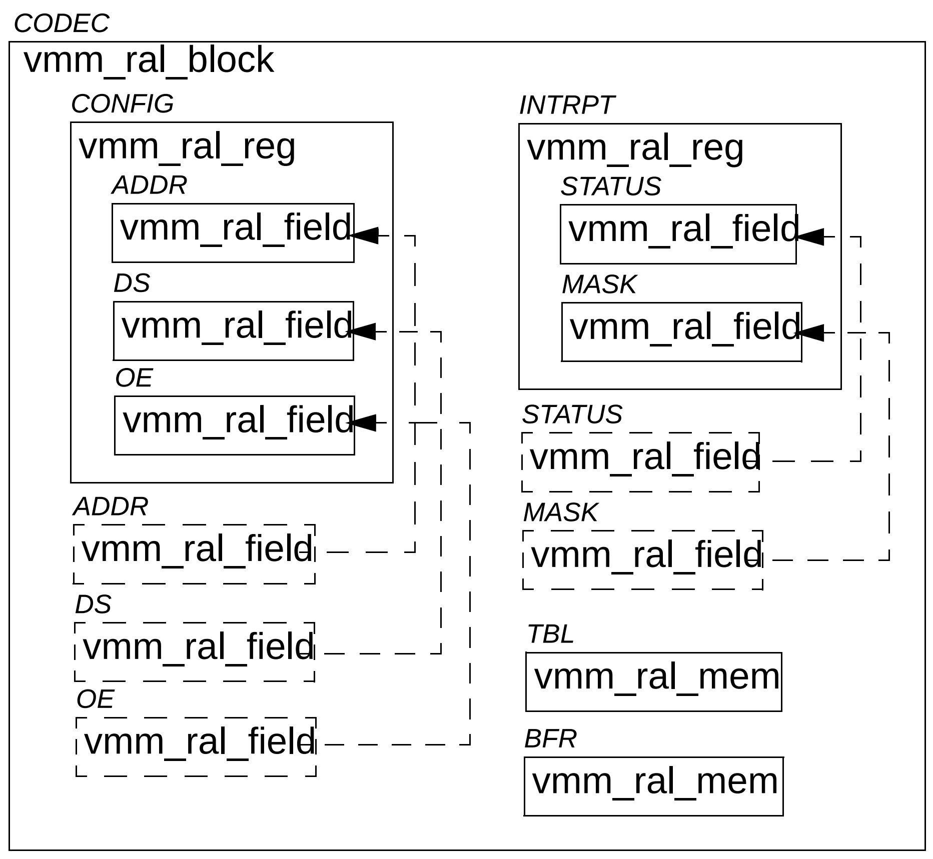


Figure 2-2 Structure of RAL Model for a Design Block



When using RAL, fields, registers and memory locations are not accessed via read and write cycles at specific addresses through a bus-functional model. Rather, they are accessed through read and write methods in their corresponding abstraction class. It is the responsibility of the RAL to turn these abstracted accesses into read and write cycles at the appropriate addresses via the appropriate bus-functional model. A RAL user never needs to worry about the specific address or location of a field, register or memory location, only the name.

For example, the field ADDR in the CONFIG register shown in Figure 2-1 can be accessed through the RAL shown in Figure 2-2 using the `CODEC.CONFIG.ADDR.read()` method. Similarly, location 7 in the BFR memory can be accessed using the `CODEC.BFR.write(7,...)` method. Furthermore, since the location of fields within physical registers is somewhat arbitrary, fields are also accessible independently of their register location. For example, the same `ADDR` field can also be accessed using the `CODEC.ADDR.read()` method. Therefore, if a field is relocated in another register, you do not need to modify a verification environment or testcase.

“Understanding the Generated Model” on page 3 details how the structure of the RAL model is created from a specification and how the names of the specification are used to create the instance names of the various abstraction classes.

Appendix B, "RAL Classes" details the functionality available in each abstraction class.

Chapter 9, "DUT Configuration" details how a RAL model is used to configure a DUT. It also explains how to write block-level register and memory access operations that are portable from a block-level environment to a system-level environment.

Mirroring

The RAL abstraction 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 RAL abstraction model has is the read and write access to those registers. If the DUT internally modifies the content of any field or register through its normal operations (for example, by setting a status bit or incrementing an accounting counter), the mirrored value becomes outdated.

The RAL abstraction model takes every opportunity to update its mirrored value. Upon every read operation, whether via a physical interface or back-door access, the mirror for the read register is updated. Upon every write operation, whether via a physical interface or back-door access, the new mirror value for the written register is predicted based on the access modes of the bits in the register. Resetting a RAL abstraction model sets the mirror to the reset value specified in the model.

A mirror is not a scoreboard. It can accurately predict the content of registers that are not updated by the design. However, it cannot determine if an updated value is correct or not. For example, a one-bit field of mode “`vmm_ral::A1`” can return a value of 0 in one read operation and a value of 1 in the subsequent read operation. The mirror notices the change in value and updates itself, but it cannot assume that the raising of that bit is invalid. However, if the same bit goes from 1 to 0 without an intervening write operation or a reset to clear, the mirror flags the updated value as invalid.

You can access mirrored values in zero-time by using the “vmm_ral_field::get()” or “vmm_ral_reg::get()” methods. You can also update the mirror by using the “vmm_ral_field::mirror()”, “vmm_ral_reg::mirror()” or “vmm_ral_block_or_sys::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 or system updates the mirror for all fields and registers it contains. Updating a mirror in a large block or system 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 zero-time by using the "vmm_ral_field::set()" or "vmm_ral_reg::set()" methods. Once a mirror value has been written to, 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 "vmm_ral_reg::update()" or "vmm_ral_block_or_sys::update()" methods. If the new mirrored value matches the old mirrored value, the register is not updated. Updating a block or system with its mirror updates all fields and registers it contains with their corresponding mirror values. Updating a large block or system may take a lot of simulation time if physical write cycles are used, whereas updating using back-door access usually takes zero-time. Synopsys recommends that you use this update-from-mirror process when configuring the DUT to minimize the number of write operations performed.

Memories Are Not Mirrored

Memories can be quite large. That is why they are usually modelled 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 modelled. 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. Both the memory model of the DUT and the memory mirror, would end up being fully populated, and duplicating 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 “vmm_ral_mem::peek()” or “vmm_ral_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.

If you specify the necessary `hdl_node` attributes in the RALF specification, memories modelled inside a Verilog DUT using single-dimension unpacked arrays of packed bits, have their back-door access automatically generated. You must provide a user-defined back-door mechanism if the memory is modelled using an associative array, discrete registers or uses a third-party memory model (such as the DesignWare memory models). Refer to Chapter 10, "Back-door Access" for additional information.

3

Installing and Using RAL

Every shipment of VCS or Vera includes RAL. Therefore, no further installation is required. If you are using SystemVerilog with an older version of VCS that does not include a RAL installation, or you wish to use a more recent version of the SystemVerilog implementation of RAL, you can use a separate installation. First, you must obtain the VMM open source distribution available at http://vmmcentral.org and install it in some directory identified by the `$VMM_HOME` environment variable.

Minimum Requirements

You must have, at a minimum, the specified versions of the following tools:

- VCS or Pioneer 2005.06-SP1
- Vera 2005.06

- Perl 5.6
- TCL

Installation Instructions

Perform the following steps to install the VMM Open Source distribution:

1. Create an installation directory.

   This directory must be accessible by all users.

2. Decompress and expand the VMM Open Source package.

   Install the package in the installation directory created in step 1.

3. Define the `VMM_HOME` environment variable.

   For the remainder of this document, the installation directory will be referred to as `${VMM_HOME}`.

4. Add `${VMM_HOME}/shared/bin` to your `PATH` environment variable.

Compiling and Running OpenVera Code With Vera

To make the predefined RAL classes visible in your OpenVera code, put the following directive at the top of the relevant source files:

```
#include "vmm_ral.vrh"
```

Vera will automatically locate the header file and load the appropriate object file found in the Vera distribution.

Compiling and Running OpenVera Code With VCS

To make the predefined RAL classes visible in your OpenVera code, you must specify the following file before any OpenVera source files when compiling using VCS:

```
vcs -ntb -ntb_opts rvm ...\
   ${VCS_HOME}/etc/rvm/vmm_ral.vrp ...
```

Compiling and Running SystemVerilog Code

To make the predefined RAL classes visible in your SystemVerilog code, include the following directive at the top of the relevant source files:

```
‘include "vmm_ral.sv"
```

...and specify the following command-line option:

```
% vcs ... -ntb_opts rvm ...
```

This automatically includes the source code for the predefined classes in your simulation found in the VCS installation.

Alternatively, you can simply list the following file on the command line before any file that uses the RAL classes:

```
${VCS_HOME}/etc/rvm/vmm_ral.sv
```

Using a VMM Open Source Distribution

You must indicate to VCS where to locate the source files by using the following command-line option and by omitting the `"-ntb_opts rvm"` option:

```
+incdir+${VMM_HOME}/sv
```

Alternatively, you can simply list the following file, along with the include directory, on the command line before any file that uses the RAL classes:

```
% vsc ... +incdir+${VMM_HOME}/sv \
          ${VMM_HOME}/sv/vmm_ral.sv ...
```

4

Register and Memory Specification

The Register Abstraction Layer File (RALF) is used to specify all the registers and memories in the design under verification. It is used to generate the object-oriented register and memory high-level abstraction layer. The first step in a project is to create a RALF description. Appendix A, "RALF Syntax" contains detailed syntax and documentation for the RALF description.

As you add and modify fields, registers, and memories, you can update the RALF description many times during a project. You can then regenerate the abstraction layer multiple times without requiring modifications to the existing environment or tests.

Systems, Blocks, Registers, and Fields

In RAL, a design is a block or a system of blocks. The smallest functional unit that can be verified is a block. Systems are designs composed of blocks. Systems can also be composed of smaller systems of blocks, called subsystems.

There must be at least one block in a RALF description. The top-level construct describing the design under verification can be a `block` or `system` construct. The top-level block is identified when the RAL code is generated, therefore, a single RALF description may contain descriptions of multiple blocks and systems. The following example shows the RALF description of a design block:

Example 4-1 RALF Description of a Design Block

```
block blk_name {
   ...
}
```

Systems are composed of subsystems or blocks. Blocks are composed of registers and memories. There can be no registers or memories directly in a system. If a design has system-wide registers or memories, they should be described in a block named, for example, `system_wide`. The following example shows the RALF description of a system:

Example 4-2 RALF Description of a System

```
system sys_name {
   ...
   block blk_name ...
   system subsys_name ...
}
```

Registers are composed of fields. Fields are concatenated to form a register, with optional unused bits between fields. A register must contain at least one field. The following example shows the RALF description of registers and memories in a block:

Example 4-3   RALF Description of Registers and Memories in a Block

```
block blk_name {
   ...
   register reg_name ...
   register reg_name ...
   ...
   memory mem_name ...
}
```

The field is the basic unit of the RAL. Fields are accessed atomically, independently of their location within a register or other fields. Therefore, fields can be moved within or across registers without having to modify the code that uses them. The following example shows the RALF description of fields in a register:

Example 4-4   RALF Description of Fields in a Register

```
register reg_name {
   ...
   field fld_name ...
   field fld_name ...
}
```

Reusability and Composition

RALF descriptions are intended to describe designs that can be arbitrarily combined and reused to create larger designs. There is no need for a RALF description of a block or subsystem to be aware of the context in which the block or subsystem is going to be used. In RALF descriptions, blocks and subsystems are described as stand-alone designs.

Although a RALF can describe an entire design inline, as in Example 4-5, a description can also instantiate blocks, registers and fields as required. The granularity of the description is arbitrary and you should plan for it to maximize reuse.

Example 4-5   Inlined RALF Description

```
system sys_name {
   ...
   block blk_name {
      ...
      register reg_name {
         ...
         field fld_name {
            ...
         }
      }
      ...
      memory mem_name {
         ...
      }
   }
}
```

RALF descriptions can include other RALF descriptions of smaller designs. Included descriptions can be reused and instantiated to compose the description of a larger design. The following example illustrates how this can be done:

Example 4-6   Hierarchical RALF Description

```
field fld_name {
   ...
}

register reg_name {
   ...
   field fld_name ;
}

memory mem_name {
   ...
}

block blk_name {
   ...
   register reg_name;
   memory mem_name;
}

system sys_name {
   ...
   block blk_name;
}
```

Naming

The names of fields, registers, memories, blocks, and systems are very important because these names are used to identify their corresponding abstraction class in the RAL abstraction model.

The following naming conventions apply to the names elements within a RALF description:

- Names must not be OV or SV reserved keywords

  These names are used as the name of abstraction classes in the generated OV or SV code. Therefore, they cannot be the same as reserved keywords in OV or SV.

- Field names should be unique within a block

Each block abstraction class contains a class property for each field contained in all of its registers, regardless of the specific register where it is located. If unique, the name of the field class property within the block abstraction class is the name of the field. In this case, fields can be moved within or across physical registers without affecting the verification environment or tests. Regardless of field name uniqueness, the block abstraction class contains another field class property referring to each field using the concatenation of the register and field name. See "Registers" for additional information.

Example 4-7   Field Class Properties in a Block Abstraction Class

```
block blk_name {
   register reg_name {
      field fld1;
      field fld2;
   }
   register xyz {
      field fld2;
   }
}

Yields:

class ral_block_blk_name;
   ...
   vmm_ral_field fld1, reg_name_fld1;
   vmm_ral_field reg_name_fld2;
   ...
   vmm_ral_field xyz_fld2
endclass
```

- Register names must be unique within a block and should be unique from field names.

  Each block abstraction class contains a class property for each register it contains. The name of the register class property within the block abstraction class is the name of the register and must, therefore, be unique and should be different from field names.

Example 4-8   Register Abstraction Classes in a Block Abstraction Class

```
block blk_name {
   register reg_name {
      field fld1;
      field fld2;
   }
}

Yields:

class ral_block_blk_name;
   ral_reg_blk_name_reg_name reg_name;
   vmm_ral_field             fld1, reg_name_fld1;
   vmm_ral_field             fld2, reg_name_fld2;
endclass
```

- Memory names must be unique within a block and unique from register names and should be unique from field names.

  Each block abstraction class contains a class property for each memory it contains. The name of the memory class property within the block abstraction class is the name of the memory and must, therefore, be unique and different from register names. It should also be different from field names.

Example 4-9   Memory Abstraction Classes in a Block Abstraction Class

```
block blk_name {
   register reg_name {
      field fld1;
      field fld2;
   }
   memory mem_name;
}

Yields:

class ral_block_blk_name;
   ral_reg_blk_name_reg_name reg_name;
   vmm_ral_field             fld1, reg_name_fld1;
   vmm_ral_field             fld2, reg_name_fld2;
   ral_mem_blk_name_mem_name mem_name;
```

```
endclass
```

- Block and subsystem names must be unique within a system.

  Each system abstraction class contains a class property for each block and subsystem it contains. The name of the block and subsystem class property within the system abstraction class is the name of the block or subsystem. Therefore, block and subsystem names must be unique.

- Independently defined names of registers, memories, blocks, and systems must be, respectively, globally unique within a RALF description.

  Each independently defined RALF element corresponds to a generated abstraction class in the RALF model (see "Understanding the Generated Model"). The names of these elements are used to generate the name of the corresponding class. Class names must be globally unique in SystemVerilog and OpenVera. Therefore, the names of independently defined registers, memories, blocks, and systems must be globally unique, otherwise they will generate identical abstraction class names.

  This requirement does not apply to elements defined inline within another definition.

Note:

Instantiated fields, registers, memories, blocks and subsystems can be renamed. With all of these naming requirements, it would be very difficult to have reusable RALF descriptions. Descriptions would need to know of their contexts to ensure uniqueness. Nor would it be possible to describe a design that contains multiple instances of the same block. Fortunately, any element of a RALF description can be renamed when instantiated to ensure uniqueness.

Example 4-10 Renaming RALF Elements

```
block blk_name {
   ...
}

system sys_name {
   ...
   block blk_name=blk1;
   block blk_name=blk2;
}
```

Hierarchical Descriptions and Composition

A RAL description can have independently specified registers, memories, blocks, and subsystems. You can instantiate these elements in higher level elements to create complete design descriptions.

When you specify registers, memories, blocks and subsystems, you also independently and explicitly specify their physical width as a number of bytes. Therefore, a block can be composed of registers and memories of smaller or larger width. Similarly, systems can be composed of blocks of smaller or larger width.

If you instantiate an element in a wider element, the value of the narrower element is justified to the least-significant bit and the most significant bits are padded with zero or truncated.

If you instantiate an element in a narrower element, the value of the wider element is split into the minimum number of narrower values.

You can specify splitting as:

- **Big Endian -** The most-significant bits are split into the lower addresses in the narrower address space. A 5-byte wide value of 0x1234567890 would be split into three 2-byte narrower values at increasing addresses in the following order: 0x0012, 0x3456 and 0x7890.
- **Little Endian -** The least-significant bits are split into the lower addresses in the narrower address space. A 5-byte wide value of 0x1234567890 would be split into three 2-byte narrower values at increasing addresses in the following order: 0x7890, 0x3456 and 0x0012.
- **Big FIFO -** All split values are accessed at the same physical address in the narrower address space. The most-significant bits are accessed first. A 5-byte wide value of 0x1234567890 would be split into three consecutive 2-byte narrower values at the same address in the following order: 0x0012, 0x3456 and 0x7890.
- **Little FIFO -** All split values are accessed at the same physical address in the narrower address space. The least-significant bits are accessed first. A 5-byte wide value of 0x1234567890 would be split into three consecutive 2-byte narrower values at the same address in the following order: 0x7890, 0x3456 and 0x0012.

Arrays and Register Files

Many designs have identical registers or groups of registers located in consecutive memory locations. These registers could be described by explicitly specifying each register, ignoring the fact that they are identical.

Example 4-11 Explicit specification of register arrays

```
register reg_name {
   ...
}
block blk_name {
   ...
   register reg_name=reg_0;
   register reg_name=reg_1;
   ...
   register reg_name=reg_7;
}
```

The repetitive process could be simplified by using the TCL *for-loop* command. Using the for-loop only simplifies the syntactical requirements of the specification. It does not change the RAL model that will be ultimately generated.

Example 4-12 Iterated explicit specification of register arrays

```
register reg_name {
   ...
}
block blk_name {
   ...
   for {set n 0} {$n < 8} {incr n} {
      register reg_name=reg_$n;
   }
}
```

The problem with explicitly enumerating consecutive registers is that they have unique names. It will not be possible to randomly index or iterate over their RAL model when writing SystemVerilog or OpenVera code that uses these consecutive registers.

Specifying consecutive registers using a register array will result in an array being available to be indexed or iterated on at runtime, not just at specification time. See "Arrays" for more details on the code generation process for arrays.

Example 4-13 Specification of register arrays

```
register reg_name {
   ...
}
block blk_name {
   ...
   register reg_name[8];
   register regX[5] {
      ...
   }
}
```

A sequence of register arrays will locate them in consecutive memory locations. For example, the specification in Example 4-13 will result in the following address map: `reg_name[0]`, `reg_name[1]`, `...` `reg_name[7]`, `regX[0]`, `regX[1]`, `...` `regX[4]`. If sequences of register groups, or interleaved register arrays are required, then you should a register file array. The specification in Example 4-14 will yield the following address map: `reg[0].reg_name`, `reg[0].X`, `reg[1].reg_name`, `...` `reg[4].reg_name`, `reg[4].X`.

Example 4-14 Specification of register file arrays

```
register reg_name {
   ...
}
block blk_name {
      ...
      regfile reg[5] {
         register reg_name;
         register X {
            ...
         }
      }
   }
```

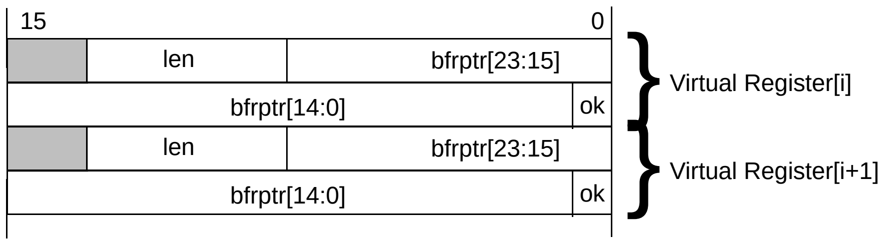
Virtual Fields and Virtual Registers

By default, fields and registers are assumed to be implemented in individual, dedicated hardware structures with a constant and permanent physical location such as a set of D flip-flops. In contrast, virtual fields and virtual registers are implemented in memory or RAM. Their physical location and layout is created by an agreement between the hardware and the software, not by their physical implementation.

Virtual fields and registers can be modelled using RAL by creating a logical overlay on a RAL memory model that can then be accessed as if they were real physical fields and registers. The RAL model of the memory itself remains available for directly accessing the raw memory without regard to any virtual structure it may contain.

Virtual fields define continuous bits in one or more memory locations and can span a memory location boundary. Virtual fields are contained in virtual registers. Virtual registers define continuous whole memory locations. They can span multiple memory locations but are always composed of entire memory locations, never fractions of memory locations.

Figure 4-1   Virtual Field and Virtual Register Structure



Virtual registers are always arrays because the usual reason they are virtual is that there are a large number of them and implementing them in a RAM instead of individual flip-flops is most efficient. Arrays of virtual registers are associated with a memory. The association of a virtual register array with a memory can be static (for example, specified in the RALF file) or dynamic (for example, specified at runtime through user code).

Static virtual registers are associated with a specific memory and are located at specific offsets within that memory. The association is specified in the RALF file and is created by the code generator. This association is permanent and cannot be broken at runtime.

Example 4-15   Static virtual register array

```
block MAC {
   ...
   memory DMABFRS { ... }
   ...
   virtual register CHANNEL[1024] DMABFRS@0 {
      field {...};
      ...
   }
}
```

Dynamic virtual registers are dynamically associated with a user-specified memory and are located at user-specified offsets within that memory at runtime. The dynamic allocation of virtual register arrays can also be performed randomly by a Memory

Allocation Manager instance. The structure of the virtual registers is specified in the RALF file, but the number of virtual registers in the array and its association with a memory is specified in the SystemVerilog or OpenVera code and must be correctly implemented by the user. Dynamic virtual registers arrays can be relocated or resized at runtime.

Example 4-16   Dynamic virtual register specification

```
block MAC {
   ...
   memory DMABFRS { ... }
   ...
   virtual register CHANNEL {
      field {...};
      ...
   }
}
```

Example 4-17   Implementing dynamic virtual registers

```
ral_model.MAC.CHANNEL.implement(1024,
                                ral_model.MAC.DMABFRS,
                                0);
```

Example 4-18   Randomly implementing dynamic virtual registers

```
ral_model.MAC.CHANNEL.allocate(1024,
                           ral_model.MAC.DMABFRS.mam);
```

Because virtual fields and virtual registers are implemented in memory, their content is not mirrored by the RAL model.

Multiple Physical Interfaces

Some designs may have more than one physical interface, each with accessible registers or memories. Some registers or memories may even be accessible via more than one physical interfaces and be shared.

A physical interface is called a domain. Only blocks and systems can have domains. Domains contain registers and memories. If a block or system has only one physical interface, there is no need to specify a domain for that interface.

For example, the block "bridge" shown in Example 4-19 specifies a block with two physical interfaces and a register accessible from both interfaces at offset 0 in their respective address spaces.

Example 4-19 Specification for a two-domain block

```
register xfer {
   bits 32;
   access rw;
   shared (xfer_reg);
}

block bridge {
   bytes 4;
   domain apb {
      register xfer;
   }
   domain ahb {
      register xfer;
   }
}
```

Some physical interfaces may have different transactions used for configuration than the transactions used for normal operations. For example, PCI interfaces have `configuration write` transactions that are different from normal `write` transactions.

Configuration transactions are typically used to set a base address and other decoding information required by normal transactions. Because configuration transactions are used separately from normal transactions, and normal transactions cannot occur until the DUT has been suitably configured using configuration transactions, configuration and normal transactions on the same physical interface must be modelled as separate physical interfaces.

Systems with multiple domains can instantiate blocks with a single domain. A domain must be entirely instantiated within a system domain, that is, a block-level or subsystem-level domain cannot be split between two system-level domains. Different block-level or subsystem-level domains can be instantiated in the same system-level domain but in different address offsets.

When instantiating a multiple-domain block or sub-system in a multiple-domain system, the same name and `hdl_path` must be used for all instances. This creates a single instance of the block or subsystem with its various domains instantiated in different domains.

Example 4-20 shows a specification of a multiple-domain instantiation. Notice how the same instance name "br" and HDL path are used in both cases. Example 4-21 shows the corresponding abstraction model of the system. Notice how domains do not create an additional abstraction scope.

Example 4-20 Instantiating a two-domain block in a two-domain system

```
system amba {
   bytes 4;
   domain apb {
      block bridge.apb=br (amba_bus.bridge);
   }
   domain ahb {
      block bridge.ahb=br (amba_bus.bridge);
   }
}
```

Example 4-21   Model of a two-domain block in a two-domain system

```
class ral_block_bridge extends vmm_ral_block;
   ral_reg_xfer xfer;
   ...
endclass

class ral_sys_amba extends vmm_ral_sys;
   ral_block_bridge br;
   ...
endclass
```

5

Code Generation

Once a description of the available registers and memories in a design is available, `ralgen` can automatically generate the RAL abstraction model for these registers and memories. Test cases, firmware, device drivers and DUT configuration code use this model to access the registers and memories through an object-oriented abstraction layer. The predefined tests also use this model to verify the functional correctness of the registers and memories.

Generating a RAL Model

To generate the RAL model, use the following command:

```
% ralgen [options] -t topname -l sv|ov {-I dir}
{filename.ralf}
```

Where:

`-t topname`

The name of the top-level block or system description in the RALF file that entirely describes the design under verification.

`-l sv|ov`

Specifies SystemVerilog (`sv`) or OpenVera (`ov`) as the implementation language for the generated code. Depending on which option you specify, the RAL model for the entire design is generated in either a file named `ral_`*`topname`*`.sv` or `ral_`*`topname`*`.vr` in the current working directory.

`-I dir`

An optional list of directories that `ralgen` searches for sourced Tcl files.

`filename``.ralf`

The name of the files containing the RALF description. Although, the `.ralf` extension is not required, Synopsys recommends you specify it. If you specify more than one file, they are parsed as if they had been concatenated together.

Options

The following options are available:

`-b`

Generate the back-door access code for those registers and memories where a complete `hdl_path` has been specified.

`-c a`

Generate the “Address Map” functional coverage model. The `-c` option may be specified multiple times.

`-c b`

Generate the “Register Bits” functional coverage model. The `-c` option may be specified multiple times.

`-c f`

Generate the “Field Values” functional coverage model. The `-c` option may be specified multiple times.

`-e`

Generate empty constraint blocks for every abstract class.

Understanding the Generated Model

The generated abstraction model is a function of the RALF description used to generate it. Therefore, understanding how the generation process works will help you use the generated model based on the knowledge of the RALF description.

The generated abstraction model is described using a bottom-up approach, in the order in which the classes are generated and then compiled. If you prefer to read a top-down description, simply read the following sections (“Fields”, “Registers”, “Register Files”, “Virtual Registers”, “Memories”, “Blocks”, and “Systems”) in the reverse order.

Fields

No abstraction class is generated for a field definition. Instead, each field is modeled by an instance of the `vmm_ral_field` class (for details, see “vmm_ral_field”).

The instance of that class is stored in a property of the class modeling the register that instantiates it and the block that instantiates the register.

Registers

An abstraction class is generated for each register definition. For each:

- Independently defined register named `regnam`, there is a class named `ral_reg_regnam`
- Register named `regnam` defined inline in the specification of a block named `blknam`, there is a class named `ral_reg_blknam_regnam`
- Register named `regnam` defined inline in the specification of a register file named `filnam` in a block named `blknam`, there is a class named `ral_reg_blknam_filnam_regnam`

In all cases, the register abstraction class is derived from the `vmm_ral_reg` class (for details, see “vmm_ral_reg”).

All virtual methods defined in the `vmm_ral_reg` class are overloaded in the register model class. Each virtual method is overloaded to implement register-specific behavior of the register as defined in the RALF description. No new methods are added to the register abstraction class.

As shown in Example 5-1, the register abstraction class contains a class property for each field it contains. The name of the property is the name of the field. There are no properties for unused or reserved fields.

Example 5-1 Register Model Class for Register in Example A-4

```
class ral_reg_CTRL extends vmm_ral_reg;
   vmm_ral_field TXE;
   vmm_ral_field RXE;
   vmm_ral_field PAR;
   vmm_ral_field DTR;
   vmm_ral_field CTS;
   ...
endclass: ral_reg_CTRL
```

Instances of this class are found in the block abstraction class for the blocks instantiating this register.

Register Files

An abstraction class is generated for each register file definition. For each register file named `filnam` defined inline in the specification of a block named `blknam`, there is a class named `ral_regfile_blknam_filnam`. The register abstraction class is not derived from any base class because it is purely a container for the registers instantiated in the register file.

The register file container class contains a class property for each register it contains. See the “Blocks” on page 9 section for a description of those properties.

Example 5-2 Register File Specification and Corresponding Model

```
block dma_ctrl {
   regfile chan {
      register src {
         field addr { ... }
      }
      register dst {
         field addr { ... }
      }
      register count {
         field n_bytes { ... }
      }
      register ctrl {
         field TXE { ... }
         field BSY { ... }
      }
   }
}
```

Corresponding abstraction model:

```
class ral_regfile_dma_ctrl_chan;
   ral_reg_dma_ctrl_chan_src src;
   vmm_ral_field             src_addr;

   ral_reg_dma_ctrl_chan_dst dst;
   vmm_ral_field             dst_addr;

   ral_reg_dma_ctrl_chan_count count;
   vmm_ral_field               n_bytes, count_n_bytes;
   vmm_ral_field               TXE, ctrl_TXE;
   vmm_ral_field               BSY, ctrl_BSY;
   ...
endclass: ral_reg_dma_ctrl_chan
```

Instances (usually arrays of instances) of this class are found in the block abstraction class for the blocks instantiating this register file.

Virtual Registers

An abstraction class is generated for each virtual register array definition. For each independently defined virtual register array named `vregnam`, there is a class named `ral_vreg_vregnam`. For each virtual register array named `vregnam` defined inline in the specification of a block named `blknam`, there is a class named `ral_vreg_blknam_vregnam`. In both cases, the virtual register array abstraction class is derived from the "vmm_ral_vreg" class (for details, see "vmm_ral_reg"). A single abstraction class is used for all virtual registers in the array.

All virtual methods defined in the "vmm_ral_vreg" class are overloaded in the virtual register array abstraction class. Each virtual method is overloaded to implement register-specific behavior of the virtual register array as defined in the RALF description. No new methods are added to the virtual register array abstraction class.

As shown in Example 5-3, the virtual register array abstraction class contains a class property for each virtual field it contains. The name of the property is the name of the field. There are no properties for unused or reserved fields, and unlike register arrays, a single instance of the virtual register array abstraction class is used to model the complete virtual register array.

Example 5-3   Virtual Register Abstraction Class

```
block blk1 {
   memory ram0 { ... }

   virtual register dma[256] ram0@0x0000 {
      field len { ... }
      field bfrptr { ... }
      field ok { ... }
   }
}
```

Corresponding abstraction model:

```
class ral_vreg_blk1_dma extends vmm_ral_vreg;
   vmm_ral_vfield len;
   vmm_ral_vfield bfrptr;
   vmm_ral_vfield ok;
   ...
endclass: ral_vreg_blk1_dma
```

```
class ral_block_blk1 extends vmm_ral_block;
   vmm_ral_mem       ram0;
   ral_vreg_blk1_dma dma;
   ...
endclass: ral_block_blk1
```

A single instance (not an array of instance) of this class is found in the block abstraction class for the blocks instantiating a virtual register array.

Memories

An abstraction class is generated for each memory definition. For each independently defined memory named `memnam`, there is a class named `ral_mem_memnam`. For each memory named `memnam` defined inline in the specification of a block named `blknam`, there is a class named `ral_mem_blknam_memnam`.

In both cases, the memory abstraction class is derived from the `vmm_ral_mem` class (for details, see "vmm_ral_mem").

All virtual methods defined in the `vmm_ral_mem` class are overloaded in the memory abstraction class. Each virtual method is overloaded to implement memory-specific behavior of the memory as defined in the RALF description. No new methods are added to the memory abstraction class.

As shown in Example 5-4, the memory abstraction class contains no additional class properties.

Example 5-4 Memory Abstraction Class for Memory in Example A-9

```
class ral_mem_ROM extends vmm_ral_mem;
   ...
endclass: ral_mem_ROM
```

Instances of this class are found in the block abstraction class for the blocks instantiating this memory.

Blocks

An abstraction class is generated for each block definition. For each independently defined block named `blknam`, there is a class named `ral_block_blknam`. For each block named `blknam` defined inline in the specification of a system named `sysnam`, there is a class named `ral_block_sysnam_blknam`. In both cases, the block abstraction class is derived from the `vmm_ral_block` class (for details, see "vmm_ral_block").

All virtual methods defined in the `vmm_ral_block` class are overloaded in the block abstraction class. Each virtual method is overloaded to implement block-specific behavior of the block as defined in the RALF description. No new methods are added to the block abstraction class.

As shown in Example 5-5 and Example 5-6, the block abstraction class contains a class property for each register and register file it contains. The name of the register or register file property is the name of the register or file. The block abstraction class also contains one or two class properties for each field it contains. The name of each field property is the name of the field (if unique within the register) and the name of the register concatenated with the name of the field, respectively. There are no properties for unused or reserved fields.

Important:

It is preferable that field names be unique across blocks. Therefore, each field has a property with the same name in the block abstraction class that instantiates them. If you move the field to another physical register, you can use this uniquely-named field property to reduce testbench maintenance. If you use the name that is prefixed with the register name, you must modify testbenches if the field is relocated to another physical register.

Example 5-5   Block Abstraction Class for Block in Example A-10

```
class ral_block_uart extends vmm_ral_block;
   ral_reg_CTRL  CTRL;
   vmm_ral_field TXE, CTRL_TXE;
   vmm_ral_field RXE, CTRL_RXE;
   vmm_ral_field PAR, CTRL_PAR;
   vmm_ral_field DTR, CTRL_DTR;
   vmm_ral_field CTS, CTRL_CTS;

   ral_mem_tx_bfr tx_bfr;
   ...
endclass: ral_block_uart
```

Example 5-6   Block Abstraction Class for Block in Example A-12

```
ral_block_bridge extends vmm_ral_block;
   ral_reg_flags     pci_flags;
   vmm_ral_field     pci_flags_cts;
   vmm_ral_field     pci_flags_dtr;

   ral_reg_data_xfer to_ahb;
   vmm_ral_field     to_ahb_data;

   ral_reg_data_xfer frm_ahb;
   vmm_ral_field     frm_ahb_data;

   ral_reg_flags     ahb_flags;
   vmm_ral_field     ahb_flags_cts;
   vmm_ral_field     ahb_flags_dtr;

   ral_reg_data_xfer to_pci;
   vmm_ral_field     to_pci_data;
   ral_reg_data_xfer frm_pci;
   vmm_ral_field     frm_pci_data;
   ...
endclass: ral_block_bridge
```

Instances of this class are found in the system model class for the systems instantiating this block.

Arrays

If a block contains a register array or register file array, the class property for the register array or register file array is declared as a fixed sized array to the corresponding register abstraction class or register file container class. Similarly, the field properties for the fields contained in the register array are declared as a fixed sized array of `vmm_ral_field` classes.

Example 5-7   Array Specifications and Corresponding Model

```
block b1 {
   register r1[32] {
      field f1 { ... }
   }
   regfile  rf[16] {
      register r1 {
         field f1 { ... }
      }
      register r2[4] {
         field f1 { ... };
      }
   }
}
```

Corresponding abstraction model:

```
class ral_regfile_b1_rf;
   ral_reg_b1_rf_r1 r1;
   vmm_ral_field    r1_f1;

   ral_reg_b1_rf_r2 r2[4];
   vmm_ral_field    f2_f1[4];
   ...
```

```
endclass: ral_regfile_b1_rf

class ral_block_b1 extends vmm_ral_block;
   ral_reg_b1_r1 r1[32];
   vmm_ral_field f1[32], r1_f1[32];

   ral_regfile_b1_rf rf[16]
   ...
endclass: ral_block_b1
```

Systems

An abstraction class is generated for each system definition. For each independently defined system named `sysnam`, there is a class named `ral_sys_sysnam`. For each subsystem named `subnam` defined inline in the specification of a system named `sysnam`, there is a class named `ral_sys_sysnam_subnam`.

In both cases, the system abstraction class is derived from the `vmm_ral_sys` class (for details, see "vmm_ral_sys").

All virtual methods defined in the `vmm_ral_sys` class are overloaded in the system abstraction class. Each virtual method is overloaded to implement system-specific behavior of the system as defined in the RALF description. No new methods are added to the system abstraction class.

As shown in Example 5-8 and Example 5-9, the system abstraction class contains a class property for each block and subsystem it contains. The name of the block or subsystem property is the name of the block or system. For blocks with multiple domains, the name of the blocks and subsystems are also available prefixed with the domain name.

Example 5-8 System Abstraction Class for Example A-13

```
class ral_sys_SoC extends vmm_ral_sys;
   ral_block_uart uart0;
   ral_block_uart uart1;
   ...
endclass: ral_sys_SoC
```

Example 5-9 System Abstraction Class for Example A-14

```
class ral_sys_SoC extends vmm_ral_sys;
   ral_block_uart uart0, ahb_uart0;
   ral_block_uart uart1, ahb_uart1,
   ral_block_bridge ahb_br;
   ral_block_bridge pci_br;
   ...
endclass: ral_sys_SoC
```

Arrays

If a system contains a block array or subsystem array, the class property for the block array or subsystem array is declared as a fixed sized array of the corresponding block abstraction class or system abstraction class.

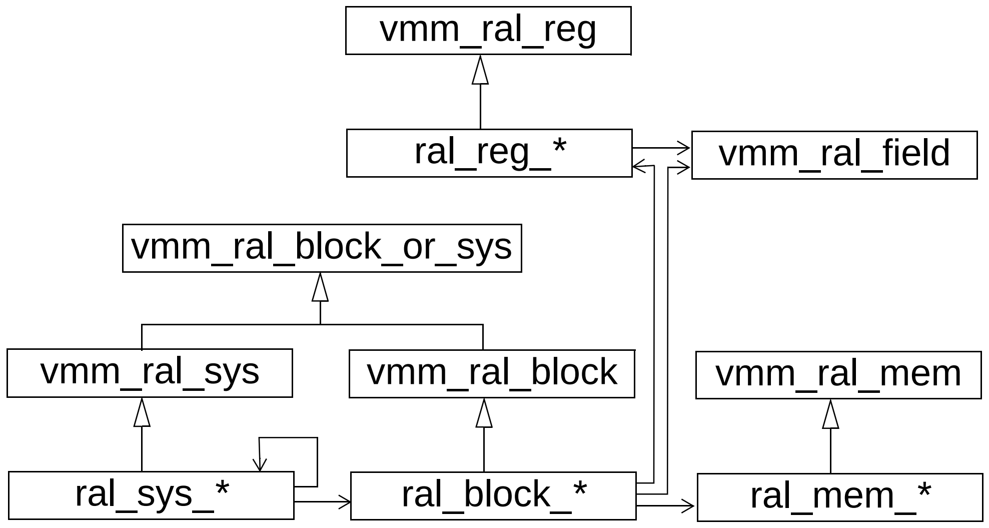
Example 5-10 System Abstraction Class with Block Array

```
class ral_sys_SoC extends vmm_ral_sys;
   ral_block_uart uart[2]
   ...
endclass: ral_sys_SoC
```

UML Diagram

The following UML diagram shows the relationships of the various classes in a complete RAL model. For detailed information, refer to Appendix B which contains detailed descriptions of each class, in alphabetical order.

Figure 5-1   UML Diagram of Complete RAL Model



6

Physical-layer Transactors

The generated abstraction layer provides a high-level interface mechanism to the registers and memories in a design. Testbenches can thus be written in terms of register and memory access without having to worry about addresses or bus cycles. However, the abstraction layer must eventually turn these accesses into physical read and write cycles to the design under verification.

It is not possible to write a generic abstraction layer on top of a specific physical interface. Thus, the RAL assumes a generic physical interface that can perform read and write cycles at specified addresses and return read and cycle completion status information. The RAL assumes it is connected to a generic transactor `vmm_rw_xactor` (see “vmm_rw_xactor”), extended from `rvm_xactor`. This generic transactor executes generic transactions described by the `vmm_rw_access` transaction descriptor. The behavior of the generic transactor is predefined in its own extension of the `vmm_xactor::main()` method which invokes two new virtual methods to execute burst or single transactions.

These generic transactions must be translated and executed on the actual physical interface of the design under verification. You accomplish this by instantiating an appropriate bus-functional model in an extension of the `vmm_rw_xactor` class and translating the generic `vmm_rw_access` transactions to appropriate transactions on the bus-functional model in an extension of the `vmm_rw_xactor::execute_single()` or `vmm_rw_xactor::execute_burst()` virtual methods. It is not necessary to further extend the `vmm_xactor::main()` method, although it can be done if you require more autonomous behavior.

The structure of a translation transactor is shown in Figure 6-1. The UML diagram of a translation transactor is shown in Figure 6-2.

Figure 6-1 Structure of Translation Transactors

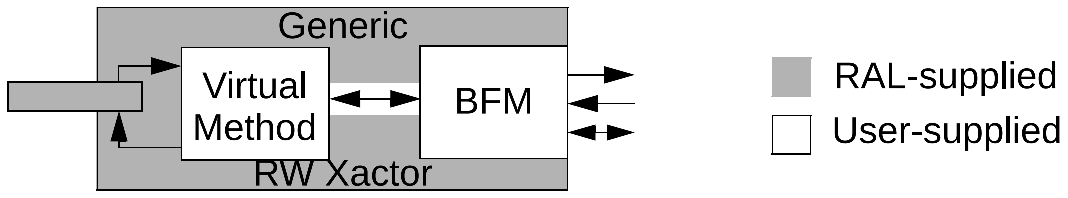
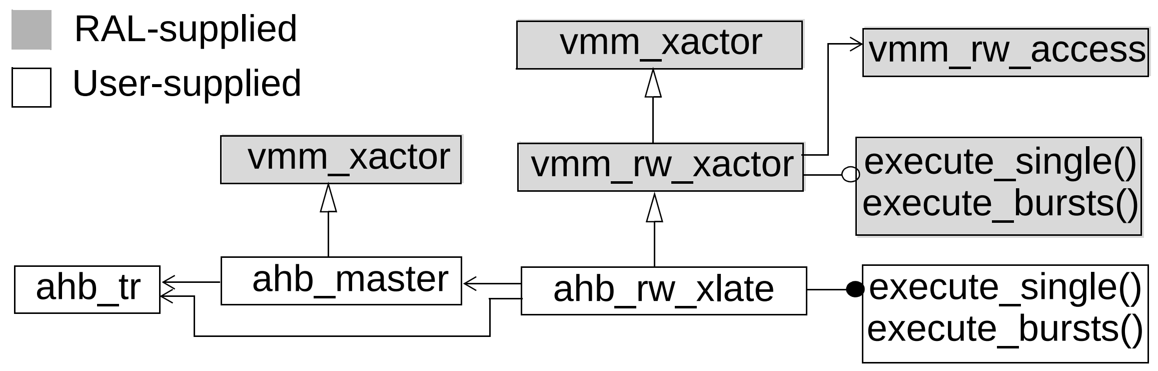


Figure 6-2 UML Diagram of Translation Transactor



The `vmm_xactor::start_xactor()`, `vmm_xactor::stop_xactor()`, and `vmm_xactor::reset_xactor()` methods must be overloaded to appropriately start, stop and reset the physical-level BFM instantiated in the generic transactor and used to execute the generic transactions when the generic transactor is started, stopped or reset.

The `execute_single()` task has no default implementation and must be overloaded. The `execute_burst()` task has a default implementation that executes burst transactions as equivalent single transactions. If the physical protocol does not support burst transactions, the `execute_burst()` task need not be overloaded.

You must define a translation transactor for each unique physical protocol on the design. Multiple physical interfaces using the same physical protocol can use multiple instances of the same translation transactor class.

Example 6-1 shows a translation transactor that executes the generic transactions on a AHB bus.

Example 6-1   Executing Generic Transactions on AHB Interface

```
class ahb_rw_xlate extends vmm_rw_xactor;
   ahb_master ahb;
   ahb_tr     randomized_tr;

   function new(string                instance,
                int unsigned          stream_id,
                ...
                vmm_rw_access_channel exec_chan = null);
      super.new("AHB RAL Master", instance, stream_id,
                exec_chan);

      this.ahb = new(...);
   endfunction: new

   virtual task execute_single(vmm_rw_access tr);
      // Translate the generic RW into a AHB RW
      ahb_tr xtr = new;

      xtr.data_id     = tr.data_id;
      xtr.scenario_id = tr.scenario_id;
      xtr.stream_id   = tr.stream_id;

      this.randomized_tr.randomize() with {
         addr == tr.addr << 2;
         if (tr.kind == vmm_rw::WRITE) {
            kind == ahb_tr::WRITE;
            data == tr.data;
         } else kind == ahb_tr::READ;
      };
      $cast(xtr, this.randomized_tr.copy());
      // Execute physical transaction,
      // assuming a blocking completion model.
      this.ahb.in_chan.put(xtr);
      if (tr.kind == vmm_rw::READ) tr.data = xtr.data;
      if (xtr.status == ahb_tr::NO_ERR)
         tr.status = vmm_rw::IS_OK;
      else tr.status = vmm_rw::ERROR;
    endtask: execute_single

   virtual void function start_xactor();
      super.start_xactor();
      this.ahb.start_xactor();
   endfunction: start_xactor
   ...
```

```
endclass: ahb_rw_xlate
```

Resetting Transactors and Disabling Threads

You should avoid resetting transactors involved in executing physical accesses in the middle of an operation. This may cause a mismatch between the mirrored value in the RAL model and the hardware. If an instance of an "vmm_rw_xactor" is reset using the method while an access transaction is executed, the transaction is forcibly completed with a `vmm_rw::RETRY` status.

A similar problem arises when a thread performing RAL operations is disabled or terminated. Information internal to the RAL model may be left in a spurious state. It will be necessary to invoke the "vmm_ral_reg::reset()" method on the register that was being accessed by the disabled or terminated thread.

7

Verification Environment

Testbenches and tests need a verification environment on which to execute. The predefined tests that come with RAL require some specific elements to be present in the verification environment on which they are built.

A RAL-based verification environment integrates a RAL model with the physical-level transactors that perform the read and write operation and with the design under verification.

A RAL-based environment must be derived from the `vmm_ral_env` class, which is itself derived from the `rvm_env` or `vmm_env` class. The `vmm_ral_env` class is a simple extension of its base class and only serves to introduce two predefined elements: a reset task to reset the design under verification repeatedly during simulation, and an instance of the `vmm_ral_access` transactor. For additional information about the `vmm_ral_env` class, see “vmm_ral_env”.

Once the RAL-based verification environment is suitable for executing the predefined RAL tests, you can expand it to be used as the verification environment for all user-defined tests. However, it is important that the design under verification be as idle and inactive as possible after the completion of the `vmm_ral_env::reset_dut()` method. The predefined tests built on top of the verification environment expect that the value of registers will remain unchanged between accesses. If a field value cannot be guaranteed to remain constant between accesses, its access mode should be specified as `other`.

A RAL-based verification environment is VMM compliant.

Top-level Module

The first step in creating a RAL-based verification environment is to create a top-level module instantiating the design under verification. The top-level module also instantiates all signals needed to connect to the pins of the design as well as clock generators.

Example 7-1 Top-level Module

```
module tb_top;

reg clk = 0;
wire rst = 0;
...
design dut(..., clk, rst, ...);

always #5ns clk = ~clk;

endmodule:tb_top
```

SystemVerilog Interfaces

If you are using SystemVerilog, the top-level module also includes all interface instances required by the physical-level transactors. The wires in the interface may be directly connected to the pins of the design under verification or they may be connected to the wires or variables inside the interface instances used by the design.

Example 7-2 SystemVerilog Interface Instances in Top-level Module

```
module tb_top;
...
ahb_if ahb0();

design dut(..., ahb0.hadr, ahb0.hdat, ...);
...
endmodule:tb_top
```

OpenVera Interfaces

If you are using OpenVera, the next step is to specify the `interface` instances and the binding of `interface` signals to virtual port instances. They are used to connect to the design under verification.

The physical-level transactors may come with macros to facilitate the specification of the `interface` and virtual port bindings.

Example 7-3 OpenVera Interface and Virtual Port Bindings

```
interface ahb0 {
   input         clk CLOCK       hdl_node "tb_top.pclk";
   output [31:0] adr PHOLD   #1  hdl_node "tb_top.haddr";
   inout  [31:0] dat PSAMPLE #-1
                     PHOLD   #1  hdl_node "tb_top.hdat"
   ...
}
```

```
bind ahb_ma_port ahb0_ma {
   adr ahb0.adr;
   dat ahb0.dat;
   ...
}
```

Environment Class

The next step is to create the environment class. The environment class must be derived from the `vmm_ral_env` class.

Example 7-4 RAL-Based Environment Class

```
class tb_env extends vmm_ral_env;
   ...
endclass: tb_env
```

The RAL-based environment base class has similar requirements to the `rvm_env` and `vmm_env` base classes. Additionally, there are three requirements for its implementation: RAL Model, Physical-level Transactors and Reset Tasks. The following sections describe these requirements.

RAL Model

An instance of the generated RAL model must exist. The name of the top-level class in the RAL model depends on the RALF description used to generate it.

You must then register the RAL model with the RAL access transactor instantiated by the base class, using the `vmm_rw_access::set_model()` method.

Example 7-5 RAL Model Instance for Block Design

```
class tb_env extends vmm_ral_env;
   ral_block_uart ral_model;
   ...
   function new();
      this.ral_model = new;
      this.ral.set_model(this.ral_model);
   endfunction: new
   ...
endclass: tb_env
```

Example 7-6 RAL Model Instance for System Design

```
class tb_env extends vmm_ral_env;
   ral_sys_SoC ral_model;
   ...
   function new();
      this.ral_model = new;
      this.ral.set_model(this.ral_model);
   endfunction: new
   ...
endclass: tb_env
```

Physical-level Transactors

The physical-level transactors required to translate between the generic read/write transactions and the actual physical interfaces on the design, must also be instantiated. According to the VMM guidelines, they are instantiated in the `build()` method.

An additional requirement of the RAL-based environment is that their instances must be registered with the RAL access transactor instantiated in the base class. If there is more than one domain (for example, physical interface) in the design under verification, the domain name corresponding to the physical interface driven by the transactor instance must be specified.

Example 7-7 and Example 7-8 show physical interface transactors for single-domain and two-domain designs.

Example 7-7 Physical Interface Transactor Instance for Single-domain Design

```
class tb_env extends vmm_ral_env;
   ...
   ral_ahb_master ahb;
   ...
   virtual function void build();
      super.build();

      this.ahb = new(...);
      this.ral.add_xactor(this.ahb);
      ...
   endfunction: build
   ...
endclass: tb_env
```

Example 7-8 Physical Interface Transactor Instance for Two-domain Design

```
class tb_env extends vmm_ral_env;
   ...
   ahb_rw_xlate ahb;
   pci_rw_xlate pci;
   ...
   virtual function void build();
      super.build();

      this.ahb = new(...);
      this.ral.add_xactor(this.ahb, "ahb");
      this.pci = new(...);
      this.ral.add_xactor(this.pci, "pci");
      ...
   endfunction: build
   ...
endclass: tb_env
```

Reset Task

Finally, instead of overloading the `vmm_env::reset_dut()` or `rvm_env::reset_dut_t()` method, the hardware reset sequence is specified by overloading the `vmm_ral_env::hw_reset()` method. The `vmm_ral_env::reset_dut()` method calls this new method to perform a hardware reset during the simulation process. Example 7-7 illustrates a hardware reset.

Example 7-9 Specifying Hardware Reset Sequence

```
class tb_env extends vmm_ral_env;
   ...
   virtual task hw_reset();
      tb_top.rst <= 1;
      repeat (3) @ (posedge tb_top.clk);
      tb_top.rst <= 0;
   endfunction: hw_reset
   ...
endclass: tb_env
```

8

Executing Pre-defined Tests

It is now possible to execute any of the predefined tests that come with RAL to verify the proper operation of the registers and memories in the design under verification. We recommend that you start with the simplest test—the hardware reset test—to debug the RAL-based environment, the physical transactors, and the design under verification to a level where they can be taken through more complicated tests.

Some of the predefined tests require that back-door access be available for registers or memories. See Chapter 10, "Back-door Access" for details on providing back-door access.

The predefined tests expect a file to be included before the `program` statement. In OpenVera, that file must be named `ral_env.vrh`. In SystemVerilog, the name of that file may be specified by defining the VMM_RAL_TEST_PRE_INCLUDE symbol. If that symbol is not defined, the file named ral_env.svh is includedthat file must be named `ral_env.svh`. You can use it to include other header or source files necessary to compile the test without errors according to your compilation strategy. You can also use it to set the timescale for the test. In SystemVerilog, a second file may be included inside the program block. the name of that file is specified by defining the VMM_RAL_TEST_POST_INCLUDE symbol. If that symbol is not defined, no file is included. You can use this second inclusion point to include files that must be outside of the $unit package or $root module, such as the generated RAL model if it contains hierarchical references for backdoor accesses.the default name of the verification environment used by the predefined tests is `tb_env`. If the verification environment has a different name, specify it by defining the RAL_TB_ENV macro in the `ral_env.vrh` or `ral_env.svh` file as follows:

```
‘define RAL_TB_ENV my_env
```

SystemVerilog

To compile and simulate the hardware reset test (named `hw_reset`) using SystemVerilog, use the following commands. You need to add the necessary command-line options and arguments to correctly compile all of the required transactors, the verification environment and design files.

```
% vcs -sverilog -ntb_opts rvm \
      ... $VCS_HOME/etc/rvm/sv/RAL/tests/hw_reset.sv ...
% ./simv
```

Using a VMM Open Source Installation

To compile and simulate the hardware reset test (named `hw_reset`) using a VMM Open Source distribution, use the following commands. You need to add the necessary command-line options and arguments to correctly compile all of the required transactors, the verification environment and design files.

```
% ... +incdir+$VMM_HOME/sv \
      $VMM_HOME/sv/RAL/tests/hw_reset.sv ...
```

OpenVera with VCS

To compile and simulate the hardware reset test (named `hw_reset`) using OpenVera under VCS (NTB), use the following commands. You need to add the necessary command-line options and arguments to correctly compile all of the required transactors, the verification environment and design files.

```
% vcs -ntb -ntb_opts rvm \
      $VCS_HOME/etc/rvm/vmm_ral.vrp \
      ... $VCS_HOME/etc/rvm/ov/RAL/tests/hw_reset.vr ...
% ./simv
```

OpenVera with Vera

To compile and simulate hardware reset test (named `hw_reset`) using OpenVera under Vera, use the following commands. You need to add the necessary command-line options and arguments to correctly compile all of the required transactors, the verification environment and design files.

```
% vera -cmp -vlog -I. -I $VERA_HOME/include ... \
       $VERA_HOME/pub_src/RAL/tests/hw_reset.vr .
% vcs -vera +vera_load=hw_reset.vro \
      +vera_mload=ral_env.vrl \
      ... vera_shell.v ...
% ./simv
```

Predefined Tests

The following predefined tests are included in RAL. You can augment or modify them to better verify your design.

`gen_html (SV only)`

Generates (crude) HTML documentation of the RAL model.

`hw_reset`

Applies hardware reset to the design and read all registers in the design. Verifies that the value read for each register corresponds to the specified reset value.

`bit_bash`

Verifies that all bits operate as specified, assuming that the DUT is completely idle. Does not test bits of mode *`vmm_ral::OTHER`* and *`vmm_ral::USERn`*.

`reg_access (SV only)`

Exercises all registers with a back-door access available using the following process:

- Skip the register if it contains unpredictable fields, such as `OTHER`
- Write `~reset` using frontdoor

- Check register content using backdoor
- Write reset value using backdoor
- Check register content using frontdoor

`mem_walk`

Walks through all addresses in `vmm_ral::RW` memories using the following process for address `k`:

- Write `~k` at address `k`
- If `k>0`, read address `k-1` and expect `~(k-1)`
- if `k>0`, write `k-1` at address `k-1`
- if last address, read address `k` and expect `~k`

`mem_access`

Walks through all addresses in memories with a back-door access available using the following process for address `k`:

- Write random value `v` at address `k` through frontdoor
- If memory is RW, expect `v` through back-door read
- Write `~v` through back-door write
- Read and expect `~k` through front-door read

`shared_access`

Exercises all shared registers and memories using the following process for each domain (requires at least one domain with READ capability or back-door access).

- Write a random value using the domain

- Check the content using all other domains, checking access rights

9

DUT Configuration

Once the correct operation of the registers and memories inside the DUT have been confirmed using the predefined tests, you must implement functional tests to verify the actual functionality of the design. This usually requires that you configure the DUT in some way.

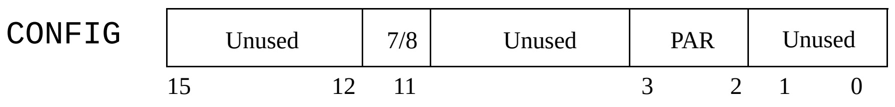
The `vmm_ral_env` verification environment is a VMM-compliant environment, therefore, the configuration of the DUT is implemented with an extension of the `vmm_ral_env::cfg_dut()` method, as shown in Example 9-1.

Example 9-1 Extension of the cfg_dut() Method in the Environment

```
class tb_env extends vmm_ral_env;
   ...
   virtual task cfg_dut();
      super.cfg_dut();
      ...
   endtask: cfg_dut
   ...
endclass: tb_env
```

The verification environment configures the DUT by writing to configuration fields by using the RAL model. All fields, registers, and memories are directly accessible through the RAL abstraction model by using the appropriate hierarchical reference to the field, register, or memory abstraction model. See "Understanding the Generated Model" for a description of how the structure of the RALF specification is used to generate the structure of the RAL abstraction model. For example, a UART with a configuration register as shown in Figure 9-1, could be configured through its RAL model as shown in Example 9-2.

Figure 9-1 UART Configuration Register



Example 9-2 Configuration Through the RAL Model

```
virtual task cfg_dut();
   vmm_rw::status_e status;
   super.cfg_dut();
   case (this.cfg.parity)
      NONE: this.ral_model.PAR.write(status, 2'b00);
      ODD : this.ral_model.PAR.write(status, 2'b01);
      EVEN: this.ral_model.PAR.write(status, 2'b10);
   endcase
   this.ral_model.DATA8.write(status,
                              this.cfg.use_8_bits);
endtask: cfg_dut
```

You can minimize the number of write cycles by using the mirror update capability of RAL. In the implementation of the DUT configuration process shown in Example 9-3, a single physical write cycle is performed instead of two, because both fields are located in the same register. Should the selected configuration correspond to the default configuration of the DUT, no write cycles are necessary or executed.

Example 9-3   Minimizing Physical Configuration Transactions

```
virtual task cfg_dut();
   vmm_rw::status_e status;
   super.cfg_dut();
   case (this.cfg.parity)
      NONE: this.ral_model.PAR.set(2’b00);
      ODD : this.ral_model.PAR.set(2’b01);
      EVEN: this.ral_model.PAR.set(2’b10);
   endcase
   this.ral_model.DATA8.set(this.cfg.use_8_bits);
   this.ral_model.update(status);
endtask: cfg_dut
```

If it is necessary to reuse the block configuration procedure between the block-level environment and the system-level environment, you should implement the bl ock configuration procedure in a task, as shown in Example 9-4. The block configuration task, takes as an argument, the block RAL model used to configure the block.

Example 9-4   Block Configuration Task

```
task uart_config(uart_cfg       cfg,
                 ral_block_uart ral_model);
   vmm_rw::status_e status;
   case (cfg.parity)
      NONE: ral_model.PAR.set(2’b00);
      ODD : ral_model.PAR.set(2’b01);
      EVEN: ral_model.PAR.set(2’b10);
   endcase
   ral_model.DATA8.set(cfg.use_8_bits);
   ral_model.update(status);
endtask: uart_config

virtual task cfg_dut();
   super.cfg_dut();
   uart_config(this.cfg, this.ral_model);
endtask: cfg_dut
```

Should the UART be present in a system, you can reuse the configuration task in the system environment. As shown in Example 9-5, you can use the same configuration task to configure multiple instances of the block.

Example 9-5   Reusing Block Configuration Task in System Configuration

```
virtual task cfg_dut();
   super.cfg_dut();
   uart_config(this.cfg.uart0, this.ral_model.uart0);
   uart_config(this.cfg.uart1, this.ral_model.uart1);
endtask: cfg_dut
```

10

Back-door Access

A 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 backdoor improves the efficiency of verifying registers and memories because 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 name 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.

Back-door access is limited only by the capabilities of the underlying language and simulation environment.

Back-door Read/Write vs. Peek/Poke

You can perform back-door access to registers and memory by using either the following read/write methods:

- “vmm_ral_field::read()”
- “vmm_ral_field::write()”
- “vmm_ral_mem::read()”
- “vmm_ral_mem::write()”
- “vmm_ral_reg::read()”
- “vmm_ral_reg::write()”

...or the following peek/poke methods:

- “vmm_ral_field::peek()”
- “vmm_ral_field::poke()”
- “vmm_ral_mem::peek()”
- “vmm_ral_mem::poke()”

- "vmm_ral_reg::peek()"
- "vmm_ral_reg::poke()"

The `peek()` methods return the actual value read using the backdoor 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 a `read-update` 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 frontdoor.

Note: Some design implementations may not allow values to be poked. For example, a read-only register that is implemented by sampling signal values or constants cannot be poked. In such instances, a `vmm_rw::ERROR` status should be returned.

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 zeroes 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 (unless the entire register is composed of read-only fields, in which case, the entire write operation is ignored).

Generated Backdoors

Automatically-generated back-door mechanisms are associated with their corresponding register or memory abstraction class when the RAL model containing these registers and memories is instantiated. However, in order to enable the automatic generation of back-door access, it is necessary to specify the hierarchical path to the HDL structures that implement the register or memory. This is accomplished by using the `hdl_path` attributes in “field”, “register”, “regfile”, “memory”, “block” and “system” instantiations of the RALF specification.

The generated backdoor simply concatenates the path elements specified in the individual `hdl_path` attributes to form the complete path to the target register or memory. For example, the RALF file shown in Example 10-1 would yield the path `S1_TOP_PATH.b1_i.dec.r1_reg` to the register `r1`.

Example 10-1   RALF Description with hdl_path Specifications

```
system s1 {
   ...
   block b1 (b1_i) @’h1000 {
     ...
      register r1 dec.r1_reg {
         ...
      }
   }
}
```

For a path to be well-formed, a RALF “regfile”, “block” or “system” must correspond to a design module or entity instance. For example, the (partial) RTL code shown in Example 10-2 represents the structure of the design matching the specification in Example 10-1.

Example 10-2 RTL Structure

```
module b1(...);
   ...
   always @ (posedge clk)
   begin: dec
      reg [7:0] r1_reg;
      if (rst) r1_reg <= 0;
      else if (...) r1_reg <= ...;
   end
   ...
endmodule

module s1(...);
   ...
   b1 b1_i(...);
   ...
endmodule

module tb_top;
   ...
   s1 dut(...);
   ...
endmodule
```

The absolute path to the instance of the DUT that corresponds to the RAL model is specified by defining the `*name*_TOP_PATH` symbol where name is the uppercase name of the top-level block or system in the RAL model. Using the structure shown in Example 10-2, the `S1_TOP_PATH` symbol must be defined to `tb_top.dut`, as shown in the following:

```
   % vcs ... +define+S1_TOP_PATH=tb_top.dut ... \
         ral_s1.sv ...
```

Arrays

If the RALF specification contains arrays of “system”, “block”, “regfile” or “register” instances (see “Arrays and Register Files” for more details on arrays of instances), the `hdl_path` attribute must contain a `%d` placeholder that will be replaced with the decimal index value of the instance in the array.

For example, the RALF file shown in Example 10-3 would generate the following paths to access the different instances of register `r1` located in each instance of the block `b1`:

- `S1_TOP_PATH.b1_i0.dec.r1_reg`
- `S1_TOP_PATH.b1_i1.dec.r1_reg`

Example 10-3   RALF Description with Array hdl_path Specifications

```
system s1 {
   ...
   block b1[2] (b1_i%d) @’h1000 +’h0010 {
     ...
      register r1 dec.r1_reg {
         ...
      }
   }
}
```

Of course, this implies that the module or `reg` instances corresponding to the arrays have matching names, as shown in Example 10-4.

Example 10-4 RTL Structure with Arrays of Instances

```
module s1(...);
   ...
   b1 b1_i0(...);
   b1 b1_i1(...);
   ...
endmodule
```

VHDL

Automatic back-door access generation is not currently supported for VHDL designs, or registers located in a portion of the design described using VHDL or encapsulated using VHDL (for example, inside a VHDL donut).

Target Structures

The automatically-generated back-door access code must make certain assumptions about the nature of the HDL code used to implement the register and memory being accessed. Although there are almost unlimited ways you can implement a register, there are only a few styles that are supported by the back-door access generator. It is important that, when implementing registers and memories in RTL code, a suitable coding style be used.

The following guidelines outline the restrictions on RTL structures used to implement registers and memories to enable automatic generation of their back-door access. Some of these restrictions may be removed in the future as the capabilities of the back-door access generator are improved.

If the target structures do not meet the requirements for automatic generation of back-door access, a user-defined back-door access mechanism must be created, as specified in "User-defined Backdoors" on page 15.

Writable Fields and memories must be implemented using "reg".

When performing a back-door write operation, a blocking procedural assignment is used. This requires that the target of the assignment be a `reg`.

Read-only fields may be implemented using wire, parameter or Boolean expression.

Such structures cannot be written to, therefore, only the read back-door access to a read-only field is generated. Attempting a back-door write to a read-only field will result in an error.

Example 10-5 Read-only Field Implemented Using an Expression

```
always @ (*)
begin
   if (wr) rdat = ’Z;
   else case (addr)
     ...
     16’h0010: rdat = {fifo_fl, fifo_mt};
     ...
   endcase
end
```

Example 10-6   RALF Description for Read-only Field

```
register r1 @’h0010{
   bytes 2;
   field mt (fifo_mt) {
      bits   1;
      reset  1;
      access ro;
   }
   field fl (fifo_fl) {
      bits   1;
      reset  0;
      access ro;
   }
}
```

Example 10-7   Alternative RALF Description for Read-only Field

```
register r1 (fifo_fl, fifo_mt) @’h0010{
   bytes 2;
   field mt {
      bits   1;
      reset  1;
   }
   field fl {
      bits   1;
      reset  0;
   }
}
```

A register may implement all of its fields in a single "reg".

A register may be composed of more than one field. All these different fields may be implemented in the same `reg` that implements the overall register. This implies that all bits in the register, up to the most-significant bits of the most-significant field, are implemented and there are no reserved or unused bits between fields. In that case, no `hdl_path` should be specified in field instantiations in the register specification.

For example, the register specified using the `register` definition shown in Example 10-8, can be implemented using the RTL code shown in Example 10-9. The `reg` named `r1_reg` is used to implement fields `f1` and `f2`.

Example 10-8   Register with Multiple Fields

```
register r1 (r1_reg) @'h0010{
   bytes 2;
   field f1 {
      bits   4;
      reset  4'hA;
   }
   field f2 {
      bits   8;
      reset  8'h55;
   }
}
```

Example 10-9   Single-reg Implementation of Register with Multiple Fields

```
reg [11:0] r1_reg;
always @ (posedge clk)
begin
   if (rst) r1_reg <= {8'h55, 4'hA};
   else if (wr) case (addr)
      ...
      16'h0010: r1_reg <= wdat;
      ...
   endcase
end

always @ (*)
begin
   if (wr) rdat = 'Z;
   else case (addr)
     ...
     16'h0010: rdat = r1_reg;
     ...
   endcase
end
```

If per-field peek()/poke() operations are required (not yet supported), each field instance should have its respective bit slice specified in its `hdl_path` attribute. For example, the register specified using the `register` definition shown in Example 10-10, can also be implemented using the RTL code shown in Example 10-9.

Example 10-10 Register with Multiple Fields

```
register r1 @'h0010{
   bytes 2;
   field f1 (r1_reg[3:0]) {
      bits   4;
      reset  4'hA;
   }
   field f2 (r1_reg[11:4]) {
      bits   8;
      reset  8'h55;
   }
}
```

A register may implement its fields in separate "reg".

A register may be composed of more than one field. All these different fields may be implemented in different `regs` that each implement one field. The register is the concatenation of these individual `regs`. This implementation allows reserved or unused bits between fields. In that case, the `hdl_path` must be specified in field instantiations in the register specification.

For example, the register specified using the `register` definition shown in Example 10-11, can be implemented using the RTL code shown in Example 10-12. The `regs` named `f1_reg` and `f2_reg` are used to implement fields `f1` and `f2` respectively. Additionally, both Example 10-5 and Example 10-6 show an example of a register implemented using separate constructs for separate read-only fields.

Example 10-11   Register with Multiple Fields

```
register r1 @’h0010{
   bytes 2;
   field f1 (f1_reg) {
      bits   4;
      reset  4’hA;
   }
   field f2 (f2_reg) @8 {
      bits   4;
      reset  4’h5;
   }
}
```

Example 10-12   Multiple-reg Implementation of Register with Multiple Fields

```
reg [3:0] f1_reg, f2_reg;
always @ (posedge clk)
begin
   if (rst) begin
      f1_reg <= 4’hA;
      f2_reg <= 4’h5};
   end
   else if (wr) case (addr)
      ...
      16’h0010: begin
         f1_reg <= wdat[3:0];
         f2_reg <= wdat[11:8];
      end
      ...
   endcase
end

always @ (*)
begin
   if (wr) rdat = ’Z;
   else case (addr)
     ...
     16’h0010: rdat = {f2_reg, 4’h0, f1_reg};
     ...
   endcase
end
```

A field may be implemented using multiple "reg".

Like registers, a field may be implemented as separate `regs`. For example, the register specified using the `register` definition shown in Example 10-13, can be implemented using the RTL code shown in Example 10-14. The `regs` named `f2a_reg` and `f2b_reg` are used to implement field `f2`.

Example 10-13 Field Implemented with Multiple regs

```
register r1 @'h0010{
   bytes 2;
   field f1 (f1_reg) {
      bits   4;
      reset  4'hA;
   }
   field f2 (f2a_reg, f2b_reg) @8 {
      bits   4;
      reset  4'h5;
   }
}
```

Example 10-14 Multiple-reg Implementation of a Fields

```
reg [3:0] f1_reg, f2a_reg, f2b_reg;
always @ (posedge clk)
begin
   if (rst) begin
      f1_reg <= 4'hA;
      {f2a_reg, f2b_reg} <= 4'h55};
   end
   else if (wr) case (addr)
      ...
      16'h0010: begin
         f1_reg <= wdat[3:0];
         {f2a_reg, f2b_reg} <= wdat[11:4];
      end
      ...
   endcase
end

always @ (*)
begin
   if (wr) rdat = 'Z;
   else case (addr)
     ...
     16'h0010: rdat = {f2a_reg, f2b_reg, f1_reg};
     ...
   endcase
end
```

A register may have a mix of read-only and writable fields.

Read-only fields cannot be written to, even with a backdoor. A register containing a mix of read-only and writable fields will skip the read-only fields during a back-door write operation.

A memory must be implemented using a single unpacked array.

A memory is accessed using the offset of the memory as the index of the array storing its content. Two memories cannot be modeled using the same array nor can a memory be implemented using the concatenation of multiple arrays (either bit-wise or address-wise).

For example, the memory specified using the `memory` definition shown in Example 10-15, can be implemented using the RTL code shown in Example 10-16. The `reg` named `m1_reg` is used to implement the entire memory.

Example 10-15   Memory Specification

```
memory m1 (m1_reg) @'h1000{
   size 1k;
   bits 16;
}
```

Example 10-16   Implementation of Memory with Unpacked Array

```
reg [15:0] m1_reg[1024];
always @ (posedge clk)
begin
   if (wr) casex (addr)
      ...
      16’b0001_00xx_xxxx_xxxx: m1_reg[addr[9:0]] <= wdat;
      ...
   endcase
end

always @ (*)
begin
   if (wr) rdat = ’Z;
   else casex (addr)
     ...
     16’b0001_00xx_xxxx_xxxx: rdat = m1_reg[addr[9:0]];
     ...
   endcase
end
```

Note: Automatic generation of back-door access to memories modeled using DesignWare models is not yet supported.

User-defined Backdoors

User-defined back-door mechanisms are instantiated and associated with their corresponding register or memory abstraction class in the implementation of the `vmm_ral_env::build()` method.

A user-defined register backdoor is provided through an extension of the “vmm_ral_reg_backdoor” class. A back-door write operation is implemented in the “vmm_ral_reg_backdoor::write()” virtual method whereas a back-door read operation is implemented in the "vmm_ral_reg_backdoor::read()" virtual method. This back-door access is then associated with a specific register through the "vmm_ral_reg::set_backdoor()" method.

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

If a memory contains error detection and correction codes (ECC), the memory backdoor must handle the generation of ECC bits on write operations. For more information, see "ECC Backdoor Access".

Implementing a Register Backdoor in OpenVera with Verilog DUT

The following steps detail and illustrate how to implement a register backdoor when using an OpenVera verification environment on a Verilog DUT. Note that it is a different approach than the one used in the automatically generated backdoor in OpenVera because this approach requires less typing and maintenance.

1. Create a port-less module containing read and write tasks. If a port-less module containing register back-door access tasks already exists, you can use the same module. If this is not the first register backdoor, go directly to step 2.

```
module backdoors;

   task reg_read(input  integer id,
                 output [63:0]  data);
   begin
      case (id)
         ...
         default: begin
            $display("Invalid register identifier %0d",
                     id);
            $finish;
         end
      endcase
   end
   endtask

   task reg_write(input integer id,
                  input [63:0]  data);
   begin
      case (id)
         ...
         default: begin
            $display("Invalid register identifier %0d",
                     id);
            $finish;
         end
      endcase
   end
   endtask

endmodule
```

2. Add a new choice to both `case` statements using the same integer value. The integer value must be unique and different from the other choices in the `case` statements. That integer value is now the unique integer identifier for that register.

```
task reg_read(...);
begin
   case (id)
      ...
      4: ...
      default: ...
   endcase
end
endtask

task reg_write(...);
```

```
begin
   case (id)
      ...
      4: ...
      default: ...
   endcase
end
endtask
```

3. Identify the absolute hierarchical access path to the register, starting with the topmost module. Identify the simulation constructs used to implement the register. The path and simulation constructs are subject to change should the structure of the DUT change. The path and constructs may also change between an RTL model and a gate-level model.

   For illustration purposes, the register is implemented using two `regs` named `field1` and `field2`, with an unused bit between them.

4. In the new choice in the read task, implement the necessary statements to read the current value of the register and return it in the `data` argument.

```
task reg_read(...);
begin
   case (id)
      ...
      4: data = {tb_top.dut...field2, 1’b0,
                 tb_top.dut...field1};
      default: ...
   endcase
end
endtask
```

5. In the new choice in the write task, implement the necessary statements to set the current value of the register to the value specified in the `data` argument.

```
task reg_write(...);
begin
```

```
   case (id)
      ...
      4: begin
            reg unused;
            {tb_top.dut...field2, unused,
             tb_top.dut...field1} = data;
         end
      default: ...
   endcase
end
endtask
```

6. In the file that defines the verification environment class (extended from `vmm_ral_env`), define the register back-door read and register back-door write tasks as available external tasks. If this definition already exists, because this is not the first register backdoor to be implemented, go to step 8.

```
hdl_task reg_bkdr_rd(integer        id,
                     var bit [63:0] data)
   "backdoors.reg_read";

hdl_task reg_bkdr_wr(integer    id,
                     bit [63:0] data)
   "backdoors.reg_write";
```

7. In the same file, create an extension of the "vmm_ral_reg_backdoor" class with an integer identifier assigned at construction time. Use this integer identifier when calling the Verilog tasks in the extensions of the virtual methods.

```
class reg_backdoors extends vmm_ral_reg_backdoor {
   local id;

   task new(integer id)
   {
      this.id = id;
   }

   virtual function vmm_rw::status_e
      write_t(bit [63:0] data,
              integer    data_id,
              integer    scenario_id,
              integer    stream_id)
   {
      reg_bkdr_wr(this.id, data);
      write_t = vmm_rw::IS_OK;
   }

   virtual function vmm_rw::status_e
      read_t(var bit [63:0]  data,
             integer    data_id,
             integer    scenario_id,
             integer    stream_id)
   {
      mem_bkdr_rd(this.id, data);
      read_t = vmm_rw::IS_OK;
   }
}
```

8. In the extension of the `vmm_ral_env::build()` method, allocate an instance of the extended "vmm_ral_reg_backdoor" class with the unique integer identifier used in the `case` statement choices created in step 2. Then associate that new instance with the corresponding register abstraction class in the RAL model.

```
   task tb_env::build()
   {
      super.build();
      ...
      {
         reg_backdoors bkdr = new(4);
         this.ral_model.regname.set_backdoor(bkdr);
      }
   }
```

Implementing a Register Backdoor in SystemVerilog

The following steps detail and illustrate how to implement a register backdoor when using a SystemVerilog verification environment and DUT:

1. In the file that defines the verification environment class (extended from `vmm_ral_env`), create an extension of the "vmm_ral_reg_backdoor" class with an integer identifier assigned at construction time. If this class extension exists because this is not the first register backdoor, go directly to step 2.

```
class reg_backdoors extends vmm_ral_reg_backdoor;
   local int id;

   function new(int id);
      this.id = id;
   endfunction: new

   virtual task write(output vmm_rw::status_e status,
                      input  bit [63:0]       data,
                      input  int              data_id,
                    input  int              scenario_id,
                     input  int              stream_id);
      case (id)
         ...
         default: begin
            $display("Invalid register identifier %0d",
                     id);
            $finish;
         end
      endcase
   endtask: write

   virtual task read(output vmm_rw::status_e status,
                     output bit [63:0]       data,
                     input  int              data_id,
                   input  int              scenario_id,
                    input  int              stream_id);
      case (id)
         ...
         default: begin
            $display("Invalid register identifier %0d",
                     id);
            $finish;
         end
      endcase
   endtask: write
endclass: reg_backdoors
```

2. Add a new choice to both `case` statements using the same integer value. This integer value is now the unique integer identifier for that register.

```
virtual task write(output vmm_rw::status_e status,
                   input  bit [63:0]       data,
                   input  int              data_id,
                   input  int              scenario_id,
                   input  int              stream_id);
   case (id)
      ...
      4: ...
      default: ...
   endcase
endtask: write

virtual task read(output vmm_rw::status_e status,
                  output bit [63:0]       data,
                  input  int              data_id,
                  input  int              scenario_id,
                  input  int              stream_id);
   case (id)
      ...
      4: ...
      default: ...
   endcase
endtask: read
```

3. Identify the absolute hierarchical access path to the register, starting with the topmost module. Identify the simulation constructs used to implement the register. The path and simulation constructs are subject to change should the structure of the DUT change. They may also change between an RTL model and a gate-level model.

   For illustration purposes, the register is implemented using two `reg` named `field1` and `field2`, with an unused bit between them.

4. In the new choice in the read virtual task, implement the necessary statements to read the current value of the register and return it in the `data` argument.

```
virtual task read(output vmm_rw::status_e status,
                  output bit [63:0]       data);
  case (id)
     ...
     4: data = {tb_top.dut...field2, 1'b0,
                tb_top.dut...field1};
     default: ...
  endcase
endtask: read
```

5. In the new choice in the write task, implement the necessary statements to set the current value of the register to the value specified in the `data` argument.

```
virtual task write(output vmm_rw::status_e status,
                   input  bit [63:0]       data,
                   input  int              data_id,
                  input  int              scenario_id,
                   input  int              stream_id);
  case (id)
     ...
     4: begin
           bit unused;
           {tb_top.dut...field2, unused,
            tb_top.dut...field1} = data;
        end
     default: ...
  endcase
endtask: write
```

6. In the extension of the `vmm_ral_env::build()` method, allocate an instance of the extended "vmm_ral_reg_backdoor" class with the unique identifier used in the case statement choice created in step 2. Then associate that new instance with the corresponding register abstraction class in the RAL model.

```
function void tb_env::build();
   super.build();
   ...
```

```
      begin
         reg_backdoors bkdr = new(4);
         this.ral_model.regname.set_backdoor(bkdr);
      end
   endfunction: build
```

Implementing a Memory Backdoor in OpenVera with DWMM

The following steps detail and illustrate how to implement a memory backdoor when using an OpenVera verification environment on a DesignWare Memory Model (DWMM):

1. Include the file `${VMM_HOME}/ov/RAL/example/dwmm/vmm_ral_dwmm_backdoor.vr` in the file that implements the verification environment (based on `vmm_ral_env`). If this file has already been included because this is not the first memory backdoor, go directly to step 2.
2. Identify the model ID of the `DWMM` instance.
3. In the extension of the `vmm_ral_env::build()` method, allocate an instance of the `vmm_ral_dwmm_backdoor` class with the `DWMM` instance identifier identified in step 2. Then associate that new instance with the corresponding memory abstraction class in the RAL model.

```
task tb_env::build()
{
   super.build();
   ...
   {
      vmm_ral_dwmm_backdoor bkdr = new(2);
      this.ral_model.memname.set_backdoor(bkdr);
   }
}
```

Implementing a Memory Backdoor in OpenVera with Verilog DUT

The following steps detail and illustrate how to implement a memory backdoor when using an OpenVera verification environment on a Verilog DUT:

1. Create a port-less module containing read and write tasks. If a port-less module containing register back-door access tasks already exists, you can use this module. If these tasks exist because this is not the first memory backdoor, go directly to step 2.

```
module backdoors;

   task mem_read(input  integer id,
                 input  [63:0]  offset,
                 output [63:0]  data);
   begin
      case (id)
         ...
         default: begin
            $display("Invalid memory identifier %0d",
                     id);
            $finish;
         end
      endcase
   end
   endtask

   task mem_write(input integer id,
                  input [63:0]  offset,
                  input [63:0]  data);
   begin
      case (id)
         ...
         default: begin
            $display("Invalid memory identifier %0d",
                     id);
            $finish;
         end
      endcase
   end
   endtask

endmodule
```

An integer identifier—arbitrarily assigned but *unique*—is used to identify the memory requiring back-door access. It will be necessary to translate from that identifier to the actual hierarchical access path in the `case` statement.

2. Add a new choice to both `case` statements using the same integer value. The integer value must be unique from all of the other integer values in the respective `case` statements. This integer value is now the unique integer identifier for that memory.

```
task mem_read(...);
begin
   case (id)
      ...
      2: ...
      default: ...
   endcase
end
endtask

task mem_write(...);
begin
   case (id)
      ...
      2: ...
      default: ...
   endcase
end
endtask
```

3. Identify the absolute hierarchical access path to the memory, starting with the topmost module. Identify the simulation constructs used to implement the memory. The path and simulation constructs are subject to change should the structure of the DUT changes. They may also change between an RTL model and a gate-level model.

For illustration purposes, the memory is implemented using two 8-bit arrays named `byte0` and `byte1`, with a parity bit. Bit #16 is used as an indication of the parity validity.

4. In the new choice in the read task, implement the necessary statements to read the current value of the memory at the specified offset and return it in the `data` argument.

```
task reg_read(...);
begin
   case (id)
      ...
      2: begin
           reg par;
           data[15:0] = {tb_top.dut...byte1[offset],
                         tb_top.dut...byte0[offset]};
           par = ^data[15:0];
           data[16] = par ^
                      tb_top.dut...parity[offset];
         end
      default: ...
   endcase
end
endtask
```

5. In the new choice in the write task, implement the necessary statements to set the current value of the memory at the specified address to the value specified in the `data` argument.

```
task reg_write(...);
begin
   case (id)
      ...
      2: begin
            reg par = (^data[15:0]) ^ data[16];
            tb_top.dut...parity[offset] = par;
            {tb_top.dut...byte1[offset],
            tb_top.dut...byte0[offset]} = data;
         end
      default: ...
   endcase
end
endtask
```

6. In the file that defines the verification environment class (extended from `vmm_ral_env`), define the memory back-door read and write tasks as available external tasks. If this definition already exists, because this is not the first memory backdoor to be implemented, go to step 8.

```
hdl_task mem_bkdr_rd(integer        id,
                     bit [63:0]     offset,
                     var bit [63:0] data)
   "backdoors.mem_read";

hdl_task mem_bkdr_wr(integer    id,
                     bit [63:0] offset,
                     bit [63:0] data)
   "backdoors.mem_write";
```

7. In the same file, create an extension of the "vmm_ral_mem_backdoor" class with an integer identifier assigned at construction time. Use this integer identifier when calling the Verilog tasks in the extensions of the virtual methods.

```
class mem_backdoors extends vmm_ral_mem_backdoor {
   local id;

   task new(integer id)
   {
      this.id = id;
   }

   virtual function vmm_rw::status_e
      write_t(bit [63:0] offset,
              bit [63:0] data,
              integer    data_id,
              integer    scenario_id,
              integer    stream_id)
   {
      mem_bkdr_wr(this.id, offset, data);
      write_t = vmm_rw::IS_OK;
   }

   virtual function vmm_rw::status_e
      read_t(bit [63:0] offset,
                 var bit [63:0]  data,
                 integer    data_id,
                 integer    scenario_id,
                 integer    stream_id)
      {
         mem_bkdr_rd(this.id, offset, data);
         read_t = vmm_rw::IS_OK;
      }
   }
```

8. In the extension of the `vmm_ral_env::build()` method, allocate an instance of the extended "vmm_ral_mem_backdoor" class with the unique identifier used in the case statement choice created in step 2. Then, associate that new instance with the corresponding memory abstraction class in the RAL model.

```
task tb_env::build()
{
   super.build();
   ...
   {
      mem_backdoors bkdr = new(2);
      this.ral_model.memname.set_backdoor(bkdr);
   }
}
```

Implementing a Memory Backdoor in SystemVerilog

The following steps detail and illustrate how to implement a memory backdoor when using a SystemVerilog verification environment and DUT:

1. In the file that defines the verification environment class (extended from `vmm_ral_env`), create an extension of the "vmm_ral_mem_backdoor" class with an integer identifier assigned at construction time. If this class extension exists, because this is not the first memory backdoor, go directly to step 2.

```
class mem_backdoors extends vmm_ral_mem_backdoor;
   local int id;
   function new(int id);
      this.id = id;
   endfunction: new

   virtual task write(output vmm_rw::status_e status,
                      input  bit [63:0]       offset,
                      input  bit [63:0]       data,
                      input  int              data_id,
                    input  int              scenario_id,
                     input  int              stream_id);
      case (id)
         ...
         default: begin
            $display("Invalid memory identifier %0d",
                     id);
            $finish;
         end
      endcase
   endtask: write

   virtual task read(output vmm_rw::status_e status,
                     input  bit [63:0]       offset,
                     output bit [63:0]       data,
                     input  int              data_id,
                   input  int              scenario_id,
                    input  int              stream_id);
      case (id)
         ...
         default: begin
            $display("Invalid memory identifier %0d",
                     id);
            $finish;
         end
      endcase
   endtask: write
endclass: mem_backdoors
```

2. Add a new choice to both `case` statements using the same integer value. This integer value is now the unique integer identifier for that memory.

```
virtual task write(output vmm_rw::status_e status,
                   input  bit [63:0]       offset,
                   input  bit [63:0]       data,
                     input  int              data_id,
                     input  int              scenario_id,
                     input  int              stream_id);
   case (id)
      ...
      2: ...
      default: ...
   endcase
endtask: write

virtual task read(output vmm_rw::status_e status,
                  input  bit [63:0]       offset,
                  output bit [63:0]       data,
                  input  int              data_id,
                  input  int              scenario_id,
                  input  int              stream_id);
   case (id)
      ...
      2: ...
      default: ...
   endcase
endtask: read
```

3. Identify the absolute hierarchical access path to the memory, starting with the topmost module. Identify the simulation constructs used to implement the memory. The path and simulation constructs are subject to change should the structure of the DUT change. They may also change between an RTL model and a gate-level model.

   For illustration purposes, the memory is implemented using an associative array.

4. In the new choice in the read virtual task, implement the necessary statements to read the current value of the memory at the specified location and return it in the `data` argument.

```
virtual task read(output vmm_rw::status_e status,
                  input  bit [63:0]       offset,
                  output bit [63:0]       data);
   case (id)
      ...
      2: begin
```

```
            if (tb_top.dut...ram.exists(offset))
               data = tb_top.dut...ram[offset];
            else data = ’0;
         end
      default: ...
   endcase
endtask: read
```

5. In the new choice in the write task, implement the necessary statements to set the current value of the memory at the specified location to the value specified in the `data` argument.

```
virtual task write(output vmm_rw::status_e status,
                   input  bit [63:0]       offset,
                   input  bit [63:0]       data,
                   input  int              data_id,
                   input  int              scenario_id,
                   input  int              stream_id);
   case (id)
      ...
      2: tb_top.dut...ram[offset] = data;
      default: ...
   endcase
endtask: write
```

6. In the extension of the `vmm_ral_env::build()` method, allocate an instance of the extended "vmm_ral_mem_backdoor" class with the unique identifier used in the `case` statement choice created in Step 2. Then associate that new instance with the corresponding memory abstraction class in the RAL model.

```
function void tb_env::build();
   super.build();
   ...
   begin
      mem_backdoors bkdr = new(2);
      this.ral_model.memname.set_backdoor(bkdr);
   end
endfunction: build
```

11

User-defined Field Access

The mirror included with RAL is not intended to be a scoreboard for the field values. It is a best guess effort at the current value of the fields based on observed read and write access to the DUT. If the value of a field is modified by the DUT, or otherwise updated, it is not possible to constantly update it to reflect field values updated or modified by the design. However, if the content of the field can be predicted based on the write and read operations to it, then the mirror can accurately predict its content and its correct operation can be verified.

The RAL contains several predefined field access modes, as specified in "field". The access modes are used, in combination with observed read and write operations, to determine the expected value of a field. You can use the `OTHER` and `USER`n access modes to specify that the behavior of the field does not fall within any of the predefined access modes.

Although most fields fall within one of the predefined categories, it is possible to design a field that behaves predictably but differently from the predefined access modes. It is possible to implement any user-defined behavior through "vmm_ral_field" callback extensions of the "vmm_ral_field_callbacks::pre_write()" and "vmm_ral_field_callbacks::post_read()" methods.

For example, a field that can only be written once it has been read, can, at least once, be modelled using the `vmm_ral::OTHER` or `vmm_ral::USERx` access mode and the following callback extensions:

```
class write_after_read extends vmm_ral_field_callbacks;
   local bit ok_to_write = 1;
   virtual task pre_write(vmm_ral_field         field,
                          ref bit [63:0]        wdat,
                          ref vmm_ral::path_e   path,
                          ref string            domain);
      if (!ok_to_write) wdat = field.get();
      ok_to_write = 0;
   endtask: pre_write

   virtual task post_read(input vmm_ral_field     field,
                          ref   bit [63:0]        rdat,
                          input vmm_ral::path_e   path,
                          input string            domain,
                         ref   vmm_rw::status_e  status);
      ok_to_write = 1;
   endtask: post_read
endclass: write_after_read
```

You can modify the behavior of any field to the user-specified behavior by simply registering the callback extension with the appropriate field abstraction class instance as follows:

```
begin
   write_after_read cb = new;
   this.ral_model.blk.magic_field.append_callback(cb);
end
```

```
this.ral_model.get_fields(flds);
foreach (flds[i]) begin
   if (flds[i].get_access() == ram_ral::USER0) begin
      write_after_read cb = new;
      flds[i].append_callback(cb);
   end
end
```

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. However, it is questionable whether or not it is suitable or functionally correct to use the field in that fashion.

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 depends on the hardware functionality. If the *hardware* does not prevent the subsequent write operation, then the field should be specified as read-write because that would accurately reflect the actual behavior of the field.

If you wish to include usage assertions to specific fields (for example, specifying that a configuration field is never written to more than once, despite the fact that it is physically possible), use a callback extension registered with the field as shown in the following example:

```
class config_once extends vmm_ral_field_callbacks;
   local bit written = 0;
   virtual task pre_write(vmm_ral_field          field,
                          ref bit [63:0]         wdat,
                          ref vmm_ral::path_e    path,
                          ref string            domain);
      if (this.written) ‘vmm_error(field.log, "...");
   endtask: pre_write

   virtual task post_read(input vmm_ral_field     field,
                          ref   bit [63:0]        rdat,
                          input vmm_ral::path_e   path,
                          input string            domain,
                         ref   vmm_rw::status_e  status);
      this.written = 1;
   endtask: post_read

   task reset();
      this.written = 0;
   endtask: reset
endclass: config_once
```

These usage assertions should be registers in the `vmm_env::config_dut()` method extension to avoid them from being triggered by the predefined tests. For more information, see “Predefined Tests”.

12

Memories with ECC

By default, every bit in each location in a memory is assumed to be user-defined. However, in certain high-reliability designs, the content of memory locations may be protected by extra error detection and correction codes (ECC). Each memory location contains additional ECC bits that the design creates during write operations. The design checks the ECC bits during read operations and indicates an error if a discrepancy is detected.

The presence and value of the ECC bits are transparent to the outside world. However, it is important that their functionality be verified. Furthermore, if you use backdoor write access, it is important to set the ECC bits properly so that an error is not indicated if the memory location is subsequently read through a physical access.

The number of possible memory protection mechanisms is potentially infinite and only limited by the imagination and requirements of the designers. To support arbitrary memory protection mechanisms, it is possible to augment the default unprotected access mechanism with any user-defined protection scheme.

ECC Backdoor Access

When performing backdoor write operations, it is necessary to correctly set the ECC bits to avoid ECC errors when a physical interface subsequently reads these memory locations. It may also be useful or necessary to have direct access to the ECC bits because these bits are created and used entirely within the design, and can only be accessed through backdoor access.

The default backdoor data path is 64-bits wide. This width is defined by the size of the `data` argument in all of the `read()` and `write()` methods in RAL, not the hardware being verified. As long as data and ECC bits for each memory location fit in the 64-bit data path, it is possible to access ECC bits through the existing RAL memory backdoor mechanism. If more than 64 bits are necessary to access the ECC bits along with the data, simply define the `VMM_RAL_DATA_WIDTH` symbol to the required width (see "Maximum Data Size" for more details). For example, a 64-bit memory with an 8-bit ECC protection code would require that the `VMM_RAL_DATA_WIDTH` be defined as 72 or greater.

When using a memory backdoor access, simply concatenate the ECC bits to the data bits, as shown in Example 12-1. To avoid having the users compute the ECC bit values before or after each backdoor access, it is better to transfer an indication of the ECC bit correctness rather than the ECC bit value itself. ECC is about correctness, not absolute value. This further allows the encapsulation of the ECC algorithm within the memory model.

Example 12-1 Reading and Writing ECC Bits Through Backdoor Access

```
virtual task write(output vmm_rw::status_e status,
                   input  bit [63:0]       offset,
                   input  bit [63:0]       data,
                   input  int              data_id,
                   input  int              scenario_id,
                   input  int              stream_id);
    // Corrupt ECC bit if it is set in the data to write
    data[35:32] = this.ecc.compute(data[31:0])
                 ^ data[35:32];
   tb_top.dut.mem0[offset] = data;
   status = vmm_rw::IS_OK;
endtask: write

virtual task read(output vmm_rw::status_e status,
                  input  bit [63:0]       offset,
                  output bit [63:0]       data,
                  input  int              data_id,
                  input  int              scenario_id,
                  input  int              stream_id);
   // Report invalid ECC bits
   data = tb_top.dut.mem0[offset];
   data[35:32] = this.ecc.compute(data[31:0])
                ^ data[35:32];
   status = vmm_rw::IS_OK;
endtask: read
```

ECC Physical Access

It may be useful or necessary to inject errors in the ECC or data bits to verify the functionality of the error detection circuitry in the design.

Due to the fact that these bits are created and used entirely within the design, they are not visible or accessible when performing physical access. However, it is possible to gain visibility over the ECC bits before and after a physical access through extensions of the `vmm_ral_mem` callback methods "vmm_ral_mem_callbacks::post_write()" and "vmm_ral_mem_callbacks::pre_read()".

After the completion of a physical write operation, it is possible to modify the data or ECC bits that were stored in the memory through a direct access, as shown in Example 12-2. Any subsequent read operation to that memory location indicates an ECC error.

Example 12-2 Injecting ECC Errors

```
virtual task post_write(vmm_ral_mem           mem,
                        bit [63:0]            offset,
                        bit [63:0]            wdat,
                        vmm_ral::path_e       path,
                        string                domain,
                        ref vmm_rw::status_e  status);
   bit [35:0] ecc_wdat;
   ecc_wdat = {this.ecc.compute(wdat), wdat[31:0]};
   if (error_on_write) begin
      ecc_wdat ^= 1'b1 << error_on_bit;
   end
   tb_top.dut.mem0[offset] = ecc_wdat;
endtask: post_write
```

Before a physical read operation starts, it is possible to modify the data or ECC bits that are stored in the memory location that is about to be read through a direct access. The subsequent read operation to that memory location indicates an ECC error. It is a good idea to restore the original content of the memory location after the read operation is complete to avoid further ECC errors when reading that memory location. Otherwise, this error injection mode would be no different than an error injected during the write operation.

```
virtual task pre_read(vmm_ral_mem           mem,
                      ref bit [63:0]        offset,
                      ref vmm_ral::path_e   path,
                      ref string            domain);
   if (error_on_read) begin
      ecc_rdat = tb_top.dut.mem0[offset];
      tb_top.dut.mem0[offset] = ecc_rdat
                              ^ (1'b1 << error_on_bit);
   end
endtask: pre_read
```

13

Non-linear, Non-mapped Access

By default, the entire address space of registers and memories is assumed to be linearly mapped into the address space of the block that instantiates it. Each register or location in a memory corresponds to a unique address in the block.

However, you can use different access 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.

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

Non-mapped register or memory does not consume any address space. Their offset should be specified as `@none` when their specification is instantiated in a "block" or "regfile".

User-defined Register Access

A user-defined register access is provided by an extension of the "vmm_ral_reg_frontdoor" class. A user-defined write operation is implemented in the "vmm_ral_reg_frontdoor::write()" virtual method whereas a user-defined read operation is implemented in the "vmm_ral_reg_frontdoor::read()" virtual method. This user-defined register access mechanism is then associated with a specific memory by the "vmm_ral_reg::set_frontdoor()" method.

User-defined register access mechanisms are instantiated and associated with their corresponding register abstraction class in the implementation of the `vmm_ral_env::build()` method.

The following steps detail and illustrate how to implement a user-defined register access mechanism, using an indexed access mechanism:

1. In the file that defines the verification environment class (extended from `vmm_ral_env`), create an extension of the "vmm_ral_reg_frontdoor" class. Provide a reference to any element of the RAL model or environment that needs to be used to perform read or write access through the constructor.

```
class indexed_reg extends vmm_ral_reg_frontdoor;
   local vmm_ral_reg offset;
   local vmm_ral_reg data;
   local bit [7:0]   addr;

   function new(vmm_ral_reg offset,
                vmm_ral_reg data
```

```
                 bit [7:0]   addr);
      this.offset = offset;
      this.data   = data;
      this.addr   = addr;
   endfunction: new

   virtual task write(output vmm_rw::status_e status,
                      input  bit [63:0]       data,
                      input  int              data_id,
                     input  int              scenario_id,
                     input  int               stream_id);
      this.offset.write(status, this.addr,
                        data_id, scenario_id, stream_id);
      if (status != vmm_rw::IS_OK) return;
      this.data.write(status, data,
                      data_id, scenario_id, stream_id);
   endtask: write

   virtual task read(output vmm_rw::status_e status,
                     output bit [63:0]       data,
                     input  int              data_id,
                    input  int              scenario_id,
                    input  int               stream_id);
      this.offset.write(status,this. addr,
                        data_id, scenario_id, stream_id);
      if (status != vmm_rw::IS_OK) return;
      this.data.read(status, data,
                     data_id, scenario_id, stream_id);
   endtask: read
endclass: indexed_reg
```

2. In the extension of the `vmm_ral_env::build()` method, allocate an instance of the extended "vmm_ral_reg_frontdoor" class. Then associate that new instance with the corresponding register abstraction class in the RAL model.

```
function void tb_env::build();
   super.build();
   ...
   begin
      indexed_reg idxreg = new(this.ral_model.mem_idx,
                               this.ral_model.mem_data, 'h0F);
      this.ral_model.regname.set_frontdoor(idxreg);
   end
endfunction: build
```

User-defined Memory Access

A user-defined memory access is provided by an extension of the “vmm_ral_mem_frontdoor” class. A user-defined write operation is implemented in the “vmm_ral_mem_frontdoor::write()” virtual method whereas a user-defined read operation is implemented in the “vmm_ral_mem_frontdoor::read()” virtual method. This user-defined memory access mechanism is then associated with a specific memory by the “vmm_ral_mem::set_frontdoor()” method.

User-defined memory access mechanisms are instantiated and associated with their corresponding memory abstraction class in the implementation of the `vmm_ral_env::build()` method.

The following steps detail and illustrate how to implement a user-defined memory access mechanism, using an indexed access mechanism:

1. In the file that defines the verification environment class (extended from `vmm_ral_env`), create an extension of the “vmm_ral_mem_frontdoor” class. Provide a reference to any element of the RAL model that needs to be used to perform read or write access through the constructor.

```
class indexed_mem extends vmm_ral_mem_frontdoor;
   local vmm_ral_reg offset;
   local vmm_ral_reg data;

   function new(vmm_ral_reg offset,
                vmm_ral_reg data);
      this.offset = offset;
      this.data   = data;
   endfunction: new
   virtual task write(output vmm_rw::status_e status,
                      input  bit [63:0]       offset,
                      input  bit [63:0]       data,
                      input  int              data_id,
                    input  int              scenario_id,
                    input  int               stream_id);
      this.offset.write(status, offset,
                        data_id, scenario_id, stream_id);
      if (status != vmm_rw::IS_OK) return;
      this.data.write(status, data,
                      data_id, scenario_id, stream_id);
   endtask: write

   virtual task read(output vmm_rw::status_e status,
                     input  bit [63:0]       offset,
                     output bit [63:0]       data,
                     input  int              data_id,
                   input  int              scenario_id,
                   input  int               stream_id);
      this.offset.write(status, offset,
                        data_id, scenario_id, stream_id);
      if (status != vmm_rw::IS_OK) return;
      this.data.read(status, data,
                     data_id, scenario_id, stream_id);
   endtask: read
endclass: indexed_mem
```

2. In the extension of the `vmm_ral_env::build()` method, allocate an instance of the extended “vmm_ral_mem_frontdoor” class. Then associate that new instance with the corresponding memory abstraction class in the RAL model.

```
function void tb_env::build();
   super.build();
   ...
   begin
      indexed_mem idxram = new(this.ral_model.mem_idx,
                               this.ral_model.mem_data);
      this.ral_model.memname.set_frontdoor(idxram);
   end
endfunction: build
```

14

Functional Coverage Model

Optionally, you can generate a RAL model with one or more predefined functional coverage models to measure how thoroughly the various host-accessible elements are exercised by your functional verification suite.

The default generated RAL model does not contain any functional coverage model. To generate a coverage model, `ralgen` must be invoked with the `-c` option. The argument to the `-c` option determines which coverage model is included in the RAL model:

Use `-c` b to generate the register bits coverage model.

-c a

Generate the address map coverage model.

-c f

Generate the field value coverage model.

Multiple functional coverage models can be generated in the same RAL model by specifying the `-c` option multiple times or specifying multiple arguments to a single `-c` option. For example, the following commands are equivalent:

```
% ralgen -c b -c a ...
% ralgen -c ba ...
```

Even though the generated RAL model may contain one or more functional coverage models, they are not enabled by default. This is necessary in order to reduce the memory footprint of a RAL model, as some functional coverage models can be significant in size, and to improve the runtime performance of simulations as the collection of coverage metrics and the writing of functional coverage databases incurs a significant overhead. Therefore, It is necessary to explicitly enable a functional coverage model when a RAL model is first constructed.

The functional coverage models are enabled through the `cover_on` argument of the "vmm_ral_block::new()" or "vmm_ral_sys::new()" methods as shown in Example 14-1. Once enabled, measurement for all functional coverage models is enabled by default.

Example 14-1   Enabling Functional Coverage Models

```
ral_my_block ral_model = new(vmm_ral::REG_BITS +
vmm_ral::ADDR_MAP);

ral_my_block ral_model = new(vmm_ral::ALL_COVERAGE);
```

If it is necessary to dynamically turn off functional coverage measurement, every block and system RAL abstraction class has a `set_cover()` method that dynamically turns functional coverage measurement on or off, as specified by the argument of the method. If the `set_cover()` method is called on a higher-level abstraction class using the “vmm_ral_block_or_sys::set_cover()” method, it is automatically invoked for all the lower-level abstraction classes it contains.

The following methods are useful for controlling the functional coverage model available in a RAL abstraction model:

- `vmm_ral_block_or_sys::has_cover() .................. page 139`
- `vmm_ral_block_or_sys::set_cover() .................. page 140`
- `vmm_ral_block_or_sys::is_cover_on() ................ page 142`
- `vmm_ral_block::new() ............................... page 153`
- `vmm_ral_block::has_cover() ......................... page 139`
- `vmm_ral_block::set_cover() ......................... page 140`
- `vmm_ral_block::is_cover_on() ....................... page 142`
- `vmm_ral_sys::new() ................................. page 362`
- `vmm_ral_sys::has_cover() ........................... page 139`
- `vmm_ral_sys::set_cover() ........................... page 140`
- `vmm_ral_sys::is_cover_on() ......................... page 142`

Predefined Functional Coverage Models

The following functional coverage models are available to be generated in the RAL model. Different models target a different perspective of the register verification process and should be used when appropriate.

Because functional models can be large in size and significantly impact runtime performance, they should be used carefully, at the right level of design granularity and only when their coverage points are targeted. Once filled to satisfaction, functional coverage models should no longer be generated—although their metrics should be preserved and continued to be reported.

Register Bits

This model is generated using the `-c b` command-line option for every register specified with a "+b" cover attribute. The coverage model is constructed by specifying the `vmm_ral::REG_BITS` symbol.

This model is designed to confirm that every specified bit in a RAL model has been thoroughly exercised and is implemented as specified. This functional model can be quite large and is, therefore, best used at the block level.

This functional coverage model is implemented by instances of `ral_cvr_reg_regname::reg_bits` coverage groups. In a block, there is one coverage group instance per register, for each domain instantiating the register. There is a coverage point for every field defined in the register and a bin to measure whether each individual bit of a field has been read and written through the domain physical interface as a 0 and a 1, respectively. This model does not measure backdoor accesses. The coverage model does not include unused or reserved bits.

Address Map

This model is generated using the `-c a` command-line option for every register and memory specified with a "+a" cover attribute. The coverage model is constructed by specifing the `vmm_ral::ADDR_MAP` symbol.

This model is designed to confirm that the address map of a design has been thoroughly exercised. It is best used at the top-level.

Address map coverage is implemented at the block level and supports address coverage of registers (including any registers in register files) and memories. Because fields cannot be physically accessed, they are not considered in the address map coverage. Virtual registers, being a logical structure imposed on a memory, are not included in the address map coverage either: it is assumed that if the address map coverage model of the memory containing the virtual registers is covered, the address map coverage model for the virtual registers can be considered covered as well.

The address map functional coverage model is composed of the ral_cvr_block_<block_name>::[<domain_name>_]addr_map coverage groups. For each block, there is one coverage group instance per domain in each block instance. In each coverage group (i.e. domain), there is a coverage point for each register (including each registers in register arrays and register files) and a coverage point for each memory in the block.

A register coverage point contains only one bin named "accessed". The bin is covered whenever the register is accessed using a read or a write operation.

A memory coverage point contains three bins. The first bin, named "first_location_accessed", is covered when the first location in the memory is accessed using a read or a write operation. The second bin, named "last_location_accessed", is covered when the last location in the memory is accessed using a read or a write operation. The third bin, named "other_locations_accessed", is covered when anyone of the remaining locations in the memory is accessed using a read or write operation.

Address map coverage measurement happens automatically during any front door read or write operation. Back-door accesses do not contribute toward the address map functional coverage.

Field Values

This model is generated using the `-c f` command-line option for every register specified with a "+f" cover attribute. The coverage model is constructed by specifing the `vmm_ral::FIELD_VALS` symbol.

This model is designed to confirm that every configuration of a design has been verified. It is best used at the top-level.

Field value coverage model is implemented at the register level and supports value coverage of all fields and cross coverage between fields and other cross coverage points within the same register. Field value coverage is not supported for virtual fields/registers.

The field valye functional coverage model is composed of the ral_reg_<reg_name>::field_values coverage groups. There is one coverage group instance per register instance. In each coverage group, there is a coverage point for each field in the register, except for "unused" and "reserved" fields.

By default, if the size of a field is 4 bits or less, the corresponding coverage point contains a bin for each possible value of that field. If the size of the field is greater than 4 bits, the corresponding coverage point contains three bins: the first bin, named "min", corresponds to the minimum value of that field (or ’0); the second bin, named "max", corresponds to the maximum value of that field (or ’1); and the third bin, named "others" corresponds to all other values of that field. The weight of a coverpoint is equal to the number of bins in that point.

User-Defined Field Value Coverage Bins

If the default field value bins are not suitable, there are many ways coverage bins can be defined for a coverage corresponding to a field value. In all cases, the weight of the coverage point will be equal to the number of bins.

If symbolic values are defined for a field using the "enum" property, a bin is implicitly defined for each symbolic value. The field specification shown in Example 14-2 will create three bins, named "AA", "BB" and "CC", each corresponding to field values 0, 1 and 15 respectively.

Example 14-2   Defining implicit coverage bins via symbolic field values

```
field f2 {
     bits 8;
     enum { AA, BB, CC=15 }
   }
```

User-defined bins can be explicitly specified using the "coverpoint" attribute. Example 14-3 illustrates how multiple coverage bins and bin arrays can be defined using numerical as well as symbolic field values, sets of values and ranges of values. The semantics of the bin specification is identical to the equivalent bin specification in SystemVerilog, as specified in section 18.4 of the 1800-2005 SystemVerilog Language Reference Manual.

Example 14-3   Defining explicit coverage bins

```
field f2 {
     bits 8;
     enum { AA, BB, CC=15 }
     coverpoint {
         bins AAA     = { 0, 12 }
         bins BBB []  = { 1, 2, AA, CC }
         bins CCC [3] = { 14,15, [ BB : 10 ] }
         bins DDD     = default
     }
```

```
    }
```

User Defined Cross Coverage Specification

A cross coverage point between different field values within the same register can be specified using the "cross" attribute. If a user-defined cross-coverage point is labelled, it is possible to use that cross-coverage point in another cross-coverage point.

Example 14-4   User-defined cross-coverage point

```
register r {
    field f1 {...}
    field f2 {...}
    field f3 {...}

    cross f1 f2 {
        label xyz;
    }
    cross xyz f3;
}
```

RALF cover attribute

By default, all applicable elements in a RAL models are included in the address map and register bits coverage models and all are excluded from the field value coverage model. The "cover" attribute can be used to specify the portions of the RAL model that should be included in or excluded from a coverage model.

All elements in a RAL model can be specified with a "cover" attribute to specify whether it and all of the sub-elements it contains are to be included in or excluded from a particular coverage mode. The address map, register bits and field value coverage models are identified by the letters "a", "b" and "f" respectively. A model element is included in or excluded from a coverage model by prefixing its identifying letter with a "+" or a "-’ respectively. For example, the attribute "cover +a+b-f" specifies that this element is included in the address map and register bits coverage model but not in the field values coverage model.

The coverage attribute for a RAL element are automatically inherited from the higher-level element. If a coverage model is not specified in a "cover" attribute, the inclusion or exclusion for that model is inherited from the higher level. For example, the attribute "cover +f" specified that this element (and all of its lower-level elements) are to be included in the field value coverage model but it does not say anything about the inclusion or exclusion of this element with respect to the other coverage models.

It is important to note that, unless a system, block, register file or register contains a "cover +f" attribute, no field value coverage model will be generated.

Example 14-5   Inherited cover attributes

```
system top {
    block b {
        cover -a+f            #-a+b+f
        …
        register r1 {
            cover -f          #+a+b-f
        }
        register r2 {
            cover -b          #+a-b+f
        }
    }
    system sub {
        cover +f              #+a+b+f
        …
    }
}
```

If a "cover" attribute is specified outside the "domain" attribute of a multi-domain block or system, it applies to all domains specified in that block or system. A "cover" attribute specified inside a "domain" attribute applies to all registers and memories instantiated in that domain.

User-defined Functional Coverage Model

Additional functional coverage group instances can be added to a RAL abstraction model to implement a user-defined functional coverage model. Sampling of user-defined functional coverage points is best implemented in callback extensions.

The sampling of user-defined functional coverage points should be dynamically controlled by checking the return value of the `is_cover_on()` method of the appropriate RAL abstraction class.

15

Randomizing Field Values

A RAL model can specify constraints on field values. If a field is specified with a `constraint` attribute, its value can be randomized. If a field is specified with no `constraint` attributes, it is a constant field that is never randomized. If you require an unconstrained field that can be randomized, specify the field with an empty `constraint` attribute. For example, fields `f1` and `f2` in Example 15-1 are randomized but field `f3` is not.

Within a field specification, the constraints specify the valid values for the field independently of any other field value. Within a register specification, the constraints specify constraints on field values based on the register where the field is instantiated or other field values within the register. Within a block or system specification, the constraints specify constraints on field values based on the block or system where the field is instantiated or other field values within the block or system.

Example 15-1 Field Constraints

```
field f1 {
   bits 8;
   constraint spec {
      value <= ’h80;
   }
}

register r {
   field f1;
   field f2 {
      bits 8;
      constraint consistency {
         f1.value == f2.value;
      }
   }
   field f3 {
      bits 2;
   }
}
```

Example 15-2 RAL Model for Example 15-1

```
class ral_r1 extends vmm_ral_reg;
   rand vmm_ral_field f1;
   rand vmm_ral_field f2;

   constraint f1_spec {
      f1.value < ’h80;
   }
   constraint consistency {
      f1.value == f2.value;
   }
   constraint user_defined;
}
```

Field constraints are inlined in the register class that instantiates the field to minimize the possibility of randomly selecting inconsistent field values. Constraints declared in a `field` property in the RAL description are not visible in the field abstraction class because they are inlined in the register class that instantiates the field and not in the field itself. If a field descriptor is directly randomized, it is therefore unconstrained. Therefore, do not directly randomize field descriptors. To randomize the content of fields subject to their constraints, the register, block, or system descriptor must be randomized. Once randomized, the field values can be written or updated into the DUT.

Example 15-3   Improperly Randomizing Fields

```
ral_model.r1.f1.randomize();
```

Example 15-4   Properly Randomizing Fields

```
ral_model.r1.randomize();
```

The content of memories cannot be randomized.

Adding Constraints

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.

You can add additional constraints by using `randomize() with {}` when randomizing a register, block and system abstraction class.

Example 15-5   Randomizing a Register Descriptor with Additional Constraints

```
ok = ral_model.r1.randomize() with {
   f1.value == 0;
};
```

You may also add additional constraints by defining the undefined `user_defined` external constraint block available in register abstraction classes containing randomized fields, and all block and system abstraction classes. The undefined external constraint block is included in the generated RAL model only if the `-ext_ud` command-line option has been used when invoking `ralgen`. This undefined external constraint block can then be defined by simply adding the necessary out-of-class constraint definition to the simulated code. No tests need to be modified.

Example 15-6 Adding External Constraints

```
constraint ral_r1::user_defined {
   f1.value == 0;
}
```

Undefined external constraint block declarations are not included by default in the RAL model to avoid the many warning messages that are produced when such a constraint is loaded. It may be possible (but not necessarily advisable) to disable these warning messages. For example, the warning message shown in Example 15-7 can be disabled using the "`+warn=noBCNACMBP`" command-line option with VCS.

Example 15-7 Undefined Constraint Block Warning Message from VCS

```
Warning-[BCNACMBP] Body of constraint in non-abstract class
must be present
        Constraint 'user_defined' has no body
```

Example 15-8 Undefined Constraint Block Warning Message from Vera

```
Vera Warning: Method "user_defined()" is declared in class
"ral_r1", but not defined
```

You cannot add constraints by extending a register, block or system abstraction class. It is important that you do not replace references to abstraction class instances in the RAL model by instances of extended classes because internal references are maintained in addition to the occurrences explicitly defined in the RAL model.

Relaxing Constraints

You can relax constraints specified in a RAL definition to inject errors by turning the corresponding constraint block OFF. The name of the constraint blocks are the same as the name of the constraints specified in the RAL description and are found in the corresponding block, system or register descriptor, except for field-specific constraints. Field-specific constraints are inlined in the register descriptor where the field is instantiated and the name of the constraint block is prefixed with the name of the unique field name within that register.

Example 15-9

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

16

Maximum Data Size

By default, the maximum size of fields, registers and memories is limited to 64 bits. This limitation is enforced by all methods having `bit [63:0]` data arguments. Smaller fields, registers and memories are intrinsically supported by using the SystemVerilog and OpenVera automatic value extension and truncation.

RAL version 1.1.0 and greater supports variable maximum data size. The maximum data size may be reduced to save memory in large RAL models. It may also be increased to support larger fields, register or memory values. The following instructions use a 128-bit maximum data size example as an illustration. The number of bits does not have to be a power of two and can be smaller than 64.

`ralgen` will issue a warning about the necessity of defining a new maximum data size should a RALF file specify fields, registers or memories with more than 64 data bits.

The maximum data size is defined globally, for all RAL model components. If different RAL models, with different maximum data size requirements are included in the same simulation, you must use the largest maximum data size.

When specifying a different maximum data size, all data arguments in the methods documented in Appendix B are redefined using `bit [VMM_RAL_DATA_WIDTH-1:0]` instead of `bit [63:0]`.

Maximum Data Size in SystemVerilog

By default, the maximum data size in SystemVerilog is 64 bits. You can change this default to a larger or smaller value by defining the `VMM_RAL_DATA_WIDTH` symbol to a suitable value.

You can define the symbol value using a command-line option, as shown in the following example:

```
% vcs ... +define+VMM_RAL_DATA_WIDTH=128 ...
```

You can also specify the symbol value in SystemVerilog source files, before the first inclusion of the `vmm_ral.sv` file, as shown in the following example:

```
‘define VMM_RAL_DATA_WIDTH 128
‘include "vmm_ral.sv"
```

Maximum Data Size in OpenVera

By default, the maximum data size in OpenVera is 64 bits. You can change this default to a larger or smaller value by defining the `VMM_RAL_DATA_WIDTH` symbol to a suitable value.

Maximum Data Size using Vera

The VRO file shipped with Vera or in a separate VMM installation, is precompiled for a maximum data size of 64 bits. If a different maximum data size is required, a new VRO file must first be created.

The VRO file is created by compiling the `vmm_ral.vrp` file with the symbol value defined using a command-line option. If compiling from a VMM installation shipped with Vera, use the following command:

```
vera -cmp -vip -DVMM_RAL_DATA_WIDTH=128 \
     ${VERA_HOME}/vrp/vmm_ral.vrp .
```

If compiling from a separate VMM installation, use the following command:

```
vera -cmp -vip -DVMM_RAL_DATA_WIDTH=128 \
     ${VMM_HOME}/ov/vrp/vmm_ral.vrp .
```

You must specify the <u>same</u> maximum data size when compiling any source code that includes the `vmm_ral.vrh` file. You can accomplish this through a command-line symbol definition, or by explicitly defining the symbol in a source file before the first inclusion of the VRH file, as shown in one of the following Example:

```
vera -cmp -DVMM_RAL_DATA_WIDTH=128 ...
```

or:

```
#define VMM_RAL_DATA_WIDTH 128
#include "vmm_ral.vrh"
```

Important: When using a separate VMM installation, you must use a fully qualified path to include the VRH file, as shown in the following example:

```
#define VMM_RAL_DATA_WIDTH 128
#include ".../path/to/vmm/installation/ov/include/
vmm_ral.vrh"
```

Maximum Data Size Using VCS

You can define the symbol by using a command-line option, as shown in the following example:

```
vcs ... -ntb_define VMM_RAL_DATA_WIDTH=128 ...
```

You can also specify the symbol value in OpenVera source files, before the first inclusion of the `vmm_ral.vrp` file, as shown in the following example:

```
#define VMM_RAL_DATA_WIDTH 128
#include "vmm_ral.vrp"
```

A

RALF Syntax

A RALF description is a Tcl 8.5 file. Therefore, it is possible to use programming constructs such as loops and variables to rapidly and concisely construct large register sets and memory definitions. You can also use the Tcl `source` command to perform multiple and hierarchical register specification management. Also, you can use Tcl expressions to specify register offset values, base values and register names.

The semi-colon is used as a separator and is not necessary immediately after or before a closing curly brackets.

RALF Construct Summary

Grammar Notation

The following notations are used to specify the exact syntax of RALF descriptions:

`normal`	Literal items
italics	User-specified identifiers
[...]	Optional items
<...>	Repeated items, 1 to *N* times
[<...>]	Optional repeated items, 0 to *N* times
...|...	A choice of items

Reserved Words

In addition to the SystemVerilog and OpenVera reserved words, the following words are reserved and cannot be used as user-defined identifiers:

access	field	regfile
bits	hard_reset	register
block	hdl_path	reset
bytes	initial	shared
constraint	left_to_right	size
doc	memory	soft_reset
domain	noise	system
endian	read	virtual write
		write

Useful Tcl Commands

Considering a RALF description is a Tcl file, the full power of the Tcl language becomes available. The following Tcl commands are likely to be useful:

`#`*`comment`*

Indicates single-line comments with characters following a # considered as comments.

`set` *`name value`*

Sets the specified variable to the specified value. Allows the use of variable names as mnemonics, using Tcl syntax to set and get variable values.

`source` *`filename`*

Includes the specified Tcl file. Inclusion of files enable hierarchical RALF descriptions. The filename can have an absolute path or relative path.

```
for {set i 0} {$i < 10} {incr i} {
        ...
    }
```

For loops can be used to concisely create multiple fields, registers, memories and blocks specifications. Any RALF property value can be based on the value of the loop index variable or other variables.

```
if {$var} {
        ...
    }
```

Conditionally interprets Tcl statements or RALF specifications. Allows the selection or exclusion of elements in a RALF description.

You can view a complete list of available Tcl commands by visiting the following web address:

http://www.tcl.tk/man/tcl8.5/TclCmd/contents.htm

Tcl Syntax and FAQ

The Tcl syntax rules can be found by visiting the following web address:

http://www.tcl.tk/man/tcl8.5/TclCmd/Tcl.htm

Note that ralgen preprocesses the RALF file to escape some of its syntax elements that have special meaning in Tcl. For example, the `[` and `]` used to specify arrays are properly escaped to avoid command substitution.

Whitespace

It is important to note how Tcl breaks a command into separate words on whitespaces, quoted (") and bracketed ({ and }) text. Therefore, a RALF file is sensitive to whitespace. Do not use whitespace in your code if none is shown in this appendix. Where a whitespace is shown, at least one must be present. For example, the following syntax is invalid because the `{` is considered as part of the *`field`* command’s second argument and not a separate token:

```
## This is wrong
   field REVISION_ID @2{
      bits 8;
   }
```

This example is valid because a space is required to separate the { from the preceding Tcl command argument:

```
## This is right
   field REVISION_ID @2 {
      bits 8;
   }
```

Trailing Comments

A common mistake occurs when trying to add a trailing comment to a RALF construct using the following (erroneous) syntax:

```
register my_reg {
      ...
   } # my_reg
```

Considering that Tcl commands terminate at the end-of-line, the trailing comment is considered part of the register command. To have the trailing comment be properly interpreted as a comment, the previous Tcl command should be explicitly terminated with a semicolon, as shown in the following (correct) syntax:

```
register my_reg {
      ...
   }; # my_reg
```

field

A field defines an atomic set of consecutive bits. Fields are concatenated into registers.

Syntax

```
field name [{
   <properties>
}]
```

Defines a field with the specified name. If you specify the name `unused` or `reserved`, it specifies unused or reserved bits within a register and you can specify only the `bits` property. Unused bits are assumed to be read-only and have a permanent value of zero. If another behavior is expected of unused or reserved bits, such as a different read-back value, you must specify an explicit field for them.

Properties

The following properties can be used to specify the field;

```
[bits n;]
```

Specifies the number of bits in the field. If not specified, defaults to 1. This property can only be specified once.

```
[access    rw|ro|wo|w1|ru|w1c|rc|a1|a0|
           other|user0|user1|user2|user3|dc;]
```

Specifies the functionality of all the bits in the field when the field is written or read.

A field can be:

writeable	`rw`
read-only	`ro`
write-only	`wo`
write-once	`w1`
read-only, but value updated by the design	`ru`
write a 1 to bitwise-clear	`w1c`
clear on read	`rc`
auto-set by the design	`a1`
auto-cleared by the design	`a0`
other	`other`

user-defined behavior	`user0`
	`user1`
	`user2`
	`user3`

don't care	`dc`

A write-only field always reads back as all zeroes when read, but reads back with the last written value when peeked.

If writing to a field may cause the content of other fields to be modified, or if the behavior of a field depends on the value of another field, then specify `other`, `userN` or `dc`. All other access modes assume that fields are independent of each other.

When using the "vmm_ral_field::mirror()" method with the `check` argument specified as `vmm_ral::VERB`, the mirrored value of the field is assumed to contain the expected value of the field for all field modes except don't-care (`dc`). If a field is specified as `dc`, its mirrored value behaves as if it were a `rw` field, but it is never compared against a value read from the design.

If a field has different access modes in different domain, specify the sum of all modes in the `field` property, then put access restrictions in the shared `register` property instantiation. For example, a field that is read-only in one domain and write-only in another must be specified as read-write, with the register instance in the first domain specified as write-only and in the second domain as read-only.

By default, a field is writeable (`rw`).

```
[reset|hard_reset value;]
```

Specifies the hard reset value for the field. By default, a value of 0 is used.

Supports unknown (x or X) and high-impedance (z or Z) bits in *value*. However, such bits are eventually converted to 0 in the RAL Base Class because the reset *value* in the RAL Base Class is a 2-state value.

```
[soft_reset value;]
```

Specifies the soft reset value for the field. By default, a field is not affected by a soft reset.

Supports unknown (x or X) and high-impedance (z or Z) bits in *value*. However, such bits are eventually converted to 0 in the RAL Base Class because the soft reset *value* in the RAL Base Class is a 2-state value.

```
[<constraint name [{
     <expressions>
}]>]
```

Specifies constraints on the field value when it is randomized. The constraints are not interpreted by the generation script and must be valid SystemVerilog or OpenVera expressions. The identifier `value` is used to refer to the value of the field.

If a `constraint` property is not specified, the field cannot be randomized. If an unconstrained but random field is required, simply specify an empty constraint block.

```
[enum { <name[=val],> }]
```

Defines symbolic names for field values. If a value is no explicitly specified for a symbolic name, the value is the value of the previous name plus one—or zero if it is the first name.

```
[cover <+|- b|f>
```

Specifies if the bits in this fields are to be included (+b) in or excluded (-b) from the register-bit coverage model.

Specifies if the field value coverage point for this field is an explicit goal (+f), in which case its weight will be equal to the number of specified or implicit bins. If it is specified as an implicit goal (-f) as part of a cross-coverage point, its coverage point weight will be equal to zero.

```
[<coverpoint {
   <bins name [[[n]]] = { <n|[n:n],> } | default>
}>]
```

Explicitly specifies the bins in the field value coverpoint for this field. The semantics of the bin specification is identical to the SystemVerilog coverage bin specification, as defined in section 18.4 of the 1800-2005 Language Reference Manual.

```
[doc {
     <text>
   }]
```

Specifies user documentation for the field, using HTML formatting tags. This feature is currently not supported.

Example

Example A-1   1-bit read/write Field

```
field tx_en;
```

Example A-2   2-bit Randomizable Field

```
field PAR {
   bits 2;
   reset 2'b11;
   constraint valid {
      value != 2'b00;
   }
}
```

Example A-3   Explicitly specified coverage bins

```
field f2 {
     bits 8;
     enum { AA, BB, CC=15 }
     coverpoint {
         bins AAA     = { 0, 12 }
         bins BBB []  = { 1, 2, AA, CC }
         bins CCC [3] = { 14,15, [ BB : 10 ] }
         bins DDD     = default
     }
   }
```

register

A register defines a concatenation of fields. Registers are used in register files and blocks.

Syntax

```
register name {
   <properties>
}
```

Defines a register with the specified name.

Properties

The following properties can be used to specify the register.

`[bytes n;]`

Specifies the number of bytes in the register. The total number of bits in the fields in this register cannot exceed this number of bytes. If this property is not specified, the width of the register is the minimum integral number of bytes necessary to implement all fields contained in the register.

`[left_to_right;]`

By default, fields are concatenated starting from the least-significant bit of the register. If this property is specified, fields are concatenated starting from the most-significant side of the register, but justified to the least-significant side. When using a left-to-right specification style, the first field cannot have a bit offset specified: the offset of the first field will depend on the size of and spacing between the other fields.

```
[<field name[=rename] [(hdl_path)] [@bit_offset];
[<field name [(hdl_path)] [@bit_offset] {
      <field properties>
   }>]
```

Defines and instantiates the specified field in this register. The first form specifies an instance of a previously-defined field description. The second form defines a new field description and instantiates it in the register file.

Fields separated by unused or reserved bits can be separated by specifying a field named `unused` or `reserved` of the appropriate width or by using a bit offset. A bit offset, from the least-significant bit in the register can be specified. If no bit offset is specified, the field is located immediately to the left (or right if the `left_to_right` property is specified) of the previously instantiated field.

The optional `(hdl_path)` is the hierarchical reference, within the register, to the HDL structure implementing the field. If an `(hdl_path)` is specified, direct hierarchical access to the field can be automatically generated by concatenating it with the `(hdl_path)` of the enclosing register. The `(hdl_path)` can be an expression and it must be enclosed between parentheses.

By default, the bit offset represents the position of the least-significant bit of the field with respect to the least-significant bit of the register. A value of 0 indicates a field starting in the least-significant bit of the register. If the `left_to_right` property is specified, the bit offset is specified as the offset of the most-significant bit of the field from the most-significant used bit in the register. The position of the most-significant used bit in the register, is a function of the size of, and spacing between all specified fields as fields are always left-justified, even when specifying a left-to-right order.

You must specify at least one `field` property.

Any gap in the register before and after fields is assumed to be made of unused bits that are read-only and have a permanent value of zero. If another behavior is expected from unused or reserved bits, an explicit field must be specified for them.

```
[<constraint name [{
      <expression>
   }]>]
```

Specifies constraints on the value of the fields it contains when it is randomized. The constraints are not interpreted by the generation script and must be valid SystemVerilog or OpenVera expressions. The identifier `fieldname.value` refers to the value of a field.

`[noise ro|rw|no;]`

Specifies if and how this register can be accessed during normal operations of the design without affecting the configuration or functional correctness of the device. By default, a register can be read at any time (`ro`). If `rw` is specified, this register can also be written. If no is specified, this register cannot be accessed in any way during normal operations. Currently unsupported.

`[shared [(`*`hdl_path`*`)];]`

Specifies that this register is physically shared by all domains in a block that instantiates it. This property can only be used in a stand-alone register specification.

The (`hdl_path`) specifies the hierarchical access path to the physical register. It is used instead of the (`hdl_path`) specified in the block instantiating it. If an (`hdl_path`) is specified, direct hierarchical access to the shared register can be automatically generated by concatenating it with the (`hdl_path`) of the enclosing block. The (`hdl_path`) must be enclosed in parentheses.

`[cover <+|- a|b|f>`

Specifies if the address of this register should be excluded (-a) from the block's address map coverage model.

Specifies if the bits in this register are to be included (+b) in or excluded (-b) from the register-bit coverage model.

Specifies if the fields in this registers should be included (+f) in or excluded (-f) from the field value coverage model.

```
[<cross <name,> [{
   label name
}]>]
```

Specify a cross-coverage point for two or more fields or cross-coverage point. To be able to use a cross-coverage point in another cross-coverage point, it must be labelled.

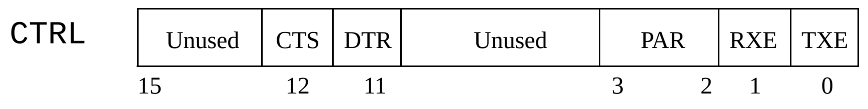
```
[doc {
      <text>
   }]
```

Specifies user documentation for the register, using HTML formatting tags. Currently unsupported.

Example

The following Example are different ways to specify the register illustrated in Figure A-1.

Figure A-1   Register Specification



Example A-4   Specification for Register in Figure A-1

```
register CTRL {
   field TXE {}
   field RXE {}
   field PAR {
      bits 2;
      reset 2’b11;
   }
   field DTR @11 {
      access ru;
   }
   field CTS {
      access ru;
      reset 1;
   }
}
```

Example A-5 Specification for register in Figure A-1

```
   source Example A-2
   register CTRL {
      bytes 2;
      left_to_right;
      field CTS {
         access ru;
         reset 1;
      }
      field DTR {
         access ru;
      }
      field unused {
         bits 7;
      }
      field PAR;
      field RXE {}
      field TXE {}
   }
```

Example A-6 User-defined cross-coverage point

```
   register r {
       field f1 {...}
       field f2 {...}
       field f3 {...}

       cross f1 f2 {
           label xyz;
       }
       cross xyz f3;
   }
```

regfile

A register file defines a collection of consecutive registers. Register files are used in blocks.

Syntax

```
regfile name {
   <properties>
}
```

Defines a register file with the specified name.

Properties

The following properties can be used to specify the register file.

```
[<register name[=rename][[n]] [(hdl_path)]
   [@offset] [read|write];>]
```

```
[<register name[[n]] [(hdl_path)] [@offset] {
      <property>
   }]
```

The first form specifies an instance of a previously-defined register description. The second form defines a new register description and instantiates it in the register file. An inlined register description cannot contain the `shared` property. Access to a shared register can be further restricted to read or write in a particular instance.

A register may be instantiated at an explicit address offset within the register file. If not specified, the register is instantiated at the next available address, starting with 0. The number of addresses occupied by a register depends on the width of the register and the endian property of the block defining the register file. If a register is not mapped in the address space of the block (see "Non-linear, Non-mapped Access"), the offset may be specified as `@none` to indicate that the register does not consume any address locations.

If a numerical index is specified, an array of registers is instantiated. Register arrays are located at consecutive address offsets, starting with register[0]. Instantiating an array of register is logically equivalent to explicitly instantiating all of the individual registers explicitly. The only difference is that they will be accessible as an array in the generated SystemVerilog or OpenVera code.

The optional `(hdl_path)` is the hierarchical reference, within the block, to the HDL structure implementing the register. If an `(hdl_path)` is specified, direct hierarchical access to the register can be automatically generated by concatenating it with the `(hdl_path)` of the enclosing system and any HDL expression specified in the register. The `(hdl_path)` can be an expression and it must be enclosed between parentheses. The `(hdl_path)` for a register array must include a %d placeholder that will be replaced with the decimal index of the register in the array.

If more than one register with the same name is instantiated in the same register file, it must be renamed to a unique name within the register file.

You must specify at least one register property.

```
[<constraint name [{
      <expression>
   }]>]
```

Specifies constraints on the value of the registers and fields it contains when it is randomized. The constraints are not interpreted by the generation script and must be valid SystemVerilog or OpenVera expressions.

```
[cover <+|- a|b|f>
```

Specifies if the registers in this register file are to be included (+) in or excluded (-) from the address map (a), register bits (b) or field value (f) coverage model.

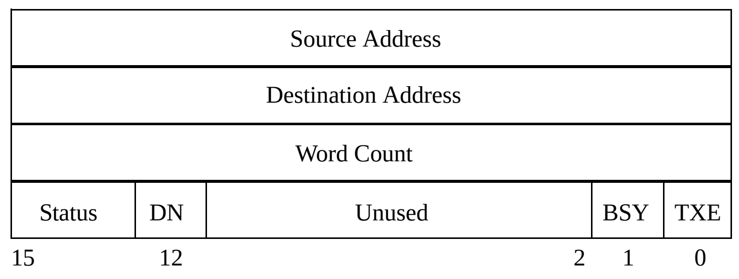
```
[doc {
      <text>
   }]
```

Specifies user documentation for the register file, using HTML formatting tags. Currently unsupported.

Example

The primary purpose of register files is to define arrays of groups of registers. For example, the register group illustrated in Figure A-2 is used to configure a DMA channel. The block RALF specification shown in Example A-7 illustrates how a 16-channel DMA controller might be described.

Figure A-2 DMC Channel Configuration Registers Specification


Source Address
Destination Address
Word Count
Status
DN
Unused
BSY
TXE
15
12
2
1
0

Example A-7   Specification for multi-channel DMA controller

```
block dma_ctrl {
   regfile chan[16] {
      register src {
         bytes 2;
         field addr {
            bits 16;
         }
      }
      register dst {
         bytes 2;
         field addr {
            bits 16;
         }
      }
      register count {
         bytes 2;
         field n_bytes {
            bits 16;
         }
      }
      register ctrl {
         bytes 2;
         field TXE {
            bits 1;
            access a0;
         }
         field BSY {
            bits 1;
            access ro;
         }
         field DN @12 {
            bits 1;
            access ro;
         }
         field status {
            bits 3;
            access ro;
         }
      }
   }
}
```

memory

A memory defines a region of consecutively addressable locations. Memories are used in blocks.

Syntax

```
memory name {
   <property>
}
```

Defines a memory with the specified name.

Properties

The following properties can be used to specify the memory.

`size` *`m`*`[k|M|G];`

Specifies the number of consecutive addresses in the memory where each location has the number of bits specified by the ***`bits`*** property. The size may also include a unit. In that case, the specified size is multiplied by:

- 1024 (k)
- 2^20 (M)
- 2^30 (G)

This property is required.

`bits` *`n`*`;`

Specifies the number of bits in each memory location. The total number of bits in the memory is the specified number of bits multiplied by the specified size. This property is required.

`[access rw|ro;]`

Specifies if the memory is a RAM (`rw`) or a ROM (`ro`). By default, a memory is a RAM.

`[initial x|0|1|addr|`*`literal`*`[++|--];]`

Specifies the initial content of the memory is to be filled with unknowns (`x`), filled with zeroes (`0`), filled with ones (`1`), set to the physical address value (`addr`), or set to a constant (*`literal`*), incrementing (*`literal`*`++`) or decrementing (*`literal`*`--`) literal value.

The content of the memory is initialized to the specified pattern when the `vmm_ral_mem::initialize()` method in its abstraction class is invoked. By default, a memory is initialized with unknowns (`x`).

Initialization requires that backdoor access to the memory content be available.

`[noise ro|rw|unused|no;]`

Specifies if and how this memory can be accessed during normal operations of the design without effecting the configuration of the device. By default, a memory can be read at any time (`ro`). If `rw` is specified, this memory can also be written. If `unused` is specified, only unused memory locations can be read and written. If no is specified, this memory cannot be accessed in any way during normal operations. Currently unsupported.

`[shared [(`*`hdl_path`*`)];]`

Specifies that this memory is physically shared by all domains in a block that instantiates it. Can only be used in a standalone memory specification.

The optional `(hdl_path)` specifies the hierarchical access path to the physical memory. It is used in lieu of the `(hdl_path)` specified in the block instantiating it. If an `(hdl_path)` is specified, direct hierarchical access to the shared memory can be automatically generated by concatenating it with the `(hdl_path)` of the enclosing block. The `(hdl_path)` must be enclosed between parentheses.

`[cover <+|- a>`

Specifies if this memory is to be included (+a) in or excluded (-a) from the address map coverage model.

```
[doc {
      <text>
   }]
```

Specifies user documentation for the memory, using HTML formatting tags.

Example

Example A-8 64 KB RAM

```
memory dma_bfr {
   bits 8;
   size  64k;
}
```

Example A-9   2 KB ROM

```
memory tx_bfr {
   bits    16;
   size    1024;
   access  ro;
   initial 0++;
}
```

virtual register

A `virtual register` defines a concatenation of virtual fields. Virtual registers are used in blocks.

Syntax

```
virtual register name {
   <properties>
}
```

Defines a virtual register with the specified name.

Properties

The following properties can be used to specify the virtual register.

`[bytes n;]`

Specifies the number of bytes in the register. The total number of bits in the fields in this register cannot exceed this number of bytes. The actual number of memory locations used by the virtual register is the minimum integral number of memory locations required to provide the specified number of bytes. If this property is not specified, the width of the register is the minimum integral number of memory locations necessary to implement all fields contained in the register.

```
[left_to_right;]
```

By default, fields are concatenated starting from the least-significant bit of the register. If this property is specified, fields are concatenated starting from the most-significant side of the register but justified to the least-significant side. When using a left-to-right specification style, the first field cannot have a bit offset specified: the offset of the first field will depend on the size of, and spacing between, the other fields.

```
[<field name[=rename] [@bit_offset];

[<field name [@bit_offset] {
      bits n;
      [doc {
         <text>
      }]
   }>]
```

Defines and instantiates the specified virtual field with the specified number of bits in this virtual register. The first form specifies an instance of a previously-defined field description where only the `bits` property is considered (all other properties are ignored). The second form defines a new field description and instantiates it in the register file.

Please refer to the specification of the `field` property in the "register" construct for more details on how they are physically laid out.

At least one `field` property must be specified.

All bits in a virtual register, including unused and reserved bits have their access modes defined by the access mode of the underlying memory used to implement it and the domain used to access them.

```
[doc {
     <text>
  }]
```

Specifies user documentation for the register/field, using HTML formatting tags. Currently unsupported.

block

A block defines a set of registers and memories. Registers are concatenated into blocks. A block can have more than one physical interface. Registers and memories can be shared across physical interfaces within a block.

Syntax

```
block name {
   <property>
}
```

Specifies a design block with the specified name and a single physical interface.

```
block name {
   domain name {
      <property>
   }
   <domain name {
      <property>
   }>
   [doc { <text> }]
}
```

Specifies a design block with the specified name and multiple physical interfaces. The name of each domain specifies the name of the corresponding physical interface. At least two domains must be specified. This form of the block specification can have a `doc` property outside of the `domain` specification.

The name of the block is used to generate block-specific unique identifiers.

Properties

The following properties can be used to specify the block and its domains.

```
bytes n;
```

Specifies the number of bytes that can be accessed concurrently and uniquely addressed through the physical interface. This property is required.

```
[endian little|big|fifo_ls|fifo_ms;]
```

Specifies how wider registers and memories are mapped onto multiple accesses over the physical interface. See "Hierarchical Descriptions and Composition" for a description of the various mapping modes. By default, little endian is used.

```
[<register name[=rename][[n]] [(hdl_path)]
   [@offset] [+incr] [read|write];>]
```

```
[<register name[[n]] [(hdl_path)] [@offset] [+incr]
   {
      <property>
   }]
```

The first form specifies an instance of a previously-defined register description. The second form defines a new register description and instantiates it in the block. An inlined register description cannot contain the `shared` property. Access to a shared register can be further restricted to read or write in a particular instance.

A register may be instantiated at an explicit address offset within the block. If not specified, the register is instantiated at the next available address, starting with 0. The number of addresses occupied by a register depends on the width of the register and the endian property of the block. If a register is not mapped in the address space of the block (see "Non-linear, Non-mapped Access"), the offset may be specified as `@none` to indicate that the register does not consume any address locations.

If a numerical index is specified, an array of register is instantiated. Register arrays are located at consecutive address offsets, starting with register [0]. If an increment value is specified, the offset of each register in the register array is incremented by the specified increment. Instantiating an array of register is logically equivalent to explicitly instantiating all of the individual registers explicitly. The only difference is that they will be accessible as an array in the generated SystemVerilog or OpenVera code.

The optional `(hdl_path)` is the hierarchical reference, within the block, to the HDL structure implementing the register. If an `(hdl_path)` is specified, direct hierarchical access to the register is automatically generated by concatenating it with the `(hdl_path)` of the enclosing system and any `(hdl_path)` expression specified in the register. The `(hdl_path)` can be an expression and it must be enclosed between parentheses. The `(hdl_path)` for a register array must include a "%d" placeholder that will be replaced with the decimal index of the register in the array.

If more than one register with the same name is instantiated in the same block, it must be renamed to a unique name within the block.

You must specify at least one register or memory property.

Registers must have unique addresses, therefore, it is not possible to describe a block containing a read-only register and a write-only register sharing the same physical address. If it is not possible to avoid this implementation structure, specify a single register with a field of `other` bits.

```
[<regfile name[=rename][[n]] [(hdl_path)] [@offset]
   [+incr];>]
```

```
[<regfile name[[n]] [(hdl_path)] [@offset] [+incr]
   {
      <property>
   }]
```

The first form specifies an instance of a previously-defined register file description. The second form defines a new register file description and instantiates it in the block.

If a numerical index is specified, an array of register files is instantiated. Register file arrays are located at consecutive address offsets, starting with register [0], separated by the specified offset increment. The offset increment is required and only valid when instantiating a `regfile` array. Instantiating an array of register files is logically equivalent to explicitly instantiating all of the individual register files explicitly. The only difference is that they will be accessible as an array in the generated SystemVerilog or OpenVera code.

Register files are usually used to specify arrays of register groups. Arrays of register files yield a different address map than register arrays. See "Arrays and Register Files" for more details.

The optional (`hdl_path`) is the hierarchical reference, within the block, to the HDL structure implementing the register file. Direct hierarchical access to the register file can be automatically generated by concatenating the specified (`hdl_path`) with the (`hdl_path`) of the enclosing system and the (`hdl_path`) specified in the registers. The (`hdl_path`) must be enclosed between parentheses. The (`hdl_path`) for a register file array must include a "%d" placeholder that will be replaced with the decimal index of the register file in the array.

```
[<memory name[=rename] [(hdl_path)] [@offset]
      [read|write];>]
```

```
[<memory name [(hdl_path)] [@offset] {
     <property>
  }>]
```

The first form specifies an instance of a previously-defined memory description. The second form defines a new memory description and instantiates it in the block. An inlined memory description cannot contain the `shared` property. The access to a shared memory can be further restricted to read or write in a particular instance.

A memory may be instantiated at an explicit address offset within the block. If not specified, the memory is instantiated at the next available address, starting with 0. The number of addresses occupied by a memory depends on the size and width of the memory, and the endian property of the block. If a memory is not mapped in the address space of the block (see “Non-linear, Non-mapped Access”), the offset may be specified as `@none` to indicate that the memory does not consume any address locations.

The optional (`hdl_path`) is the hierarchical reference, within the block, to the HDL structure implementing the memory. If an (`hdl_path`) is specified, direct hierarchical access to the memory can be automatically generated by concatenating it with the (`hdl_path`) of the enclosing system. The (`hdl_path`) must be enclosed between parentheses.

If more than one memory with the same name is instantiated in the same block, it must be renamed to a unique name within the block.

At least one register or memory property must be specified.

```
[<virtual register name[=rename][[n] mem@offset
   [+incr]];>]
```

```
[<virtual register name[[n] mem@offset [+incr]] {
      <property>
   }]
```

The first form instantiates an array of a previously-defined virtual register description in the block. The second form instantiates an array of a new virtual register description.

If a memory association is specified, the array of virtual register is statically implemented in the specified memory starting at the specified offset. If an increment value is specified, the implementation offset of each virtual register in the virtual register array is incremented by the specified increment. If a memory association is not specified, the virtual register is still instantiated in the block but must be dynamically associated with an implementation memory using the “vmm_ral_vreg::implement()” or “vmm_ral_vreg::allocate()” method before it can be used.

If more than one array of virtual registers with the same name is associated in the same block, it must be renamed to a unique name within the memory.

Because virtual registers are implemented in memory, it is possible to describe overlapping virtual register arrays.

```
[<constraint name [{
      <expression>
   }]>]
```

Specifies constraints used when the content of the registers in the block is randomized. The constraints are not interpreted by the generation script and must be valid SystemVerilog or OpenVera expressions. Constraints at this level should specify cross-register constraints.

Constraints cannot be used to constrain the content of memories or virtual registers.

```
[cover <+|- a|b|f>
```

Specifies if the registers and memories in this block are to be included (+) in or excluded (-) from the address map (a), register bits (b) or field value (f) coverage model. If specified inside a "domain", applies to that domain only.

```
[doc {
      <text>
   }]
```

Specifies user documentation for the block or domain, using HTML formatting tags.

Example

Example A-10 Block With Single Physical Interface

```
source Example A-5
source Example A-9
block uart {
   bytes 1;
   endian little;
   register CTRL;
   memory tx_bfr @'h00100;
}
```

Example A-11 Block With Register Array

```
block multi_chan {
   bytes 1;
   endian little;
   register CHAN_CTRL[32] @'h0200 {
      bytes 2;
      ...
   };
}
```

Example A-12 Block With Two Physical Interfaces

```
register data_xfer {
   bytes 4;
   field data {
      bits 32;
   }
   shared;
}
register flags {
   field cts {
      access ru;
      reset 1;
   }
   field dtr {
      access ru;
   }
}
block bridge {
   domain pci {
     bytes 4;
     register pci_flags;
```

```
         register data_xfer=to_ahb write;
         register data_xfer=frm_ahb read;
      }
      domain ahb {
         bytes 4;
         register ahb_flags;
         register data_xfer=to_pci write;
         register data_xfer=frm_pci read;
      }
   }
```

system

A `system` defines a design composed of blocks or subsystems. A system can be used to create larger systems.

Syntax

```
system name {
   <property>
}
```

Specifies a system with the specified name and a single physical interface.

```
system name {
   domain name {
      <property>
   }
   <domain name {
      <property>
   }>
   [doc { <text> }]
}
```

Specifies a system with the specified name and multiple physical interfaces. The name of each domain specifies the name of the corresponding physical interface. At least two domains must be specified. This form of the `system` specification can have a `doc` property outside of the `domain` specification.

The name of the system is used to generate system-specific unique identifiers.

Properties

The following properties can be used to specify the system and its domains.

`bytes` *`n`*`;`

Specifies the number of bytes that can be accessed concurrently and uniquely addressed through the physical interface. This property is required.

```
[endian little|big|fifo_ls|fifo_ms;]
```

Specifies how wider blocks and subsystems are mapped onto multiple accesses over the physical interface. See "Hierarchical Descriptions and Composition" for a description of the various mapping modes. By default, little endian is used.

```
[<block name[[.domain]=rename][[n]] [(hdl_path)]
           @offset [+incr];>]
```

```
[<block name[[n]] [(hdl_path)] @offset [+incr] {
      <property>
   }]
```

The first form specifies an instance of a previously-defined block description. The second form defines a new block description and instantiates it in the system.

A block must be instantiated at an explicit address offset within the system. If the base address of the block is programmable, specify the default (after reset) base address.

If a numerical index is specified, an array of blocks is instantiated. Block arrays are located at consecutive address offsets, starting with block[0], separated by the specified offset increment. The offset increment is required and only valid when instantiating a block array. Instantiating an array of blocks is logically equivalent to explicitly instantiating all of the individual blocks explicitly. The only difference is that they will be accessible as an array in the generated SystemVerilog or OpenVera code.

The optional (`hdl_path`) is the hierarchical reference, within the system, to the HDL structure implementing the block. Direct hierarchical access to the registers and memories in the block can be automatically generated by concatenating the specified (`hdl_path`) with the (`hdl_path`) of any enclosing system and the (`hdl_path`) to the registers and memories within the block. The (`hdl_path`) must be enclosed between parentheses. The (`hdl_path`) for a block array must include a "%d" placeholder that will be replaced with the decimal index of the block in the array.

If more than one block with the same name is instantiated in the same block, it must be renamed to a unique name within the block. A reference to a domain within a block uses a composite name and must be renamed to a single name that is a valid SystemVerilog or OpenVera user-defined identifier.

At least one `block` or `system` property must be specified.

```
[<system name[[.domain]=rename][[n]] [(hdl_path)]
           @offset [+incr];>]
```

```
[<system name[[n]] [(hdl_path)] @offset [+incr] {
     <property>
   }]
```

The first form specifies an instance of a previously-defined subsystem description. The second form defines a new subsystem description and instantiates it in the system.

A subsystem must be instantiated at an explicit address offset within the system. If the base address of the subsystem is programmable, specify the default (after reset) base address.

If a numerical index is specified, an array of subsystems is instantiated. Subsystem arrays are located at consecutive address offsets, starting with `subsys[0]`, separated by the specified offset increment. The offset increment is required and only valid when instantiating a subsystem array. Instantiating an array of subsystems is logically equivalent to explicitly instantiating all of the individual subsystems explicitly. The only difference is that they will be accessible as an array in the generated SystemVerilog or OpenVera code.

The optional (`hdl_path`) is the hierarchical reference, within the system, to the HDL structure implementing the subsystem. Direct hierarchical access to the registers and memories in the subsystem can be automatically generated by concatenating the specified (`hdl_path`) with the (`hdl_path`) of any enclosing system and the (`hdl_path`) to the registers and memories within the subsystem. The (`hdl_path`) must be enclosed between parentheses. The (`hdl_path`) for a subsystem array must include a "%d" placeholder that will be replaced with the decimal index of the subsystem in the array.

If more than one subsystem with the same name is instantiated in the same system, it must be renamed to a unique name within the system. A reference to a domain within a subsystem uses a composite name and must be renamed to a single name that is a valid SystemVerilog or Openvera user-defined identifier.

At least one `block` or `system` property must be specified.

```
[<constraint name [{
      <expression>
   }]>]
```

Specifies constraints used when the content of the registers and memories in the system is randomized. The constraints are not interpreted by the generation script and must be valid SystemVerilog or OpenVera expressions. Constraints at this level should specify cross-register constraints.

```
[cover <+|- a|b|f>
```

Specifies if the registers and memories in this block are to be included (+) in or excluded (-) from the address map (a), register bits (b) or field value (f) coverage model. If specified inside a "domain", applies to that domain only.

```
[doc {
      <text>
   }]
```

Specifies user documentation for the system or domain, using HTML formatting tags. Currently unsupported.

Example

Example A-13 System With Single Physical Interface

```
source Example A-10
system SoC {
   bytes 1;
   endian little;
   block uart[2] @'hF0000 +'h01000;
}
```

Example A-14   System With Two Physical Interfaces

```
source Example A-10
source Example A-12
system SoC {
   domain ahb {
      bytes 4;
      block uart[2] @’hF0000 +’h01000;
      block bridge.ahb=br @0;
   }
   domain pci {
      bytes 4;
      block bridge.pci=br @0;
   }
}
```

B

RAL Classes

This appendix provides detailed documentation of the classes that compose a RAL model. All RAL classes generated from a user specification are extended from one of the RAL base classes. Extensions do not add any methods, therefore, the documentation of the base class is the same as the documentation for the generated class.

The OpenVera and SystemVerilog classes have identical functionality and features, therefore, they are documented together. The heading used to introduce a method uses the SystemVerilog name. The OpenVera name will be identical, except for a few instances where a `_t` suffix is appended to indicate that it may be a blocking method.

Usage examples are usually specified in a single language but that should not deter users of the other language as they would be almost identical. This document prefers to provide more different examples than almost identical examples in each language.

This appendix documents the classes in alphabetical order, and documents methods in each class in a logical order, where methods that accomplish similar results are documented sequentially. Additionally, this appendix provides a summary of all available methods with cross references to their detailed documentation at the beginning of each class specification.

RAL Class Summary

vmm_mam

This class is a memory allocation management utility class similar to C’s `malloc()` and `free()`. A single instance of this class is used to manage a single, contiguous address space. This memory allocation management class is used by any application-level process that requires reserved space in the memory. The section of memory (called a region) will remain reserved until it is explicitly released.

Summary

vmm_mam::new()

Create a new instance of a memory allocation manager.

SystemVerilog

```
function new(string name,
   vmm_mam_cfg cfg,
   vmm_ral_mem mem = null);
```

OpenVera

```
task new(string name,
   vmm_mam_cfg cfg,
   vmm_ral_mem mem = null);
```

Description

Create an instance of a memory allocation manager with the specified name. This instance manages all memory region allocation within the address range specified in the configuration descriptor.

If a reference to a RAL memory abstraction class is provided, the memory locations within the regions can be accessed through the region descriptor, using the Xref and Xref methods.

The specified name is used as the instance name of the message interface found in the “`vmm_mam::log`” class property.

Example

Example B-1

```
class ral_mam extends vmm_mam;
   ...
   function new(vmm_mam_cfg cfg);
      super.new("Memory allocation",cfg);
      `vmm_note(log,"vmm_mam::new() method is verified.");
   endfunction
   ...
endclass
```

vmm_mam::log

Message service interface for the memory allocation manager.

SystemVerilog

```
vmm_log log;
```

OpenVera

```
rvm_log log;
```

Description

Message service interface used to issue messages from this memory allocation manager. The name of the interface is hard-coded as "Memory Allocation Manager". The instance name of the interface is the name of the manager specified in the constructor. These names may be modified afterward using the `vmm_log::set_name()` or `vmm_log::set_instance()` methods.

Example

Example B-2

```
class ral_mam extends vmm_mam;
   ...
   vmm_log log;
   function new(vmm_mam_cfg cfg);
      super.new("Memory allocation",cfg);
      log = new("VMM MAM","log");
      log.set_name("VMM Memory Log");
      `vmm_note(log,$psprintf("vmm_mam::log=>Created 
user-defined String ::%s",
                log.get_name()));
```

```
    endfunction
    ...
endclass
```

vmm_mam::default_alloc

Default memory region allocator.

SystemVerilog

```
vmm_mam_allocator default_alloc;
```

OpenVera

```
vmm_mam_allocator default_alloc;
```

Description

Default object instance that is randomized in the `request_region()` method when allocating a region at a random address and no allocator object is specified.

Example

Example B-3

```
class tb_env extends vmm_ral_env;
   ...
   vmm_mam_cfg mam_cfg;
   vmm_mam_region mam_reg;
   ...
   function new();
      super.new();
      this.mam_cfg = new();
   endfunction

   virtual function void gen_cfg();
      super.gen_cfg();
      if (!this.mam_cfg.randomize()) begin
         `vmm_fatal(log, "Failed to randomize Memory 
configuration");
```

```
      end
      ...
   endfunction

   virtual function void build();
      super.build();
      ...
      mam_reg    = mam.request_region(mam_cfg.n_bytes);
      if(mam.default_alloc == null)
       `vmm_fatal(log, "Creation of Default alloc instance 
is Failed");
   endfunction
   ...
endclass
```

vmm_mam::reconfigure()

Reconfigure the managed address space.

SystemVerilog

```
function vmm_mam_cfg reconfigure(vmm_mam_cfg cfg = null);
```

OpenVera

```
function vmm_mam_cfg reconfigure(vmm_mam_cfg cfg = null);
```

Description

Optionally modify the maximum and minimum addresses of the address space managed by the allocation manager, allocation mode, or locality. The number of bytes per memory location cannot be modified once an allocation manager has been constructed.

Returns the previous configuration.

All currently allocated regions must fall within the new address space.

Example

Example B-4

```
class tb_env extends vmm_ral_env;
   ...
   if (!this.mam_cfg.randomize()) begin
      `vmm_fatal(log, "Failed to randomize Memory
configuration");
   end
   if (!this.recfg.randomize()) begin
      `vmm_fatal(log, "Failed to randomize Memory
configuration");
   end
   ...
   mam_cfg = mam.reconfigure(this.recfg);
   ...
endclass
```

vmm_mam::reserve_region()

Reserve a specific memory region.

SystemVerilog

```
function vmm_mam_region reserve_region(bit [63:0]
start_offset,
    int unsigned n_bytes);
```

OpenVera

```
function vmm_mam_region reserve_region(bit [63:0]
start_offset,
    integer    n_bytes);
```

Description

Reserve a memory buffer of the specified number of bytes starting at the specified offset in the memory. A descriptor of the reserved region is returned. If the specified region cannot be reserved, *`null`* is returned.

It may not be possible to reserve a region because it overlaps with an already-allocated region or it lies outside the address range managed by the memory manager.

Example

Example B-5

```
class tb_env extends vmm_ral_env;
   ...
   virtual function void build();
      super.build();
      ...
      mam_region =
mam.reserve_region(mam_cfg.start_offset,mam_cfg.n_bytes);
      if(mam_region == null)
        begin
       `vmm_fatal(log,$psprintf({"Failed to reserve Memory 
Location from ",
                  "%0h location."},mam_cfg.start_offset));
        end
      ...
   endfunction
   ...
endclass
```

vmm_mam::request_region()

Request a memory region.

SystemVerilog

```
function vmm_mam_region request_region(
   int unsigned n_bytes, vmm_mam_allocator alloc = null);
```

OpenVera

```
function vmm_mam_region request_region(integer n_bytes,
    vmm_mam_allocator alloct = null);
```

Description

Request and reserve a memory buffer of the specified number of bytes starting at a random location in the memory. If an allocator is specified, it is randomized to determine the start offset of the region. If no allocator is specified, the allocator found in the "`vmm_mam::default_alloc`" class property is randomized.

A descriptor of the allocated region is returned. If no region can be allocated, `null` is returned. It may not be possible to allocate a region because there is no area in the memory with enough consecutive locations to meet the size requirements or because there is another contradiction when randomizing the allocator.

If the memory allocation is configured to `vmm_mam::THRIFTY` or `vmm_mam::NEARBY` (see the "`vmm_mam_cfg::mode`" and "`vmm_mam_cfg::locality`" class properties, respectively), a suitable region is first sought procedurally. If no suitable region is found, then the allocator is randomized. The allocator is immediately randomized when the memory allocation is configured as `vmm_mam::BROAD` and `vmm_mam::GREEDY`.

Example

Example B-6

```
class tb_env extends vmm_ral_env;
   ...
   mam_reg = mam.request_region(mam_cfg.n_bytes);
   if(mam_reg == null)
     begin
       `vmm_fatal(log,$psprintf("Failed to request Memory
            Location for %d bytes.", mam_cfg.n_bytes));
     end
   ...
endclass
```

vmm_mam::release_region()

Release a previously allocated memory region.

SystemVerilog

```
function void release_region(vmm_mam_region region);
```

OpenVera

```
task release_region(vmm_mam_region region);
```

Description

Release the specified previously allocated memory region. An error is issued if the specified region has not been previously allocated or is no longer allocated.

Example

Example B-7

```
TBD
```

vmm_mam::release_all_regions()

Release all allocated memory regions.

SystemVerilog

```
function void release_all_regions();
```

OpenVera

```
task release_all_regions();
```

Description

Release all allocated memory regions.

Example

Example B-8

```
TBD
```

vmm_mam::psdisplay()

Human-readable description of allocated memory regions.

SystemVerilog

```
function string psdisplay(string prefix = "");
```

OpenVera

```
function string psdisplay(string prefix = "");
```

Description

Create a human-readable description of the state of the memory manager and the currently allocated regions. Each line of the description is prefixed with the specified prefix.

Example

Example B-9

```
...
   mam_region =
mam.reserve_region(mam_cfg.start_offset,mam_cfg.n_bytes);
   //Display Allocated Memory [Starting Location:Ending
Location]
   `vmm_note(log,$psprintf("%s",mam.psdisplay("Allocated
Memory")));
   ...
```

vmm_mam::for_each()

Iterate over allocated memory regions.

SystemVerilog

```
function vmm_mam_region for_each(bit reset = 0);
```

OpenVera

```
function vmm_mam_region for_each(bit reset = 0);
```

Description

Iterate over all currently allocated regions. If `reset` is non-zero, reset the iterator and return the first allocated region. Returns `null` when there are no additional allocated regions to iterate on.

Example

Example B-10

```
TBD
```

vmm_mam::get_memory()

Return the RAL abstraction class for the managed memory.

SystemVerilog

```
function vmm_ral_mem get_memory();
```

Description

Return the reference to the RAL memory abstraction class for the memory implementing the locations managed by this instance of the allocation manager. Returns `null` if no memory abstraction class was specified at construction time.

Example

Example B-11

```
TBD
```

vmm_mam_allocator

An instance of this class is randomized to determine the starting offset of a randomly allocated memory region. This class can be extended to provide additional constraints on the starting offset, such as word alignment or location of the region within a memory page.

Summary

vmm_mam_allocator::len

Number of addressable locations required.

SystemVerilog

```
int unsigned len;
```

OpenVera

```
integer len;
```

Description

Set by the memory manager before every randomization. Specifies the number of memory locations (not necessarily bytes) required in the region.

Example

Example B-12

```
class tb_env extends vmm_ral_env;
   ...
   virtual function void build();
      super.build();
      ...
      mam_reg = mam.request_region(mam_cfg.n_bytes);
      if(mam.default_alloc == null)
       `vmm_fatal(log, "Creation of Default alloc instance 
is Failed");

     `vmm_note(log,$psprintf(" vmm_mam_allocator::len ==> 
%0d",mam.default_alloc.len));
   endfunction
   ...
endclass
```

vmm_mam_allocator::min_offset

Minimum address offset under management.

SystemVerilog

```
bit [63:0] min_offset;
```

OpenVera

```
bit [63:0] min_offset;
```

Description

Set by the memory manager before every randomization. Specifies the minimum address offset in the memory under management. The `vmm_mam_allocator_valid` constraint block constrains the *“`vmm_mam_allocator::start_offset`”* class property to fall within the range defined by this property and the *“`vmm_mam_allocator::max_offset`”* property—minus the required number of memory locations.

Example

Example B-13

```
...
   mam_reg = mam.request_region(mam_cfg.n_bytes);
   if(mam.default_alloc == null)
     `vmm_fatal(log, "Creation of Default alloc instance is 
Failed");

   `vmm_note(log,$psprintf("vmm_mam_allocator::min_offset 
==> %0h",
                           mam.default_alloc.min_offset));
   ...
```

vmm_mam_allocator::max_offset

Maximum address offset under management.

SystemVerilog

```
bit [63:0] max_offset;
```

OpenVera

```
bit [63:0] max_offset;
```

Description

Set by the memory manager before every randomization. Specifies the maximum address offset in the memory under management. The `vmm_mam_allocator_valid` constraint block constrains the “`vmm_mam_allocator::start_offset`” class property to fall within the range defined by this property and the “`vmm_mam_allocator::min_offset`” property—minus the required number of memory locations.

Example

Example B-14

```
...
   mam_reg = mam.request_region(mam_cfg.n_bytes);
   if(mam.default_alloc == null)
     `vmm_fatal(log, "Creation of Default alloc instance is 
Failed");

   `vmm_note(log,$psprintf("vmm_mam_allocator::max_offset 
==> %0h",
                           mam.default_alloc.max_offset));
   ...
```

vmm_mam_allocator::in_use[$]

Currently allocated memory regions.

SystemVerilog

```
vmm_mam_region in_use[$];
```

OpenVera

```
vmm_mam_region in_use[$];
```

Description

Set by the memory manager before every randomization. Specifies the memory regions currently allocated. The *vmm_mam_allocator_no_overlap* constraint block constrains the “vmm_mam_allocator::start_offset” class property—minus the required number of memory locations—to fall outside of the allocated regions.

Example

Example B-15

```
int total_rsvd_region;
mam_region =
mam.reserve_region(mam_cfg.start_offset,mam_cfg.n_bytes);
   foreach(mam.default_alloc.in_use[i])
     total_rsvd_region ++;
   `vmm_note(log,$psprintf("Total Allocated(Reserved)
Regions are ==> %0d",
                            total_rsvd_region));
   ...
```

vmm_mam_allocator::start_offset

Starting offset of the randomly allocated region.

SystemVerilog

```
rand bit [63:0] start_offset;
```

OpenVera

```
rand bit [63:0] start_offset;
```

Description

After a successful randomization, this class property will contain a valid starting offset for a memory region of the desired length.

Example

Example B-16

```
...
   mam_reg = mam.request_region(mam_cfg.n_bytes);
   if(mam.default_alloc == null)
     `vmm_fatal(log, "Creation of Default alloc instance is
Failed");

  `vmm_note(log,$psprintf("vmm_mam_allocator::start_offset
==> %0h",
                         mam.default_alloc.start_offset));
   ...
```

vmm_mam_cfg

This class is used to specify the memory managed by an instance of a “`vmm_mam`” memory allocation manager class.

Summary

vmm_mam_cfg::n_bytes

Number of bytes in each addressable location.

SystemVerilog

```
rand int unsigned n_bytes;
```

OpenVera

```
rand integer n_bytes;
```

Description

Total number of bytes in each memory location. The `vmm_mam_cfg_valid` constraint blocks constrains this class property to the {1:64} range.

Although the memory allocation manager will operate properly with any positive value for this property, it does not make much physical sense to have values that are not powers of two (for example, 1, 2, 4, 8, etc...).

Example

Example B-17

```
class tb_env extends vmm_ral_env;
   ...
   if (!this.mam_cfg.randomize()) begin
      `vmm_fatal(log, "Failed to randomize Memory 
configuration");
   end
   ...
   `vmm_note(log,$psprintf("vmm_mam_cfg::n_bytes => 
%0d",mam_cfg.n_bytes));
```

```
   ...
endclass
```

vmm_mam_cfg::start_offset

Offset of the first addressable location under management.

SystemVerilog

```
rand bit [63:0] start_offset;
```

OpenVera

```
rand bit [63:0] start_offset;
```

Description

The starting offset of the consecutive memory area managed by the memory manager instance.

The `vmm_mam_cfg_valid` constraint block constrains this class property to be less than "`vmm_mam_cfg::end_offset`".

Example

Example B-18

```
class tb_env extends vmm_ral_env;
   ...
   if (!this.mam_cfg.randomize()) begin
      `vmm_fatal(log, "Failed to randomize Memory 
configuration");
   end
   ...
   `vmm_note(log,$psprintf("vmm_mam_cfg::start_offset => 
64'h%0h",
             mam_cfg.start_offset));
   ...
endclass
```

vmm_mam_cfg::end_offset

Offset of the last addressable location under management.

SystemVerilog

```
rand bit [63:0] end_offset;
```

OpenVera

```
rand bit [63:0] end_offset;
```

Description

The ending offset of the consecutive memory area managed by the memory manager instance.

The `vmm_mam_cfg_valid` constraint block constrains this class property to be greater than “`vmm_mam_cfg::start_offset`”.

Example

Example B-19

```
class tb_env extends vmm_ral_env;
   ...
   if (!this.mam_cfg.randomize()) begin
      `vmm_fatal(log, "Failed to randomize Memory
configuration");
   end
   ...
   `vmm_note(log,$psprintf("vmm_mam_cfg::end_offset =>
64'h%0h",
             mam_cfg.end_offset));
   ...
endclass
```

vmm_mam_cfg::mode

Memory allocation mode.

SystemVerilog

```
rand enum {vmm_mam::GREEDY, vmm_mam::THRIFTY} mode;
```

OpenVera

```
rand enum {vmm_mam::GREEDY, vmm_mam::THRIFTY} mode;
```

Description

Configures how new memory regions are allocated by the "`vmm_mam::request_region()`" method. If set to `vmm_mam::THRIFTY`, memory used in previously allocated, but now freed regions is preferred. If set to `vmm_mam::GREEDY`, previously unused memory is preferred.

Example

Example B-20

```
class tb_env extends vmm_ral_env;
   ...
   if (!this.mam_cfg.randomize()) begin
      `vmm_fatal(log, "Failed to randomize Memory 
configuration");
   end
   ...
   `vmm_note(log,$psprintf("vmm_mam_region::mode      ==> 
%0s",
             mam_cfg.mode.name));
   ...
endclass
```

vmm_mam_cfg::locality

Memory allocation locality.

SystemVerilog

```
rand enum {vmm_mam::BROAD, vmm_mam::NEARBY} locality;
```

OpenVera

```
rand enum {vmm_mam::BROAD, vmm_mam::NEARBY} locality;
```

Description

Configures where in memory, in relation to currently allocated regions, new memory is allocated by the “`vmm_mam::request_region()`” method. If set to `vmm_mam::NEARBY`, memory near or adjacent to currently allocated regions is preferred. If set to `vmm_mam::BROAD`, the entire memory space is used without preference.

Example

Example B-21

```
class tb_env extends vmm_ral_env;
   ...
   if (!this.mam_cfg.randomize()) begin
      `vmm_fatal(log, "Failed to randomize Memory 
configuration");
   end
   ...
   `vmm_note(log,$psprintf("vmm_mam_region::locality   ==> 
%0s",
             mam_cfg.locality.name));
   ...
endclass
```

vmm_mam_region

This class is used by the memory allocation manager to describe allocated memory regions. Instances of this class should not be created directly, therefore, this appendix does not document the constructor. Instances of this class should be created only from within the memory manager, in the "`vmm_mam::reserve_region()`" and "`vmm_mam::request_region()`" methods.

Summary

vmm_mam_region::get_start_offset()

Return the starting offset of the allocated region.

SystemVerilog

```
function bit [63:0] get_start_offset();
```

OpenVera

```
function bit [63:0] get_start_offset();
```

Description

Return the starting offset, within the memory, of the allocated region.

Example

Example B-22

```
class tb_env extends vmm_ral_env;
   ...
   virtual function void build();
      super.build();
      mam_region =
mam.reserve_region(mam_cfg.start_offset,mam_cfg.n_bytes);
      if(mam_region == null)
        begin
       `vmm_fatal(log,$psprintf({"Failed to reserve Memory
Location from ",
                  "%0h location."},mam_cfg.start_offset));
        end

`vmm_note(log,$psprintf("vmm_mam_region::get_start_offset
==> %h",
                mam_region.get_start_offset()));
   ...
   endfunction
```

```
        ...
    endclass
```

vmm_mam_region::get_end_offset()

Return the ending offset of the allocated region.

SystemVerilog

```
function bit [63:0] get_end_offset();
```

OpenVera

```
function bit [63:0] get_end_offset();
```

Description

Return the ending offset, within the memory, of the allocated region.

Example

Example B-23

```
class tb_env extends vmm_ral_env;
   ...
   mam_region =
mam.reserve_region(mam_cfg.start_offset,mam_cfg.n_bytes);
   if(mam_region == null)
     begin
       `vmm_fatal(log,$psprintf({"Failed to reserve Memory
Location from %0h ",
                  "location."},mam_cfg.start_offset));
     end
  `vmm_note(log,$psprintf("vmm_mam_region::get_end_offset
==> %h",
             mam_region.get_end_offset()));
   ...
endclass
```

vmm_mam_region::get_len()

Return the number of memory locations in the allocated region.

SystemVerilog

```
function int unsigned get_len();
```

OpenVera

```
function integer get_len();
```

Description

Return the number of consecutive memory locations (not necessarily bytes) in the allocated region.

Example

Example B-24

```
class tb_env extends vmm_ral_env;
   ...
   mam_region =
mam.reserve_region(mam_cfg.start_offset,mam_cfg.n_bytes);
   if(mam_region == null)
     begin
       `vmm_fatal(log,$psprintf({"Failed to reserve Memory
Location from %0h ",
                  "location."},mam_cfg.start_offset));
     end
    `vmm_note(log,$psprintf("vmm_mam_region::get_len
==> %d",
               mam_region.get_len()));
   ...
endclass
```

vmm_mam_region::get_n_bytes()

Return the number of memory locations in the allocated region.

SystemVerilog

```
function int unsigned get_n_bytes();
```

OpenVera

```
function integer get_n_bytes();
```

Description

Return the number of consecutive bytes in the allocated region. If the managed memory contains more than one byte per address, the number of bytes in an allocated region may be greater than the number of requested or reserved bytes.

Example

Example B-25

```
class tb_env extends vmm_ral_env;
   ...
   mam_region =
mam.reserve_region(mam_cfg.start_offset,mam_cfg.n_bytes);
   if(mam_region == null)
     begin
       `vmm_fatal(log,$psprintf({"Failed to reserve Memory
Location from %0h ",
                  "location."},mam_cfg.start_offset));
     end
  `vmm_note(log,$psprintf("vmm_mam_region::get_n_bytes
==> %d",
             mam_region.get_n_bytes()));
   ...
```

```
endclass
```

vmm_mam_region::get_memory()

Return the RAL memory abstraction class for the region.

SystemVerilog

```
function vmm_ral_mem get_memory();
```

Description

Return the reference to the RAL memory abstraction class for the memory implementing this allocated memory region. Returns `null` if no memory abstraction class was specified for the allocation manager that allocated this region.

Example

Example B-26

```
TBD
```

vmm_mam_region::get_virtual_registers()

Return the RAL abstraction class for the virtual register set.

SystemVerilog

```
function vmm_ral_vreg get_virtual_registers();
```

Description

Return the reference to the RAL virtual register abstraction class for the set of virtual registers implemented in the allocated region. Returns `null` if the memory region is not known to implement virtual registers.

Example

Example B-27

```
vmm_mam_region::get_virtual_registers() [page 208]

  vmm_ral_vreg vregs[];
  ral_model.memory_name.get_virtual_registers(vregs);
  foreach (vregs[i]) begin
     vregs[i].display();
  end
```

vmm_mam_region::psdisplay()

Human-readable description of allocated memory.

SystemVerilog

```
function string psdisplay(string prefix = "");
```

OpenVera

```
function string psdisplay(string prefix = "");
```

Description

Create a human-readable description of the allocated region. Each line of the description is prefixed with the specified prefix.

Example

Example B-28

```
...
   mam_region =
mam.reserve_region(mam_cfg.start_offset,mam_cfg.n_bytes);
   //Display Allocated Region [Min Location:Max Location]

`vmm_note(log,$psprintf("%s",mam_region.psdisplay("Allocat
ed Region")));
   ...
```

vmm_mam_region::read()

Reads a region location from the design.

SystemVerilog

```
task read(
   output vmm_rw::status_e status,
   input  bit [63:0]       offset,
   output bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e read_t(
   bit [63:0]       offset,
   var bit [63:0]   value,
   vmm_ral::path_e  path = vmm_ral::DEFAULT,
   string           domain = "",
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Reads the current value of the memory region location from the design using the specified access path. If the memory is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data/rvm_data` class properties in the `vmm_rw_access` transaction descriptors that are necessary to execute this read operation. This allows the physical and back-door read access to be traced back to the higher-level transaction that caused the access to occur.

This method can only be used on a region allocated by an allocation manager that is associated with a RAL memory abstraction class at construction time.

Example

Example B-29

```
TBD
```

vmm_mam_region::write()

Writes a region location in the design.

SystemVerilog

```
task write(
   output vmm_rw::status_e status,
   input  bit [63:0]       offset,
   input  bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int              stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e write_t(
   bit [63:0]       offset,
   bit [63:0]       value,
   vmm_ral::path_e  path = vmm_ral::DEFAULT,
   string           domain = "",
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer          stream_id = -1)
```

Description

Writes the specified value at the specified region location in the design using the specified access path. If the memory is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data/rvm_data` class properties in the `vmm_rw_access` transaction descriptors that are necessary to execute this write operation. This allows the physical and back-door write access to be traced back to the higher-level transaction that caused the access to occur.

This method can only be used on a region allocated by an allocation manager that is associated with a RAL memory abstraction class at construction time.

Example

Example B-30

```
TBD
```

vmm_mam_region::burst_read()

Perform a burst-read operation on the region.

SystemVerilog

```
task burst_read(
   output vmm_rw::status_e  status,
   input  vmm_ral_mem_burst burst,
   output bit [63:0]        value[],
   input  vmm_ral::path_e   path = vmm_ral::DEFAULT,
   input  string            domain = "",
   input  int               data_id = -1,
   input  int               scenario_id = -1,
   input  int               stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e read_t(
   vmm_ral_mem_burst burst,
   var bit [63:0]    value[*],
   vmm_ral::path_e   path = vmm_ral::DEFAULT,
   string            domain = "",
   integer           data_id = -1,
   integer           scenario_id = -1,
   integer           stream_id = -1)
```

Description

Burst-read the current values of the region locations specified by the burst descriptor. If the memory is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or the `vmm_ral_access::burst_read()` method. This allows the physical and back-door read access to be traced back to the higher-level transaction that caused the access to occur.

This method can only be used on a region allocated by an allocation manager that is associated with a RAL memory abstraction class at construction time.

Example

Example B-31

```
TBD
```

vmm_mam_region::burst_write()

Perform a burst-write operation on the region.

SystemVerilog

```
task burst_write(
   output vmm_rw::status_e  status,
   input  vmm_ral_mem_burst burst,
   input  bit [63:0]        value[],
   input  vmm_ral::path_e   path = vmm_ral::DEFAULT,
   input  string            domain = "",
   input  int               data_id = -1,
   input  int               scenario_id = -1,
   input  int               stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e write_t(
   vmm_ral_mem_burst burst,
   bit [63:0]        value[*],
   vmm_ral::path_e   path = vmm_ral::DEFAULT,
   string            domain = "",
   integer           data_id = -1,
   integer           scenario_id = -1,
   integer           stream_id = -1)
```

Description

Burst-write the specified values in the region locations specified by burst descriptor. If the memory is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or the `vmm_ral_access::burst_write()` method. This allows the physical and back-door write access to be traced back to the higher-level transaction that caused the access to occur.

This method can only be used on a region allocated by an allocation manager that is associated with a RAL memory abstraction class at construction time.

Example

Example B-32

```
TBD
```

vmm_mam_region::peek()

Peek a region location from the design.

SystemVerilog

```
task peek(
   output vmm_rw::status_e status,
   input  bit [63:0]       offset,
   output bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int              stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e peek_t(
   bit [63:0]       offset,
   var bit [63:0]   value,
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer          stream_id = -1)
```

Description

Reads the current value of the region location from the design using a back-door access.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door read access to be traced back to the higher-level transaction that caused the access to occur.

This method can only be used on a region allocated by an allocation manager that is associated with a RAL memory abstraction class at construction time.

Example

Example B-33

```
TBD
```

vmm_mam_region::poke()

Poke a region location in the design.

SystemVerilog

```
task poke(
   output vmm_rw::status_e status,
   input  bit [63:0]       offset,
   input  bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e poke_t(
   bit [63:0]       offset,
   bit [63:0]       value,
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Deposit the specified value at the specified region location in the design using a back-door access. Depending on the design model implementation, it may be possible to modify the content of a read-only memory.

The optional value of the arguments:

```
data_id
scenario_id
stream_id arguments
```

...are passed to the back-door access method. This allows the physical and back-door write access to be traced back to the higher-level transaction that caused the access to occur.

This method can only be used on a region allocated by an allocation manager that is associated with a RAL memory abstraction class at construction time.

Example

Example B-34

```
TBD
```

vmm_ral

Utility class for global symbolic values. Each set of symbolic values is specified using enumerated types. The symbolic values are accessed using a fully-qualified name, such as `vmm_ral::RW`.

A separate encapsulating class is used to minimize the length of these identifiers and to make them easier to share across classes.

Summary

vmm_ral::path_e

Symbolic values identifying the path used to access a register or memory within the design.

SystemVerilog

```
vmm_ral::BFM
vmm_ral::BACKDOOR
vmm_ral::DEFAULT
```

OpenVera

```
vmm_ral::BFM
vmm_ral::BACKDOOR
vmm_ral::DEFAULT
```

Description

`vmm_ral::BFM`

Accesses the register or memory through the appropriate physical interface transactor.   Also known as front-door access.

`vmm_ral::BACKDOOR`

Accesses the register or memory directly through an appropriate back-door mechanism, usually a hierarchical path into the design.

`vmm_ral::DEFAULT`

Accesses the register or memory using the default path specified in the enclosing block or system abstraction class.

Example

Example B-35

```
this.ral_model.status_reg.read(status, value,
                               vmm_ral::BACKDOOR);
```

Example B-36

```
status = uart_ral.status_reg.tx_en.write_t(1,
                                           vmm_ral::BFM);
```

vmm_ral::access_e

Symbolic values identifying the behavior of the bits implementing a field when read or written.

SystemVerilog

```
vmm_ral::RW
vmm_ral::RO
vmm_ral::WO
vmm_ral::W1
vmm_ral::RU
vmm_ral::RC
vmm_ral::W1C
vmm_ral::A0
vmm_ral::A1
vmm_ral::OTHER
vmm_ral::USER0
vmm_ral::USER1
vmm_ral::USER2
vmm_ral::USER3
vmm_ral::DC
```

OpenVera

```
vmm_ral::RW
vmm_ral::RO
vmm_ral::WO
vmm_ral::W1
vmm_ral::RU
vmm_ral::RC
vmm_ral::W1C
vmm_ral::A0
vmm_ral::A1
vmm_ral::OTHER
vmm_ral::USER0
vmm_ral::USER1
vmm_ral::USER2
vmm_ral::USER3
vmm_ral::DC
```

Description

`vmm_ral::RW`

The content of the field can be read and written. The content of the field is never modified by the design.

`vmm_ral::RO`

The content of the field can be read. Writing has no effect on its content. The content of the field is never modified by the design. It may be possible to modify the content of the field using “`vmm_ral_field::poke()`”.

`vmm_ral::WO`

The content of the field can be written. The value read is always zero and has no correlation with the content of the field. You can obtain the actual content of the field by using “`vmm_ral_field::peek()`”.

`vmm_ral::W1`

The content of the field can be read but can be physically written only once after a reset. The write-once mechanism prevents further modifications of the content of the field after the first “`vmm_ral_field::write()`” operation. The content of the field can be written using “`vmm_ral_field::poke()`” any number of times. The field can only be written again after a reset operation.

`vmm_ral::RU`

The content of the field can be read. Writing has no effect on its content. The content of the field can be modified by the design. It may be possible to modify the content of the field using “`vmm_ral_field::poke()`”.

`vmm_ral::RC`

The content of the field can be read. The content of the field is cleared after each read operation. Writing has no effect on its content. The content of the field is modified by the design. The actual content of the field can be obtained without modifying it using “`vmm_ral_field::peek()`”.

`vmm_ral::W1C`

The content of the field can be read. Writing to the field clears all bits corresponding to the bits that are set in the value being written. Bits that are cleared have no effect on its content. The content of the field is modified by the design. Using “`vmm_ral_field::poke()`” can force the bits to a specific value.

`vmm_ral::A1`

The content of the field can be read and cleared. Writing a 1 has no effect. The content of the field can also be set by the design. Using “`vmm_ral_field::poke()`” can force the bits to a specific value.

`vmm_ral::A0`

The content of the field can be read and set. Writing a 0 has no effect. The content of the field can also be cleared by the design. Using “`vmm_ral_field::poke()`” can force the bits to a specific value.

`vmm_ral::DC`

The content of the field is never checked for correctness (don't care) against the mirrored value when using the `“vmm_ral_field::mirror()”` method. The content of the mirror is updated as if the field was a `vmm_ral::RW` field. Use when the content of the field is not really predictable.

`vmm_ral::OTHER`

The effect of reading or writing the field on its content is unknown or may have effects on the content of other fields. The mirrored value assumes a `rw` behavior.

`vmm_ral::USER`*n*

The effect of read or writing the field on its content is user-defined. The mirrored value assumes a `rw` behavior.

Example

Example B-37

```
vmm_ral_field fields[];
ral_model.status_reg.get_fields(fields);
foreach (fields[i]) begin
   if (fields[i].get_access() != vmm_ral::RO) begin
      ‘vmm_error(log, "A status bit is not read-only");
   end
end
```

Example B-38

```
vmm_ral_field fields[*];
reg.get_fields(fields);
foreach (fields, i) {
   if (fields[i].get_access() == vmm_ral::W1C &&
       value[i] == 1’b0) begin
      value[i] = 1’b1;
   end
end
status = reg.write_t(value);
```

vmm_ral::check_e

Symbolic values identifying the behavior when an expected value does not match an actual value.

SystemVerilog

```
vmm_ral::QUIET
vmm_ral::VERB
```

OpenVera

```
vmm_ral::QUIET
vmm_ral::VERB
```

Description

`vmm_ral::QUIET`

No message reporting the discrepancy is reported.

`vmm_ral::VERB`

A message reporting the discrepancy is reported through an appropriate instance of the message service interface.

Example

Example B-39

```
this.ral_model.mirror(status, vmm_ral::QUIET,
                      vmm_ral::BACKDOOR);
```

Example B-40

```
status = this.ral_model.mirror_t(vmm_ral::VERB);
```

vmm_ral::endianness_e

Symbolic values identifying the endianness of values when mapped to a narrower data path.

SystemVerilog

```
vmm_ral::NO_ENDIAN
vmm_ral::LITTLE_ENDIAN
vmm_ral::BIG_ENDIAN
vmm_ral::LITTLE_FIFO
vmm_ral::BIG_FIFO
```

OpenVera

```
vmm_ral::NO_ENDIAN
vmm_ral::LITTLE_ENDIAN
vmm_ral::BIG_ENDIAN
vmm_ral::LITTLE_FIFO
vmm_ral::BIG_FIFO
```

Description

`vmm_ral::NO_ENDIAN`

It is not possible to map wide values onto discrete values on a narrower data path. Used as default value only.

`vmm_ral::LITTLE_ENDIAN`

Values are mapped onto a narrower data path using consecutive addresses, with the least significant bits in the lower addresses.

`vmm_ral::BIG_ENDIAN`

Values are mapped onto a narrower data path using consecutive addresses, with the most significant bits in the lower addresses.

`vmm_ral::LITTLE_FIFO`

Values are mapped onto a narrower data path using consecutive values at the same addresses, with the least significant bits transferred first.

`vmm_ral::BIG_FIFO`

Values are mapped onto a narrower data path using consecutive values at the same addresses, with the most significant bits transferred first.

Example

Example B-41

```
ral_blk_myblk  ral_blk  =  new ( vmm_ral ::  LITTLE_ENDIAN);
```

vmm_ral::reset_e

Symbolic values identifying the type of reset.

SystemVerilog

```
vmm_ral::HARD
vmm_ral::SOFT
```

OpenVera

```
vmm_ral::HARD
vmm_ral::SOFT
```

Description

`vmm_ral::HARD`

A hard reset.

`vmm_ral::VERB`

A soft reset.

Example

Example B-42

```
class tb_vmm_ral extends vmm_ral;
   ...
   reset_e rst_inst;

   function new();
      super.new();
   endfunction
   ...
endclass
```

```
class tb_env extends vmm_ral_env;
   ...
   tb_vmm_ral tb_ral;
   ...
   function new();
     super.new();
     this.tb_ral = new();
   endfunction

   virtual task hw_reset();
     tb_ral.rst_inst = vmm_ral::HARD;
     `vmm_note(log,$psprintf("reset_e =
%0d",tb_ral.rst_inst));
   endtask
   ...
endclass
```

vmm_ral::coverage_e

Symbolic values identifying functional coverage models.

SystemVerilog

```
vmm_ral::NO_COVERAGE
vmm_ral::REG_BITS
vmm_ral::ADDR_MAP
vmm_ral::FIELD_VALS
vmm_ral::ALL_COVERAGE
```

OpenVera

```
vmm_ral::NO_COVERAGE
vmm_ral::REG_BITS
vmm_ral::ADDR_MAP
vmm_ral::FIELD_VALS
vmm_ral::ALL_COVERAGE
```

Description

Symbolic values identifying functional coverage models. In order to use this class, the corresponding functional coverage models must have been previously generated using the `-c` option of `ralgen`. See `“Predefined Functional Coverage Models” on page 121` for more details.

`vmm_ral::NO_COVERAGE`

No coverage models.

`vmm_ral::REG_BITS`

The `“Register Bits”` coverage model.

`vmm_ral::ADDR_MAP`

The “Address Map” coverage model. This model can only be dynamically controlled through a block or system abstraction class.

`vmm_ral::FIELD_VALS`

The “Field Values” coverage model. This model can only be dynamically controlled through a block or system abstraction class.

`vmm_ral::ALL_COVERAGE`

All known coverage models.

Example

Example B-43

```
class ral_reg_SAMPLE extends vmm_ral_reg;
   ...
   local virtual function void domain_coverage(string
domain,bit rights,int idx);
     if (this.can_cover(vmm_ral::REG_BITS)) begin
         ral_cvr_reg_oc_ethernet_INT_MASK cg;
         cg = new(this.get_fullname(), domain);
         this.reg_bits[idx] = cg;
     end
   endfunction
   ...
endclass
```

vmm_ral_access

RAL component managing accesses to registers and memories through physical interfaces. Also provides an access-by-address and access-by-name service to registers and memories.

Summary

vmm_ral_access::set_model()

Associates a RAL abstraction model with the RAL physical access component.

SystemVerilog

```
function void set_model(vmm_ral_block_or_sys model)
```

OpenVera

```
task set_model(vmm_ral_block_or_sys model)
```

Description

Associates the specified RAL abstraction model with the RAL physical access component instance. Once a model is associated with an access component, registers, fields and memories can be accessed through the RAL.

A model can be associated with only one access component. Similarly, an access component can be associated with only one abstraction model.

It is possible to have multiple instances of the access component associated with their respective abstraction model.

Example

Example B-44

```
class my_env extends vmm_ral_env;
   ral_block_my_block model;

   function new();
      this.model = new;
      super.ral.set_model(this.model);
   endfunction: new
endclass: my_env
```

Example B-45

```
class my_env extends rvm_env {
   test_cfg cfg;
   vmm_ral_access ral[];
   ral_block_my_block model[];
   ...
   function build();
      this.ral = new [this.cfg.n];
      this.model = new [this.cfg.n];
      foreach (this.ral, i) {
         this.model[i] = new;
         this.ral[i].set_model(this.model[i]);
      end
   }
   ...
}
```

vmm_ral_access::get_model()

Returns the RAL abstraction model associated with the RAL physical access component.

SystemVerilog

```
function vmm_ral_block_or_sys get_model()
```

OpenVera

```
function vmm_ral_block_or_sys get_model()
```

Description

Return the RAL abstraction model that was associated with the RAL physical access component instance using the “vmm_ral_access::set_model()” method.

Example

Example B-46

```
if (this.ral.get_model() == null) begin
   ‘vmm_fatal(log, "No RAL model was specified");
end
```

vmm_ral_access::add_xactor()

Associates a physical-level transactor with a domain in the RAL abstraction model.

SystemVerilog

```
function void add_xactor(vmm_rw_xactor xact,
                         string        domain = "")
```

OpenVera

```
task add_xactor(vmm_rw_xactor xact,
               string        domain = "")
```

Description

Associates the specified physical-level transactor with the specified domain in the RAL abstraction model. The specified domain must exist in the model and only one transactor can be associated with a particular domain.

The physical-level transactor is implicitly started when added to the RAL abstraction model. This allows the predefined RAL tests to execute and the RAL model to later be used to configure the DUT in the `vmm_env::cfg_dut()` step, before the `vmm_env::start()` step has been executed. The physical-level transactor can be explicitly started in the `vmm_env::start()` step without adverse effects.

If a domain has no physical-level transactor associated with it, it is not possible to perform physical accesses to its registers and memories.

This method must be called after “vmm_ral_access::set_model()” has been called.

Example

Example B-47

```
class my_env extends vmm_ral_env;
   ral_ahb_master ahb_ma;
   ...
   function void build();
      super.build();
      this.ahb_ma = new(...);
      super.ral.add_xactor(this.ahb_ma);
   endfunction: build
   ...
endclass: my_env
```

Example B-48

```
class my_env extends vmm_ral_env {
   ral_pci_master pci_ma;
   ral_pci_config pci_cfg;
   ...
   task build() {
      super.build();
      this.pci_ma = new(...);
      super.ral.add_xactor(this.pci_ma, "PCI");
      this.pci_cfg = new(...);
      super.ral.add_xactor(this.pci_cfg, "CFG");
   }
   ...
}
```

vmm_ral_access::read()

Reads a value from a specified physical address.

SystemVerilog

```
task read(output vmm_rw::status_e status,
          input  bit [63:0]       addr,
          output bit [63:0]       data,
          input  int              n_bits = 64,
          string                  domain = "",
          input  int              data_id = -1,
          input  int              scenario_id = -1,
          input  int             stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e read_t(bit [63:0] addr,
                               var bit [63:0] data,
                               integer   n_bits = 64,
                               string    domain = "",
                               integer   data_id = -1,
                               integer   scenario_id = -1
                               integer   stream_id = -1)
```

Description

Reads a value from the specified physical address through the physical interface of the specified domain.

Returns an indication of the success or failure of the operation.

The optional value of the arguments:

```
n_bits
data_id
scenario_id
stream_id
```

...are assigned to their corresponding vmm_data/rvm_data class property in the “vmm_rw_access” transaction descriptor used to execute the read cycle. This allows the read cycle to be traced back to the higher-level transaction that caused the cycle to occur.

The mirrored content of any register or memory located at that address is not updated. This method is provided if low-level read operations are necessary. Reading of fields, registers or memory locations should be done using the “vmm_ral_field::read()”, “vmm_ral_reg::read()” or “vmm_ral_mem::read()” methods, respectively.

Example

Example B-49

```
this.ral.read(status, ’h0000, value);
if (status != vmm_rw::OK) begin
   vmm_error(log, "Error reading from 0x0000");
end
```

Example B-50

```
for (i = 0; i < ’hFFFF; i++) {
   status = this.ral.read_t(i, value, , "AHB");
   if (status != vmm_rw::OK) {
      vmm_error(log, psprintf("Error reading from AHB’h%h",
                              i));
   }
}
```

vmm_ral_access::write()

Writes a value at a specified physical address.

SystemVerilog

```
task write(output vmm_rw::status_e status,
           input  bit [63:0]       addr,
           input  bit [63:0]       data,
           input  int              n_bits,
           input  string           domain = "",
           input  int              data_id = -1,
           input  int              scenario_id = -1,
           input  int             stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e write_t(bit [63:0] addr,
                                  bit [63:0] data,
                                  integer    n_bits,
                                  string     domain = "",
                                  integer    data_id = -1,
                               integer    scenario_id = -1,
                                 integer    stream_id = -1)
```

Description

Writes the specified value at the specified physical address through the physical interface of the specified domain.

Returns an indication of the success or failure of the operation.

The optional value of the arguments:

```
n_bits
data_id
scenario_id
stream_id
```

...are assigned to their corresponding `vmm_data/rvm_data` class property in the “`vmm_rw_access`” transaction descriptor used to execute the write cycle. This allows the write cycle to be traced back to the higher-level transaction that caused the cycle to occur.

The mirrored content of any register or memory located at that address is not updated. This method is provided if low-level write operations are necessary. Writing of fields, registers or memory locations should be done using the “`vmm_ral_field::write()`”, “`vmm_ral_reg::write()`” and “`vmm_ral_mem::write()`” methods, respectively.

Example

Example B-51

```
this.ral.write(status, ’h0000, ’h0003);
if (status != vmm_rw::OK) begin
   vmm_error(log, "Error writing to 0x0000");
end
```

Example B-52

```
for (i = 0; i < ’hFFFF; i++) {
   status = this.ral.write_t(i, random(), , "AHB");
   if (status != vmm_rw::OK) {
      vmm_error(log, psprintf("Error writing to AHB’h%h",
                              i));
   }
}
```

vmm_ral_access::burst_read()

Reads a series value from a specified set of physical addresses.

SystemVerilog

```
task burst_read(output vmm_rw::status_e status,
                input  bit [63:0]        start,
                input  bit [63:0]        incr,
                input  bit [63:0]        max,
                input  int               n_beats,
                output bit [63:0]        data[],
                input  vmm_data          user = null,
                input  int               n_bits = 64,
                string                   domain = "",
                input  int               data_id = -1,
                input  int              scenario_id = -1,
                input  int               stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e burst_read_t(
     input  bit [63:0] start,
                 input  bit [63:0] incr,
                 input  bit [63:0] max,
                 input  integer    n_beats,
                 output bit [63:0] data[],
                 input  vmm_data   user = null,
                 input  integer    n_bits = 64,
                 string            domain = "",
                 input  integer    data_id = -1,
                 input  integer    scenario_id = -1,
                 input  integer    stream_id = -1)
```

Description

Reads a set of values using a burst read cycle through the physical interface of the specified domain. The following parameters are used to populate the `“vmm_rw_burst”` descriptor that will eventually be executed by the `“vmm_rw_xactor::execute_burst()”` method:

```
start
incr
max
n_beats
user
```

Returns an indication of the success or failure of the operation.

The mirrored content of any register or memory located in the burst area is not updated. This method is provided if low-level burst-read operations are necessary. Burst-read operations should be done using the `“vmm_ral_mem::burst_read()”` method.

Example

Example B-53

```
...
   this.ral.burst_read(status,'h0000,'h1000, value);
   if (status != vmm_rw::OK) begin
   `vmm_error(log, "Error reading a series value from a 
0x1000");
   end
   ...
```

vmm_ral_access::burst_write()

Writes a series of values at a specified set of physical addresses.

SystemVerilog

```
task burst_write(output vmm_rw::status_e status,
                 input  bit [63:0]       start,
                 input  bit [63:0]       incr,
                 input  bit [63:0]       max,
                 input  bit [63:0]       data[],
                 input  vmm_data         user = null,
                 input  int              n_bits = 64,
                 string                  domain = "",
                 input  int              data_id = -1,
                 input  int              scenario_id = -1,
                 input  int              stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e burst_write_t(
     input  bit [63:0] start,
                 input  bit [63:0] incr,
                 input  bit [63:0] max,
                 input  bit [63:0] data[],
                 input  vmm_data   user = null,
                 input  integer    n_bits = 64,
                 string            domain = "",
                 input  integer    data_id = -1,
                 input  integer    scenario_id = -1,
                 input  integer    stream_id = -1)
```

Description

Writes a set of value using a burst write cycle through the physical interface of the specified domain. The following parameters are used to populate the “`vmm_rw_burst`” descriptor that will eventually be executed by the “`vmm_rw_xactor::execute_burst()`” method:

```
start
incr
max
user
```

The number of beats is assumed to be equal to the number of data values to be written.

Returns an indication of the success or failure of the operation.

The mirrored content of any register or memory located in the burst area is not updated. This method is provided if low-level burst-write operations are necessary. Burst-write operations should be done using the “`vmm_ral_mem::burst_write()`” method.

Example

Example B-54

```
...
   this.ral.burst_write(status,'h0000,'h1000,'h100F);
   if (status != vmm_rw::OK) begin
  vmm_error(log, "Error writing a series value at 0x1000");
   end
   ...
```

vmm_ral_access::default_path

Specifies the default path to use when accessing registers and memories.

SystemVerilog

```
vmm_ral::path_e default_path
```

OpenVera

```
vmm_ral::path_e default_path
```

Description

Specifies the default path to use when accessing fields, registers or memories in the design. This default path can be superseded by the default access path in the system, sub-system and block abstraction models. This default path can also be superseded on a per-access basis.

The value of this property must be `vmm_ral::BFM` or `vmm_ral::BACKDOOR`. Whereas it is the topmost and ultimate default path specification, it cannot be `vmm_ral::DEFAULT`.

This default path is not used when using the “`vmm_ral_access::read()`” and “`vmm_ral_access::write()`” methods.

Example

Example B-55

```
super.ral.default_path = vmm_ral::BACKDOOR;
this.ral_model.soc.blk.default_path = vmm_ral::BFM;
this.ral_model.mirror();
```

vmm_ral_access::set_by_name()

Sets the mirror value of the specified register.

SystemVerilog

```
virtual function bit set_by_name(
   string      name,
   bit [63:0] value)
```

OpenVera

```
virtual function bit set_by_name(
   string      name,
   bit [63:0] value)
```

Description

Locates the register with the specified name and sets its value mirrored in the RAL abstraction model. The actual register in the design is not written or updated. See “vmm_ral_reg::set()” for more details on the operation of the mirror.

See “vmm_ral_block_or_sys::get_reg_by_name()” for details on how the register is located. Returns TRUE if a unique register of the specified name is found in the RAL model. Returns FALSE otherwise.

It is better to use the “vmm_ral_reg::set()” method directly in the RAL abstraction class for a register in the RAL model (accessed using hierarchical references - see “Understanding the Generated Model” on page 37) rather than using this method with a hard-coded name.

Example

Example B-56

```
Use:

   ral_model.blk.ctrl_reg.set(’h0001);

instead of:

   ral.set_by_name("ctrl_reg", ’h0001);
```

Example B-57

```
string reg_name;
reg_name = compute_name();
if (!this.ral.set_by_name(reg_name, ’hFFFF)) {
   rvm_error(log, {"No such register: ", reg_name};
}
```

vmm_ral_access::get_by_name()

Returns the mirror value of the specified register.

SystemVerilog

```
virtual function bit get_by_name(
   input  string      name,
   output bit [63:0] value)
```

OpenVera

```
virtual function bit get_by_name(
   string          name,
   var bit [63:0] value)
```

Description

Locates the register with the specified name and returns its value mirrored in the RAL abstraction model. The actual register in the design is not read. See "vmm_ral_reg::get()" for more details on the operation of the mirror.

See "vmm_ral_block_or_sys::get_reg_by_name()" for details on how the register is located. Returns TRUE if a unique register of the specified name is found in the RAL model. Returns FALSE otherwise.

It is better to use the "vmm_ral_reg::get()" method directly in the RAL abstraction class for a register in the RAL model (accessed using hierarchical references - see "Understanding the Generated Model" on page 37) rather than using this method with a hard-coded name.

Example

Example B-58

```
Use:

   ral_model.blk.status_reg.get(status);

instead of:

   ral.get_by_name("status_reg", status);
```

Example B-59

```
string reg_name;
reg_name = compute_name();
if (!this.ral.get_by_name(reg_name, val)) {
   rvm_error(log, {"No such register: ", reg_name};
}
printf("%s = %h\n", reg_name, val);
```

vmm_ral_access::read_by_name()

Reads a value from a specified named register.

SystemVerilog

```
task read_by_name(
        output vmm_rw::status_e status,
        input  string           name,
        output bit [63:0]       data,
        input  vmm_ral::path_e  path = DEFAULT,
        input  string           domain = "",
        input  int              data_id = -1,
        input  int              scenario_id = -1,
        input  int             stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e read_by_name_t(
   string          name,
   var bit [63:0]  data,
   vmm_ral::path_e path = DEFAULT,
   string          domain = "",
   integer         data_id = -1,
   integer         scenario_id = -1,
   integer        stream_id = -1)
```

Description

Locates the register with the specified name and performs the specified read operation through its RAL abstraction model. The mirror is updated.

See “`vmm_ral_block_or_sys::get_reg_by_name()`” for details on how the register is located.

It is better to use the “`vmm_ral_reg::read()`” method directly in the RAL abstraction class for a register in the RAL model (accessed using hierarchical references - see “Understanding the Generated Model” on page 37) rather than using this method with a hard-coded name.

Example

Example B-60

```
Use:

   ral_model.blk.status_reg.read(status, val);

instead of:

   ral.read_by_name(status, "status_reg", val);
```

Example B-61

```
string reg_name;
reg_name = compute_name();
status = this.ral.read_by_name_t(reg_name, val);
if (status != vmm_rw::IS_OK) {
   rvm_error(log, {"No such register: ", reg_name};
}
else printf("%s = %h\n", reg_name, val);
```

vmm_ral_access::write_by_name()

Writes a value from a specified named register.

SystemVerilog

```
task write_by_name(
        output vmm_rw::status_e status,
        input  string           name,
        output bit [63:0]       data,
        input  vmm_ral::path_e  path = DEFAULT,
        input  string           domain = "",
        input  int              data_id = -1,
        input  int              scenario_id = -1,
        input  int             stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e write_by_name_t(
   string          name,
   var bit [63:0]  data,
   vmm_ral::path_e path = DEFAULT,
   string          domain = "",
   integer         data_id = -1,
   integer         scenario_id = -1,
   integer        stream_id = -1)
```

Description

Locates the register with the specified name and performs the specified write operation through its RAL abstraction model. The mirror is updated.

See “vmm_ral_block_or_sys::get_reg_by_name()” for details on how the register is located.

It is better to use the “vmm_ral_reg::write()” method directly in the RAL abstraction class for a register in the RAL model (accessed using hierarchical references - see “Understanding the Generated Model” on page 37) rather than using this method with a hard-coded name.

Example

Example B-62

```
Use:

   ral_model.blk.ctrl_reg.write(status, ’h0003);

instead of:

   ral.write_by_name(status, "ctrl_reg", ’h0003);
```

Example B-63

```
string reg_name;
reg_name = compute_name();
status = this.ral.write_by_name_t(reg_name, ’hFFFF);
if (status != vmm_rw::IS_OK) {
   rvm_error(log, {"No such register: ", reg_name};
}
```

vmm_ral_access::read_mem_by_name()

Reads a value from a specified named memory.

SystemVerilog

```
task read_mem_by_name(
        output vmm_rw::status_e status,
        input  string           name,
        input  bit [63:0]       offset,
        output bit [63:0]       data,
        input  vmm_ral::path_e  path = DEFAULT,
        input  string           domain = "",
        input  int              data_id = -1,
        input  int              scenario_id = -1,
        input  int             stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e read_mem_by_name_t(
   string          name,
   bit [63:0]      offset,
   var bit [63:0]  data,
   vmm_ral::path_e path = DEFAULT,
   string          domain = "",
   integer         data_id = -1,
   integer         scenario_id = -1,
   integer        stream_id = -1)
```

Description

Locates the memory with the specified name and performs the specified read operation through its RAL abstraction model. The mirror is updated.

See “`vmm_ral_block_or_sys::get_mem_by_name()`” for details on how the memory is located.

It is better to use the “vmm_ral_mem::read()” method directly in the RAL abstraction class for a memory in the RAL model (accessed using hierarchical references - see “Understanding the Generated Model” on page 37) rather than using this method with a hard-coded name.

Example

Example B-64

```
Use:

   ral_model.blk.mem0.read(status, 0, val);

instead of:

   ral.read_mem_by_name(status, "mem0", 0, val);
```

Example B-65

```
string mem_name;
mem_name = compute_name();
status = this.ral.read_mem_by_name_t(mem_name, 0, val);
if (status != vmm_rw::IS_OK) {
   rvm_error(log, {"No such memory: ", mem_name};
}
else printf("%s[0] = %h\n", mem_name, val);
```

vmm_ral_access::write_mem_by_name()

Writes a value from a specified named register.

SystemVerilog

```
task write_mem_by_name(
        output vmm_rw::status_e status,
        input  string           name,
        input  bit [63:0]       offset,
        output bit [63:0]       data,
        input  vmm_ral::path_e  path = DEFAULT,
        input  string           domain = "",
        input  int              data_id = -1,
        input  int              scenario_id = -1,
        input  int             stream_id = -1)
```

OpenVera

```
function vmm_rw::status_e write_mem_by_name_t(
   string          name,
   bit [63:0]      offset,
   bit [63:0]      data,
   vmm_ral::path_e path = DEFAULT
   string          domain = "",
   integer         data_id = -1,
   integer         scenario_id = -1,
   integer        stream_id = -1)
```

Description

Locates the memory with the specified name and performs the specified write operation through its RAL abstraction model. The mirror is updated.

See “`vmm_ral_block_or_sys::get_mem_by_name()`” for details on how the memory is located.

It is better to use the “`vmm_ral_mem::write()`” method directly in the RAL abstraction class for a memory in the RAL model (accessed using hierarchical references - see “`Understanding the Generated Model” on page 37`) rather than using this method with a hard-coded name.

Example

Example B-66

```
Use:

   ral_model.blk.mem0.write(status, 0, ’h0003);

instead of:

   ral.write_mem_by_name(status, "mem0", 0, ’h0003);
```

Example B-67

```
string mem_name;
mem_name = compute_name();
status = this.ral.write_mem_by_name_t(mem_name, 0, ’hFFFF);
if (status != vmm_rw::IS_OK) {
   rvm_error(log, {"No such memory: ", mem_name};
}
```

vmm_ral_block_or_sys

Virtual base class for block and system descriptors. Provides functionality that is identical between blocks and systems.

Summary

vmm_ral_block_or_sys::log

Message service interface.

SystemVerilog

```
vmm_log log
```

OpenVera

```
rvm_log log
```

Description

Message service interface instance for the block or system descriptor. A single message service interface instance is shared by all block and system abstraction class instances.

Example

Example B-68

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   ...
   function new(vmm_ral_sys sys);
      super.new("VMM RAL System",sys);
      log = new("VMM RAL BLOCK/SYS","Log");
   endfunction
   ...
endclass
```

vmm_ral_block_or_sys::get_name()

Returns the name of the block or system.

SystemVerilog

```
virtual function string get_name()
```

OpenVera

```
virtual function string get_name()
```

Description

Returns the name of the block or system corresponding to the instance of the descriptor.

Example

Example B-69

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   ...
   function new(vmm_ral_sys sys);
      super.new("VMM RAL System",sys);
      log = new("VMM RAL BLOCK/SYS","Log");

`vmm_note(log,$psprintf({"vmm_ral_block_or_sys::get_name=>
Gets the name",
                           " of the block or system: %0s"},
                               log.get_name()));
   endfunction
   ...
endclass
```

vmm_ral_block_or_sys::get_type()

Returns the type name of the block or system.

SystemVerilog

```
virtual function string get_type()
```

OpenVera

```
virtual function string get_type()
```

Description

Returns the name of the block or system corresponding to the declaration of the descriptor.

This name is usually the same as the instance name returned by "`vmm_ral_block_or_sys::get_name()`", except when the instance name has been renamed because of multiple instances of the same block or system, or because a domain in a multiple-domain block or system is instantiated.

Example

Example B-70

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   ...
   function new(vmm_ral_sys sys);
      super.new("VMM RAL System",sys);
      log = new("VMM RAL BLOCK/SYS","Log");

`vmm_note(log,$psprintf({"vmm_ral_block_or_sys::get_type=>
Gets the type",
                           " name of the block or system: %0s"},
                                    log.get_type()));
   endfunction
   ...
endclass
```

vmm_ral_block_or_sys::get_fullname()

Returns the fully-qualified name of the block or system.

SystemVerilog

```
virtual function string get_fullname()
```

OpenVera

```
virtual function string get_fullname()
```

Description

Returns the hierarchical name of the block or system corresponding to the instance of the descriptor. The name of the top-level block or system is not included in the fully-qualified name as it is implicit for every RAL model.

Example

Example B-71

```
`vmm_note(log, $psprintf("Block Name:
%s\n",ral_model.block_name.get_fullname()));
```

vmm_ral_block_or_sys::get_domains()

Returns the name of the domains in the block or system.

SystemVerilog

```
function void get_domains(ref string names[])
```

OpenVera

```
task get_domains(var string names[*])
```

Description

Fills the specified dynamic array with the names of all the domains in the block or system. The order of the domain names is not specified.

Example

Example B-72

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   ...
   function new(vmm_ral_sys sys, string domain);
     super.new("VMM RAL System",sys,"");
     log = new("VMM RAL BLOCK/SYS","Log");
   endfunction
   ...
   this.get_domains("");

`vmm_note(log,$psprintf({"vmm_ral_block_or_sys::get_domain
s=>Gets the name",
                            " name of the domains in the
block or system.",
                     " size of names = %0d"},names.size()));
   ...
endclass
```

vmm_ral_block_or_sys::get_external_domain()

Returns the name of the top-level domain.

SystemVerilog

```
function string get_external_domain(string domain)
```

OpenVera

```
function string get_external_domain(string domain)
```

Description

Return the name of the top-level domain that instantiates the specified domain of this block or system.

Example

Example B-73

```
string reg_domains[];
vmm_ral_block b = reg.get_parent();
reg.get_domains(domains);
$write("Register %s is in domain %s\n",
       reg.get_fullname(),
b.get_external_domain(domains[0]));
```

vmm_ral_block_or_sys::get_parent()

Returns the system that instantiates this block or system.

SystemVerilog

```
virtual function vmm_ral_sys get_parent()
```

OpenVera

```
virtual function ram_ral_sys get_parent()
```

Description

Returns a reference to the descriptor of the system that includes the block or system corresponding to the descriptor instance. If this is the top-level block or system, returns null.

Example

Example B-74

```
vmm_ral_block_or_sys::get_parent() : [Page 263]

  vmm_ral_sys parent;
  parent = ral_model.block_name.get_parent();
  parent.display();
```

vmm_ral_block_or_sys::get_base_addr()

Returns the base address of the block or system.

SystemVerilog

```
virtual function bit [63:0] get_base_addr(string domain = "")
```

OpenVera

```
virtual function bit [63:0] get_base_addr(string domain = "")
```

Description

Returns the base address of the specified domain of the block or system in the address space of the immediately instantiating system.

If this is the top-level block or system, always returns 0.

Example

Example B-75

```
bit [`VMM_RAL_ADDR_WIDTH-1:0] addr;
addr = ral_model.block_name.get_base_addr();
```

vmm_ral_block_or_sys::C_addr_of()

Gets the base address of the block or system for the C API.

SystemVerilog

```
function int C_addr_of()
```

Description

Returns the address of the block or system in the C API address space. The returned address is designed to be passed to the RAL C API, as described in Chapter 17, "C Interface" and is not designed to be used as the actual physical base address of the block or system.

Example

Example B-76

```
To be added in a future release.
```

vmm_ral_block_or_sys::set_offset()

Modify the base address of the block or system.

SystemVerilog

```
virtual function bit set_offset(bit [63:0] offset,
    string domain = "")
```

Description

Dynamically relocate the base address of the specified domain in the block or subsystem in the address space of the immediately instantiating system. The new address range for the block or subsystem must not be occupied by another block or subsystem. Note that after using this method, the behavior of the RAL model will be different from the RALF specification.

Returns TRUE of the relocation was succesful. Returns FALSE if the specified domain does not exist in the immediately enclosing system or the new base address creates an overlap between this block or subsystem address range and another block or subsystem.

It is not possible to relocate the base address of the top-level system because is it not instantiated anywhere.

Example

Example B-77

```
ral_model.block_name.set_offset('h1000);
```

vmm_ral_block_or_sys::get_n_bytes()

Returns the width of the physical interface on this block or system.

SystemVerilog

```
virtual function int unsigned get_n_bytes(
   string domain = "")
```

OpenVera

```
virtual function integer get_n_bytes(
   string domain = "")
```

Description

Returns the width, in number of bytes, of the physical interface of the block or system for the specified domain.

Example

Example B-78

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   ...
   function new(vmm_ral_sys sys string domain);
     super.new("VMM RAL System",sys,"", 24);
     log = new("VMM RAL BLOCK/SYS","Log");
   endfunction
   ...
   `vmm_note(log,$psprintf({"
vmm_ral_block_or_sys::get_n_bytes => Gets the",
                         " width of the physical interface
on this block",
                        " or system. Width (bytes) = %0d"},
                            this.get_n_bytes("")));
```

```
    ...
endclass
```

vmm_ral_block_or_sys::get_endian()

Returns the endianness of the physical interface on this block or system.

SystemVerilog

```
virtual function vmm_ral::endianness_e get_endian(
   string domain = "")
```

OpenVera

```
virtual function vmm_ral::endianness_e get_endian(
   string domain = "")
```

Description

Returns the endianness of the physical interface of the block or system for the specified domain.

Example

Example B-79

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   ...
   function new(vmm_ral_sys sys,string domain);
     super.new("VMM RAL
System",sys,"",vmm_ral::LITTLE_ENDIAN);
     log = new("VMM RAL BLOCK/SYS","Log");
   endfunction
   ...
   `vmm_note(log,$psprintf({"
vmm_ral_block_or_sys::get_endian => Gets the",
                            " endianness of the physical
interface on this",
                                               " block or system ENDIAN = %0d"},
                                                  this.get_endian("")));
       ...
   endclass
```

vmm_ral_block_or_sys::default_access

Specifies the default access path for this block or system.

SystemVerilog

```
vmm_ral::path_e default_access
```

OpenVera

```
vmm_ral::path_e default_access
```

Description

Specifies the default access path when reading and writing registers and memories in this block or system.

If set to `vmm_ral::DEFAULT`, uses the default access path of the parent system. If set to `vmm_ral::DEFAULT` and this is the top-level block or system, uses the default access path of the RAL access interface. See “`vmm_ral_access::default_path`” for further details.

Example

Example B-80

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   vmm_ral::path_e default_access = vmm_ral::DEFAULT;
   ...
   function new(vmm_ral_sys sys,string domain);
     super.new("VMM RAL 
System",sys,"",vmm_ral::LITTLE_ENDIAN);
     log = new("VMM RAL BLOCK/SYS","Log");
   endfunction
   ...
endclass
...
this.ral.write(status, 'h0000, 'h000F);
if(status != vmm_rw::OK) begin
  `vmm_error(log, "Error reading from 0x0000");
end
...
```

vmm_ral_block_or_sys::get_default_access()

Returns the default access path for this block or system.

SystemVerilog

```
virtual function vmm_ral::path_e get_default_access()
```

OpenVera

```
virtual function vmm_ral::path_e get_default_access()
```

Description

Determines the default access path for this block or system, based on the value specified in its `default_access` property and its parent systems. See “`vmm_ral_block_or_sys::default_access`” for more details.

Example

Example B-81

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   vmm_ral::path_e default_access = vmm_ral::DEFAULT;
   ...
   function new(vmm_ral_sys sys,string domain);
     super.new("VMM RAL 
System",sys,"",vmm_ral::LITTLE_ENDIAN);
     log = new("VMM RAL BLOCK/SYS","Log");
   endfunction
   ...

`vmm_note(log,$psprintf({"vmm_ral_block_or_sys::get_defaul
t_access => Gets",
                            " the default access path for
block or system ",
                            "Default Access =
%0d"},this.get_default_access()));
   ...
endclass
```

vmm_ral_block_or_sys::display()

Displays a description of the block or system to `stdout`.

SystemVerilog

```
virtual function void display(string prefix = "",
                              string domain = "")
```

OpenVera

```
virtual task display(string prefix = "",
                     string domain = "")
```

Description

Displays the image created by the “`vmm_ral_block_or_sys::psdisplay()`” method to the standard output.

Example

Example B-82

```
ral_model.block_name.display();
```

vmm_ral_block_or_sys::psdisplay()

Creates a human-readable description of the block or system.

SystemVerilog

```
virtual function string psdisplay(string prefix = "",
                                  string domain = "")
```

OpenVera

```
virtual function string psdisplay(string prefix = "",
                                  string domain = "")
```

Description

Creates a human-readable description of the block or system, including the registers and memories it contains. Each line of the description is prefixed with the specified prefix.

If a domain is specified, only a description of that domain is created. Otherwise, a description of all domains within the block or system is created.

Example

Example B-83

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   ...
   function new(vmm_ral_sys sys,string domain);
     super.new("VMM RAL 
System",sys,"",vmm_ral::LITTLE_ENDIAN);
     log = new("VMM RAL BLOCK/SYS","Log");
   endfunction
   ...

`vmm_note(log,$psprintf({"vmm_ral_block_or_sys::psdisplay=
>Creates a human-",
                           " readable description of the
block or system: %s",
                      this.psdisplay("RAL_BLK_SYS",""))));
   ...
endclass
```

vmm_ral_block_or_sys::get_fields()

Returns all fields in this block or system.

SystemVerilog

```
virtual function void get_fields(
   ref vmm_ral_field fields[],
   input string     domain = "")
```

OpenVera

```
virtual task get_fields(
   var vmm_ral_field fields[*],
   string           domain = "")
```

Description

Fills the specified dynamic array with the descriptor for all of the fields contained in the block or system. If a domain is specified, only the fields accessible through the specified domain are returned. The order in which the fields are located in the array is not specified.

Example

Example B-84

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   vmm_ral_field SAMPLE;
   ...
   function new(vmm_ral_sys sys,string domain);
     super.new("VMM RAL 
System",sys,"",vmm_ral::LITTLE_ENDIAN);
     log = new("VMM RAL BLOCK/SYS","Log");
     this.SAMPLE = new(this,"SAMPLE","");
   endfunction
   ...
   this.get_fields(this.SAMPLE,"");
   ...
endclass
```

vmm_ral_block_or_sys::get_field_by_name()

Returns the field with the specified name in this block or system.

SystemVerilog

```
virtual function vmm_ral_field get_field_by_name(
   string name)
```

OpenVera

```
virtual function vmm_ral_field get_field_by_name(
   string name)
```

Description

Finds a field with the specified name in the block or system and returns its descriptor. If no fields are found, returns null.

Field name uniqueness is guaranteed only within registers. Therefore, if used on a system or block with more than one field having the same name, this method returns the first field found.

Example

Example B-85

```
class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   vmm_ral_field SAMPLE1;
   vmm_ral_field SAMPLE2;
   vmm_ral_field temp;
   ...
   function new(vmm_ral_sys sys,string domain);
     super.new("VMM RAL
System",sys,"",vmm_ral::LITTLE_ENDIAN);
```

```
      log = new("VMM RAL BLOCK/SYS","Log");
      this.SAMPLE1 = new(this,"SAMPLE1","");
      this.SAMPLE2 = new(this,"SAMPLE2","");
    endfunction
    ...
    temp = this.get_field_by_name("SAMPLE");
    ...
endclass
```

vmm_ral_block_or_sys::get_registers()

Returns all registers in this block or system.

SystemVerilog

```
virtual function void get_registers(
   ref vmm_ral_reg regs[],
   input string   domain = "")
```

OpenVera

```
virtual task get_registers(
   var vmm_ral_reg regs[*],
   string         domain = "")
```

Description

Fills the specified dynamic array with the descriptor for all of the registers contained in the block or system. If a domain is specified, only the registers accessible by the specified domain are returned. The order in which the registers are located in the array is not specified.

Example

Example B-86

```
vmm_ral_reg regs[];
  ral_model.block_name.get_registers(regs);
  foreach (regs[i]) begin
      regs[i].display();
  end
```

vmm_ral_block_or_sys::get_reg_by_name()

Returns the register with the specified name in this block or system.

SystemVerilog

```
virtual function vmm_ral_reg get_reg_by_name(
   string name)
```

OpenVera

```
virtual function vmm_ral_reg get_reg_by_name(
   string name)
```

Description

Finds a register with the specified name in the block or system and return its descriptor. If no registers are found, returns null.

Register name uniqueness is guaranteed only within blocks. Therefore, if used on a system with more than one register having the same name, this method returns the first register found.

Example

Example B-87

```
class ral_reg_REG_SAMPLE extends vmm_ral_reg;
  ...
  rand vmm_ral_field RX_M;
  rand vmm_ral_field TX_M;
  ...
  function new(vmm_ral_block blk);
     super.new(parent, name);
     this.RX_M = new(this, "RX_M");
     this.TX_M = new(this, "TX_M");
  endfunction
```

```
   ...
endclass

class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   ral_reg_REG_SAMPLE REG_SAMPLE;
   vmm_ral_reg temp;
   ...
   function new(vmm_ral_sys sys,string domain);
     super.new("VMM RAL
System",sys,"",vmm_ral::LITTLE_ENDIAN);
     log = new("VMM RAL BLOCK/SYS","Log");
     this.REG_SAMPLE = = new(this, "REG_SAMPLE");
   endfunction
   ...
   temp = this.get_reg_by_name("REG_SAMPLE");
   ...
endclass
```

vmm_ral_block_or_sys::get_reg_by_offset()

Gets the register at the specified offset in this block or system.

SystemVerilog

```
virtual function vmm_ral_reg get_reg_by_offset(
   bit [63:0]  offset,
   string      domain = "")
```

Description

Finds the register located at the specified offset within the block or system address space in the specified domain and returns its descriptor. If no register is found at the specified offset, returns NULL.

The entire register may occupy more than one offset within the address space of the block or system if it is wider than the physical interface. In such cases, this function looks for the start (lowest) address of the register’s address space.

This function has two implementation versions – a default version and a faster version, which takes more memory than the default version.

In the default (slower) version, all registers of the underlying blocks and subsystems are searched, short-circuiting the search wherever possible, to check if a register exists at the specified block/system level offset/address.

The faster version uses associative arrays to cache block level register offsets, which helps to speed up the search, given an offset. Memory consumption is higher in the faster version, due to caching. You can activate the faster version by defining the runtime macro VMM_RAL_FAST_SRCH.

Example

Example B-88

```
vmm_ral_reg register;
  register = ral_model.block_name.get_reg_by_offset('h10);
  if (register == null) `vmm_error(log, "wrong offset");
```

vmm_ral_block_or_sys::get_virtual_fields()

Returns all virtual fields in this block or system.

SystemVerilog

```
virtual function void get_virtual_fields(
   ref vmm_ral_vfield fields[],
   input string      domain = "")
```

Description

Fills the specified dynamic array with the descriptor for all of the virtual fields contained in the block or system. If a domain is specified, only the fields implemented in memories accessible through the specified domain are returned. The order in which the fields are located in the array is not specified.

Example

Example B-89

```
vmm_ral_block_or_sys::get_virtual_fields() [page 279]

  vmm_ral_vfield vfields[];
  ral_model.block_name.get_virtual_fields(vfields);
  foreach (vfields[i]) begin
     vfields[i].display();
  end
```

vmm_ral_block_or_sys::get_virtual_field_by_name()

Returns the virtual field with the specified name in this block or system.

SystemVerilog

```
virtual function vmm_ral_vfield get_virtual_field_by_name(
   string name)
```

Description

Finds a virtual field with the specified name in the block or system and returns its descriptor. If no fields are found, returns null.

Field name uniqueness is guaranteed only within virtual registers. Therefore, if used on a system or block with more than one field having the same name, this method returns the first field found.

Example

Example B-90

```
vmm_ral_block_or_sys::get_virtual_field_by_name() [page
280]

  vmm_ral_vfield vfield;
  vfield =
ral_model.block_name.get_virtual_field_by_name("virtual_fi
eld_name");
  if (vfield == null) `vmm_error(log, "specified field
doesn't exists");
```

vmm_ral_block_or_sys::get_virtual_registers()

Returns all virtual registers in this block or system.

SystemVerilog

```
virtual function void get_virtual_registers(
   ref vmm_ral_vreg regs[],
   input string    domain = "")
```

Description

Fills the specified dynamic array with the descriptor for all of the virtual registers contained in the block or system. If a domain is specified, only the registers implemented in memories accessible by the specified domain are returned. The order in which the registers are located in the array is not specified.

Example

Example B-91

```
vmm_ral_block_or_sys::get_virtual_registers() [page 281]

  vmm_ral_vreg vregs[];
  ral_model.block_name.get_virtual_registers(vregs);
  foreach (vregs[i]) begin
     vregs[i].display();
  end
```

vmm_ral_block_or_sys::get_vreg_by_name()

Returns the virtual register with the specified name in this block or system.

SystemVerilog

```
virtual function vmm_ral_vreg get_vreg_by_name(
   string name)
```

Description

Finds a virtual register with the specified name in the block or system and return its descriptor. If no registers are found, returns null.

Register name uniqueness is guaranteed only within blocks. Therefore, if used on a system with more than one register having the same name, this method returns the first register found.

Example

Example B-92

```
vmm_ral_block_or_sys::get_vreg_by_name() [page 282]

  vmm_ral_vreg vreg;
  vreg = 
ral_model.block_name.get_vreg_by_name("virtual_reg_name");
  if (vreg == null) `vmm_error(log, "specified virtual 
register doesn't exists");
```

vmm_ral_block_or_sys::get_memories()

Returns all memories in this block or system.

SystemVerilog

```
virtual function void get_memories(
   ref vmm_ral_mem mems[],
   input string   domain = "")
```

OpenVera

```
virtual task get_memories(
   var vmm_ral_mem mems[*],
   string         domain = "")
```

Description

Fills the specified dynamic array with the descriptor for all of the memories contained in the block or system. If a domain is specified, only those memories accessible in the specified domain are returned. The order in which the memories are located in the array is not specified.

Example

Example B-93

```
vmm_ral_mem memories[];
  ral_model.block_name.get_memories(memories);
  foreach (memories[i]) begin
      memories[i].display();
  end
```

vmm_ral_block_or_sys::get_mem_by_name()

Returns the memory with the specified name in this block or system.

SystemVerilog

```
virtual function vmm_ral_mem get_mem_by_name(
   string name)
```

OpenVera

```
virtual function vmm_ral_mem get_mem_by_name(
   string name)
```

Description

Finds a memory with the specified name in the block or system and returns its descriptor. If no memories are found, returns null.

Memory name uniqueness is guaranteed only within blocks. Therefore, if used on a system with more than one memory having the same name, this method returns the first memory found.

Example

Example B-94

```
class my_ral_block_or_sys extends vmm_ral_block_or_sys;
   ...
   rand my_ral_mem my_mem;
   vmm_ral_mem temp;
   ...
   temp = this.get_mem_by_name("my_mem");
   ...
endclass
```

vmm_ral_block_or_sys::get_constraints()

Returns the constraint blocks in this block or system.

SystemVerilog

```
virtual function void get_constraints(
   ref string names[])
```

OpenVera

```
virtual task get_constraints(
   var string names[*])
```

Description

Fills the specified dynamic array with the names of the constraint blocks in this block or system. Does not include the constraint blocks in the registers or fields in this block or system. The location of each constraint block name in the array is not defined.

Example

Example B-95

```
class my_ral_block_or_sys extends vmm_ral_block_or_sys;
   ...
   rand vmm_ral_field MINFL;
   ...
   constraint MINFL_spec {
      MINFL.value == 'h40;
   }
   ...
   function new(vmm_ral_sys sys);
      ...
      Xadd_constraintsX("MINFL_spec");
   endfunction
   ...
endclass
...
my_ral_block_or_sys my_blk_sys;
...
string str[];
...
this.ral_model.my_blk_sys.get_constraints(str);
foreach (str[i])
   `vmm_note(log,$psprintf("Constraint Name is
%0s",str[i]));
...
```

vmm_ral_block_or_sys::has_cover()

Query available functional coverage models.

SystemVerilog

```
virtual function int has_cover(vmm_ral::coverage_e models)
```

OpenVera

```
virtual function integer has_cover(vmm_ral::coverage_e
models)
```

Description

Queries which of the specified functional coverage models are available. Multiple functional coverage models can be specified by adding the functional coverage model identifiers.

Returns the sum of the available functional coverage models.

A functional coverage model is available only if it has been generated by `ralgen` (see “Predefined Functional Coverage Models” on page 121) then enabled when calling “vmm_ral_block::new()” or “vmm_ral_sys::new()”.

Example

Example B-96

```
ral_model.set_cover(ral_model.has_cover(vmm_all::ALL_CO
VERAGE));
```

vmm_ral_block_or_sys::set_cover()

Turns functional coverage measurement on or off.

SystemVerilog

```
virtual function int set_cover(vmm_ral::coverage_e is_on)
```

OpenVera

```
virtual function integer set_cover(vmm_ral::coverage_e
is_on)
```

Description

Turns the collection of functional coverage measurements on or off for this block or system and all subsystems, blocks, registers, fields and memories within it. The functional coverage measurement is turned on for every coverage model specified. Multiple functional coverage models can be specified by adding the functional coverage model identifiers. All other functional coverage models are turned off.

Returns the sum of all functional coverage models whose measurements were previously on.

This method can only control the measurement of functional coverage models that have been generated by `ralgen` (see “Predefined Functional Coverage Models” on page 121) then enabled when calling “vmm_ral_block::new()” or “vmm_ral_sys::new()”.

See the “vmm_ral_block_or_sys::has_cover()” method to identify the available functional coverage models.

Example

Example B-97

```
ral_model.set_cover(vmm_all::NO_COVERAGE));

ral_model.set_cover(vmm_all::REG_BITS +
vmm_all::ADDR_MAP);
```

vmm_ral_block_or_sys::is_cover_on()

Queries if functional coverage measurement is on or off.

SystemVerilog

```
virtual function bit is_cover_on(vmm_ral::coverage_e is_on)
```

OpenVera

```
virtual function bit is_cover_on(vmm_ral::coverage_e is_on)
```

Description

Returns TRUE of measurement for all of the specified functional coverage models that are currently on. Multiple functional coverage models can be specified by adding the functional coverage model identifiers. See “vmm_ral_block_or_sys::set_cover()” for more details.

Example

Example B-98

```
class my_ral_block_or_sys extends vmm_ral_block_or_sys;
   ...
   function new(vmm_ral_sys sys,int cover_on);
     super.new("VMM RAL System",sys,vmm_ral::NO_COVERAGE);
      log = new("VMM RAL BLOCK/SYS","Log");

`vmm_note(log,$psprintf({"vmm_ral_block_or_sys::is_cover_o
n=>Queries if",
                          "functional coverage measurement
is on or off.",
                               " =
%b"},this.is_cover_on(vmm_ral::REG_BITS));
   endfunction
   ...
endclass
```

vmm_ral_block_or_sys::reset()

Resets the mirror values in this block or system.

SystemVerilog

```
virtual function void reset(string           domain = "",
    vmm_ral::reset_e kind = vmm_ral::HARD)
```

OpenVera

```
virtual task reset(string           domain = "",
    vmm_ral::reset_e kind = vmm_ral::HARD)
```

Description

Sets the mirror value of all registers in the block or system to the specified hard or soft reset value. Does not actually set the value of the registers in the design, only the values mirrored in their corresponding descriptor in the RAL model.

If a domain is specified, only those registers accessible through the specified domain are reset.

The mirror values of memories is not modified.

Example

Example B-99

```
ral_model.block_name.reset();
```

vmm_ral_block_or_sys::needs_update()

Queries if a mirrored value in this block or system has been set.

SystemVerilog

```
virtual function bit needs_update()
```

OpenVera

```
virtual function bit needs_update()
```

Description

If a mirror value has been modified in the RAL model without actually updating the actual register, the mirror and state of the registers are outdated. This method returns TRUE if the state of the registers needs to be updated to match the mirrored values (or vice-versa).

The mirror values, or actual content of registers, are not modified. For additional information, see "vmm_ral_block_or_sys::update()" or "vmm_ral_block_or_sys::mirror()".

Example

Example B-100

```
class ral_reg_REG_SAMPLE extends vmm_ral_reg;
  ...
  rand vmm_ral_field RX_M;
  rand vmm_ral_field TX_M;
  ...
  function new(vmm_ral_block blk);
     super.new(parent, name);
     this.RX_M = new(this, "RX_M");
     this.TX_M = new(this, "TX_M");
  endfunction
  ...
endclass

class ral_blk_sys_TEMP extends vmm_ral_block_or_sys;
   ...
   vmm_log log;
   ral_reg_REG_SAMPLE REG_SAMPLE;
   ...
   function new(vmm_ral_sys sys,string domain);
     super.new("VMM RAL 
System",sys,"",vmm_ral::LITTLE_ENDIAN);
     log = new("VMM RAL BLOCK/SYS","Log");
     this.REG_SAMPLE = = new(this, "REG_SAMPLE");
   endfunction
   ...

`vmm_note(log,$psprintf({"vmm_ral_block_or_sys::needs_upda
te=>Queries if a"
                            "mirrored value in this block 
or system has been",
                        " set. = %b",this.needs_update()));
   ...
endclass
```

vmm_ral_block_or_sys::update()

Updates registers to match mirrored values in this block or system.

SystemVerilog

```
virtual task update(
   output vmm_rw::status_e status,
   input vmm_ral::path_e  path = vmm_ral::DEFAULT)
```

OpenVera

```
virtual function vmm_rw::status_e update_t(
   vmm_ral::path_e path = vmm_ral::DEFAULT)
```

Description

Using the minimum number of write operations, updates the content of the registers in the design to match the mirrored values. The update can be performed using the physical interfaces (front-door access) or back-door (zero-time) access.

This method performs the reverse operation of "vmm_ral_block_or_sys::mirror()".

Example

Example B-101

```
bit update;
vmm_rw::status_e status;
update = ral_model.block_name.needs_update();
if (update == 1) begin
   ral_model.block_name.update(status);
end
```

vmm_ral_block_or_sys::mirror()

Updates the mirrored value of registers in this block or system to match the design.

SystemVerilog

```
virtual task mirror(
   output vmm_rw::status_e status,
   input  vmm_ral::check_e  check = vmm_ral::QUIET,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT)
```

OpenVera

```
virtual function vmm_rw::status_e mirror_t(
   vmm_ral::check_e  check = vmm_ral::QUIET,
   vmm_ral::path_e  path  = vmm_ral::DEFAULT)
```

Description

Updates the content of the registers mirror values to match their corresponding values in the design. The mirroring can be performed using the physical interfaces (front-door access) or back-door (zero-time) access. If the `check` argument is specified as `vmm_ral::`VERB, an error message is issued if the current mirrored value does not match the actual value in the design.

This method performs the reverse operation of `“vmm_ral_block_or_sys::update()”`.

Example

Example B-102

```
vmm_rw::status_e status;
ral_model.block_name.mirror(status);
```

vmm_ral_block_or_sys::readmemh()

Initializes the registers and memories in the block or system.

SystemVerilog

```
virtual task readmemh(string filename)
```

OpenVera

```
virtual task readmemh_t(string filename)
```

Description

Not yet implemented.

Initializes the content of all registers and memories in the design using the values in the specified file. The values are updated using the default access path. See “`vmm_ral_block_or_sys::writememh()`” for details.

The format of the file is not specified.

Example

Example B-103

```
TBD
```

vmm_ral_block_or_sys::writememh()

Dumps the value of all registers and memories in the block or system.

SystemVerilog

```
virtual task writememh(string filename)
```

OpenVera

```
virtual task writememh_t(string filename)
```

Description

Not yet implemented.

Dumps the content of all registers and memories in the design to the specified file. The file can then be used as an input for the `“vmm_ral_block_or_sys::readmemh()”` method. The values are obtained using the default access path.

The format of the file is not specified.

Example

Example B-104

```
TBD
```

vmm_ral_block

Block descriptor class derived from “`vmm_ral_block_or_sys`”.

Summary

vmm_ral_block::new()

Creates an instance of a RAL model.

SystemVerilog

```
function new(vmm_ral::coverage_e cover_on =
vmm_ral::NO_COVERAGE);
```

OpenVera

```
task new(vmm_ral::coverage_e cover_on =
vmm_ral::NO_COVERAGE);
```

Description

Creates an instance of a RAL model with the corresponding block as the top-level structural element.

The `cover_on` argument specifies the functional coverage models to be enabled in the RAL model. Multiple functional coverage models may be specified by adding their symbolic names. Only functional coverage models that were generated by `ralgen` using the `-c` option can be enabled. Because the functional coverage models affect the memory footprint and runtime performance of a RAL model, they should be enabled only when relevant.

It is not possible to enable a functional coverage model at a later time, but it is possible to turn the measurement of a functional coverage model off then back on using the `“vmm_ral_block_or_sys::set_cover()”` method.

Example

Example B-105

```
ral_sys_mysys ral_model = new(vmm_ral::REG_BITS +
                              vmm_ral::ADDR_MAP);
```

vmm_ral_env

Base class, derived from `vmm_env`, to be used when creating RAL-based verification environments. This section documents only the differences or additions in this class in comparison to the base class. You can find the documentation for the properties and methods inherited as-is in the *Reference Verification Methodology User's Guide* and the *Verification Methodology Manual for SystemVerilog*.

Summary

vmm_ral_env::new()

Creates a RAL-based environment.

SystemVerilog

```
function new(string name = "RAL-Based Verif Env")
```

OpenVera

```
task new(string name = "RAL-Based Verif Env")
```

Description

Creates a new instance of a RAL-based environment base class. This method is usually called by the constructor of a user-defined extension of this class using `super.new()`.

Example

Example B-106

```
class tb_env extends vmm_ral_env;
   ...
   function new();
      super.new("RAL Based RTL Env");
      this.cfg = new;
   endfunction
   ...
endclass
```

vmm_ral_env::ral

RAL access interface object instance.

SystemVerilog

```
vmm_ral_access ral
```

OpenVera

```
vmm_ral_access ral
```

Description

Instance of the RAL access interface used to access registers and memories in the design verified by this environment. The instance is allocated in the constructor for this class and should not be modified, replaced, nulled or reallocated. See "`vmm_ral_access`" for additional information.

Example

Example B-107

```
class tb_env extends vmm_ral_env;
   ...
   if(this.ral != null)
     begin
      `vmm_note(log,{"Instance of vmm_ral_access class has
been created. ",
                 "Instance name is ral."});
     end
   else
     begin
     `vmm_error(log,"Creation of instance of vmm_ral_access
class is failed.");
     end
```

```
      ...
   endclass
```

vmm_ral_env::hw_reset

Performs a hardware reset operation.

SystemVerilog

```
virtual task hw_reset()
```

OpenVera

```
virtual task hw_reset_t()
```

Description

This method must be overloaded in a user-extension of this base class. It performs a complete hardware reset of the design. The design must be in its reset state when the task returns.

This method is called by the default implementation of the `“vmm_ral_env::reset_dut”` method.

Example

Example B-108

```
class tb_env extends vmm_ral_env;
   ...
   virtual task hw_reset();
      tb_top.reset <= 1;
      `vmm_note(log,"Entire System Hardware is reseted.");
      @ (posedge tb_top.clk);
      tb_top.reset <= 0;
   endtask
   ...
endclass
...
initial
```

```
        begin
           tb_env env = new();
           env.reset_dut();
        end
```

vmm_ral_env::sw_reset

Performs a software reset operation.

SystemVerilog

```
virtual task sw_reset()
```

OpenVera

```
virtual task sw_reset_t()
```

Description

This method must be overloaded in a user-extension of this base class. It performs a complete software reset of the design. The design must be in its reset state when the task returns.

Example

Example B-109

```
class tb_env extends vmm_ral_env;
   ...
   virtual task sw_reset();
      tb_top.reset <= 1;
      `vmm_note(log,"Entire System Software is reseted.");
      @ (posedge tb_top.clk);
      tb_top.reset <= 0;
   endtask

   virtual task reset_dut();
     super.reset_dut();
     this.sw_reset();
   endtask
   ...
endclass
```

vmm_ral_env::reset_dut

Hardware reset step in a simulation sequence.

SystemVerilog

```
virtual task reset_dut()
```

OpenVera

```
virtual task reset_dut_t()
```

Description

This method calls the "`vmm_ral_env::hw_reset`" method to reset the design. It should not need to be overloaded in a user-defined extension of this class.

It is important that the design under verification be as idle and inactive as possible after the completion of this method. The predefined tests built on top of the verification environment expect that the value of registers will remain unchanged between accesses.

Example

Example B-110

```
class tb_env extends vmm_ral_env;
   ...
   virtual task hw_reset();
   ...
   ...
   endtask
   ...
endclass
...
```

```
program test_ral_env;
  ...
  initial
    begin
       tb_env env = new();
       env.reset_dut();
    end
  ...
endprogram
```

vmm_ral_field

Field descriptors.

Summary

vmm_ral_field::log

Message service interface.

SystemVerilog

```
vmm_log log
```

OpenVera

```
rvm_log log
```

Description

Message service interface instance for the field descriptor. A single message service interface instance is shared by all field abstraction class instances.

Example

Example B-111

```
class my_ral_field extends vmm_ral_field;
   ...
   vmm_log log;
   ...
   function new(vmm_ral_reg rg);
      super.new("VMM RAL REGISTER",rg);
      log = new("VMM RAL REGISTER","Log");
     `vmm_note(log,"vmm_ral_field::log instance created.");
      ...
   endfunction
   ...
endclass
```

vmm_ral_field::get_name()

Returns the name of the field.

SystemVerilog

```
virtual function string get_name()
```

OpenVera

```
virtual function string get_name()
```

Description

Returns the name of the field corresponding to the instance of the descriptor.

Example

Example B-112

```
  vmm_ral_field fields[];
  ral_model.block_name.reg1.get_fields(mems);
  foreach (fields[i]) begin
      `vmm_note(log, $psprintf("Field Name: %s\n",
fields[i].get_name()));
  end
```

vmm_ral_field::get_fullname()

Returns the fully-qualified name of the field.

SystemVerilog

```
virtual function string get_fullname()
```

OpenVera

```
virtual function string get_fullname()
```

Description

Returns the hierarchical name of the field corresponding to the instance of the descriptor. The name of the top-level block or system is not included in the fully-qualified name as it is implicit for every RAL model.

Example

Example B-113

```
  vmm_ral_field fields[];
  ral_model.block_name.reg1.get_fields(mems);
  foreach (fields[i]) begin
      `vmm_note(log, $psprintf("Field Name: %s\n",
fields[i].get_fullname()));
  end
```

vmm_ral_field::get_register()

Returns the register that instantiates this field.

SystemVerilog

```
virtual function vmm_ral_reg get_register()
```

OpenVera

```
virtual function ram_ral_reg get_register()
```

Description

Returns a reference to the descriptor of the register that includes the field corresponding to the descriptor instance.

Example

Example B-114

```
  vmm_ral_reg register;
  register = 
ral_model.block_name.reg1.field_1.get_register();
  register.display();
```

vmm_ral_field::get_lsb_pos_in_register()

Returns the offset of the least-significant bit of the field.

SystemVerilog

```
virtual function int unsigned get_lsb_pos_in_register()
```

OpenVera

```
virtual function integer get_lsb_pos_in_register()
```

Description

Returns the index of the least significant bit of the field in the register that instantiates it. An offset of 0 indicates a field that is aligned with the least-significant bit of the register.

Example

Example B-115

```
  vmm_ral_field fields[];
  ral_model.block_name.reg1.get_fields(fields);
  foreach (fields[i]) begin
     `vmm_note(log, $psprintf("Offset of LSB of the field:
%s\n", fields[i].get_lsb_pos_in_register()));
  end
```

vmm_ral_field::get_n_bits()

Returns the width of the field.

SystemVerilog

```
virtual function int unsigned get_n_bits()
```

OpenVera

```
virtual function integer get_n_bits()
```

Description

Returns the width, in number of bits, of the field.

Example

Example B-116

```
  vmm_ral_field fields[];
  ral_model.block_name.reg1.get_fields(fields);
  foreach (fields[i]) begin
      `vmm_note(log, $psprintf("Width of the field: %d\n", 
fields[i].get_n_bits()));
  end
```

vmm_ral_field::get_access()

Returns the access mode of the field.

SystemVerilog

```
virtual function vmm_ral::access_e get_access(string domain
= "")
```

OpenVera

```
virtual function vmm_ral::access_e get_access(string domain
= "")
```

Description

Returns the specification of the behavior of the field when written and read through the optionally specified domain.

If the register containing the field is shared across multiple domains, a domain must be specified. The access mode of a field in a specific domain may be restricted. For example, a RW field may only be writable through one of the domains and read-only through all of the other domains.

Example

Example B-117

```
class my_ral_reg extends vmm_ral_reg;
   ...
   rand my_ral_field my_field;
   ...
   `vmm_note(log,$psprintf({"vmm_ral_field::get_access =>
Gets the access mode",
             " of the field.=
%0d"},this.my_field.get_access("")));
```

```
   ...
endclass
```

vmm_ral_field::set_access()

Set the access mode of the field.

SystemVerilog

```
virtual function vmm_ral::access_e
set_access(vmm_ral::access_e mode)
```

OpenVera

```
virtual function vmm_ral::access_e
set_access(vmm_ral::access_e mode)
```

Description

Set the access mode of the field to the specified mode and return the previous access mode.

WARNING! Using this method will modify the behavior of the RAL model from the behavior specified in the original specification.

Example

Example B-118

```
class my_ral_reg extends vmm_ral_reg;
   ...
   rand vmm_ral_field my_ral_field;
   ...
   function new(vmm_ral_block blk);
      super.new("VMM RAL BLOCK",blk);
      log = new("VMM RAL BLOCK","Log");
   endfunction
   ...
   my_ral_field my;
   ...
   this.my.set_access(vmm_ral::RW);
   ...
   `vmm_note(log,$psprintf({"After Setting the access mode
of the field.",
           " access mode = %0d"},this.my.get_access("")));
   ...
endclass
```

vmm_ral_field::display()

Displays a description of the field to stdout.

SystemVerilog

```
virtual function void display(string prefix = "")
```

OpenVera

```
virtual task display(string prefix = "")
```

Description

Displays the image created by the "vmm_ral_field::psdisplay()" method on the standard output.

Example

Example B-119

```
vmm_ral_field fields[];
ral_model.block_name.reg1.get_fields(fields);
foreach (fields[i]) begin
    fields[i].display();
end
```

vmm_ral_field::psdisplay()

Creates a human-readable description of the field.

SystemVerilog

```
virtual function string psdisplay(string prefix = "")
```

OpenVera

```
virtual function string psdisplay(string prefix = "")
```

Description

Creates a human-readable description of the field and its current mirrored value. Each line of the description is prefixed with the specified prefix.

Example

Example B-120

```
vmm_ral_field fields[];
ral_model.block_name.reg1.get_fields(fields);
foreach (fields[i]) begin
   `vmm_note(log, $psprintf("psdisplay of the field: %s\n",
fields[i].psdisplay()));
end
```

vmm_ral_field::set()

Sets the mirror value of the field.

SystemVerilog

```
virtual function void set(bit [63:0] value)
```

OpenVera

```
virtual task set(bit [63:0] value)
```

Description

Sets the mirror value of the field to the specified value. Does not actually set the value of the field in the design, only the value mirrored in its corresponding descriptor in the RAL model. Use the `“vmm_ral_reg::update()”` method to update the actual register with the mirrored value or the `“vmm_ral_field::write()”` method to set the actual field and its mirrored value.

The final value in the mirror is a function of the field access mode and the set value, just like a normal physical write operation to the corresponding bits in the hardware. As such, this method (when eventually followed by a call to `“vmm_ral_reg::update()”`) is a zero-time functional replacement for the `“vmm_ral_field::write()”` method. For example, the mirrored value of a read-only field is not modified by this method, and the mirrored value of a write-once field can only be set if the field has not yet been written to using a physical (for example, front-door) write operation.

To modify the mirrored value to a specific value, regardless of the access mode, and thus use the RAL mirror as a scoreboard for the register values in the DUT, use the "vmm_ral_field::predict()" method.

Example

Example B-121

```
$write("Setting the mirrored value to the field to specified
value");
ral_model.fld.set(8'hff);
```

vmm_ral_field::predict()

Force the mirror value of the field.

SystemVerilog

```
virtual function bit predict(bit [63:0] value)
```

OpenVera

```
virtual function bit predict(bit [63:0] value)
```

Description

Force the mirror value of the field to the specified value. Does not actually force the value of the field in the design, only the value mirrored in its corresponding descriptor in the RAL model. Use the `“vmm_ral_reg::update()”` method to update the actual register with the mirrored value or the `“vmm_ral_field::write()”` method to set the actual field and its mirrored value.

The final value in the mirror is the specified value, regardless of the access mode. For example, the mirrored value of a read-only field is modified by this method, and the mirrored value of a read-update field can be updated to any value predicted to correspond to the value in the corresponding physical bits in the design.

Returns FALSE if this method is called while the register containing the field is being read or written, therefore, rendering the prediction unreliable. Returns TRUE otherwise.

Example

Example B-122

```
...
this.ral_model.my_reg.my_field.predict(data);
`vmm_note(log,$psprintf("Forced Value ==> %0h",data));
`vmm_note(log,$psprintf("Previous Value ==> %h",
          this.ral_model.my_reg.my_field.get()));

this.ral_model.my_reg.write(status,data);
`vmm_note(log,$psprintf("Written Value ==> %h",
          this.ral_model.my_reg.my_field.get()));
...
```

vmm_ral_field::get()

Returns the mirror value of the field.

SystemVerilog

```
virtual function bit [63:0] get()
```

OpenVera

```
virtual function bit [63:0] get()
```

Description

Returns the mirror value of the field. Does not actually read the value of the field in the design, only the value mirrored in its corresponding descriptor in the RAL model.

The mirrored value of a write-only field is the value that was set or written and assumed to be stored in the bits implementing the field. Even though a physical read operation of a write-only field returns zeroes, this method returns the assumed content of the field.

Use the “`vmm_ral_field::read()`” method to read the actual field and update its mirrored value.

Example

Example B-123

```
bit [63:0] data;
data = ral_model.block_name.reg1.field_1.get();
```

vmm_ral_field::reset()

Resets the mirror values in the field.

SystemVerilog

```
virtual function void reset(vmm_ral::reset_e kind =
vmm_ral::HARD)
```

OpenVera

```
virtual task reset(vmm_ral::reset_e kind = vmm_ral::HARD)
```

Description

Sets the mirror value of the field to the specified reset value. Does not actually reset the value of the field in the design, only the value mirrored in the descriptor in the RAL model.

The value of a write-once (`vmm_ral::W1`) field can be subsequently modified each time a hard reset is applied.

Example

Example B-124

```
ral_model.block_name.reg1.field_1.reset();
```

vmm_ral_field::set_reset()

Modify the reset value.

SystemVerilog

```
virtual function logic [63:0] set_reset(
     logic [63:0]      value,
     vmm_ral::reset_e kind = vmm_ral::HARD)
```

OpenVera

```
virtual function bit [63:0] set_reset(
     bit [63:0]       value,
     vmm_ral::reset_e kind = vmm_ral::HARD)
```

Description

Modify the reset value of the field to the specified value and return the previously-defined reset value. A soft-reset value specified as all X's indicates no soft-reset value.

<u>WARNING</u>! Using this method will modify the behavior of the RAL model from the behavior specified in the original register specification.

Example

Example B-125

```
   ...
   bit [63:0] usr_rst;
   ...
   this.ral_model.my_reg.my_field.reset();
   `vmm_note(log,$psprintf("Default Field Reset Value ==> 
%0h",
             this.ral_model.my_reg.my_field.get()));
   this.ral_model.my_reg.my_field.set_reset(usr_rst);
   this.ral_model.my_reg.my_field.reset();
   `vmm_note(log,$psprintf("User-defined Field Reset Value
==> %0h",
              this.ral_model.my_reg.my_field.get()));
   ...
```

vmm_ral_field::needs_update()

Queries if the mirrored value for this field has been set.

SystemVerilog

```
virtual function bit needs_update()
```

OpenVera

```
virtual function bit needs_update()
```

Description

If the mirror value has been modified in the RAL model without actually updating the actual register, the mirror and state of the registers are outdated. This method returns TRUE if the state of the field needs to be updated to match the mirrored values (or vice-versa).

The mirror value or actual content of the field are not modified. See "`vmm_ral_reg::update()`" or "`vmm_ral_reg::mirror()`".

Example

Example B-126

```
  bit update;
  vmm_rw::status_e status;
  update =
ral_model.block_name.reg1.field_1.needs_update();
```

vmm_ral_field::read()

Reads a field value from the design.

SystemVerilog

```
virtual task read(
   output vmm_rw::status_e status,
   output bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e read_t(
   var bit [63:0]   value,
   vmm_ral::path_e  path = vmm_ral::DEFAULT,
   string           domain = "",
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Reads the current value of the field from the design using the specified access path. If a back-door access path is used, the effect of reading the field through a physical access is mimicked. For example, a write-only field will return zeroes.

If the field is located in a register shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data/rvm_data` class properties in the `“vmm_rw_access”` transaction descriptors that are necessary to execute this read operation. This allows the physical and back-door read accesses to be traced back to the higher-level transaction that caused the access to occur.

The mirrored value of the field, and all other fields located in the same register, is updated with the value read from the design. The mirror value of write-only fields are never updated during a read operation.

Example

Example B-127

```
...
fork
  begin
    wait(written);
    this.ral_model.my_reg.my_field.read(status,rd_data);
    `vmm_note(log,$psprintf("Read Value:: %0h with status
%0s", rd_data,status.name));
      written = 1'b0;
    end
join_none
   ...
```

vmm_ral_field::write()

Sets a field value in the design.

SystemVerilog

```
virtual task write(
   output vmm_rw::status_e status,
   input  bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e write_t(
   bit [63:0]       value,
   vmm_ral::path_e  path = vmm_ral::DEFAULT,
   string           domain = "",
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Writes the specified field value in the design using the specified access path. If a back-door access path is used, the effect of writing the field through a physical access is mimicked. For example, a read-only field will not be written.

If the field is located in a register shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data/rvm_data` class properties in the “`vmm_rw_access`” transaction descriptors that are necessary to execute this write operation. This allows the physical and back-door write accesses to be traced back to the higher-level transaction that caused the access to occur.

If the register, where this field is physically located, contains other fields, the following values are used for the other bits in the register, based on each field access mode:

```
vmm_ral::RW
vmm_ral::RO
vmm_ral::WO
vmm_ral::W1
vmm_ral::RU
vmm_ral::OTHER
vmm_ral::USERn
```

The value for the other field is written using the mirrored content of the field.

```
vmm_ral::RC
vmm_ral::W1C
vmm_ral::A0
```

The value for the other field is written as all zeroes.

```
vmm_ral::A1
```

The value for the other field is written as all ones.

If the register containing this field contains other write-once fields, these other fields are written at the same time as this field. If a physical (for example, front-door) access is used, it is not possible to modify their content without first resetting the model. It may be preferable to use a `set()/update()` approach. Once write-once fields are written through a physical (for example, front-door) access, their value can no longer be modified.

The mirrored value of the fields are updated based on the written value and the specified behavior of the field contents after a write operation.

Example

Example B-128

```
class my_ral_reg extends vmm_ral_reg;
   rand vmm_ral_field my_field_1;
   rand vmm_ral_field my_field_2;

   function new(...);
      super.new(...);
     //Here vmm_ral::A1 is value for the access_e enum. You
can change it
      //to get the required value for perticular field.
      this.my_field_1 = new(this, "my_field_1", 1,
vmm_ral::RW, ...);
      this.my_field_1 = new(this, "my_field_1", 1,
vmm_ral::A1, ...);
      ...
   endfunction
   ...
endclass
...
fork
  begin
```

```
this.ral_model.my_reg.my_field_1.write(status,data,vmm_ral
::BACKDOOR);
   `vmm_note(log,$psprintf(" Written Value:: %0h with status
%0s",
              data,status.name);
    written = 1'b1;
  end
  begin
    wait(written);
    this.ral_model.my_reg.my_field_1.read(status,rd_data);
    `vmm_note(log,$psprintf("my_field_1 Value:: %0h with
status %0s",
              rd_data,status.name);

    this.ral_model.my_reg.my_field_2.read(status,rd_data);
    `vmm_note(log,$psprintf("my_field_2 Value:: %0h with
status %0s",
              rd_data,status.name);
    written = 1'b0;
  end
join_none
...
```

vmm_ral_field::peek()

Peek a field value from the design.

SystemVerilog

```
virtual task peek(
   output vmm_rw::status_e status,
   output bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e peek_t(
   var bit [63:0]   value,
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Peek the current value of the field from the design using a back-door access. The value of the field in the design is not modified, regardless of the access mode.

The optional value of the `data_id`, `scenario_id` and `stream_id` arguments are passed to the back-door access method. This allows the physical and back-door read accesses to be traced back to the higher-level transaction which caused the access to occur.

The mirrored value of the field, and all other fields located in the same register, is updated with the value peeked from the design.

Example

Example B-129

```
vmm_ral_block  b;
bit [63:0]  value;
begin
b.register.fld.peek(status,value);
 `vmm_note(log, $psprintf("b.register.fld   %h", value));
end
```

vmm_ral_field::poke()

Poke a field value in the design.

SystemVerilog

```
virtual task poke(
   output vmm_rw::status_e status,
   input  bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e poke_t(
   bit [63:0]       value,
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Deposit the specified field value in the design using a back-door access. The value of the field is updated, regardless of the access mode.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door write accesses to be traced back to the higher-level transaction that caused the access to occur.

If the register, where this field is physically located, contains other fields, the current value of the other fields are peeked first then poked back in.

Example

Example B-130

```
vmm_ral_block  b;
b.register.fld.poke(status, 8'hAB);
if (status != vmm_rw::IS_OK)  begin
`vmm_error(log, $psprintf("Update status was  %s",
status.name()));
end
```

vmm_ral_field::mirror()

Updates the mirrored value of this field to match the design.

SystemVerilog

```
virtual task mirror(
   output vmm_rw::status_e status,
   input vmm_ral::check_e  check = vmm_ral::QUIET,
   input vmm_ral::path_e   path = vmm_ral::DEFAULT,
   input string           domain = "")
```

OpenVera

```
virtual function vmm_rw::status_e mirror_t(
   vmm_ral::check_e  check = vmm_ral::QUIET,
   vmm_ral::path_e   path  = vmm_ral::DEFAULT,
   string           domain = "")
```

Description

Updates the content of the field mirror value for all the fields in the same register to match the current values in the design. The mirroring can be performed using the physical interfaces (frontdoor) or “`vmm_ral_field::peek()`” (backdoor). If the `check` argument is specified as `vmm_ral::VERB`, an error message is issued if the current mirrored value of the entire register does not match the actual value in the design.

The content of a write-only field is mirrored and optionally checked only if a `vmm_ral::BACKDOOR` access path is used to read the register containing the field.

If the field is located in a register shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

Example

Example B-131

```
   ...
   this.ral_model.my_reg.my_field.set(data);
   fork
     begin
       this.ral_model.my_reg.my_field.mirror(status);
      `vmm_note(log,$psprintf(" Mirrored Value:: %0h with
status %0s",
                 data,status.name);
       mirrored = 1'b1;
     end
   join_none
   ...
```

vmm_ral_field::append_callback()

Appends a callback extension instance.

SystemVerilog

```
function void append_callback(vmm_ral_field callbacks cbs)
```

OpenVera

```
task append_callback(vmm_ral_field_callbacks cbs)
```

Description

Appends the specified callback extension instance to the registered callbacks for this field descriptor. Callbacks are invoked in the order of registration.

Note that field callback methods will be invoked before their corresponding "vmm_ral_reg" callback methods.

Example

Example B-132

```
program test;
...

class my_ral_field_callbacks extends
vmm_ral_field_callbacks;
...
endclass

my_ral_field_callbacks cb;
...
initial
   begin
```

```
        ...
        cb = new();
        env.ral_model.my_field.append_callback(cb);
        ...
    end
...
endprogram
```

vmm_ral_field::prepend_callback()

Prepends a callback extension instance.

SystemVerilog

```
function void prepend_callback(vmm_ral_field_callbacks cbs)
```

OpenVera

```
task prepend_callback(vmm_ral_field_callbacks cbs)
```

Description

Prepends the specified callback extension instance to the registered callbacks for this field descriptor. Callbacks are invoked in the reverse order of registration.

Note that field callback methods will be invoked before their corresponding "`vmm_ral_reg`" callback methods.

Example

Example B-133

```
program test;
...

class my_ral_field_callbacks extends
vmm_ral_field_callbacks;
...
endclass

my_ral_field_callbacks cb;
...
initial
   begin
```

```
      ...
      cb = new();
      env.ral_model.my_field.prepend_callback(cb);
      ...
   end
...
endprogram
```

vmm_ral_field::unregister_callback()

Removes a callback extension instance.

SystemVerilog

```
function void unregister_callback(vmm_ral_field_callbacks
cbs)
```

OpenVera

```
task unregister_callback(vmm_ral_field_callbacks cbs)
```

Description

Removes the specified callback extension instance from the registered callbacks for this field descriptor. A warning message is issued if the callback instance has not been previously registered.

Example

Example B-134

```
program test;
...

class my_ral_field_callbacks extends
vmm_ral_field_callbacks;
...
endclass

my_ral_field_callbacks cb;
...
initial
   begin
      ...
      cb = new();
      env.ral_model.my_field.append_callback(cb);
      //Can't call second time for same instance.
     env.ral_model.my_field.append_callback(cb);  //Wrong
Way
      ...
      env.ral_model.my_field.unregister_callback(cb);
      ...
   end
...
endprogram
```

vmm_ral_field_callbacks

Field descriptors.

Summary

vmm_ral_field_callbacks::pre_write()

OOP callback invoked before writing a field.

SystemVerilog

```
virtual task pre_write(vmm_ral_field       field,
                       ref bit [63:0]      wdat,
                       ref vmm_ral::path_e path,
                      ref string          domain)
```

OpenVera

```
virtual task pre_write_t(vmm_ral_field       field,
                         var bit [63:0]      wdat,
                         var vmm_ral::path_e path,
                        var string          domain)
```

Description

This callback method is invoked before a value is written to a field in the DUT. The written value, if modified, modifies the actual value that will be written. The path and domain used to write to the field can also be modified.

This callback method is only invoked when the "vmm_ral_field::write()" or the "vmm_ral_reg::write()" method is used to write to the field inside the DUT. This callback method is not invoked when only the mirrored value is written to using the "vmm_ral_field::set()" method.

Because writing a field causes the register to be written, and therefore all of the other fields it contains to also be written, all registered "vmm_ral_field_callbacks::pre_write()"

methods with the fields contained in the register will also be invoked, then all registered “`vmm_ral_reg_callbacks::pre_write()`” methods with the register containing the field will also be invoked.

Example

Example B-135

```
TBD
```

vmm_ral_field_callbacks::post_write()

OOP callback invoked after writing a field.

SystemVerilog

```
virtual task post_write(vmm_ral_field         field,
                        bit [63:0]            wdat,
                        vmm_ral::path_e       path,
                        string                domain
                       ref vmm_rw::status_e status)
```

OpenVera

```
virtual task post_write_t(vmm_ral_field         field,
                          bit [63:0]            wdat,
                          vmm_ral::path_e       path,
                          string                domain
                         var vmm_rw::status_e status)
```

Description

This callback method is invoked after a value is written to a field in the DUT. The `wdat` value is the final mirrored value in the register as reported by the “`vmm_ral_field::get()`” method.

This callback method is only invoked when the “`vmm_ral_field::write()`” or the “`vmm_ral_reg::write()`” method is used to write to the field inside the DUT. This callback method is not invoked when only the mirrored value is written to using the “`vmm_ral_field::set()`” method.

Because writing a field causes the register to be written and, therefore, all of the other fields it contains to also be written, all registered “`vmm_ral_field_callbacks::post_write()`”

methods with the fields contained in the register will also be invoked. At this point, all registered `“vmm_ral_reg_callbacks::post_write()”` methods with the register containing the field will also be invoked.

Example

Example B-136

```
TBD
```

vmm_ral_field_callbacks::pre_read()

OOP callback invoked before reading a field.

SystemVerilog

```
virtual task pre_read(vmm_ral_field       field,
                      ref vmm_ral::path_e path,
                     ref string         domain)
```

OpenVera

```
virtual task pre_read_t(vmm_ral_field       field,
                        var vmm_ral::path_e path,
                       var string         domain)
```

Description

This callback method is invoked before a value is read from a field in the DUT. The path and domain used to read from the field can be modified.

This callback method is only invoked when the “`vmm_ral_field::read()`” or the “`vmm_ral_reg::read()`” method is used to read from the field inside the DUT. This callback method is not invoked when only the mirrored value is read using the “`vmm_ral_field::get()`” method.

Because reading a field causes the register to be read and, therefore, all of the other fields it contains to also be read, all registered “`vmm_ral_field_callbacks::pre_read()`” methods with the fields contained in the register will also be invoked. At this point, all registered “`vmm_ral_reg_callbacks::pre_read()`” methods with the register containing the field will also be invoked.

Example

Example B-137

```
TBD
```

vmm_ral_field_callbacks::post_read()

OOP callback invoked after reading a field.

SystemVerilog

```
virtual task post_read(input vmm_ral_field     field,
                       ref   bit [63:0]        rdat,
                       input vmm_ral::path_e   path,
                       input string            domain
                      ref   vmm_rw::status_e status)
```

OpenVera

```
virtual task post_read_t(vmm_ral_field         field,
                         var bit [63:0]        rdat,
                         vmm_ral::path_e       path,
                         string                domain
                        var vmm_rw::status_e status)
```

Description

This callback method is invoked after a value is read from a field in the DUT. The `rdat` and `status` values are the values that are ultimately returned by the “`vmm_ral_field::read()`” method and can be modified.

This callback method is only invoked when the “`vmm_ral_field::read()`” or the “`vmm_ral_reg::read()`” method is used to read from the field inside the DUT. This callback method is not invoked when only the mirrored value is read from using the “`vmm_ral_field::get()`” method.

Because reading a field causes the register to be read and, therefore, all of the other fields it contains to also be read, all registered “`vmm_ral_field_callbacks::post_read()`”

methods with the fields contained in the register will also be invoked. At this point, all registered `“vmm_ral_reg_callbacks::post_read()”` methods with the register containing the field will also be invoked.

Example

Example B-138

```
TBD
```

vmm_ral_mem

Memory descriptors.

Summary

vmm_ral_mem::log

Message service interface.

SystemVerilog

```
vmm_log log
```

OpenVera

```
rvm_log log
```

Description

Message service interface instance for the memory descriptor. A single message service interface instance is shared by all memory abstraction class instances.

Example

Example B-139

```
class my_mem extends vmm_ral_mem;
   vmm_log log;
   ...
   function new(...);
      super.new(...);
      log = new("my_ral_mem","log");
      `vmm_note(log,"Log instance is successfully created 
for vmm_ral_mem.");
      ...
   endfunction
   ...
endclass
```

vmm_ral_mem::mam

Default allocation manager.

SystemVerilog

```
const vmm_mam mam
```

Description

Default memory allocation manager that can be used to dynamically allocate or reserve regions of consecutive locations. All regions allocated by this allocation manager instance are associated with this memory. By default, the entire memory address space is available for allocation.

This allocation manager must not be replaced with another allocation manager. Instead, it should be reconfigured to match the application requirements.

See the *VMM Environment Composition User Guide* (**subenv_user_guide.pdf**) for more information and detailed documentation on the memory allocation manager.

Example

Example B-140

```
class my_mem extends vmm_ral_mem;
   ...
   if(this.mam == null)
     `vmm_error(log,"Failed to create vmm_mam class instant
in vmm_ral_mem.");
   else
      `vmm_note(log,"vmm_mam class instant is created
successfully.");
```

```
     ...
endclass
```

vmm_ral_mem::get_name()

Returns the name of the memory.

SystemVerilog

```
virtual function string get_name()
```

OpenVera

```
virtual function string get_name()
```

Description

Returns the name of the memory corresponding to the instance of the descriptor.

Example

Example B-141

```
  vmm_ral_mem mems[];
  ral_model.block_name.get_memories(mems);
  foreach (mems[i]) begin
      `vmm_note(log, $psprintf("Memory Name: %s\n",
mems[i].get_name()));
  end
```

vmm_ral_mem::get_fullname()

Returns the fully-qualified name of the memory.

SystemVerilog

```
virtual function string get_fullname()
```

OpenVera

```
virtual function string get_fullname()
```

Description

Returns the hierarchical name of the memory corresponding to the instance of the descriptor. The name of the top-level block or system is not included in the fully-qualified name as it is implicit for every RAL model.

Example

Example B-142

```
  vmm_ral_mem mems[];
  ral_model.block_name.get_memories(mems);
  foreach (mems[i]) begin
      `vmm_note(log, $psprintf("Memory Name: %s\n",
mems[i].get_fullname()));
  end
```

vmm_ral_mem::get_domains()

Returns the name of the domains sharing this memory.

SystemVerilog

```
function void get_domains(ref string names[])
```

OpenVera

```
task get_domains(var string names[*])
```

Description

Fills the specified dynamic array with the names of all the block-level domains that can access this memory. The order of the domain names is not specified.

Example

Example B-143

```
string domains[];
ral_model.block_name.mem1.get_domains(domains)
foreach (domains[i]) begin
    $display("Domain Name: %s\n", domains[i]);
end
```

vmm_ral_mem::get_block()

Returns the block that instantiates this memory.

SystemVerilog

```
virtual function vmm_ral_block get_block()
```

OpenVera

```
virtual function ram_ral_block get_block()
```

Description

Returns a reference to the descriptor of the block that includes the memory corresponding to the descriptor instance.

Example

Example B-144

```
vmm_ral_block blk;
blk = ral_model.block_name.mem1.get_block();
blk.display();
```

vmm_ral_mem::get_offset_in_block()

Returns the address of a memory location within the block address space.

SystemVerilog

```
virtual function bit [63:0] get_offset_in_block(
   input bit [63:0] mem_addr = 0,
   input string    domain = "")
```

OpenVera

```
virtual function bit [63:0] get_offset_in_block(
   bit [63:0] mem_addr = 0,
   string     domain = "")
```

Description

Returns the address of the specified location in the memory within the overall address space of the block that instantiates it. By default, returns the base address of the memory. If the memory is shared between multiple physical interfaces, a domain must be specified.

If the memory location is wider than the physical interface of the block, the lowest address value is returned.

Example

Example B-145

```
  bit [`VMM_RAL_ADDR_WIDTH-1:0] addr;
  addr = ral_model.block_name.mem1.get_offset_in_block();
  addr = 
ral_model.block_name_shared.mem2.get_offset_in_block("d1")
; //multiple domain block
```

vmm_ral_mem::get_address_in_system()

Returns the address of a memory location within the design address space.

SystemVerilog

```
virtual function bit [63:0] get_address_in_system(
   input bit [63:0] mem_addr = 0,
   input string    domain = "")
```

OpenVera

```
virtual function bit [63:0] get_address_in_system(
   bit [63:0] mem_addr = 0,
   string     domain = "")
```

Description

Returns the address of the specified location in the memory within the overall address space of the design. By default, returns the base address of the memory. If the memory is shared between multiple physical interfaces, a domain must be specified.

If the memory location is wider than the physical interface used to access it, the lowest address value is returned.

Example

Example B-146

```
  bit [`VMM_RAL_ADDR_WIDTH-1:0] addr;
 addr = ral_model.block_name.mem1.get_address_in_system();
  addr =
ral_model.block_name_shared.mem2.get_address_in_system("d1
"); //multiple domain block
```

vmm_ral_mem::get_size()

Returns the number of unique locations in this memory.

SystemVerilog

```
virtual function int unsigned get_size()
```

OpenVera

```
virtual function integer get_size()
```

Description

Returns the number of unique memory locations in this memory.

Example

Example B-147

```
int size;
size = ral_model.block_name.mem1.get_size();
```

vmm_ral_mem::get_n_bits()

Returns the number of bits in each memory location.

SystemVerilog

```
virtual function int unsigned get_n_bits()
```

OpenVera

```
virtual function integer get_n_bits()
```

Description

Returns the width, in number of bits, of each memory location in the memory.

Example

Example B-148

```
int width;
width = ral_model.block_name.mem1.get_n_bits();
```

vmm_ral_mem::get_n_bytes()

Returns the number of bits in each memory location.

SystemVerilog

```
virtual function int unsigned get_n_bytes()
```

Description

Returns the width, in number of bits, of each memory location within the memory. If the number of bits is not a multiple of eight, some bits in the most significant byte are not implemented.

Example

Example B-149

```
int size;
size = ral_model.block_name.mem1.get_n_bytes();
```

vmm_ral_mem::get_access()

Returns the access mode of the memory.

SystemVerilog

```
virtual function vmm_ral::access_e get_access(string domain
= "")
```

OpenVera

```
virtual function vmm_ral::access_e get_access(strong domain
= "")
```

Description

Returns the specification of the behavior of the memory when written and read. If the memory is shared in more than one domain, a domain name must be specified.

If access restrictions are present when accessing a memory through the specified domain, the access mode returned takes the access restrictions into account. For example, a read-write memory accessed through a domain with read-only restrictions would return `vmm_ral::RO`.

Example

Example B-150

```
class my_mem extends vmm_ral_mem;
   ...
   vmm_ral::access_e access;
   access = this.get_access();
   `vmm_log(log,$psprintf("Memory Access is
%0s",access.name);
```

```
    ...
endclass
```

vmm_ral_mem::get_rights()

Returns the access rights of the memory.

SystemVerilog

```
virtual function vmm_ral::access_e get_rights(string domain
= "")
```

OpenVera

```
virtual function vmm_ral::access_e get_rights(string domain
= "")
```

Description

Returns the access rights of a memory. Returns `vmm_ral::RW`, `vmm_ral::RO`, or `vmm_ral::WO`. The access rights of a memory is always `vmm_ral::RW`, unless it is a shared memory with access restrictions in a particular domain.

If the memory is shared in more than one domain, a domain name must be specified. If the memory is not shared in the specified domain, an error message is issued and `vmm_ral::RW` is returned.

Example

Example B-151

```
  vmm_ral::access_e rights;
  rights = ral_model.block_name.mem1.get_rights();
  rights =
ral_model.block_name_shared.mem2.get_rights("d1"); //
rights of shared memory mem1 in domain "d1"
```

vmm_ral_mem::get_virtual_fields()

Returns all virtual fields implemented in this memory.

SystemVerilog

```
virtual function void get_virtual_fields(
   ref vmm_ral_vfield fields[])
```

Description

Fills the specified dynamic array with the descriptor for all of the virtual fields implemented in this memory. The order in which the fields are located in the array is not specified.

Example

Example B-152

```
vmm_ral_mem::get_virtual_fields() [page 351]

  vmm_ral_vfield vfields[];
  ral_model.memory_name.get_virtual_fields(vfields);
  foreach (vfields[i]) begin
     vfields[i].display();
  end
```

vmm_ral_mem::get_virtual_field_by_name()

Returns the virtual field with the specified name in this memory.

SystemVerilog

```
virtual function vmm_ral_vfield get_virtual_field_by_name(
   string name)
```

Description

Finds a virtual field with the specified name implemented in this memory and returns its descriptor. If no fields are found, returns null.

Field name uniqueness is guaranteed only within virtual registers. Therefore, if used on a memory implementing more than one field having the same name, this method returns the first field found.

Example

Example B-153

```
vmm_ral_mem::get_virtual_field_by_name() [page 352]

  vmm_ral_vfield vfield;
  vfield =
ral_model.memory_name.get_virtual_field_by_name("virtual_f
ield_name");
  if (vfield == null) `vmm_error(log, "specified field
doesn't exists");
```

vmm_ral_mem::get_virtual_registers()

Returns all virtual registers in this memory.

SystemVerilog

```
virtual function void get_virtual_registers(
   ref vmm_ral_vreg regs[])
```

Description

Fills the specified dynamic array with the descriptor for all of the virtual registers implemented in this memory. The order in which the registers are located in the array is not specified.

Example

Example B-154

```
vmm_ral_mem::get_virtual_registers() [page 353]

  vmm_ral_vreg vregs[];
  ral_model.memory_name.get_virtual_registers(vregs);
  foreach (vregs[i]) begin
     vregs[i].display();
  end
```

vmm_ral_mem::get_vreg_by_name()

Returns the virtual register with the specified name in this memory.

SystemVerilog

```
virtual function vmm_ral_vreg get_vreg_by_name(
   string name)
```

Description

Finds a virtual register with the specified name implemented in this memory and returns its descriptor. If no registers are found, returns null.

Example

Example B-155

```
vmm_ral_mem::get_vreg_by_name() [page 354]

  vmm_ral_vreg vreg;
  vreg = 
ral_model.memory_name.get_vreg_by_name("virtual_reg_name")
;
  if (vreg == null) `vmm_error(log, "specified virtual 
register doesn't exists");
```

vmm_ral_mem::display()

Displays a description of the memory to stdout.

SystemVerilog

```
virtual function void display(string prefix = "",
                              string domain = "")
```

OpenVera

```
virtual task display(string prefix = "",
                     string domain = "")
```

Description

Displays the image created by the "vmm_ral_mem::psdisplay()" method to the standard output.

Example

Example B-156

```
ral_model.block_name.mem1.display();
```

vmm_ral_mem::psdisplay()

Creates a human-readable description of the memory.

SystemVerilog

```
virtual function string psdisplay(string prefix = "",
                                  string domain = "")
```

OpenVera

```
virtual function string psdisplay(string prefix = "",
                                  string domain = "")
```

Description

Creates a human-readable description of the memory. Each line of the description is prefixed with the specified prefix.

If a domain is specified, the base address of the memory within that domain is used.

The content of the memory is not displayed.

Example

Example B-157

```
 `vmm_note(log, $psprintf("Register description = %s\n",
ral_model.block_name.mem1.psdisplay()));
```

vmm_ral_mem::set_frontdoor()

Defines a user-defined access mechanism for this memory.

SystemVerilog

```
function void set_frontdoor(vmm_ral_mem_frontdoor ftdr,
                           string domain = "")
```

OpenVera

```
task set_frontdoor(vmm_ral_mem_frontdoor ftdr,
                  string               domain = "")
```

Description

By default, memories are mapped linearly into the address space of the block that instantiates them. If memories are accessed using a different mechanism, a user-defined access mechanism must be defined and associated with the corresponding memory abstraction class.

See `“User-Defined Memory Access” on page 116` for an example.

vmm_ral_mem::get_frontdoor()

Returns the user-defined access mechanism for this memory.

SystemVerilog

```
function vmm_ral_mem_frontdoor get_frontdoor(
   string domain = "")
```

OpenVera

```
function vmm_ral_mem_frontdoor get_frontdoor(
   string domain = "")
```

Description

Returns the current user-defined mechanism for this memory for the specified domain. If `null`, no user-defined mechanism has been defined. A user-defined mechanism is defined by using the *“`vmm_ral_mem::set_frontdoor()`”* method.

Example

Example B-158

```
class my_mem extends vmm_ral_mem;
   ...
   if(this.get_frontdoor == null);
     `vmm_note(log,"set_frontdoor method is not called for 
this memory.");
   ...
endclass
```

vmm_ral_mem::set_backdoor()

Defines the back-door access mechanism for this memory.

SystemVerilog

```
function void set_backdoor(vmm_ral_mem_backdoor bkdr)
```

OpenVera

```
task set_backdoor(vmm_ral_mem_backdoor bkdr)
```

Description

Memories implemented using SystemVerilog unpacked arrays can be accessed using a hierarchical path. This direct back-door access is automatically generated if the necessary `hdl_path` properties are specified in the RALF description.

However, memories can be modeled using other methods, such as using DesignWare models. These memory models come with their own back-door mechanism. This method is used to associate a back-door access mechanism with a memory descriptor to enable back-door access.

A class extension implementing the back-door mechanism for DesignWare memory models has already been defined and is included in RAL.

Example

Example B-159

```
class my_block extends vmm_ral_block;
   ...
   rand my_mem mem;
   function new(...);
      super.new(...);
      mem.new(...);
      begin
        ral_mem_bkdr bkdr = new;
        this.mem.set_backdoor(bkdr);
      end
   endfunction
   ...
endclass
```

vmm_ral_mem::get_backdoor()

Returns the back-door access mechanism for this memory.

SystemVerilog

```
function vmm_ral_mem_backdoor get_backdoor()
```

OpenVera

```
function vmm_ral_mem_backdoor get_backdoor()
```

Description

Returns the current back-door mechanism for this memory. If `null`, no back-door mechanism has been defined. A back-door mechanism can be automatically defined by using the `hdl_path` properties in the RALF definition or user-defined using the `“vmm_ral_mem::set_backdoor()”` method.

Example

Example B-160

```
my_mem.write(0, 16’hABCD,
             (my_mem.get_backdoor() == null) ?
                vmm_rw::BFM : vmm_rw::BACKDOOR);
```

vmm_ral_mem::init()

Initializes the memory.

SystemVerilog

```
virtual task init(
   output bit                 is_ok,
   input  vmm_ral_mem::init_e pattern,
   input  bit [63:0]         data = 0)
```

OpenVera

```
virtual function bit init_t(
   vmm_ral_mem::init_e pattern,
   bit [63:0]         data = 0)
```

Description

Not yet implemented.

Initializes the memory and its mirrored content with the specified pattern. Requires that a back-door access to the memory be available. See “`vmm_ral_mem::init_e`” for a description of the available initialization patterns.

Returns TRUE if the initialization was successful. Returns FALSE otherwise.

Example

Example B-161

```
TBD
```

vmm_ral_mem::init_e

Symbolic values identifying the pattern with which to initialize a memory.

SystemVerilog

```
vmm_ral_mem::UNKNOWNS
vmm_ral_mem::ZEROES
vmm_ral_mem::ONES
vmm_ral_mem::VALUE
vmm_ral_mem::INCR
vmm_ral_mem::DECR
```

OpenVera

```
vmm_ral_mem::UNKNOWNS
vmm_ral_mem::ZEROES
vmm_ral_mem::ONES
vmm_ral_mem::VALUE
vmm_ral_mem::INCR
vmm_ral_mem::DECR
```

Description

See “`vmm_ral_mem::init()`” for the method that uses these symbolic values. Each symbolic value is used to define a specific memory initialization pattern as follows:

`vmm_ral_mem::UNKNOWNS`

Initializes the memory with all unknowns (’bx).

`vmm_ral_mem::ZEROES`

Initializes the memory with all zeroes (’b0).

`vmm_ral_mem::ONES`

Initializes the memory with all ones (’b1).

`vmm_ral_mem::VALUE`

Initializes the memory with a specified value.

`vmm_ral_mem::INCR`

Initializes the memory, starting with a specified value and increasing it for each location toward higher addresses.

`vmm_ral_mem::DECR`

Initializes the memory, starting with a specified value and decreasing it for each location toward lower addresses.

Example

Example B-162

```
TBD
```

vmm_ral_mem::read()

Reads a memory location from the design.

SystemVerilog

```
virtual task read(
   output vmm_rw::status_e status,
   input  bit [63:0]       mem_addr,
   output bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e read_t(
   bit [63:0]       mem_addr,
   var bit [63:0]   value,
   vmm_ral::path_e  path = vmm_ral::DEFAULT,
   string           domain = "",
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Reads the current value of the memory location from the design using the specified access path. If the memory is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

`data_id`
`scenario_id`
`stream_id`

...are passed to the back-door access method or used to set the corresponding `vmm_data/rvm_data` class properties in the `“vmm_rw_access”` transaction descriptors that are necessary to execute this read operation. This allows the physical and back-door read access to be traced back to the higher-level transaction that caused the access to occur.

Memories are not mirrored. Instead, use the `“vmm_ral_mem::peek()”` method.

Example

Example B-163

```
vmm_rw::status_e status;
bit [`VMM_RAL_DATA_WIDTH-1:0] data;
ral_model.block_name.mem1.read(status,0,data);
```

vmm_ral_mem::write()

Sets a memory location in the design.

SystemVerilog

```
virtual task write(
   output vmm_rw::status_e status,
   input  bit [63:0]       mem_addr,
   input  bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e write_t(
   bit [63:0]       mem_addr,
   bit [63:0]       value,
   vmm_ral::path_e  path = vmm_ral::DEFAULT,
   string           domain = "",
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Writes the specified value at the specified memory location in the design using the specified access path. If the memory is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

data_id
scenario_id
stream_id

...are passed to the back-door access method or used to set the corresponding vmm_data/rvm_data class properties in the "vmm_rw_access" transaction descriptors that are necessary to execute this write operation. This allows the physical and back-door write access to be traced back to the higher-level transaction that caused the access to occur.

Memories are not mirrored. Instead, use the "vmm_ral_mem::poke()" method.

Example

Example B-164

```
vmm_rw::status_e status;
ral_model.block_name.mem1.write(status,0,'hABCD);
```

vmm_ral_mem::burst_read()

Perform a burst-read operation on the memory.

SystemVerilog

```
virtual task burst_read(
   output vmm_rw::status_e  status,
   input  vmm_ral_mem_burst burst,
   output bit [63:0]        value[],
   input  vmm_ral::path_e   path = vmm_ral::DEFAULT,
   input  string            domain = "",
   input  int               data_id = -1,
   input  int               scenario_id = -1,
   input  int              stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e read_t(
   vmm_ral_mem_burst burst,
   var bit [63:0]    value[*],
   vmm_ral::path_e   path = vmm_ral::DEFAULT,
   string            domain = "",
   integer           data_id = -1,
   integer           scenario_id = -1,
   integer          stream_id = -1)
```

Description

Burst-read the current values of the memory locations specified by the burst descriptor. If the memory is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or the `“vmm_ral_access::burst_read()”` method. This allows the physical and back-door read access to be traced back to the higher-level transaction that caused the access to occur.

It is not possible to perform a burst-read operation on a memory instantiated in a block or system with a narrower data path.

Memories are not mirrored. Instead, use the `“vmm_ral_mem::peek()”` method.

Example

Example B-165

```
TBD
```

vmm_ral_mem::burst_write()

Perform a burst-write operation on the memory.

SystemVerilog

```
virtual task burst_write(
   output vmm_rw::status_e  status,
   input  vmm_ral_mem_burst burst,
   input  bit [63:0]        value[],
   input  vmm_ral::path_e   path = vmm_ral::DEFAULT,
   input  string            domain = "",
   input  int               data_id = -1,
   input  int               scenario_id = -1,
   input  int              stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e write_t(
   vmm_ral_mem_burst burst,
   bit [63:0]        value[*],
   vmm_ral::path_e   path = vmm_ral::DEFAULT,
   string            domain = "",
   integer           data_id = -1,
   integer           scenario_id = -1,
   integer          stream_id = -1)
```

Description

Burst-write the specified values in the memory locations specified by burst descriptor. If the memory is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or the "`vmm_ral_access::burst_write()`" method. This allows the physical and back-door write access to be traced back to the higher-level transaction that caused the access to occur.

It is not possible to perform a burst-write operation on a memory instantiated in a block or system with a narrower data path.

Memories are not mirrored. Instead, use the "`vmm_ral_mem::poke()`" method.

Example

Example B-166

```
TBD
```

vmm_ral_mem::peek()

Peek a memory location from the design.

SystemVerilog

```
virtual task peek(
   output vmm_rw::status_e status,
   input  bit [63:0]       mem_addr,
   output bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e peek_t(
   bit [63:0]       mem_addr,
   var bit [63:0]   value,
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Reads the current value of the memory location from the design using a back-door access.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door read access to be traced back to the higher-level transaction that caused the access to occur.

Example

Example B-167

```
vmm_rw::status_e status;
bit [`VMM_RAL_DATA_WIDTH-1:0] data;
ral_model.block_name.mem1.peek(status,0,data);
```

vmm_ral_mem::poke()

Poke a memory location in the design.

SystemVerilog

```
virtual task poke(
   output vmm_rw::status_e status,
   input  bit [63:0]       mem_addr,
   input  bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e poke_t(
   bit [63:0]       mem_addr,
   bit [63:0]       value,
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Deposit the specified value at the specified memory location in the design using a back-door access. Depending on the design model implementation, it may be possible to modify the content of a read-only memory.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door write access to be traced back to the higher-level transaction that caused the access to occur.

Example

Example B-168

```
vmm_rw::status_e status;
ral_model.block_name.mem1.poke(status,0,'hABCD);
```

vmm_ral_mem::readmemh()

Initializes the memory.

SystemVerilog

```
virtual task readmemh(string filename)
```

OpenVera

```
virtual task readmemh_t(string filename)
```

Description

Not yet implemented.

Initializes the content of all memory locations using the values in the specified file. The values are updated using the default access path. See “`vmm_ral_mem::writememh()`” for details.

The format of the file is the same as the one used by the `$readmemh` task.

Example

Example B-169

```
TBD
```

vmm_ral_mem::writememh()

Dumps the value of the memory.

SystemVerilog

```
virtual task writememh(string filename)
```

OpenVera

```
virtual task writememh_t(string filename)
```

Description

Not yet implemented.

Dumps the content of all memory locations to the specified file. The file can then be used as an input for the "`vmm_ral_mem::readmemh()`" method. The values are obtained using the default access path.

The format of the file is the same as the one used by the `$readmemh` task.

Example

Example B-170

```
TBD
```

vmm_ral_mem::append_callback()

Appends a callback extension instance.

SystemVerilog

```
function void append_callback(
   vmm_ral_mem_callbacks cbs)
```

OpenVera

```
task append_callback(
   vmm_ral_mem_callbacks cbs)
```

Description

Appends the specified callback extension instance to the registered callbacks for this memory descriptor. Callbacks are invoked in the order of registration.

Example

Example B-171

```
program test;
...

class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
...
endclass

my_ral_mem_callbacks cb;
...
initial
   begin
      ...
      cb = new();
      env.ral_model.my_mem.append_callback(cb);
      ...
   end
...
endprogram
```

vmm_ral_mem::prepend_callbacks()

Prepends a callback extension instance.

SystemVerilog

```
function void prepend_callbacks(
   vmm_ral_mem_callbacks cbs)
```

OpenVera

```
task prepend_callbacks(
   vmm_ral_mem_callbacks cbs)
```

Description

Prepends the specified callback extension instance to the registered callbacks for this memory descriptor. Callbacks are invoked in the reverse order of registration.

Example

Example B-172

```
program test;
...

class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
...
endclass

my_ral_mem_callbacks cb;
...
initial
   begin
      ...
      cb = new();
      env.ral_model.my_mem.prepend_callback(cb);
```

```
            ...
        end
    ...
    endprogram
```

vmm_ral_mem::unregister_callbacks()

Removes a callback extension instance.

SystemVerilog

```
function void unregister_callbacks(
   vmm_ral_mem_callbacks cbs)
```

OpenVera

```
task unregister_callbacks(
   vmm_ral_mem_callbacks cbs)
```

Description

Removes the specified callback extension instance from the registered callbacks for this memory descriptor. A warning message is issued if the callback instance has not been previously registered.

Example

Example B-173

```
program test;
...

class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
...
endclass

my_ral_mem_callbacks cb;
...
initial
   begin
      ...
      cb = new();
      env.ral_model.my_mem.append_callback(cb);
```

```
         //Can't call second time for same instance.
        env.ral_model.my_mem.append_callback(cb);  //Wrong Way
         ...
         env.ral_model.my_mem.unregister_callback(cb);
         ...
      end
   ...
   endprogram
```

vmm_ral_mem_backdoor

Virtual base class for back-door access to memories. Extensions of this class are automatically generated by RAL if full hierarchical paths to memories are specified through the `hdl_path` properties in RALF descriptions.

Can be extended by users to provide user-specific back-door access to memories that are not implemented in pure SystemVerilog.

Summary

vmm_ral_mem_backdoor::read()

Peek a memory location.

SystemVerilog

```
virtual task read(output vmm_rw::status_e status,
                  input  bit [63:0]       offset,
                  output bit [63:0]       data,
                  input  int              data_id,
                  input  int              scenario_id,
                 input  int              stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e read_t(
   bit [63:0]     offset,
   var bit [63:0] data,
   integer        data_id,
   integer        scenario_id,
   integer        stream_id)
```

Description

Peek the current value at the specified offset in the memory corresponding to the instance of this class. Returns the content of the memory location and an indication of the success of the operation.

The values of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the `“vmm_ral_mem::read()”` method call that requires the back-door access. This allows the read access to be traced back to the higher-level transaction that caused the access to occur.

Ideally, the execution of this method should be non-blocking.

See `“Implementing a Memory Backdoor in SystemVerilog” on page 100` for an example.

vmm_ral_mem_backdoor::write()

Poke a memory location.

SystemVerilog

```
virtual task write(output vmm_rw::status_e status,
                   input  bit [63:0]       offset,
                   input  bit [63:0]       data,
                   input  int              data_id,
                   input  int              scenario_id,
                  input  int             stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e write_t(
   bit [63:0]  offset,
   bit [63:0]  data,
   integer     data_id,
   integer     scenario_id,
   integer     stream_id)
```

Description

Deposit the specified value at the specified offset of the memory corresponding to the instance of this class. Returns an indication of the success of the operation.

The values of the arguments:

`data_id`
`scenario_id`
`stream_id`

...are the values that were optionally specified to the `“vmm_ral_mem::write()”` method call that requires the back-door access. This allows the read access to be traced back to the higher-level transaction that caused the access to occur.

Ideally, the execution of this method should be non-blocking.

If it is not possible to deposit the specified value, for example, a ROM is implemented using constants, a `vmm_rw::ERROR` status should be returned.

See `“Implementing a Memory Backdoor in SystemVerilog” on page 100` for an example.

vmm_ral_mem_burst

Descriptor for memory burst read/write operation.

Summary

vmm_ral_mem_burst::n_beats

Length of the burst.

SystemVerilog

```
rand int n_beats
```

OpenVera

```
rand integer n_beats
```

Description

Specifies the number of beats or transfers in a memory burst operation.

Example

Example B-174

```
class tb_env extends vmm_ral_env;
   ...
   vmm_ral_mem_burst cfg;
   ...
   function new();
      super.new();
      this.cfg = new();
      ...
   endfunction

   virtual function void gen_cfg();
      super.gen_cfg();
      if (!this.cfg.randomize()) begin
         `vmm_fatal(log, "Failed to randomize Memory Burst 
configuration");
      end
      ...
   endfunction
      `vmm_note(log,$psprintf("vmm_ral_mem_burst::n_beats
==> %0d",cfg.n_beats));
   ...
endclass
```

vmm_ral_mem_burst::start_offset

Starting memory offset.

SystemVerilog

```
rand bit [63:0] start_offset
```

OpenVera

```
rand bit [63:0] start_offset
```

Description

Specifies the first offset that is the target of a memory burst read or write operation.

Example

Example B-175

```
 ...
 `vmm_note(log,$psprintf("vmm_ral_mem_burst::start_offset
==> %0d",
            cfg.start_offset));
   ...
```

vmm_ral_mem_burst::incr_offset

Offset increment between beats.

SystemVerilog

```
rand bit [63:0] incr_offset
```

OpenVera

```
rand bit [63:0] incr_offset
```

Description

Memory location offset increment between individual beats of a burst read or write operation. The first beat reads or writes the memory location specified by `“vmm_ral_mem_burst::start_offset”`. The *n*th beat reads or writes the memory location specified by `“vmm_ral_mem_burst::start_offset”` + (n-1) `“vmm_ral_mem_burst::incr_offset”`.

A value of `0` implies a burst operation that repeatedly accesses the same memory location.

Example

Example B-176

```
...
  `vmm_note(log,$psprintf("vmm_ral_mem_burst::incr_offset
==> %0d",
            cfg.incr_offset));
   ...
```

vmm_ral_mem_burst::max_offset

Maximum offset for the burst.

SystemVerilog

```
rand bit [63:0] max_offset
```

OpenVera

```
rand bit [63:0] max_offset
```

Description

Limits the memory location offset range of a burst operation. Causes wrapping if subsequent beats would access a location past the specified offset.

The specified maximum offset must be a valid memory offset.

Example

Example B-177

```
   ...
  `vmm_note(log,$psprintf("vmm_ral_mem_burst::max_offset 
==> %0d",
            cfg.max_offset));
   ...
```

vmm_ral_mem_burst::user_data

Additional burst configuration information.

SystemVerilog

```
vmm_data user_data
```

OpenVera

```
rvm_data user_data
```

Description

Provides a mechanism for passing additional burst configuration information to the "`vmm_rw_xactor::execute_burst()`" method that will ultimately execute the physical burst operation. Any reference to additional user information is passed through transparently through "`vmm_rw_burst::user_data`". The additional information can be recovered by using `$cast()` or `cast_assign()`.

Example

Example B-178

```
  class my_ral_mem_burst extends vmm_ral_mem_burst;
   ...
   vmm_data user_data;
   ...
endclass
...
my_ral_mem_burst br;
...
```

vmm_ral_mem_callbacks

Memory descriptors.

Summary

vmm_ral_mem_callbacks::pre_write()

OOP callback invoked before writing a memory location.

SystemVerilog

```
virtual task pre_write(vmm_ral_mem         mem,
                       ref bit [63:0]      offset,
                       ref bit [63:0]      wdat,
                       ref vmm_ral::path_e path,
                      ref string          domain)
```

OpenVera

```
virtual task pre_write_t(vmm_ral_mem         mem,
                         var bit [63:0]      offset,
                         var bit [63:0]      wdat,
                         var vmm_ral::path_e path,
                        var string          domain)
```

Description

This callback method is invoked before a value is written to a memory location in the DUT. The written value, if modified, modifies the actual value that will be written. The address of the location within the memory, as well as the path and domain used to write to it can also be modified.

Example

Example B-179

```
...
fork
  //It will call pre_write() and post_write() callback
methods if status IS_OK.
  this.ral_model.my_mem.write(status,addr,data);
join_none
```

```
...
program test;
...
  class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
      ...
     virtual task pre_write(vmm_ral_mem                mem,
                  ref bit [`VMM_RAL_ADDR_WIDTH-1:0] offset,
                   ref bit [`VMM_RAL_DATA_WIDTH-1:0] wdat,
                     ref vmm_ral::path_e            path,
                     ref string                     domain
         `vmm_note(log,{"pre_write method is called for
vmm_ral_mem_callbacks",
                   " class"});
      endtask: pre_write
      ...
   endclass
   ...
   env.ral_model.my_mem.append_callback(cb);
   ...
endprogram
```

vmm_ral_mem_callbacks::post_write()

OOP callback invoked after writing a memory location.

SystemVerilog

```
virtual task post_write(vmm_ral_mem           mem,
                        bit [63:0]            offset,
                        bit [63:0]            wdat,
                        vmm_ral::path_e       path,
                        string                domain
                       ref vmm_rw::status_e status)
```

OpenVera

```
virtual task post_write_t(vmm_ral_mem           mem,
                          bit [63:0]            offset,
                          bit [63:0]            wdat,
                          vmm_ral::path_e       path,
                          string                domain
                         var vmm_rw::status_e status)
```

Description

This callback method is invoked after a value is successfully written to a memory location in the DUT. The `wdat` value is the value that was attempted to be written to the memory location, not necessarily the current value of that memory location. If a physical write access did not return `vmm_rw::IS_OK`, this method is not called.

Example

Example B-180

```
program test;
...
  class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
      ...
```

```
        virtual task post_write(vmm_ral_mem                mem,
                          bit [`VMM_RAL_ADDR_WIDTH-1:0] offset,
                            bit [`VMM_RAL_DATA_WIDTH-1:0] wdat,
                              vmm_ral::path_e             path,
                              string                      domain,
                           ref vmm_rw::status_e       status);
             `vmm_note(log,{"post_write method is called for
vmm_ral_mem_callbacks",
                   " class"});
      endtask
      ...
   endclass
   ...
   env.ral_model.my_mem.append_callback(cb);
   ...
endprogram
```

vmm_ral_mem_callbacks::pre_read()

OOP callback invoked before reading a memory location.

SystemVerilog

```
virtual task pre_read(vmm_ral_mem         mem,
                      ref bit [63:0]      offset,
                      ref vmm_ral::path_e path,
                     ref string          domain)
```

OpenVera

```
virtual task pre_read_t(vmm_ral_field       field,
                        var bit [63:0]      offset,
                        var vmm_ral::path_e path,
                       var string          domain)
```

Description

This callback method is invoked before a value is read from a memory location in the DUT. The address of the location in the memory, as well as the path and domain used to read from it, can be modified.

Example

Example B-181

```
...
fork
  begin
    //It will call pre_write() and post_write() callback 
methods if status IS_OK.
    this.ral_model.my_mem.write(status,addr,data);
    write_done = 1'b1;
  end
  begin
    wait(write_done);
   //It will call pre_read() and post_read() callback methods
if status IS_OK.
    this.ral_model.my_mem.read(status,addr,read_data);
    `vmm_note(log,$psprintf({"Read Data ==> %0h and Status
==> %0s",
              read_data,status.name));
    write_done = 1'b0;
  end
join_none
...
program test;
...
  class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
      ...
     virtual task pre_read(vmm_ral_mem                  mem,
                  ref bit [`VMM_RAL_ADDR_WIDTH-1:0] offset,
                      ref vmm_ral::path_e            path,
                        ref string                    dom
         `vmm_note(log,{"pre_read method is called for
vmm_mem_reg_callbacks",
                   " class"});
      endtask
      ...
   endclass
   ...
   env.ral_model.my_mem.append_callback(cb);
   ...
endprogram
```

vmm_ral_mem_callbacks::post_read()

OOP callback invoked after reading a memory location.

SystemVerilog

```
virtual task post_read(input vmm_ral_mem       mem,
                       input bit [63:0]        offset,
                       ref   bit [63:0]        rdat,
                       input vmm_ral::path_e   path,
                       input string            domain,
                      ref   vmm_rw::status_e status)
```

OpenVera

```
virtual task post_read_t(vmm_ral_mem           mem,
                         bit [63:0]            offset,
                         var bit [63:0]        rdat,
                         vmm_ral::path_e       path,
                         string                domain,
                        var vmm_rw::status_e status)
```

Description

This callback method is invoked after a value is successfully read from a memory location in the DUT. The `rdat` and `status` values are the values that are ultimately returned by the "`vmm_ral_mem::read()`" method and can be modified. If a physical read access did not return `vmm_rw::IS_OK`, this method is not called.

Example

Example B-182

```
program test;
...
  class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
```

```
        ...
      virtual task post_read(input vmm_ral_mem             mem,
                    input bit [`VMM_RAL_ADDR_WIDTH-1:0] offset,
                    ref  bit [`VMM_RAL_DATA_WIDTH-1:0] rdat,
                       input vmm_ral::path_e           path,
                       input string                   domain,
                     ref  vmm_rw::status_e           status);
           `vmm_note(log,{"post_read method is called for
vmm_ral_mem_callbacks",
                   " class"});
        endtask
        ...
    endclass
    ...
    env.ral_model.my_mem.append_callback(cb);
    ...
endprogram
```

vmm_ral_mem_callbacks::pre_burst()

OOP callback invoked before a burst operation.

SystemVerilog

```
virtual task pre_burst(vmm_ral_mem         mem,
                       vmm_rw::kind_e      kind,
                       vmm_ral_mem_burst   burst,
                       ref bit [63:0]      wdata[],
                       ref vmm_ral::path_e path,
                      ref string         domain)
```

OpenVera

```
virtual task pre_burst_t(vmm_ral_field       field,
                         vmm_rw::kind_e      kind,
                         vmm_ral_mem_burst   burst,
                         var bit [63:0]      wdata[*],
                         var vmm_ral::path_e path,
                        var string         domain)
```

Description

This callback method is invoked before a burst operation is performed. The description of the burst area, any value to be written (if it is a burst-write operation), as well as the path and domain used to perform the operation, can be modified.

Example

Example B-183

```
TBD
```

vmm_ral_mem_callbacks::post_burst()

OOP callback invoked after a burst operation.

SystemVerilog

```
virtual task post_burst(input vmm_ral_mem       mem,
                        input vmm_rw::kind_e    kind,
                        input vmm_ral_mem_burst burst,
                        ref   bit [63:0]        data[],
                        input vmm_ral::path_e   path,
                        input string            domain,
                       ref   vmm_rw::status_e status)
```

OpenVera

```
virtual task post_burst_t(    vmm_ral_field     field,
                              vmm_rw::kind_e    kind,
                              vmm_ral_mem_burst burst,
                          var bit [63:0]        data[*],
                          var vmm_ral::path_e   path,
                              string            domain,
                         var vmm_rw::status_e status)
```

Description

This callback method is invoked after a burst operation is performed. The values read (if it is a burst-read operation), as well as the status the operation, can be modified.

Example

Example B-184

```
TBD
```

vmm_ral_mem_frontdoor

Virtual base class for user-defined access to memories through a physical interface.

By default, the entire address space of a memory is mapped within the address space of the block instantiating it. Different memory locations are accessed using different addresses. If the memory is physically accessed using a non-linear, non-mapped mechanism, this base class must be user-extended to provide the physical access to the memory. See `“User-Defined Register Access” on page 114` for an example.

Summary

vmm_ral_mem_frontdoor::read()

Performs a physical read to a memory location.

SystemVerilog

```
virtual task read(output vmm_rw::status_e status,
                  input  bit [63:0]       offset,
                  output bit [63:0]       data,
                  input  int              data_id,
                  input  int              scenario_id,
                 input  int              stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e read_t(
   bit [63:0]      offset,
   var bit [63:0]  data,
   integer         data_id,
   integer         scenario_id,
   integer        stream_id)
```

Description

Performs a physical read access to the specified offset in the memory corresponding to the instance of this class. Returns the content of the memory location and an indication of the success of the operation.

The values of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the `“vmm_ral_mem::read()”` method call that requires the front-door access. This allows the read access to be traced back to the higher-level transaction that caused the access to occur.

See `“User-Defined Register Access” on page 114` for an example.

vmm_ral_mem_frontdoor::write()

Performs a physical write to a memory location.

SystemVerilog

```
virtual task write(output vmm_rw::status_e status,
                   input  bit [63:0]       offset,
                   input  bit [63:0]       data,
                   input  int              data_id,
                   input  int              scenario_id,
                  input  int              stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e write_t(
   bit [63:0]  offset,
   bit [63:0]  data,
   integer     data_id,
   integer     scenario_id,
   integer     stream_id)
```

Description

Performs a physical write access to the specified offset of the memory corresponding to the instance of this class. Returns an indication of the success of the operation.

The values of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the `“vmm_ral_mem::write()”` method call that requires the front-door access. This allows the write access to be traced back to the higher-level transaction that caused the access to occur.

See `“User-Defined Register Access” on page 114` for an example.

vmm_ral_mem_frontdoor::burst_read()

Performs a physical burst-read operation to a memory.

SystemVerilog

```
virtual task burst_read(output vmm_rw::status_e  status,
                        input  vmm_ral_mem_burst burst,
                        output bit [63:0]        data[],
                        input  int               data_id,
                      input  int               scenario_id,
                       input  int               stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e burst_read_t(
   vmm_ral_mem_burst burst,
   var bit [63:0]    data[*],
   integer           data_id,
   integer           scenario_id,
   integer          stream_id)
```

Description

Performs a physical burst-read access on the specified locations defined by the burst descriptor in the memory corresponding to the instance of this class. Returns the content of the memory locations and an indication of the success of the operation.

The values of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the `“vmm_ral_mem::burst_read()”` method call that requires the front-door access. This allows the read access to be traced back to the higher-level transaction that caused the access to occur.

Example

Example B-185

```
TBD
```

vmm_ral_mem_frontdoor::burst_write()

Performs a physical burst-write operation to a memory.

SystemVerilog

```
virtual task burst_write(output vmm_rw::status_e  status,
                         input  vmm_ral_mem_burst burst,
                         input  bit [63:0]        data[],
                         input  int               data_id,
                       input  int               scenario_id,
                        input  int               stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e burst_write_t(
   vmm_ral_mem_burst burst,
   bit [63:0]        data[*],
   integer           data_id,
   integer           scenario_id,
   integer          stream_id)
```

Description

Performs a physical burst-write access on the specified locations defined by the burst descriptor in the memory corresponding to the instance of this class. Returns an indication of the success of the operation.

The values of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the `“vmm_ral_mem::burst_write()”` method call that requires the front-door access. This allows the write access to be traced back to the higher-level transaction that caused the access to occur.

See `“User-Defined Register Access” on page 114` for an example.

vmm_ral_reg

Base class for register descriptors.

Summary

vmm_ral_reg::log

Message service interface.

SystemVerilog

```
vmm_log log
```

OpenVera

```
rvm_log log
```

Description

Message service interface instance for the register descriptor. A single message service interface instance is shared by all register abstraction class instances.

Example

Example B-186

```
class my_ral_reg extends vmm_ral_reg;
   ...
   vmm_log log;
   function new(...);
      ...
      log = new("ral_reg","log");
    `vmm_note(log,"vmm_log instance is successfully created
for vmm_ral_reg");
   endfunction
   ...
endclass
```

vmm_ral_reg::get_name()

Returns the name of the register.

SystemVerilog

```
virtual function string get_name()
```

OpenVera

```
virtual function string get_name()
```

Description

Returns the name of the register corresponding to the instance of the descriptor.

Example

Example B-187

```
  vmm_ral_reg regs[];
  ral_model.block_name.get_registers(regs);
  foreach (regs[i]) begin
      `vmm_note(log, $psprintf("Register Name: %s\n",
regs[i].get_name()));
  end
```

vmm_ral_reg::get_fullname()

Returns the fully-qualified name of the register.

SystemVerilog

```
virtual function string get_fullname()
```

OpenVera

```
virtual function string get_fullname()
```

Description

Returns the hierarchical name of the register corresponding to the instance of the descriptor. The name of the top-level block or system is not included in the fully-qualified name as it is implicit for every RAL model.

Example

Example B-188

```
  vmm_ral_reg regs[];
  ral_model.block_name.get_registers(regs);
  foreach (regs[i]) begin
      `vmm_note(log, $psprintf("Register Name: %s\n",
regs[i].get_fullname()));
  end
```

vmm_ral_reg::get_n_domains()

Returns the number of domains sharing this register.

SystemVerilog

```
function int get_n_domains()
```

OpenVera

```
function integer get_n_domains()
```

Description

Returns the number of domains that share this register. You can obtain the name of the domains with the `“vmm_ral_reg::get_domains()”` method.

Example

Example B-189

```
  int domain_number;
  domain_number =
ral_model.block_name.reg1.get_n_domains();
```

vmm_ral_reg::get_domains()

Returns the name of the domains sharing this register.

SystemVerilog

```
function void get_domains(ref string names[])
```

OpenVera

```
task get_domains(var string names[*])
```

Description

Fills the specified dynamic array with the names of all the block-level domains that can access this register. The order of the domain names is not specified.

Example

Example B-190

```
string domains[];
  ral_model.block_name.reg1.get_domains(domains)
  foreach (domains[i]) begin
      $display("Domain Name: %s\n", domains[i]);
  end
```

vmm_ral_reg::get_rights()

Returns the access rights of this register.

SystemVerilog

```
function vmm_ral::access_e get_rights(string domain = "")
```

OpenVera

```
function vmm_ral::access_e get_rights(string domain = "")
```

Description

Returns the access rights of a register. Returns `vmm_ral::RW`, `vmm_ral::RO`, or `vmm_ral::WO`. The access rights of a register is always `vmm_ral::RW`, unless it is a shared register with access restriction in a particular domain.

If the register is shared in more than one domain, a domain name must be specified. If the register is not shared in the specified domain, an error message is issued and `vmm_ral::RW` is returned.

Example

Example B-191

```
  vmm_ral::access_e rights;
  rights = ral_model.block_name.reg1.get_rights();
  rights =
ral_model.block_name_shared.reg2.get_rights("d1"); //
rights of shared register reg1 in domain "d1"
```

vmm_ral_reg::get_block()

Returns the block that instantiates this register.

SystemVerilog

```
virtual function vmm_ral_block get_block()
```

OpenVera

```
virtual function vmm_ral_block get_block()
```

Description

Returns a reference to the descriptor of the block that includes the register corresponding to the descriptor instance.

Example

Example B-192

```
vmm_ral_block blk;
blk = ral_model.block_name.reg1.get_block();
blk.display();
```

vmm_ral_reg::get_offset_in_block()

Returns the address of the register within the block address space.

SystemVerilog

```
virtual function bit [63:0] get_offset_in_block(
   input string    domain = "")
```

OpenVera

```
virtual function bit [63:0] get_offset_in_block(
   string     domain = "")
```

Description

Returns the address of the register in the overall address space of the block that instantiates it. If the register is shared between multiple physical interfaces, a domain must be specified.

If the register is wider than the physical interface of the block, the lowest address value is returned.

Example

Example B-193

```
  bit [`VMM_RAL_ADDR_WIDTH-1:0] addr;
  addr = ral_model.block_name.reg1.get_offset_in_block();
  addr = 
ral_model.block_name_shared.reg2.get_offset_in_block("d1")
; //multiple domain block
```

vmm_ral_reg::get_address_in_system()

Returns the address of the register within the design address space.

SystemVerilog

```
virtual function bit [63:0] get_address_in_system(
   input string    domain = "")
```

OpenVera

```
virtual function bit [63:0] get_address_in_system(
   string     domain = "")
```

Description

Returns the address of the register in the overall address space of the design. If the register is shared between multiple physical interfaces, you must specify a domain.

If the register is wider than the physical interface used to access it, the lowest address value is returned.

Example

Example B-194

```
  bit [`VMM_RAL_ADDR_WIDTH-1:0] addr;
 addr = ral_model.block_name.reg1.get_address_in_system();
  addr = 
ral_model.block_name_shared.reg2.get_address_in_system("d1
"); //multiple domain block
```

vmm_ral_reg::get_n_bytes()

Returns the width of the register.

SystemVerilog

```
virtual function int unsigned get_n_bytes()
```

OpenVera

```
virtual function integer get_n_bytes()
```

Description

Returns the width, in number of bytes, of the register.

Example

Example B-195

```
integer bytes;
bytes = ral_model.block_name.reg1.get_n_bytes();
```

vmm_ral_reg::get_constraints()

Returns the constraint blocks in this register.

SystemVerilog

```
virtual function void get_constraints(
   ref string names[])
```

OpenVera

```
virtual task get_constraints(
   var string names[*])
```

Description

Fills the specified dynamic array with the names of the constraint blocks in this register. The constraint blocks in the fields within this register are specified in constraint blocks with the same name as the field name. The location of each constraint block name in the array is not defined.

Example

Example B-196

```
class my_ral_reg extends vmm_ral_reg;
   ...
   rand vmm_ral_field MINFL;
   ...
   constraint MINFL_spec {
      MINFL.value == 'h40;
   }
   ..
   function new(vmm_ral_block blk);
      ...
      Xadd_constraintsX("MINFL_spec");
   endfunction
   ...
endclass
...
my_ral_reg my_reg;
...
string str[];
...
this.ral_model.my_reg.get_constraints(str);
foreach (str[i])
  `vmm_note(log,$psprintf("Constraint Name is %0s",str[i]));
...
```

vmm_ral_reg::display()

Displays a description of the register to stdout.

SystemVerilog

```
virtual function void display(string prefix = "",
                              string domain = "")
```

OpenVera

```
virtual task display(string prefix = "",
                     string domain = "")
```

Description

Displays the image created by the `“vmm_ral_reg::psdisplay()”` method to the standard output.

Example

Example B-197

```
ral_model.block_name.reg1.display();
```

vmm_ral_reg::psdisplay()

Creates a human-readable description of the register.

SystemVerilog

```
virtual function string psdisplay(string prefix = "",
                                  string domain = "")
```

OpenVera

```
virtual function string psdisplay(string prefix = "",
                                  string domain = "")
```

Description

Creates a human-readable description of the register and the fields it contains. Each line of the description is prefixed with the specified prefix.

If a domain is specified, the address of the register within that domain is used.

Example

Example B-198

```
 `vmm_note(log, $psprintf("Register description = %s\n",
ral_model.block_name.reg1.psdisplay()));
```

vmm_ral_reg::get_fields()

Returns all fields in this register.

SystemVerilog

```
virtual function void get_fields(
   ref vmm_ral_field fields[])
```

OpenVera

```
virtual task get_fields(
   var vmm_ral_field fields[*])
```

Description

Fills the specified dynamic array with the descriptor for all of the fields contained in the register. Fields are ordered from least-significant position to most-significant position within the register.

Example

Example B-199

```
vmm_ral_field fields[];
ral_model.block_name.reg1.get_fields(fields);
```

vmm_ral_reg::get_field_by_name()

Returns the field with the specified name in this register.

SystemVerilog

```
virtual function vmm_ral_field get_field_by_name(
   string name)
```

OpenVera

```
virtual function vmm_ral_field get_field_by_name(
   string name)
```

Description

Finds a field with the specified name in the register and returns its descriptor. If no fields are found, returns null.

Example

Example B-200

```
  vmm_ral_field field;
  field = 
ral_model.block_name.reg1.get_field_by_name("field_f1");
 if (field == null) `vmm_error(log, "specified field doesn't 
exists");
```

vmm_ral_reg::set_frontdoor()

Defines a user-defined access mechanism for this register.

SystemVerilog

```
function void set_frontdoor(vmm_ral_reg_frontdoor ftdr,
                           string domain = "")
```

OpenVera

```
task set_frontdoor(vmm_ral_reg_frontdoor ftdr,
                  string               domain = "")
```

Description

By default, registers are mapped linearly into the address space of the block that instantiates them. If registers are accessed using a different mechanism, a user-defined access mechanism must be defined and associated with the corresponding register abstraction class.

See “User-Defined Register Access” on page 114 for an example.

vmm_ral_reg::get_frontdoor()

Returns the user-defined access mechanism for this register.

SystemVerilog

```
function vmm_ral_reg_frontdoor get_frontdoor(
   string domain = "")
```

OpenVera

```
function vmm_ral_reg_frontdoor get_frontdoor(
   string domain = "")
```

Description

Returns the current user-defined mechanism for this register for the specified domain. If `null`, no user-defined mechanism has been defined. A user-defined mechanism is defined by using the "`vmm_ral_reg::set_frontdoor()`" method.

Example

Example B-201

```
...
my_ral_reg my_reg;
...
vmm_ral::path_e path;
...
my_ral_reg_frontdoor ftdr = new();
this.my_reg.set_frontdoor(ftdr);
...
if(my_reg.get_frontdoor() == null)
  `vmm_note(log,"Register FrontDoor is not set.");
...
```

vmm_ral_reg::set_backdoor()

Defines the back-door access mechanism for this register.

SystemVerilog

```
virtual function void set_backdoor(
   vmm_ral_reg_backdoor bkdr)
```

OpenVera

```
virtual task set_backdoor(
   vmm_ral_reg_backdoor bkdr)
```

Description

Registers implemented using SystemVerilog variables can be accessed using a hierarchical path. This direct back-door access is automatically generated if the necessary `hdl_path` properties are specified in the RALF description.

However, registers can be modeled using other methods, or be included in imported models written in different languages. This method is used to associate a back-door access mechanism with a register descriptor to enable back-door accesses.

Example

Example B-202

```
TBD
```

vmm_ral_reg::get_backdoor()

Returns the back-door access mechanism for this register.

SystemVerilog

```
virtual function vmm_ral_reg_backdoor get_backdoor()
```

OpenVera

```
virtual function vmm_ral_reg_backdoor get_backdoor()
```

Description

Returns the current back-door mechanism for this register. If `null`, no back-door mechanism has been defined. A back-door mechanism can be automatically defined by using the `hdl_path` properties in the RALF definition or user-defined using the “`vmm_ral_reg::set_backdoor()`” method.

Example

Example B-203

```
my_reg.write(16’hABCD,
             (my_reg.get_backdoor() == null) ?
                vmm_rw::BFM : vmm_rw::BACKDOOR);
```

vmm_ral_reg::set()

Sets the mirror value of a register.

SystemVerilog

```
virtual function void set(
   bit [63:0] value)
```

OpenVera

```
virtual task set(
   bit [63:0] value)
```

Description

Sets the mirror value of the fields in the register to the specified value. Does not actually set the value of the register in the design, only the value mirrored in its corresponding descriptor in the RAL model. Use the `“vmm_ral_reg::update()”` method to update the actual register with the mirrored value or the `“vmm_ral_reg::write()”` method to set the actual register and its mirrored value.

See `“vmm_ral_field::set()” on page 144` for more information on the effect of setting mirror values on fields with different access modes.

To modify the mirrored field values to a specific value, regardless of the access modes—and thus use the RAL mirror as a scoreboard for the register values in the DUT—use the `“vmm_ral_reg::predict()”` method.

Example

Example B-204

```
ral_model.block_name.reg1.set(0);
```

vmm_ral_reg::predict()

Force the mirror value of the register.

SystemVerilog

```
virtual function bit predict(bit [63:0] value)
```

OpenVera

```
virtual function bit predict(bit [63:0] value)
```

Description

Force the mirror value of the fields in the register to the specified value. Does not actually force the value of the fields in the design, only the value mirrored in their corresponding descriptor in the RAL model. Use the `“vmm_ral_reg::update()”` method to update the actual register with the mirrored value or the `“vmm_ral_reg::write()”` method to set the register and its mirrored value.

The final value in the mirror is the specified value, regardless of the access mode of the fields in the register. For example, the mirrored value of a read-only field **is** modified by this method, and the mirrored value of a read-update field can be updated to any value predicted to correspond to the value in the corresponding physical bits in the design.

Returns FALSE if this method is called while the register is being read or written, thus rendering the prediction unreliable. Returns TRUE otherwise.

Example

Example B-205

```
...
   this.ral_model.my_reg.predict(data);
   `vmm_note(log,$psprintf("Forced Value ==> %0h",data));
   `vmm_note(log,$psprintf("Previous Value==>
%h",this.ral_model.my_reg.get()));

   this.ral_model.my_reg.update();
   `vmm_note(log,$psprintf("Updated Value==>
%h",this.ral_model.my_reg.get()));
   ...
```

vmm_ral_reg::get()

Returns the mirror value of a register.

SystemVerilog

```
virtual function bit [63:0] get()
```

OpenVera

```
virtual function bit [63:0] get()
```

Description

Returns the mirror value of the fields in the register. Does not actually read the value of the register in the design, only the value mirrored in its corresponding descriptor in the RAL model.

If the register contains write-only fields, the mirrored value for those fields are the value last written and assumed to reside in the bits implementing these fields. Although a physical read operation would return zeroes for these fields, the returned mirrored value is the actual content.

Use the “`vmm_ral_reg::read()`” method to get the actual register value.

Example

Example B-206

```
bit [63:0] data;
data = ral_model.block_name.reg1.get();
```

vmm_ral_reg::reset()

Resets the mirror value of the register.

SystemVerilog

```
virtual function void reset(vmm_ral::reset_e kind =
vmm_ral::HARD)
```

OpenVera

```
virtual task reset(vmm_ral::reset_e kind = vmm_ral::HARD)
```

Description

Sets the mirror value of the fields in the register to the specified reset value. Does not actually reset the value of the register in the design, only the value mirrored in the descriptor in the RAL model.

Write-once fields in the register can be modified after a hard reset operation.

Example

Example B-207

```
ral_model.block_name.reg1.reset();
```

vmm_ral_reg::needs_update()

Queries if a mirrored value in this register has been set.

SystemVerilog

```
virtual function bit needs_update()
```

OpenVera

```
virtual function bit needs_update()
```

Description

If a mirror value has been modified in the RAL model without actually updating the actual register, the mirror and state of the register is outdated. This method returns TRUE if the state of the register needs to be updated to match the mirrored values (or vice-versa).

The mirror values or actual content of registers are not modified. See “vmm_ral_reg::update()” on page 271 or “vmm_ral_reg::mirror()” on page 272 for more details.

Example

Example B-208

```
bit update;
vmm_rw::status_e status;
update = ral_model.block_name.reg1.needs_update();
if (update == 1) begin
   ral_model.block_name.reg1.update(status);
end
```

vmm_ral_reg::update()

Updates the physical register to match mirrored values in this register descriptor.

SystemVerilog

```
virtual task update(
   output vmm_rw::status_e status,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string          domain = "")
```

OpenVera

```
virtual function vmm_rw::status_e update_t(
   vmm_ral::path_e path = vmm_ral::DEFAULT,
   string         domain = "")
```

Description

Updates the content of the register in the design to match the mirrored value, if it has been modified using one of the `set()` methods to a different value. The update can be performed using the physical interfaces (frontdoor) or `“vmm_ral_reg::poke()”` (backdoor). If the register is shared across multiple physical interfaces and physical access is used (front-door access), a domain must be specified.

This method performs the reverse operation of `“vmm_ral_reg::mirror()” on page 272`.

Example

Example B-209

```
bit update;
```

```
vmm_rw::status_e status;
update = ral_model.block_name.reg1.needs_update();
if (update == 1) begin
   ral_model.block_name.reg1.update(status);
end
```

vmm_ral_reg::mirror()

Updates the mirrored value of the register descriptor to match the design.

SystemVerilog

```
virtual task mirror(
   output vmm_rw::status_e  status,
   input  vmm_ral::check_e  check = vmm_ral::QUIET,
   input  vmm_ral::path_e   path = vmm_ral::DEFAULT,
   input  string           domain = "")
```

OpenVera

```
virtual function vmm_rw::status_e mirror_t(
   vmm_ral::check_e  check = vmm_ral::QUIET,
   vmm_ral::path_e   path  = vmm_ral::DEFAULT,
   string           domain = "")
```

Description

Updates the content of the register mirror value to match the corresponding value in the design. The mirroring can be performed using the physical interfaces (frontdoor) or "vmm_ral_reg::peek()" (backdoor). If the check argument is specified as vmm_ral::VERB, an error message is issued if the current mirrored value does not match the actual value in the design. If the register is shared across multiple physical interfaces and physical access is used (front-door access), a domain must be specified.

If the register contains write-only fields, their content is mirrored and optionally checked only if a vmm_ral::BACKDOOR access path is used to read the register.

This method performs the reverse operation of `“vmm_ral_reg::update()”`.

Example

Example B-210

```
vmm_rw::status_e status;
ral_model.block_name.reg1.mirror(status);
```

vmm_ral_reg::read()

Reads a register from the design.

SystemVerilog

```
virtual task read(
   output vmm_rw::status_e status,
   output bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e read_t(
   var bit [63:0]   value,
   vmm_ral::path_e  path = vmm_ral::DEFAULT,
   string           domain = "",
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Reads the current value of the register from the design using the specified access path. If the register is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access). If a back-door access path is used, the effect of reading the register through a physical access is mimicked. For example, clear-on-read bits in the registers will be cleared.

The optional value of the arguments:

`data_id`
`scenario_id`
`stream_id`

...are passed to the back-door access method or used to set the corresponding `vmm_data/rvm_data` class properties in the `“vmm_rw_access”` transaction descriptors that are necessary to execute this read operation. This allows the physical and back-door read access to be traced back to the higher-level transaction that caused the access to occur.

The mirrored value of the register is updated with the value read from the design. The mirrored value of any write-only field in the register is updated only if an `vmm_ral::BACKDOOR` access path is used.

Example

Example B-211

```
vmm_rw::status_e status;
bit [`VMM_RAL_DATA_WIDTH-1:0] data;
ral_model.block_name.reg1.read(status,data);
```

vmm_ral_reg::write()

Sets a register in the design.

SystemVerilog

```
virtual task write(
   output vmm_rw::status_e status,
   input  bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e write_t(
   bit [63:0]       value,
   vmm_ral::path_e  path = vmm_ral::DEFAULT,
   string           domain = "",
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Writes the specified value in the register in the design using the specified access path. If the register is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access). If a back-door access path is used, the effect of writing the register through a physical access is mimicked. For example, read-only bits in the registers will not be written.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data/rvm_data` class properties in the `“vmm_rw_access”` transaction descriptors that are necessary to execute this write operation. This allows the physical and back-door write access to be traced back to the higher-level transaction that caused the access to occur.

If the register is written using a physical (front-door) path and it contains write-once fields, it is not possible to modify the content of these fields, either through a physical or back-door access.

The mirrored value of the register location is updated based on the written value and the specified behavior of the various fields after a write operation.

Example

Example B-212

```
vmm_rw::status_e status;
ral_model.block_name.reg1.write(status,'hABCD);
```

vmm_ral_reg::peek()

Peek a register from the design.

SystemVerilog

```
virtual task peek(
   output vmm_rw::status_e status,
   output bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e peek_t(
   var bit [63:0]   value,
   vmm_ral::path_e  path = vmm_ral::DEFAULT,
   string           domain = "",
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Reads the current value of the register from the design using a back-door access.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door read access to be traced back to the higher-level transaction that caused the access to occur.

The mirrored value of the register is updated with the value read from the design.

Example

Example B-213

```
vmm_rw::status_e status;
bit [`VMM_RAL_DATA_WIDTH-1:0] data;
ral_model.block_name.reg1.peek(status,data);
```

vmm_ral_reg::poke()

Poke a register in the design.

SystemVerilog

```
virtual task poke(
   output vmm_rw::status_e status,
   input  bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int             stream_id = -1)
```

OpenVera

```
virtual function vmm_rw::status_e write_t(
   bit [63:0]       value,
   integer          data_id = -1,
   integer          scenario_id = -1,
   integer         stream_id = -1)
```

Description

Deposit the specified value in the register in the design, as-is, using a back-door access. See “vmm_ral_field::poke()” for a description of the effect on field values.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door write access to be traced back to the higher-level transaction that caused the access to occur.

The mirrored value of the register location is updated based on the written value.

Example

Example B-214

```
vmm_rw::status_e status;
ral_model.block_name.reg1.poke(status,'hABCD);
```

vmm_ral_reg::append_callback()

Appends a callback extension instance.

SystemVerilog

```
function void append_callback(
   vmm_ral_reg_callbacks cbs)
```

OpenVera

```
task append_callback(
   vmm_ral_reg_callbacks cbs)
```

Description

Appends the specified callback extension instance to the registered callbacks for this register descriptor. Callbacks are invoked in the order of registration.

Note that the corresponding “`vmm_ral_field`” callback methods will be invoked before the register callback methods.

Example

Example B-215

```
program test;
...

class my_ral_reg_callbacks extends vmm_ral_reg_callbacks;
...
endclass

my_ral_reg_callbacks cb;
...
initial
```

```
   begin
      ...
      cb = new();
      env.ral_model.my_reg.append_callback(cb);
      ...
   end
...
endprogram
```

vmm_ral_reg::prepend_callbacks()

Prepends a callback extension instance.

SystemVerilog

```
function void prepend_callbacks(
   vmm_ral_reg_callbacks cbs)
```

OpenVera

```
task prepend_callbacks(
   vmm_ral_reg_callbacks cbs)
```

Description

Prepends the specified callback extension instance to the registered callbacks for this register descriptor. Callbacks are invoked in the reverse order of registration.

Note that the corresponding “`vmm_ral_field`” callback methods will be invoked before the register callback methods.

Example

Example B-216

```
program test;
...

class my_ral_reg_callbacks extends vmm_ral_reg_callbacks;
...
endclass

my_ral_reg_callbacks cb;
...
initial
```

```
      begin
         ...
         cb = new();
         env.ral_model.my_reg.prepend_callback(cb);
         ...
      end
   ...
   endprogram
```

vmm_ral_reg::unregister_callbacks()

Removes a callback extension instance.

SystemVerilog

```
function void unregister_callbacks(
   vmm_ral_reg_callbacks cbs)
```

OpenVera

```
task unregister_callbacks(
   vmm_ral_reg_callbacks cbs)
```

Description

Removes the specified callback extension instance from the registered callbacks for this register descriptor. A warning message is issued if the callback instance has not been previously registered.

Example

Example B-217

```
program test;
...

class my_ral_reg_callbacks extends vmm_ral_reg_callbacks;
...
endclass

my_ral_reg_callbacks cb;
...
initial
   begin
      ...
      cb = new();
      env.ral_model.my_reg.append_callback(cb);
```

```
         //Can't call second time for same instance.
        env.ral_model.my_reg.append_callback(cb);  //Wrong Way
         ...
         env.ral_model.my_reg.unregister_callback(cb);
         ...
      end
   ...
   endprogram
```

vmm_ral_reg_backdoor

Virtual base class for back-door access to registers. Extensions of this class are automatically generated by RAL if full hierarchical paths to registers are specified through the `hdl_path` properties in RALF descriptions.

Can be extended by users to provide user-specific back-door access to registers that are not implemented in pure SystemVerilog.

Summary

vmm_ral_reg_backdoor::read()

Peek a register.

SystemVerilog

```
virtual task read(output vmm_rw::status_e status,
                  output bit [63:0]       data,
                  input  int              data_id,
                  input  int              scenario_id,
                 input  int             stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e read_t(
   var bit [63:0] data,
   integer    data_id,
   integer    scenario_id,
   integer    stream_id)
```

Description

Peek the current value of the register corresponding to the instance of this class. Returns the content of the register and an indication of the success of the operation.

The value of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the `“vmm_ral_reg::read()”` or `“vmm_ral_field::read()”` method call that requires the back-door access. This allows the read access to be traced back to the higher-level transaction that caused the access to occur.

The execution of this method should ideally be non-blocking.

See `“Implementing a Register Backdoor in SystemVerilog” on page 91` for an example.

vmm_ral_reg_backdoor::write()

Poke a register.

SystemVerilog

```
virtual task write(output vmm_rw::status_e status,
                   input  bit [63:0]       data,
                   input  int              data_id,
                   input  int              scenario_id,
                  input  int             stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e write_t(
   bit [63:0] data,
   integer    data_id,
   integer    scenario_id,
   integer   stream_id)
```

Description

Deposit the specified value in the register corresponding to the instance of this class. Returns an indication of the success of the operation.

The values of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the `“vmm_ral_reg::write()”` or `“vmm_ral_field::write()”` method call that requires the back-door access. This allows the write access to be traced back to the higher-level transaction that caused the access to occur.

The execution of this method should ideally be non-blocking.

If the bits in the register cannot be forced to the specified value, for example, read-only bits are implemented as constants, a `vmm_rw::ERROR` status should be returned.

See `“Implementing a Register Backdoor in SystemVerilog” on page 91` for an example.

vmm_ral_reg_callbacks

Base class for register descriptors.

Summary

vmm_ral_reg_callbacks::pre_write()

OOP callback invoked before writing a register.

SystemVerilog

```
virtual task pre_write(vmm_ral_reg         rg,
                       ref bit [63:0]      wdat,
                       ref vmm_ral::path_e path,
                      ref string         domain)
```

OpenVera

```
virtual task pre_write_t(vmm_ral_reg         rg,
                         var bit [63:0]      wdat,
                         var vmm_ral::path_e path,
                        var string         domain)
```

Description

This callback method is invoked before a value is written to a register in the DUT. The written value, if modified, changes the actual value that is written. The path and domain used to write to the register can also be modified.

This callback method is only invoked when the “vmm_ral_reg::write()” or “vmm_ral_field::write()” method is used to write to the register inside the DUT. This callback method is not invoked when only the mirrored value is written to using the “vmm_ral_reg::set()” method.

Because writing a register causes all of the fields it contains to be written, all registered “vmm_ral_field_callbacks::pre_write()” methods with the fields contained in the register will also be invoked before all registered register callback methods.

Example

Example B-218

```
...
fork
  //It will call pre_write() and post_write() callback
methods.
  this.ral_model.my_reg.write(status,data);
join_none
...
program test;
...
  class my_ral_reg_callbacks extends vmm_ral_reg_callbacks;
      ...
     virtual task pre_write(vmm_ral_reg                 rg,
                      ref bit [`VMM_RAL_DATA_WIDTH-1:0] wdat,
                         ref vmm_ral::path_e           path,
                        ref string                   domain);
         `vmm_note(log,{"pre_write method is called for
vmm_ral_reg_callbacks",
                   " class"});
      endtask: pre_write
      ...
   endclass
   ...
   env.ral_model.my_reg.append_callback(cb);
   ...
endprogram
```

vmm_ral_reg_callbacks::post_write()

OOP callback invoked after writing a register.

SystemVerilog

```
virtual task post_write(vmm_ral_reg           rg,
                        bit [63:0]            wdat,
                        vmm_ral::path_e       path,
                        string                domain
                       ref vmm_rw::status_e status)
```

OpenVera

```
virtual task post_write_t(vmm_ral_reg           rg,
                          bit [63:0]            wdat,
                          vmm_ral::path_e       path,
                          string                domain
                         var vmm_rw::status_e status)
```

Description

This callback method is invoked after a value is successfully written to a register in the DUT. The `wdat` value is the final mirrored value in the register as reported by the "`vmm_ral_reg::get()`" method. If a physical write access did not return vmm_rw::IS_OK, this method is not called.

This callback method is only invoked when the "`vmm_ral_reg::write()`" or "`vmm_ral_field::write()`" method is used to write to the register inside the DUT. This callback method is not invoked when only the mirrored value is written to using the "`vmm_ral_reg::set()`" method.

Because writing a register causes all of the fields it contains to be written, all registered "vmm_ral_field_callbacks::post_write()" methods with the fields contained in the register will also be invoked before all registered register callback methods.

Example

Example B-219

```
program test;
...
  class my_ral_reg_callbacks extends vmm_ral_reg_callbacks;
      ...
     virtual task post_write(vmm_ral_reg               rg,
                       bit [`VMM_RAL_DATA_WIDTH-1:0] wdat,
                        vmm_ral::path_e              path,
                        string                     domain,
                      ref vmm_rw::status_e        status);
         `vmm_note(log,{"post_write method is called for
vmm_ral_reg_callbacks",
                   " class"});
      endtask
      ...
   endclass
   ...
   env.ral_model.my_reg.append_callback(cb);
   ...
endprogram
```

vmm_ral_reg_callbacks::pre_read()

OOP callback invoked before reading a register.

SystemVerilog

```
virtual task pre_read(vmm_ral_reg         rg,
                      ref vmm_ral::path_e path,
                     ref string         domain)
```

OpenVera

```
virtual task pre_read_t(vmm_ral_reg         rg,
                        var vmm_ral::path_e path,
                       var string         domain)
```

Description

This callback method is invoked before a value is read from a register in the DUT. You can modify the path and domain used to read the register.

This callback method is only invoked when the `“vmm_ral_reg::read()”` or the `“vmm_ral_field::read()”` method is used to read from the register inside the DUT. This callback method is not invoked when only the mirrored value is read using the `“vmm_ral_reg::get()”` method.

Because reading a register causes all of the fields it contains to be read, all registered `“vmm_ral_field_callbacks::pre_read()”` methods with the fields contained in the register will also be invoked before all registered register callback methods.

Example

Example B-220

```
...
fork
  begin
    //It will call pre_write() and post_write() callback
methods.
    this.ral_model.my_reg.write(status,data);
    write_done = 1'b1;
  end

  begin
    wait(write_done);
    //It will call pre_read() and post_read() callback
methods.
    this.ral_model.my_reg.read(status,read_data);
    `vmm_note(log,$psprintf({"Read Data ==> %0h and Status
==> %0s",
              read_data,status.name));
    write_done = 1'b0;
  end
join_none
...
program test;
...
  class my_ral_reg_callbacks extends vmm_ral_reg_callbacks;
      ...
      virtual task pre_read(vmm_ral_reg       rg,
                            ref vmm_ral::path_e  path,
                            ref string           domain);
         `vmm_note(log,{"pre_read method is called for
vmm_ral_reg_callbacks",
                   " class"});
      endtask
      ...
   endclass
   ...
   env.ral_model.my_reg.append_callback(cb);
   ...
endprogram
```

vmm_ral_reg_callbacks::post_read()

OOP callback invoked after reading a register.

SystemVerilog

```
virtual task post_read(input vmm_ral_reg       rg,
                       ref   bit [63:0]        rdat,
                       input vmm_ral::path_e   path,
                       input string            domain
                      ref   vmm_rw::status_e status)
```

OpenVera

```
virtual task post_read_t(vmm_ral_reg           rg,
                         var bit [63:0]        rdat,
                         vmm_ral::path_e       path,
                         string                domain
                        var vmm_rw::status_e status)
```

Description

This callback method is invoked after a value is successfully read from a register in the DUT. The `rdat` and `status` values are the values that will be ultimately returned by the "`vmm_ral_reg::read()`" method and can be modified. If a physical read access did not return `vmm_rw::IS_OK`, this method is not called.

This callback method is invoked only when the "`vmm_ral_reg::read()`" or "`vmm_ral_field::read()`" method is used to read from the register inside the DUT. This callback method is not invoked when only the mirrored value is read from using the "`vmm_ral_reg::get()`" method.

Because reading a register causes all of the fields it contains to be read, all registered "vmm_ral_field_callbacks::post_read()" methods with the fields contained in the register will also be invoked before all registered register callback methods.

Example

Example B-221

```
program test;
...
  class my_ral_reg_callbacks extends vmm_ral_reg_callbacks;
      ...
     virtual task post_read(vmm_ral_reg                 rg,
                    ref bit [`VMM_RAL_DATA_WIDTH-1:0] rdat,
                      input vmm_ral::path_e           path,
                      input string                  domain,
                    ref vmm_rw::status_e            status);
         `vmm_note(log,{"post_read method is called for
vmm_ral_reg_callbacks",
                   " class"});
      endtask
      ...
   endclass
   ...
   env.ral_model.my_reg.append_callback(cb);
   ...
endprogram
```

vmm_ral_reg_frontdoor

Virtual base class for user-defined access to registers through a physical interface.

By default, different registers are mapped to different addresses in the address space of the block instantiating them. If registers are physically accessed using a non-linear, non-mapped mechanism, this base class must be user-extended to provide the physical access to these registers. See `“User-Defined Register Access” on page 114` for an example.

Summary

vmm_ral_reg_frontdoor::read()

Performs a physical register read.

SystemVerilog

```
virtual task read(output vmm_rw::status_e status,
                  output bit [63:0]       data,
                  input  int              data_id,
                  input  int              scenario_id,
                 input  int              stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e read_t(
   var bit [63:0]  data,
   integer         data_id,
   integer         scenario_id,
   integer        stream_id)
```

Description

Performs a physical read access of the register corresponding to the instance of this class. Returns the content of the register and an indication of the success of the operation.

The values of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the `“vmm_ral_reg::read()”` method call that requires the front-door access. This allows the read access to be traced back to the higher-level transaction that caused the access to occur.

See “User-Defined Register Access” on page 114 for an example.

vmm_ral_reg_frontdoor::write()

Performs a physical write to a register.

SystemVerilog

```
virtual task write(output vmm_rw::status_e status,
                   input  bit [63:0]       data,
                   input  int              data_id,
                   input  int              scenario_id,
                  input  int              stream_id)
```

OpenVera

```
virtual function vmm_rw::status_e write_t(
   bit [63:0]  data,
   integer     data_id,
   integer     scenario_id,
   integer     stream_id)
```

Description

Performs a physical write access to the register corresponding to the instance of this class. Returns an indication of the success of the operation.

The values of the arguments:

```
data_id
scenario_id
stream_id
```

...are the values that were optionally specified to the “vmm_ral_reg::write()” method call that requires the front-door access. This allows the write access to be traced back to the higher-level transaction that caused the access to occur.

See `“User-Defined Register Access” on page 114` for an example.

vmm_ral_sys

System descriptor class derived from “vmm_ral_block_or_sys”.

Summary

vmm_ral_sys::new()

Creates an instance of a RAL model.

SystemVerilog

```
function new(vmm_ral::coverage_e cover_on =
vmm_ral::NO_COVERAGE);
```

OpenVera

```
task new(vmm_ral::coverage_e cover_on =
vmm_ral::NO_COVERAGE);
```

Description

Creates an instance of a RAL model with the corresponding system as the top-level structural element.

The `cover_on` argument specifies the functional coverage models to be enabled in the RAL model. Multiple functional coverage models may be specified by adding their symbolic names. Only functional coverage models that were generated by `ralgen` using the `-c` option can be enabled. Because the functional coverage models affect the memory footprint and runtime performance of a RAL model, they should be enabled only when relevant. See "Predefined Functional Coverage Models" on page 121 for more details.

It is not possible to enable a functional coverage model at a later time, but it is possible to turn the measurement of a functional coverage model off and on using the "vmm_ral_block_or_sys::set_cover()" method.

Example

Example B-222

```
ral_sys_mysys ral_model = new(vmm_ral::ALL_COVERAGE);
```

vmm_ral_sys::get_blocks()

Returns all blocks in system.

SystemVerilog

```
virtual function void get_blocks(
   ref vmm_ral_block blocks[],
   ref string        domains[],
   input string     domain = "")
```

OpenVera

```
virtual task get_blocks(
   var vmm_ral_block blocks[*],
   var string        domains[*],
   string           domain = "")
```

Description

Fills the specified dynamic arrays with the abstraction classes and their domain name for all of the blocks directly instantiated in the system. The order in which the blocks are located in the array is not specified.

The name returned in `domains[`*`i`*`]` corresponds to the domain name within `blocks[`*`i`*`]` that is instantiated in the system.

If a domain name is specified, only the blocks instantiated in the specified domain are included.

The order in which the blocks are located in the array is not specified.

Example

Example B-223

```
  vmm_ral_block blks[];
  string domains [];
  ral_model.get_blocks(blks,domains);
  foreach (blks[i]) begin
    $display(" block name = %s :: domain name = 
%s",blks[i].get_fullname(),domains[i]);
  end
```

vmm_ral_sys::get_all_blocks()

Returns all blocks in the system and subsystems.

SystemVerilog

```
virtual function void get_all_blocks(
   ref vmm_ral_block blocks[],
   ref string        domains[],
   input string     domain = "")
```

OpenVera

```
virtual task get_all_blocks(
   var vmm_ral_block blocks[*],
   var string        domains[*],
   string           domain = "")
```

Description

Fills the specified dynamic arrays with the abstraction classes and domain names for all of the blocks instantiated in the system and subsystems it contains.

The name returned in `domains[`*`i`*`]` corresponds to the domain name within `blocks[`*`i`*`]` that is instantiated in the system.

If a domain is specified, only those blocks accessible in that domain are returned.

The order in which the blocks are located in the array is not specified.

Example

Example B-224

```
vmm_ral_block blks[];
```

```
  string domains [];
  ral_model.get_all_blocks(blks,domains);
  foreach (blks[i]) begin
    $display(" block name = %s :: domain name =
%s",blks[i].get_fullname(),domains[i]);
  end
```

vmm_ral_sys::get_block_by_name()

Returns the block with the specified name in this system.

SystemVerilog

```
virtual function vmm_ral_block get_block_by_name(
   string name)
```

OpenVera

```
virtual function vmm_ral_block get_block_by_name(
   string name)
```

Description

Finds a block with the specified name in the system and returns its descriptor. If no blocks are found, returns null.

Block name uniqueness is guaranteed only within a single system. Therefore, if used on a system with more than one block having the same name but in different subsystems, this method returns the first block found.

Example

Example B-225

```
  vmm_ral_block blk;
  blk = ral_model.get_block_by_name("block1");
 if (blk == null) `vmm_error(log, "specified block doesn't
exists");
```

vmm_ral_sys::get_subsys()

Returns all subsystems in system.

SystemVerilog

```
virtual function void get_subsys(
   ref vmm_ral_sys subsys[],
   ref string      domains[],
   input string   domain = "")
```

OpenVera

```
virtual task get_subsys(
   var vmm_ral_sys subsys[*],
   var string      domains[*],
   string         domain = "")
```

Description

Fills the specified dynamic arrays with the abstraction classes and domain names for all of the subsystems directly instantiated in the system.

The name returned in `domains[`*`i`*`]` corresponds to the domain name within `subsys[`*`i`*`]` that is instantiated in the system.

If a domain is specified, only those subsystems accessible in that domain are returned.

The order in which the subsystems are located in the array is not specified.

Example

Example B-226

```
  vmm_ral_sys sub_sys[];
  string domains [];
  ral_model.get_subsys(sub_sys,domains);
  foreach (sub_sys[i]) begin
    $display(" Sub-system name = %s :: domain name = 
%s",sub_sys[i].get_fullname(),domains[i]);
  end
```

vmm_ral_sys::get_all_subsys()

Returns all subsystems in system and subsystems.

SystemVerilog

```
virtual function void get_all_subsys(
   ref vmm_ral_subsys subsys[],
   ref string         domains[],
   input string      domain = "")
```

OpenVera

```
virtual task get_all_subsys(
   var vmm_ral_sys subsys[*],
   var string      domains[*],
   string         domain = "")
```

Description

Fills the specified dynamic arrays with the abstraction classes and domain names for all of the subsystems instantiated in the system and subsystems it contains.

The name returned in `domains[`*`i`*`]` corresponds to the domain name within `subsys[`*`i`*`]` that is instantiated in the system.

If a domain is specified, only those subsystems accessible in that domain are returned.

The order in which the subsystems are located in the array is not specified.

Example

Example B-227

```
  vmm_ral_sys sub_sys[];
  string domains [];
  ral_model.get_all_subsys(sub_sys,domains);
  foreach (sub_sys[i]) begin
    $display(" Sub-system name = %s :: domain name = 
%s",sub_sys[i].get_fullname(),domains[i]);
  end
```

vmm_ral_sys::get_subsys_by_name()

Returns the subsystem with the specified name in this system.

SystemVerilog

```
virtual function vmm_ral_sys get_subsys_by_name(
   string name)
```

OpenVera

```
virtual function vmm_ral_sys get_subsys_by_name(
   string name)
```

Description

Finds a subsystem with the specified name in the system and returns its descriptor. If no system is found, returns `null`.

Subsystem name uniqueness is guaranteed only within a single system. Therefore, if used on a system with more than one subsystem having the same name but in different subsystems, this method returns the first subsystem found.

Example

Example B-228

```
vmm_ral_sys sys;
  sys = ral_model.get_subsys_by_name("sys1");
  if (sys == null) `vmm_error(log, "specified subsystem
doesn't exists");
```

vmm_ral_vfield

Field descriptors.

Summary

vmm_ral_vfield::log

Message service interface.

SystemVerilog

```
vmm_log log
```

Description

Message service interface instance for the field descriptor. A single message service interface instance is shared by all virtual field abstraction class instances.

Example

Example B-229

```
  vmm_log log = new("Test", "Main");

 ral_model.status_reg.virtual_field.read(0, status, data);
  if (status != vmm_rw::IS_OK) begin
     `vmm_error(log, "Non-OK status when reading through 
virtual field");
  end

  log.report();
```

vmm_ral_vfield::get_name()

Returns the name of the field.

SystemVerilog

```
virtual function string get_name()
```

Description

Returns the name of the field corresponding to the instance of the descriptor.

Example

Example B-230

```
  vmm_ral_vfield vfields[];
  ral_model.status_vreg.get_fields(vfields);
  foreach (vfields[i]) begin
      `vmm_note(log, $psprintf("Virtual Field Name: %s\n",
vfields[i].get_name()));
  end
```

vmm_ral_vfield::get_fullname()

Returns the fully-qualified name of the field.

SystemVerilog

```
virtual function string get_fullname()
```

Description

Returns the hierarchical name of the field corresponding to the instance of the descriptor. The name of the top-level block or system is not included in the fully-qualified name as it is implicit for every RAL model.

Example

Example B-231

```
  vmm_ral_vfield vfields[];
  ral_model.status_vreg.get_fields(vfields);
  foreach (vfields[i]) begin
      `vmm_note(log, $psprintf("Virtual Field Hierarchical 
Name: %s\n", vfields[i].get_fullname()));
  end
```

vmm_ral_vfield::get_register()

Returns the virtual register that instantiates this field.

SystemVerilog

```
virtual function vmm_ral_vreg get_register()
```

Description

Returns a reference to the descriptor of the virtual register that includes the field corresponding to the descriptor instance.

Example

Example B-232

```
vmm_ral_vreg vreg[];
vmm_ral_vfield vfields[];
ral_model.status_vreg.get_fields(vfields);
foreach (vfields[i]) begin
   vreg[i] = vfields[i].get_register();
   vreg[i].display();
end
```

vmm_ral_vfield::get_lsb_pos_in_register()

Returns the offset of the least-significant bit of the field.

SystemVerilog

```
virtual function int unsigned get_lsb_pos_in_register()
```

Description

Returns the index of the least significant bit of the field in the virtual register that instantiates it. An offset of 0 indicates a field that is aligned with the least-significant bit of the virtual register.

Example

Example B-233

```
  vmm_ral_vfield vfields[];
  ral_model.status_vreg.get_fields(vfields);
  foreach (vfields[i]) begin
     `vmm_note(log, $psprintf("Offset of LSB of the field:
%s\n", vfields[i].get_lsb_pos_in_register()));
  end
```

vmm_ral_vfield::get_n_bits()

Returns the width of the field.

SystemVerilog

```
virtual function int unsigned get_n_bits()
```

Description

Returns the width, in number of bits, of the field.

Example

Example B-234

```
  integer count[];
  vmm_ral_vfield vfields[];
  ral_model.status_vreg.get_fields(vfields);
  foreach (vfields[i]) begin
      `vmm_note(log, $psprintf("Width of the field: %d\n", 
vfields[i].get_n_bits()));
       count[i] = vfields[i].get_n_bits();
  end
```

vmm_ral_vfield::get_access()

Returns the access mode of the field.

SystemVerilog

```
virtual function vmm_ral::access_e get_access(string domain
= "")
```

Description

Returns the specification of the behavior of the field when written and read through the optionally-specified domain.

If the register containing the field is shared across multiple domains, a domain must be specified. The access mode of a field in a specific domain may be restricted by the domain access rights of the memory implementing the field. For example, a RW field may only be writable through one of the domains and read-only through all of the other domains.

Example

Example B-235

```
  vmm_ral::access_e access;
  vmm_ral_vfield vfields[];
  ral_model.status_vreg.get_fields(vfields);
  foreach (vfields[i]) begin
    access = vfields[i].get_access();
    if (access == vmm_ral::RW) `vmm_note(log,
"vmm_ral_vfield::get_access = vmm_ral::RW\n");
    else if (access == vmm_ral::RO) `vmm_note(log,
"vmm_ral_vfield::get_access = vmm_ral::RO\n");
    else if (access == vmm_ral::WO) `vmm_note(log,
"vmm_ral_vfield::get_access = vmm_ral::WO\n");
    else `vmm_note(log,
$psprintf("vmm_ral_vfield::get_access = %d\n", access));
  end
```

vmm_ral_vfield::display()

Displays a description of the field to stdout.

SystemVerilog

```
virtual function void display(string prefix = "")
```

Description

Displays the image created by the “`vmm_ral_field::psdisplay()`” method on the standard output.

Example

Example B-236

```
vmm_ral_vfield vfields[];
ral_model.status_vreg.get_fields(vfields);
foreach (vfields[i]) begin
    vfields[i].display();
end
```

vmm_ral_vfield::psdisplay()

Creates a human-readable description of the field.

SystemVerilog

```
virtual function string psdisplay(string prefix = "")
```

Description

Creates a human-readable description of the field and its current mirrored value. Each line of the description is prefixed with the specified prefix.

Example

Example B-237

```
  vmm_ral_vfield vfields[];
  ral_model.status_vreg.get_fields(vfields);
  foreach (vfields[i]) begin
   `vmm_note(log, $psprintf("psdisplay of the field: %s\n",
vfields[i].psdisplay()));
  end
```

vmm_ral_vfield::read()

Reads a virtual field value from the design.

SystemVerilog

```
virtual task read(
   input  longint unsigned idx,
   output vmm_rw::status_e status,
   output bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int                stream_id = -1)
```

Description

Reads the current value of the field in the virtual register specified by the index from the associated memory using the specified access path.

If the field is located in a memory shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data` class properties in the "`vmm_rw_access`" transaction descriptors that are necessary to execute this read operation. This allows the physical and back-door read accesses to be traced back to the higher-level transaction that caused the access to occur.

Example

Example B-238

```
vmm_rw::status_e status;
bit [`VMM_RAL_DATA_WIDTH-1:0] data;
ral_model.status_vreg.vfield.read(0,status,data);
```

vmm_ral_vfield::write()

Sets a virtual field value in the design.

SystemVerilog

```
virtual task write(
   input  longint unsigned idx,
   output vmm_rw::status_e status,
   input  bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int                stream_id = -1)
```

Description

Writes the specified field value in the virtual register specified by the index into the associated memory using the specified access path. If a back-door access path is used, the effect of writing the field through a physical access is mimicked. For example, a read-only field will not be written.

If the virtual field is located in a memory shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data` class properties in the "`vmm_rw_access`" transaction descriptors that are necessary to execute this write operation. This allows the physical and back-door write accesses to be traced back to the higher-level transaction that caused the access to occur.

If the memory location where this virtual field is physically located contains other fields, a read-modify-write process is used to update the field value without modifying the others.

Example

Example B-239

```
vmm_rw::status_e status;
ral_model.status_vreg.vfield.write(0,status,'hABCD);
```

vmm_ral_vfield::peek()

Peek a virtual field value from the design.

SystemVerilog

```
virtual task peek(
   input  longint unsigned idx,
   output vmm_rw::status_e status,
   output bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int                stream_id = -1)
```

Description

Peek the current value of the virtual field from the associated memory using a back-door access. The value of the field in the design is not modified, regardless of the access mode.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door read accesses to be traced back to the higher-level transaction that caused the access to occur.

Example

Example B-240

```
vmm_rw::status_e status;
bit [`VMM_RAL_DATA_WIDTH-1:0] data;
ral_model.status_vreg.vfield.peek(0,status,data);
```

vmm_ral_vfield::poke()

Poke a field value in the design.

SystemVerilog

```
virtual task poke(
   input  longint unsigned idx,
   output vmm_rw::status_e status,
   input  bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int                stream_id = -1)
```

Description

Deposit the specified field value in the associated memory using a back-door access. The value of the field is updated, regardless of the access mode.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door write accesses to be traced back to the higher-level transaction that caused the access to occur.

If the memory location where this field is physically located contains other fields, the current value of the other fields are peeked first then poked back in.

Example

Example B-241

```
vmm_rw::status_e status;
ral_model.status_vreg.vfield.poke(0,status,'hABCD);
```

vmm_ral_vfield::append_callback()

Appends a callback extension instance.

SystemVerilog

```
function void append_callback(vmm_ral_vfield callbacks cbs)
```

Description

Appends the specified callback extension instance to the registered callbacks for this field descriptor. Callbacks are invoked in the order of registration.

Note that field callback methods will be invoked before their corresponding “`vmm_ral_vreg`” callback methods.

Example

Example B-242

```
write_after_read cb = new;
this.ral_model.blk.magic_vfield.append_callback(cb);
```

vmm_ral_vfield::prepend_callback()

Prepends a callback extension instance.

SystemVerilog

```
function void prepend_callback(vmm_ral_vfield_callbacks
cbs)
```

Description

Prepends the specified callback extension instance to the registered callbacks for this field descriptor. Callbacks are invoked in the reverse order of registration.

Note that field callback methods will be invoked before their corresponding "`vmm_ral_vreg`" callback methods.

Example

Example B-243

```
write_after_read cb = new;
this.ral_model.blk.magic_vfield.prepend_callback(cb);
```

vmm_ral_vfield::unregister_callback()

Removes a callback extension instance.

SystemVerilog

```
function void unregister_callback(vmm_ral_vfield_callbacks
cbs)
```

Description

Removes the specified callback extension instance from the registered callbacks for this field descriptor. A warning message is issued if the callback instance has not been previously registered.

Example

Example B-244

```
write_after_read cb = new;
this.ral_model.blk.magic_vfield.unregister_callback(cb);
```

vmm_ral_vfield_callbacks

Field descriptors.

Summary

vmm_ral_vfield_callbacks::pre_write()

OOP callback invoked before writing a field.

SystemVerilog

```
virtual task pre_write(vmm_ral_vfield      field,
                       longint unsigned    idx,
                       ref bit [63:0]      wdat,
                       ref vmm_ral::path_e path,
                         ref string            domain)
```

Description

This callback method is invoked before a value is written to a field in the DUT. The written value, if modified, changes the actual value that will be written. The path and domain used to write to the field can also be modified.

This callback method is only invoked when the “vmm_ral_vfield::write()” or “vmm_ral_vreg::write()” method is used to write to the field inside the DUT. This callback method is not invoked when the memory location is directly written to using the “vmm_ral_mem::write()” method.

Because writing a field causes the memory location to be written, and, therefore all of the other fields it contains to also be written, all registered “vmm_ral_vfield_callbacks::pre_write()” methods with the fields contained in the same memory location will also be invoked. Because the memory implementing the virtual field is accessed through its own abstraction class, all of its registered “vmm_ral_mem_callbacks::pre_write()” methods will also be invoked as a side effect.

Example

Example B-245

```
class vfield_test_cb extends vmm_ral_vfield_callbacks;

 task cb(string pretext);
    $write("\n%s : vfield_test_cb's instance %s method 
called\n",name,pretext);
 endtask

 task pre_write(vmm_ral_vfield                   field,
                longint unsigned                 idx,
                ref bit[`VMM_RAL_DATA_WIDTH-1:0] wdat,
                ref vmm_ral::path_e              path,
                ref string                       domain);
   cb("pre_write");
 endtask

endclass
```

vmm_ral_vfield_callbacks::post_write()

OOP callback invoked after writing a field.

SystemVerilog

```
virtual task post_write(vmm_ral_vfield        field,
                        longint unsigned      idx,
                        bit [63:0]            wdat,
                        vmm_ral::path_e       path,
                        string                domain
                          ref vmm_rw::status_e    status)
```

Description

This callback method is invoked after a value is written to a virtual field in the DUT.

This callback method is only invoked when the "vmm_ral_vfield::write()" or "vmm_ral_vreg::write()" method is used to write to the field inside the DUT. This callback method is not invoked when the memory location is directly written to using the "vmm_ral_mem::write()" method.

Because writing a field causes the memory location to be written, and, therefore all of the other fields it contains to also be written, all registered "vmm_ral_vfield_callbacks::post_write()" methods with the fields contained in the same memory location will also be invoked. Because the memory implementing the virtual field is accessed through its own abstraction class, all of its registered "vmm_ral_mem_callbacks::post_write()" methods will also be invoked as a side effect.

Example

Example B-246

```
class vfield_test_cb extends vmm_ral_vfield_callbacks;

 task cb(string pretext);
    $write("\n%s : vfield_test_cb's instance %s method 
called\n",name,pretext);
 endtask

 task  post_write(vmm_ral_vfield                field,
                        longint unsigned             idx,
                      bit [`VMM_RAL_DATA_WIDTH-1:0] wdat,
                       vmm_ral::path_e              path,
                       string                     domain,
                     ref vmm_rw::status_e        status);

   cb("post_write");
 endtask

endclass
```

vmm_ral_vfield_callbacks::pre_read()

OOP callback invoked before reading a field.

SystemVerilog

```
virtual task pre_read(vmm_ral_vfield      field,
                      longint unsigned    idx,
                      ref vmm_ral::path_e path,
                        ref string            domain)
```

Description

This callback method is invoked before a value is read from a field in the DUT. The path and domain used to read from the field can be modified.

This callback method is only invoked when the "`vmm_ral_vfield::read()`" method is used to read the field inside the DUT. This callback method is not invoked when the memory location containing the field is read directly using the "`vmm_ral_mem::read()`" method.

Because reading a field causes the memory location to be read, and, therefore all of the other fields it contains to also be read, all registered "`vmm_ral_vfield_callbacks::pre_read()`" methods with the fields contained in the same memory location will also be invoked. Because the memory implementing the virtual field is accessed through its own abstraction class, all of its registered "`vmm_ral_mem_callbacks::pre_read()`" methods will also be invoked as a side effect.

Example

Example B-247

```
class vfield_test_cb extends vmm_ral_vfield_callbacks;

 task cb(string pretext);
    $write("\n%s : vfield_test_cb's instance %s method 
called\n",name,pretext);
 endtask

 task pre_read(vmm_ral_vfield        field,
                         longint unsigned      idx,
                         ref vmm_ral::path_e   path,
                         ref string            domain);
   cb("pre_read");
 endtask

endclass
```

vmm_ral_vfield_callbacks::post_read()

OOP callback invoked after reading a field.

SystemVerilog

```
virtual task post_read(input vmm_ral_vfield    field,
                       input longint unsigned  idx,
                       ref   bit [63:0]        rdat,
                       input vmm_ral::path_e   path,
                       input string            domain
                         ref   vmm_rw::status_e    status)
```

Description

This callback method is invoked after a value is read from a virtual field in the DUT. The `rdat` and `status` values are the values that are ultimately returned by the “`vmm_ral_vfield::read()`” method and they can be modified.

This callback method is only invoked when the “`vmm_ral_vfield::read()`” method is used to read the field inside the DUT. This callback method is not invoked when the memory location containing the field is read directly using the “`vmm_ral_mem::read()`” method.

Because reading a field causes the memory location to be read, and, therefore all of the other fields it contains to also be read, all registered “`vmm_ral_vfield_callbacks::post_read()`” methods with the fields contained in the same memory location will also be invoked. Because the memory implementing the virtual field is accessed through its own abstraction class, all of its registered “`vmm_ral_mem_callbacks::post_read()`” methods will also be invoked as a side effect.

Example

Example B-248

```
class vfield_test_cb extends vmm_ral_vfield_callbacks;

 task cb(string pretext);
    $write("\n%s : vfield_test_cb's instance %s method 
called\n",name,pretext);
 endtask

 task post_read(vmm_ral_vfield                   field,
                longint unsigned                 idx,
                ref bit [`VMM_RAL_DATA_WIDTH-1:0] rdat,
                vmm_ral::path_e                  path,
                string                           domain,
                ref vmm_rw::status_e             status);
   cb("post_read");
 endtask

endclass
```

vmm_ral_vreg

Base class for virtual register descriptors.

Summary

vmm_ral_vreg::log

Message service interface.

SystemVerilog

```
vmm_log log
```

Description

Message service interface instance for the virtual register descriptor. A single message service interface instance is shared by all virtual register abstraction class instances.

Example

Example B-249

```
  vmm_log log = new("Test", "Main");

  ral_model.virtual_reg.read(0, status, data);
  if (status != vmm_rw::IS_OK) begin
     `vmm_error(log, "Non-OK status when reading through 
virtual register");
  end

  log.report();
```

vmm_ral_vreg::get_name()

Returns the name of the virtual register.

SystemVerilog

```
virtual function string get_name()
```

Description

Returns the name of the register corresponding to the instance of the descriptor.

Example

Example B-250

```
  vmm_ral_vreg vregs[];
  ral_model.get_virtual_registers(vregs)
  foreach (vregs[i]) begin
    `vmm_note(log, $psprintf("Virtual register Name: %s\n", 
vregs[i].get_name()));
  end
```

vmm_ral_vreg::get_fullname()

Returns the fully-qualified name of the virtual register.

SystemVerilog

```
virtual function string get_fullname()
```

Description

Returns the hierarchical name of the register corresponding to the instance of the descriptor. The name of the top-level block or system is not included in the fully-qualified name as it is implicit for every RAL model.

Example

Example B-251

```
  vmm_ral_vreg vregs[];
  ral_model.get_virtual_registers(vregs)
  foreach (vregs[i]) begin
    `vmm_note(log, $psprintf("Virtual register Name: %s\n", 
vregs[i].get_fullname()));
  end
```

vmm_ral_vreg::get_block()

Returns the block instantiating the virtual register.

SystemVerilog

```
virtual function vmm_ral_block get_block()
```

Description

Returns a reference to the descriptor of the block that includes the register corresponding to the descriptor instance.

Example

Example B-252

```
vmm_ral_block blk;
blk = ral_model.virtual_register.get_block();
blk.display();
```

vmm_ral_vreg::implement()

Dynamically implement a set of virtual registers.

SystemVerilog

```
virtual function bit implement(
     longint unsigned n,
     vmm_ral_mem      mem    = null,
     bit [63:0]       offset = 0,
     int unsigned       incr    = 0)
```

Description

Dynamically implement, resize or relocate a set of virtual registers of the specified size, in the specified memory and offset. If an offset increment is specified, each virtual register is implemented at the specified offset from the previous one. If an offset increment of 0 is specified, virtual registers are packed as closely as possible in the memory. If no memory is specified, the virtual register set is in the same memory, at the same offset using the same offset increment as originally implemented.

The initial value of the newly-implemented or relocated set of virtual registers is whatever values are currently stored in the memory now implementing them.

Returns TRUE if the memory can implement the number of virtual registers at the specified offset and increment. Returns FALSE if the memory cannot implement the specified virtual register set.

The memory region used to implement a set of virtual registers is reserved to prevent it from being allocated for another purpose by the memory's default memory allocation manager.

Statically-implemented virtual registers cannot be implemented, resized nor relocated.

Example

Example B-253

```
TBD
```

vmm_ral_vreg::allocate()

Dynamically implement a set of virtual registers.

SystemVerilog

```
virtual function vmm_mam_region allocate(
     longint unsigned n,
     vmm_mam           mam)
```

Description

Dynamically implement, resize or relocate a set of virtual registers of the specified size to a randomly allocated region of the appropriate size in the address space managed by the specified memory allocation manager.

The initial value of the newly-implemented or relocated set of virtual registers is whatever values are currently stored in the memory region now implementing them.

Returns a reference to a memory region descriptor if the memory allocation manager was able to allocate a region that can implement the number of virtual registers. Returns `null` if the memory allocation manager cannot allocate a suitable region.

Statically-implemented virtual registers cannot be implemented, resized nor relocated.

Example

Example B-254

```
TBD
```

vmm_ral_vreg::get_region()

Returns the region descriptor where virtual registers are implemented.

SystemVerilog

```
virtual function vmm_mam_region get_region()
```

Description

Returns a reference to a memory region descriptor that implements the set of virtual registers. Returns `null` if the virtual registers are not currently implemented.

A region implementing a set of virtual registers must not be released using the `vmm_mam::release_region()` method. It must be released using the “`vmm_ral_vreg::release_region()`” method.

Example

Example B-255

```
  vmm_mam_region mr;
  mr = ral_model.virtual_register.get_region();
  if (mr == null) `vmm_note(log, "failed to get_region on
virtual register");
```

vmm_ral_vreg::release_region()

Free the memory used to implement a set of virtual registers.

SystemVerilog

```
virtual function void release_region()
```

Description

Release the memory region used to implement the set of virtual registers and return it to the pool of available memory that can be allocated by the memory's default allocation manager. The virtual registers are subsequently considered as unimplemented and can no longer be accessed.

Statically-implemented virtual registers cannot be released.

Example

Example B-256

```
 vmm_mam_region mr;
 ral_model.virtual_register.release_region();
 mr = ral_model.virtual_register.get_region();
if (mr != null) `vmm_note(log, "'release_region' Failed");
```

vmm_ral_vreg::get_memory()

Returns the memory that implements this register.

SystemVerilog

```
virtual function vmm_ral_mem get_memory()
```

Description

Returns a reference to the memory abstraction class for the memory that implements the set of virtual registers corresponding to the descriptor instance.

Example

Example B-257

```
vmm_ral_mem mem;
mem = ral_model.virtual_register.get_memory();
```

vmm_ral_vreg::get_n_domains()

Returns the number of domains sharing this virtual register.

SystemVerilog

```
function int get_n_domains()
```

Description

Returns the number of domains that share the memory implementing this set of virtual registers. The name of the domains can be obtained with the "`vmm_ral_reg::get_domains()`" method.

Example

Example B-258

```
 int num_domains;
num_domains = ral_model.virtual_register.get_n_domains();
```

vmm_ral_vreg::get_domains()

Returns the name of the domains sharing this virtual register.

SystemVerilog

```
function void get_domains(ref string domains[])
```

Description

Fills the specified dynamic array with the names of all the block-level domains that can access the memory implementing this set of virtual registers. The order of the domain names is not specified.

Example

Example B-259

```
  string domains[];
  ral_model.virtual_register.get_domains(domains);
  foreach (domains[i]) begin
     `vmm_note(log, $psprintf("Domain %d = %s\n", i,
domains[i]));
  end
```

vmm_ral_vreg::get_access()

Returns the access mode of the virtual register.

SystemVerilog

```
virtual function vmm_ral::access_e get_access(string domain
= "")
```

Description

Returns the specification of the behavior of the memory used to implement the set of virtual register when written and read. If the memory is shared across more than one domain, a domain name must be specified.

If access restrictions are present when accessing a memory through the specified domain, the access mode returned takes the access restrictions into account. For example, a read-write memory accessed through a domain with read-only restrictions would return `vmm_ral::RO`.

Example

Example B-260

```
vmm_ral::access_e access;
access = ral_model.virtual_register.get_access();
```

vmm_ral_vreg::get_rights()

Returns the access rights of this virtual register.

SystemVerilog

```
function vmm_ral::access_e get_rights(string domain = "")
```

Description

Returns the access rights of the memory implementing this set of virtual registers. Returns `vmm_ral::RW`, `vmm_ral::RO` or `vmm_ral::WO`. See “`vmm_ral_mem::get_rights()`” for more details.

If the memory implementing this set of virtual registers is shared in more than one domain, a domain name must be specified. If the memory is not shared in the specified domain, an error message is issued and `vmm_ral::RW` is returned.

Example

Example B-261

```
vmm_ral::access_e rights;
rights = ral_model.virtual_register.get_rights();
```

vmm_ral_vreg::get_offset_in_memory()

Returns the address of the virtual register within the memory address space.

SystemVerilog

```
virtual function bit [63:0] get_offset_in_memory(
     longint unsigned idx)
```

Description

Returns the offset of the virtual register in the overall address space of the memory that implements it.

If the virtual register occupies more than one memory location, the lowest offset value is returned.

Example

Example B-262

```
  bit [`VMM_RAL_ADDR_WIDTH-1:0] addr;
  addr =
ral_model.virtual_register.get_offset_in_memory(3);
```

vmm_ral_vreg::get_address_in_system()

Returns the address of the virtual register within the design address space.

SystemVerilog

```
virtual function bit [63:0] get_address_in_system(
     longint unsigned idx,
     string            domain = "")
```

Description

Returns the address of the virtual register in the overall address space of the design. If the memory implementing the virtual register is shared between multiple physical interfaces, a domain must be specified.

If the virtual register is wider than the physical interface used to access it, the lowest address value is returned.

Example

Example B-263

```
  bit [`VMM_RAL_ADDR_WIDTH-1:0] addr;
  addr =
ral_model.virtual_register.get_address_in_system(2);
```

vmm_ral_vreg::get_size()

Returns the size of the virtual register array.

SystemVerilog

```
virtual function int unsigned get_size()
```

Description

Returns the number of virtual registers in the virtual register array.

Example

Example B-264

```
int size;
size = ral_model.virtual_register.get_size();
```

vmm_ral_vreg::get_n_bytes()

Returns the width of the virtual register.

SystemVerilog

```
virtual function int unsigned get_n_bytes()
```

Description

Returns the width, in number of bytes, of the virtual register.

The width of a virtual register is always a multiple of the width of the memory locations used to implement it. For example, a virtual register containing two 1-byte fields implemented in a memory with 4-bytes memory locations is 4-byte wide.

Example

Example B-265

```
int width;
width = ral_model.virtual_register.get_n_bytes();
```

vmm_ral_vreg::get_n_memlocs()

Returns the size of the virtual register array.

SystemVerilog

```
virtual function int unsigned get_n_memlocs()
```

Description

Returns the number of memory locations used by a single virtual register.

Example

Example B-266

```
int size;
size = ral_model.virtual_register.get_n_memlocs();
```

vmm_ral_vreg::get_incr()

Returns the distance between two virtual registers.

SystemVerilog

```
virtual function int unsigned get_incr()
```

Description

Returns the number of memory locations between two individual virtual registers in the same array.

Example

Example B-267

```
int dist;
dist = ral_model.virtual_register.get_incr();
```

vmm_ral_vreg::display()

Displays a description of the virtual register to stdout.

SystemVerilog

```
virtual function void display(string prefix = "",
                                string domain = "")
```

Description

Displays the image created by the "`vmm_ral_vreg::psdisplay()`" method to the standard output.

Example

Example B-268

```
    ral_model.virtual_register.display();
```

vmm_ral_vreg::psdisplay()

Creates a human-readable description of the virtual register.

SystemVerilog

```
virtual function string psdisplay(string prefix = "",
                                     string domain = "")
```

Description

Creates a human-readable description of the register and the fields it contains. Each line of the description is prefixed with the specified prefix.

If a domain is specified, the address of the register within that domain is used.

Example

Example B-269

```
`vmm_note(log, $psprintf("Virtual Register description =
%s\n", ral_model.virtual_register.psdisplay()));
```

vmm_ral_vreg::get_fields()

Returns all virtual fields in this virtual register.

SystemVerilog

```
virtual function void get_fields(
   ref vmm_ral_vfield fields[])
```

Description

Fills the specified dynamic array with the descriptor for all of the virtual fields contained in the virtual register. Fields are ordered from least-significant position to most-significant position within the register.

Example

Example B-270

```
vmm_ral_vfield vfields[];
ral_model.virtual_register.get_fields(vfields);
```

vmm_ral_vreg::get_field_by_name()

Returns the virtual field with the specified name in this virtual register.

SystemVerilog

```
virtual function vmm_ral_vfield get_field_by_name(
   string name)
```

Description

Finds a virtual field with the specified name in the register and returns its descriptor. If no fields are found, returns `null`.

Example

Example B-271

```
  vmm_ral_vfield vfield;
  vfield =
ral_model.virtual_register.get_field_by_name("virtual_fiel
d_name");
  if (vfield == null) `vmm_error(log, "specified field
doesn't exists");
```

vmm_ral_vreg::read()

Reads a virtual register from the design.

SystemVerilog

```
virtual task read(
   input  longint unsigned idx,
   output vmm_rw::status_e status,
   output bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int                stream_id = -1)
```

Description

Reads the current value of the specified virtual register from the design using the specified access path. If the memory implementing the virtual register is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data` class properties in the `“vmm_rw_access”` transaction descriptors that are necessary to execute this read operation. This allows the physical and back-door read accesses to be traced back to the higher-level transaction that caused the access to occur.

Example

Example B-272

```
vmm_rw::status_e status;
bit [`VMM_RAL_DATA_WIDTH-1:0] data;
ral_model.virtual_reg.read(0,status,data);
```

vmm_ral_vreg::write()

Sets a register in the design.

SystemVerilog

```
virtual task write(
   input  longint unsigned idx,
   output vmm_rw::status_e status,
   input  bit [63:0]       value,
   input  vmm_ral::path_e  path = vmm_ral::DEFAULT,
   input  string           domain = "",
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int                stream_id = -1)
```

Description

Writes the specified value in the specified virtual register in the design using the specified access path. If the memory implementing the virtual register is shared by more than one physical interface, a domain must be specified if a physical access is used (front-door access).

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method or used to set the corresponding `vmm_data` class properties in the “`vmm_rw_access`” transaction descriptors that are necessary to execute this write operation. This allows the physical and back-door write accesses to be traced back to the higher-level transaction that caused the access to occur.

Example

Example B-273

```
vmm_rw::status_e status;
ral_model.virtual_reg.write(0,status,'hABCD);
```

vmm_ral_vreg::peek()

Peek a virtual register from the design.

SystemVerilog

```
virtual task peek(
   input  longint unsigned idx,
   output vmm_rw::status_e status,
   output bit [63:0]       value,
   input  int              data_id = -1,
   input  int              scenario_id = -1,
   input  int                stream_id = -1)
```

Description

Reads the current value of the specified virtual register from the design using a back-door access. The memory implementing the virtual register must provide a back-door access.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door read accesses to be traced back to the higher-level transaction that caused the access to occur.

Example

Example B-274

```
vmm_rw::status_e status;
bit [`VMM_RAL_DATA_WIDTH-1:0] data;
ral_model.virtual_reg.peek(0,status,data);
```

vmm_ral_vreg::poke()

Poke a virtual register in the design.

SystemVerilog

```
virtual task poke(
   input  ilongint unsigned idx,
   output vmm_rw::status_e  status,
   input  bit [63:0]        value,
   input  int               data_id = -1,
   input  int               scenario_id = -1,
   input  int                  stream_id = -1)
```

Description

Deposit the specified value in the specified virtual register in the design, as-is, using a back-door access. The memory implementing the virtual register must provide a back-door access.

The optional value of the arguments:

```
data_id
scenario_id
stream_id
```

...are passed to the back-door access method. This allows the physical and back-door write accesses to be traced back to the higher-level transaction that caused the access to occur.

Example

Example B-275

```
vmm_rw::status_e status;
ral_model.virtual_reg.poke(0,status,'hABCD);
```

vmm_ral_vreg::append_callback()

Appends a callback extension instance.

SystemVerilog

```
function void append_callback(
   vmm_ral_vreg_callbacks cbs)
```

Description

Appends the specified callback extension instance to the registered callbacks for this virtual register descriptor. Callbacks are invoked in the order of registration.

Note that the corresponding “`vmm_ral_vfield`” callback methods will be invoked before the virtual register callback methods.

Example

Example B-276

```
  vreg_callback cb = new;
  this.ral_model.blk.magic_vreg.append_callback(cb);

//See "vmm_ral_vreg_callbacks" classes for complete
//definition of vreg_callback.
```

vmm_ral_vreg::prepend_callbacks()

Prepends a callback extension instance.

SystemVerilog

```
function void prepend_callbacks(
   vmm_ral_vreg_callbacks cbs)
```

Description

Prepends the specified callback extension instance to the registered callbacks for this register descriptor. Callbacks are invoked in the reverse order of registration.

Note that the corresponding “`vmm_ral_vfield`” callback methods will be invoked before the virtual register callback methods.

Example

Example B-277

```
  vreg_callback cb = new;
  this.ral_model.blk.magic_vreg.prepend_callback(cb);

// See "vmm_ral_vreg_callbacks" classes for complete
// definition of vreg_callback.
```

vmm_ral_vreg::unregister_callbacks()

Removes a callback extension instance.

SystemVerilog

```
function void unregister_callbacks(
   vmm_ral_vreg_callbacks cbs)
```

Description

Removes the specified callback extension instance from the registered callbacks for this register descriptor. A warning message is issued if the callback instance has not been previously registered.

Example

Example B-278

```
vreg_callback cb = new;
  this.ral_model.blk.magic_vreg.unregister_callback(cb);

//See "vmm_ral_vreg_callbacks" classes for the definition
//of vreg_callback.
```

vmm_ral_vreg_callbacks

Base class for virtual register descriptors.

Summary

vmm_ral_vreg_callbacks::pre_write()

OOP callback invoked before writing a register.

SystemVerilog

```
virtual task pre_write(vmm_ral_vreg        rg,
                       longint unsigned    idx,
                       ref bit [63:0]      wdat,
                       ref vmm_ral::path_e path,
                         ref string            domain)
```

Description

This callback method is invoked before a value is written to a virtual register in the DUT. The written value, if modified, changes the actual value that is written. The path and domain used to write to the register can also be modified.

This callback method is only invoked when the "vmm_ral_vreg::write()" method is used to write to the register inside the DUT. This callback method is not invoked when the memory location implementing a virtual register, is written to using the "vmm_ral_mem::write()" method.

Because writing a register causes all of the fields it contains to be written, all registered "vmm_ral_vfield_callbacks::pre_write()" methods with the fields contained in the register will also be invoked before all registered register callback methods. Because the memory implementing the virtual field is accessed through its own abstraction class, all of its registered "vmm_ral_mem_callbacks::pre_write()" methods will also be invoked as a side effect.

Example

Example B-279

```
class vreg_callback extends vmm_ral_vreg_callbacks;

  task display_path_e(ref vmm_ral::path_e path);
               if (path == vmm_ral::BFM) $display("Path =
vmm_ral::BFM\n");
         else if (path == vmm_ral::BACKDOOR) $display("Path
= vmm_ral::BACKDOOR\n");
          else if (path == vmm_ral::DEFAULT) $display("Path
= vmm_ral::DEFAULT\n");
               else $display("ERROR: Invlaid path_e (%d)
found\n", path);
  endtask

  virtual task pre_write(vmm_ral_vreg                   rg,
                         longint unsigned              idx,
                      ref bit [`VMM_RAL_DATA_WIDTH-1:0] wdat,
                        ref vmm_ral::path_e           path,
                       ref string                   domain);
              $display("vreg_callback: Executing Callback
task pre_write for ...\n");
               $display("Register = %s\n Index = %d\n",
rg.get_name(), idx);
               $display("Writing Data = %d'h%x\n",
`VMM_RAL_DATA_WIDTH, wdat);
               display_path_e(path);
               $display("Domain = %s", domain);
  endtask: pre_write

endclass
```

vmm_ral_vreg_callbacks::post_write()

OOP callback invoked after writing a register.

SystemVerilog

```
virtual task post_write(vmm_ral_vreg           rg,
                        longint unsigned       idx,
                        bit [63:0]             wdat,
                        vmm_ral::path_e        path,
                        string                 domain
                          ref vmm_rw::status_e    status)
```

Description

This callback method is invoked after a value is successfully written to a register in the DUT. If a physical write access did not return `vmm_rw::IS_OK`, this method is not called.

This callback method is only invoked when the "`vmm_ral_vreg::write()`" method is used to write to the register inside the DUT. This callback method is not invoked when the memory location implementing a virtual register, is written to using the "`vmm_ral_mem::write()`" method.

Because writing a register causes all of the fields it contains to be written, all registered "`vmm_ral_vfield_callbacks::post_write()`" methods with the fields contained in the register will also be invoked before all registered register callback methods. Because the memory implementing the virtual field is accessed through its own abstraction class, all of its registered "`vmm_ral_mem_callbacks::post_write()`" methods will also be invoked as a side effect.

Example

Example B-280

```
class vreg_callback extends vmm_ral_vreg_callbacks;

  task display_path_e(ref vmm_ral::path_e path);
               if (path == vmm_ral::BFM) $display("Path =
vmm_ral::BFM\n");
         else if (path == vmm_ral::BACKDOOR) $display("Path
= vmm_ral::BACKDOOR\n");
          else if (path == vmm_ral::DEFAULT) $display("Path
= vmm_ral::DEFAULT\n");
               else $display("ERROR: Invlaid path_e (%d)
found\n", path);
  endtask

  virtual task post_write(vmm_ral_vreg                 rg,
                         longint unsigned             idx,
                       bit [`VMM_RAL_DATA_WIDTH-1:0] wdat,
                        vmm_ral::path_e              path,
                       string                       domain,
                      ref vmm_rw::status_e        status);
              $display("vreg_callback: Executing Callback
task post_write for ...\n");
               $display("Register = %s\n Index = %d\n",
rg.get_name(), idx);
               $display("Wrote Data = %d'h%x\n",
`VMM_RAL_DATA_WIDTH, wdat);
               display_path_e(path);
               $display("Domain = %s", domain);
  endtask: post_write

endclass
```

vmm_ral_vreg_callbacks::pre_read()

OOP callback invoked before reading a virtual register.

SystemVerilog

```
virtual task pre_read(vmm_ral_vreg        rg,
                      longint unsigned    idx,
                      ref vmm_ral::path_e path,
                        ref string            domain)
```

Description

This callback method is invoked before a value is read from a register in the DUT. The path and domain used to read the register can be modified.

This callback method is only invoked when the "vmm_ral_vreg::read()" method is used to read to the register inside the DUT. This callback method is not invoked when the memory location implementing a virtual register, is read to using the "vmm_ral_mem::read()" method.

Because reading a register causes all of the fields it contains to be written, all registered "vmm_ral_vfield_callbacks::pre_read()" methods with the fields contained in the register will also be invoked before all registered register callback methods. Because the memory implementing the virtual field is accessed through its own abstraction class, all of its registered "vmm_ral_mem_callbacks::pre_read()" methods will also be invoked as a side effect.

Example

Example B-281

```
class vreg_callback extends vmm_ral_vreg_callbacks;

  task display_path_e(ref vmm_ral::path_e path);
               if (path == vmm_ral::BFM) $display("Path = 
vmm_ral::BFM\n");
         else if (path == vmm_ral::BACKDOOR) $display("Path 
= vmm_ral::BACKDOOR\n");
          else if (path == vmm_ral::DEFAULT) $display("Path 
= vmm_ral::DEFAULT\n");
               else $display("ERROR: Invlaid path_e (%d) 
found\n", path);
  endtask

  virtual task pre_read(vmm_ral_vreg         rg,
                        longint unsigned     idx,
                        ref vmm_ral::path_e  path,
                        ref string           domain);
              $display("vreg_callback: Executing Callback 
task pre_read for ...\n");
               $display("Register = %s\n Index = %d\n", 
rg.get_name(), idx);
               display_path_e(path);
               $display("Domain = %s", domain);
  endtask: pre_read

endclass
```

vmm_ral_vreg_callbacks::post_read()

OOP callback invoked after reading a virtual register.

SystemVerilog

```
virtual task post_read(input vmm_ral_vreg      rg,
                       input longint unsigned  idx,
                       ref   bit [63:0]        rdat,
                       input vmm_ral::path_e   path,
                       input string            domain
                         ref   vmm_rw::status_e    status)
```

Description

This callback method is invoked after a value is successfully read from a register in the DUT. The `rdat` and `status` values are the values that will be ultimately returned by the `"vmm_ral_vreg::read()"` method and can be modified. If a physical read access did not return `vmm_rw::IS_OK`, this method is not called.

This callback method is only invoked when the `"vmm_ral_vreg::read()"` method is used to read to the register inside the DUT. This callback method is not invoked when the memory location implementing a virtual register, is read to using the `"vmm_ral_mem::read()"` method.

Because reading a register causes all of the fields it contains to be written, all registered `"vmm_ral_vfield_callbacks::post_read()"` methods with the fields contained in the register will also be invoked before all registered register callback methods. Because the memory implementing the virtual field is accessed through its own abstraction class, all of its registered s methods will also be invoked as a side effect.

Example

Example B-282

```
class vreg_callback extends vmm_ral_vreg_callbacks;

  task display_path_e(ref vmm_ral::path_e path);
               if (path == vmm_ral::BFM) $display("Path =
vmm_ral::BFM\n");
         else if (path == vmm_ral::BACKDOOR) $display("Path
= vmm_ral::BACKDOOR\n");
          else if (path == vmm_ral::DEFAULT) $display("Path
= vmm_ral::DEFAULT\n");
               else $display("ERROR: Invlaid path_e (%d)
found\n", path);
  endtask

  virtual task post_read(vmm_ral_vreg                  rg,
                     longint unsigned              idx,
                   ref bit [`VMM_RAL_DATA_WIDTH-1:0] rdat,
                    input vmm_ral::path_e          path,
                    input string                 domain,
                   ref vmm_rw::status_e          status);
              $display("vreg_callback: Executing Callback
task post_read for ...\n");
               $display("Register = %s\n Index = %d\n",
rg.get_name(), idx);
               $display("Read Data = %d'h%x\n",
`VMM_RAL_DATA_WIDTH, rdat);
               display_path_e(path);
               $display("Domain = %s", domain);
  endtask: post_read

endclass
```

vmm_rw

Utility class for global symbolic values. Each set of symbolic values is specified using enumerated types. The symbolic values are accessed using a fully-qualified name, such as `vmm_rw::IS_OK`.

A separate, encapsulated class is used to minimize the length of these identifiers and to make them easier to share across classes.

Summary

vmm_rw::kind_e

Symbolic values identifying the kind of transaction to perform on a physical interface.

SystemVerilog

```
vmm_rw::READ
vmm_rw::WRITE
vmm_rw::EXPECT
```

OpenVera

```
vmm_rw::READ
vmm_rw::WRITE
vmm_rw::EXPECT
```

Description

`vmm_rw::READ`

Performs a read operation from a specified address and returns a value.

`vmm_rw::WRITE`

Performs a write operation of a specified value at a specified address.

`vmm_rw::EXPECT`

Performs a read operation from a specified address and compares the value read against a specified value.

Example

Example B-283

```
class my_rw_master extends vmm_rw_xactor;
   ...
   my_data data;
   virtual task execute_single(vmm_rw_access tr);
      ...
      if(tr.kind == vmm_rw::READ)
         data.addr == tr.addr << 2;
         ...
      ...
   endtask
   ...
endclass
```

vmm_rw::status_e

Symbolic values identifying the completion status of a transaction on a physical interface.

SystemVerilog

```
vmm_rw::IS_OK
vmm_rw::ERROR
vmm_rw::RETRY
```

OpenVera

```
vmm_rw::IS_OK
vmm_rw::ERROR
vmm_rw::RETRY
```

Description

`vmm_rw::IS_OK`

The transaction completed successfully.

`vmm_rw::ERROR`

The transaction did not complete or completed with an error indication.

`vmm_rw::RETRY`

The transaction completed with a `retry` indication.

Example

Example B-284

```
class my_rw_master extends vmm_rw_xactor;
   ...
   virtual task execute_single(vmm_rw_access tr);
      my_data data;
      ...
      data = new();
      ...
      if(data.status == my_data::OK)
         tr.status = vmm_rw::IS_OK;
      ...
   endtask
   ...
endclass
```

vmm_rw_access

Descriptor for generic bus transactions. Derived from `vmm_data/rvm_data`.

Summary

vmm_rw_access::kind

Defines the type of transaction described by this instance.

SystemVerilog

```
rand vmm_rw::kind_e kind
```

OpenVera

```
rand vmm_rw::kind_e kind
```

Description

Identifies the type of transaction that is described by this generic bus transaction descriptor instance.

Example

Example B-285

```
class my_rw_access extends vmm_rw_access;
   ...
   constraint valid_vmm_rw_access {
     kind == vmm_rw::READ;
   }
   ...
endclass
`vmm_channel(my_rw_access)
...
my_rw_access tr;
`vmm_note(log,$psprintf("vmm_rw_access::kind==>
%0d",tr.kind));
...
```

vmm_rw_access::addr

Address to read from or write to.

SystemVerilog

```
rand bit [63:0] addr
```

OpenVera

```
rand bit [63:0] addr
```

Description

Address that is the target of the transaction. If it is a `read` transaction, the specified address is read. If it is a `write` transaction, the specified address is written to.

Example

Example B-286

```
class my_rw_access extends vmm_rw_access;
   ...
   constraint valid_vmm_rw_access {
      addr > 'h0;
      addr <= 'hFFFF;
   }
   ...
endclass
...
my_rw_access tr;
...
`vmm_note(log,$psprintf("vmm_rw_access::addr==>
%0d",tr.addr));
...
```

vmm_rw_access::data

Data to write or that has been read.

SystemVerilog

```
rand bit [63:0] data
```

OpenVera

```
rand bit [63:0] data
```

Description

If it is a read transaction, it is the value that was read at the specified address. If it is a write transaction, the specified value is written at the specified address.

Example

Example B-287

```
class my_rw_access extends vmm_rw_access;
   ...
   constraint valid_vmm_rw_access {
      data = addr + 1;
   }
   ...
endclass
...
my_rw_access tr;
...
`vmm_note(log,$psprintf("vmm_rw_access::data==>
%0d",tr.data));
...
```

vmm_rw_access::n_bits

Number of valid data bits.

SystemVerilog

```
rand int n_bits
```

OpenVera

```
rand int n_bits
```

Description

Specifies the number of valid bits in the read or write cycle. The valid bits are always right justified.

This class property is constrained to be in the 1..64 range.

Example

Example B-288

```
class my_rw_access extends vmm_rw_access;
   ...
   constraint valid_vmm_rw_access {
      n_bits > 0;
      n_bits < 10;
   }
   ...
endclass
...
my_rw_access tr;
...
`vmm_note(log,$psprintf("vmm_rw_access::nbits==>
%0d",tr.n_bits));
...
```

vmm_rw_access::status

Completion status of the transaction.

SystemVerilog

```
vmm_rw::status_e status
```

OpenVera

```
vmm_rw::status_e status
```

Description

Specifies the completion status of the transaction.

Example

Example B-289

```
   ...
   my_rw_access tr;
   `vmm_note(log,$psprintf("vmm_rw_access::status==>
%0d",tr.status));
   ...
```

vmm_rw_access::new()

Creates a new instance of a transaction descriptor.

SystemVerilog

```
function new()
```

OpenVera

```
task new()
```

Description

Creates a new instance of a transaction descriptor.

Example

Example B-290

```
class my_rw_access extends vmm_rw_access;
   vmm_log log
   ...
   function new();
      super.new(this.log);
      log = new("my_rw_access","class");
      `vmm_note(log,"vmm_rw_access:: instance created.");
   endfunction
   ...
endclass
```

vmm_rw_access::psdisplay()

Creates an image representing the described transaction.

SystemVerilog

```
virtual function string psdisplay(string prefix = "")
```

OpenVera

```
virtual function string psdisplay(string prefix = "")
```

Description

Creates a human-readable image of the content of the transaction descriptor. Every line in the image is prefixed by the specified prefix. The image is returned as a string terminated with a newline.

Example

Example B-291

```
class my_rw_access extends vmm_rw_access;
   ...
  //Creates an image representing the described transaction.
   `vmm_note(log,$psprintf("%0s",this.psdisplay("RW 
Access")));
   ...
endclass
```

vmm_rw_burst

Descriptor for generic bus burst transactions. Derived from `“vmm_rw_access”`.

Summary

vmm_rw_burst::n_beats

Defines the length of the burst.

SystemVerilog

```
rand int n_beats
```

OpenVera

```
rand integer n_beats
```

Description

Specifies the number of beats or transfers in a burst transaction that is described by this generic burst transaction descriptor instance.

Example

Example B-292

```
class my_rw_burst extends vmm_rw_burst;
   ...
   constraint reasonable {
      n_beats <= 1024;
   }
   ...
endclass
...
my_rw_burst br;
`vmm_note(log,$psprintf("vmm_ral_mem_burst::n_beats ==> 
%0d",br.n_beats));
...
```

vmm_rw_burst::incr_addr

Address increment between beats.

SystemVerilog

```
rand bit [63:0] incr_addr
```

OpenVera

```
rand bit [63:0] incr_addr
```

Description

Implicit or explicit address increment between individual beats of a burst transaction. The first beat reads or writes the address specified by the `“vmm_rw_access::addr”` class property. The *n*th beat reads or writes the address specified by `“vmm_rw_access::addr”` + (*n*-1) `“vmm_rw_burst::incr_addr”`.

A value of `0` implies a burst access that repeatedly accessed the same location.

Example

Example B-293

```
class my_rw_burst extends vmm_rw_burst;
   ...
   constraint reasonable {
      incr_addr inside {0, 1, 2, 4, 8, 16, 32};
   }
   ...
endclass
...
```

```
my_rw_burst br;
`vmm_note(log,$psprintf("vmm_ral_mem_burst::incr_addr ==>
%0d",br.incr_addr));
...
```

vmm_rw_burst::max_addr

Maximum address for the burst.

SystemVerilog

```
rand bit [63:0] max_addr
```

OpenVera

```
rand bit [63:0] max_addr
```

Description

Limits the address range of the burst and causes wrapping if subsequent beats would access an address past the specified address.

Example

Example B-294

```
class my_rw_burst extends vmm_rw_burst;
   ...
   constraint reasonable {
      n_beats <= 1024;
      incr_addr inside {0, 1, 2, 4, 8, 16, 32};
   }

   constraint linear {
      incr_addr == 1;
      max_addr == addr + n_beats - 1;
   }
   ...
endclass
...
my_rw_burst br;
```

```
`vmm_note(log,$psprintf("vmm_ral_mem_burst::max_addr==>
%0d",br.max_addr));
...
```

vmm_rw_burst::data

Data to write or that has been read.

SystemVerilog

```
rand bit [63:0] data[]
```

OpenVera

```
rand bit [63:0] data[*]
```

Description

If it is a read transaction, it is the values that were read during the burst-read operation. If it is a write transaction, the specified values are written during the burst-write operation.

The number of elements in the array must be equal to the number of beats specified by “`vmm_rw_burst::n_beats`”.

Important: this class property hides “`vmm_rw_access::data`”.

Example

Example B-295

```
class my_rw_burst extends vmm_rw_burst;
   ...
   constraint vmm_rw_burst_valid {
      n_beats > 0;
      data.size() == n_beats;
   }
   ...
endclass
...
my_rw_burst br;
```

```
foreach(br.data[i])
  `vmm_note(log,$psprintf("vmm_ral_mem_burst::data ==>
%0h",br.data[i]));
...
```

vmm_rw_burst::user_data

Additional burst configuration information.

SystemVerilog

```
vmm_data user_data
```

OpenVera

```
rvm_data user_data
```

Description

Provides a mechanism for passing additional burst configuration information to the “`vmm_rw_xactor::execute_burst()`” method. Any reference to additional user information is passed through transparently. The additional information can be recovered by using `$cast()` or `cast_assign()`.

Example

Example B-296

```
class my_rw_xactor extends vmm_rw_xactor;
   ...
   my_rw_burst burst;
   virtual protected task execute_burst(vmm_rw_burst br)
     ...
     $cast(burst.user_data,br);
     ...
   endtask
   ...
endclass
```

vmm_rw_xactor

Base class, derived from `vmm_xactor/rvm_xactor`, for bus-functional models executing generic bus transactions described using `vmm_rw_access` descriptors.

Summary

vmm_rw_xactor::exec_chan

Transaction execution input channel.

SystemVerilog

```
vmm_rw_access_channel exec_chan
```

OpenVera

```
vmm_rw_access_channel exec_chan
```

Description

This channel is used to supply a stream of generic bus transaction descriptors to be executed on a physical interface. This channel implements a blocking completion model. Once the transaction has been executed, data and completion status information is back-annotated in the transaction descriptor.

Not usually used by RAL users and is used by “`vmm_ral_access`” instances for executing physical accesses to read and write registers or memory locations in the design.

Example

Example B-297

```
class my_rw_access extends vmm_rw_access;
   ...
endclass
`vmm_channel(my_rw_access)
...
class my_rw_xactor extends vmm_rw_xactor;
   ...
   my_rw_access_channel exec_chan;
```

```
      ...
      function new(string                inst,
                   int unsigned          stream_id,
                   my_rw_access_channel exec_chan = null);
         super.new("RAL Master", inst, stream_id, exec_chan);
         ...
      endfunction
      ...
   endclass
```

vmm_rw_xactor::new()

Creates a new instance of the base class.

SystemVerilog

```
function new(
   string               name,
   string               instance,
   int                  stream_id = -1,
   vmm_rw_access_channel exec_chan = null)
```

OpenVera

```
task new(
   string               name,
   string               instance,
   int                  stream_id = -1,
   vmm_rw_access_channel exec_chan = null)
```

Description

Creates a new instance of this base class, with the specified name and instance name and optional stream identifier. If a channel is specified, it is assigned to the `vmm_ral_xactor::exec_chan` class property and reconfigured with a full level of `1`. Otherwise, a new channel instance is allocated and used.

Example

Example B-298

```
function ral_wb_master::new(string name,
                            string instance,
                            int    stream_id = -1);
   super.new(name, instance, stream_id);
endfunction: new
```

vmm_rw_xactor::execute_burst()

Executes a burst transfer transaction.

SystemVerilog

```
virtual protected task execute_burst(vmm_rw_burst tr)
```

OpenVera

```
virtual protected task execute_burst_t(vmm_rw_burst tr)
```

Description

This method may be overloaded. It is called by the base class to request that a generic burst bus transaction be executed on a physical interface. Once the transaction has been executed, data and completion status information is back-annotated in the transaction descriptor then the task should return.

By default, burst transactions are executed as a series of single transactions, using `“vmm_rw_xactor::execute_single()”`. If this transactor is mapped to a physical protocol that supports burst transactions, this method should be overloaded.

This method is ultimately used to execute the following operations:

`“vmm_ral_mem::burst_read()”`
`“vmm_ral_mem::burst_write()”`
`“vmm_ral_access::burst_read()”`
`“vmm_ral_access::burst_write()”`

Example

Example B-299

```
class my_rw_master extends vmm_rw_xactor;
   ...
   function new(string                inst,
                int unsigned          stream_id,
                my_rw_access_channel exec_chan = null);
      super.new("RW Master", inst, stream_id, exec_chan);
      ...
   endfunction
   ...
   task execute_burst(vmm_rw_burst tr);
      bit [`VMM_RW_ADDR_WIDTH-1:0] addr;
      addr = tr.addr;
      tr.status = vmm_rw::IS_OK;
      ...
   endtask
   ...
endclass
...
my_rw_burst br;
my_rw_master my_xactor;
...
this.my_xactor.execute_single(tr);
...
```

vmm_rw_xactor::execute_single()

Executes a single transfer transaction.

SystemVerilog

```
virtual protected task execute_single(vmm_rw_access tr)
```

OpenVera

```
virtual protected task execute_single_t(vmm_rw_access tr)
```

Description

This method must be overloaded. It is called by the base class to request that a generic bus transaction be executed on a physical interface. Once the transaction has been executed, data and completion status information is back-annotated in the transaction descriptor then the task should return.

Example

Example B-300

```
class my_rw_master extends vmm_rw_xactor;
   ...
   my_rw_access_channel exec_chan;
   ...
   virtual task execute_single(vmm_rw_access tr);
      wb_cycle cyc;
      ...
      cyc = new();
      ...
   endtask
   ...
endclass
...
```

```
my_rw_access tr;
my_rw_master my_xactor;
...
this.my_xactor.execute_single(tr);
...
```

vmm_rw_xactor::notifications_e

Symbolic values identifying notifications indicated by the transactor.

SystemVerilog

```
vmm_rw_xactor::BURST_DONE
vmm_rw_xactor::SINGLE_DONE
```

OpenVera

```
vmm_rw_xactor::BURST_DONE
vmm_rw_xactor::SINGLE_DONE
```

Description

`vmm_rw_xactor::BURST_DONE`

Indicates that a burst cycle has been completed. The burst cycle descriptor is available as the status of the indication.

`vmm_rw_xactor::SINGLE_DONE`

Indicates that a single cycle has been completed. The single cycle descriptor is available as the status of the indication.

Example

Example B-301

```
class my_rw_master extends vmm_rw_xactor;
   ...
   typedef enum {BURST_DONE = 99990,SINGLE_DONE}
notifications_e;
   ...
   function new(string                inst,
                int unsigned          stream_id,
                my_rw_access_channel exec_chan = null);
```

```
      super.new("RW Master", inst, stream_id, exec_chan);
   endfunction
   ...
endclass
```

vmm_rw_xactor::pre_single()

AOP callback invoked before the execution of a single cycle.

SystemVerilog

```
N/A
```

OpenVera

```
task pre_single_t(vmm_rw_access tr)
```

Description

This callback method is invoked before the execution of a single cycle, before the corresponding OOP callback method is invoked, and before the `vmm_rw_xactor::execute_single()` method is invoked. The transaction descriptor, if modified, changes the transaction that is executed.

Example

Example B-302

```
TBD
```

vmm_rw_xactor::pre_burst()

AOP callback invoked before the execution of a burst cycle.

SystemVerilog

```
N/A
```

OpenVera

```
task pre_burst_t(vmm_rw_access tr)
```

Description

This callback method is invoked before the execution of a burst cycle, before the corresponding OOP callback method is invoked, and before the `“vmm_rw_xactor::execute_burst()”` method is invoked. The transaction descriptor, if modified, changes the transaction that is executed.

Example

Example B-303

```
TBD
```

vmm_rw_xactor::post_single()

AOP callback invoked after the execution of a single cycle.

SystemVerilog

```
N/A
```

OpenVera

```
task post_single_t(vmm_rw_access tr)
```

Description

This callback method is invoked after the execution of a single cycle, after the "`vmm_rw_xactor::execute_single()`" method has returned, but before the corresponding OOP callback method is invoked. The transaction descriptor must not be modified.

Example

Example B-304

```
TBD
```

vmm_rw_xactor::post_burst()

AOP callback invoked after the execution of a burst cycle.

SystemVerilog

```
N/A
```

OpenVera

```
task post_burst_t(vmm_rw_access tr)
```

Description

This callback method is invoked after the execution of a burst cycle, after the “`vmm_rw_xactor::execute_burst()`” method has returned, but before the corresponding OOP callback method is invoked. The transaction descriptor must not be modified.

Example

Example B-305

```
TBD
```

vmm_rw_xactor_callbacks

Summary

vmm_rw_xactor_callbacks::pre_single()

OOP callback invoked before the execution of a single cycle.

SystemVerilog

```
virtual task pre_single(vmm_rw_xactor xact,
                        vmm_rw_access tr)
```

OpenVera

```
virtual task pre_single_t(vmm_rw_xactor xact,
                          vmm_rw_access tr)
```

Description

This callback method is invoked before the execution of a single cycle, before the “`vmm_rw_xactor::execute_single()`” method is invoked. The transaction descriptor, if modified, changes the transaction that is executed.

Example

Example B-306

```
class my_xactor extends vmm_rw_xactor;
   ...
   my_data data; //my_data extends from vmm_rw_access
   this.wb.exec_chan.put(data);
   //This transaction object "data" is not a burst so it
main() of vmm_rw_xactor
   //will call execute_single() method and pre_single() &
post_single() Methods
   //will be invoked.

   virtual function void start_xactor();
      super.start_xactor();
      this.wb.start_xactor();
      ...
   endfunction: start_xactor
   ...
endclass
...
program test;
...
  class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
      ...
      virtual task pre_single(vmm_rw_xactor xact,
                              vmm_rw_access tr);
         `vmm_note(log,{"pre_single method is called for
vmm_ral_xactor_",
                  "callbacks class"});
      endtask
      ...
   endclass
   ...
   env.my_xactor.append_callback(cb);
   ...
endprogram
```

vmm_rw_xactor_callbacks::pre_burst()

OOP callback invoked before the execution of a burst cycle.

SystemVerilog

```
virtual task pre_burst(vmm_rw_xactor xact,
                       vmm_rw_access tr)
```

OpenVera

```
virtual task pre_burst_t(vmm_rw_xactor xact,
                         vmm_rw_access tr)
```

Description

This callback method is invoked before the execution of a burst cycle, before the “vmm_rw_xactor::execute_burst()” method is invoked. The transaction descriptor, if modified, changes the transaction that is executed.

Example

Example B-307

```
class my_xactor extends vmm_rw_xactor;
   ...
   my_rw_burst burst;
   this.wb.exec_chan.put(burst);
  //This transaction object "burst" is burst type instance
so it main() of
   //vmm_rw_xactor will call execute_burst() method and
pre_burst() & post_burst()
   //Methods will be invoked.
   ...
endclass
...
```

```
program test;
...
  class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
      ...
      virtual task pre_burst(vmm_rw_xactor xact,
                             vmm_rw_access tr);
         `vmm_note(log,{"pre_burst method is called for
vmm_ral_xactor_",
                   "callbacks class"});
      endtask
      ...
   endclass
   ...
   env.my_xactor.append_callback(cb);
   ...
endprogram
```

vmm_rw_xactor_callbacks::post_single()

OOP callback invoked after the execution of a single cycle.

SystemVerilog

```
virtual task post_single(vmm_rw_xactor xact,
                        vmm_rw_access tr)
```

OpenVera

```
virtual task post_single_t(vmm_rw_xactor xact,
                          vmm_rw_access tr)
```

Description

This callback method is invoked after the execution of a single cycle, after the “`vmm_rw_xactor::execute_single()`” method has returned. The transaction descriptor must not be modified.

Example

Example B-308

```
...
program test;
...
  class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
      ...
      virtual task post_single(vmm_rw_xactor xact,
                               vmm_rw_access tr);
         `vmm_note(log,{"post_single method is called for
vmm_ral_xactor_",
                   "callbacks class"});
      endtask
      ...
   endclass
   ...
```

```
      env.my_xactor.append_callback(cb);
      ...
   endprogram
```

vmm_rw_xactor_callbacks::post_burst()

OOP callback invoked after the execution of a burst cycle.

SystemVerilog

```
virtual task post_burst(vmm_rw_xactor xact,
                        vmm_rw_access tr)
```

OpenVera

```
virtual task post_burst_t(vmm_rw_xactor xact,
                          vmm_rw_access tr)
```

Description

This callback method is invoked after the execution of a burst cycle, after the “`vmm_rw_xactor::execute_burst()`” method has returned. The transaction descriptor must not be modified.

Example

Example B-309

```
program test;
...
  class my_ral_mem_callbacks extends vmm_ral_mem_callbacks;
      ...
      virtual task post_burst(vmm_rw_xactor xact,
                              vmm_rw_access tr);
         `vmm_note(log,{"post_burst method is called for 
vmm_ral_xactor_",
                   "callbacks class"});
      endtask
      ...
   endclass
   ...
   env.my_xactor.append_callback(cb);
```

```
      ...
   endprogram
```