Universal Verification Methodology (UVM) 1.0 Class Reference

February 2011

Notices

Accellera Standards documents are developed within Accellera and the Technical Committees of Accellera Organization, Inc. Accellera develops its standards through a consensus development process, approved by its members and board of directors, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of Accellera and serve without compensation. While Accellera administers the process and establishes rules to promote fairness in the consensus development process, Accellera does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an Accellera Standard is wholly voluntary. Accellera disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other Accellera Standard document.

Accellera does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or suitability for a specific purpose, or that the use of the material contained herein is free from patent infringement. Accellera Standards documents are supplied "**AS IS**."

The existence of an Accellera Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of an Accellera Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change due to developments in the state of the art and comments received from users of the standard. Every Accellera Standard is subjected to review periodically for revision and update. Users are cautioned to check to determine that they have the latest edition of any Accellera Standard.

In publishing and making this document available, Accellera is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is Accellera undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other Accellera Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of Accellera, Accellera will initiate action to prepare appropriate responses. Since Accellera Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, Accellera and the members of its Technical Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments for revision of Accellera Standards are welcome from any interested party, regardless of membership affiliation with Accellera. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Accellera Organization
1370 Trancas Street #163
Napa, CA 94558
USA

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. Accellera shall not be responsible for identifying patents for which a license may be required by an Accellera standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Accellera is the sole entity that may authorize the use of Accellera-owned certification marks and/or trademarks to indicate compliance with the materials set forth herein.

Authorization to photocopy portions of any individual standard for internal or personal use must be granted by Accellera Organization, Inc., provided that permission is obtained from and any required fee is paid to Accellera. To arrange for authorization please contact Lynn Horobin, Accellera, 1370 Trancas Street #163, Napa, CA 94558, phone (707) 251-9977, e-mail lynn@accellera.org. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained from Accellera.

Suggestions for improvements to the UVM 1.0 Class Reference are welcome. They should be sent to the VIP email reflector

vip-tc@lists.accellera.org

The current Working Group's website address is

www.accellera.org/activities/vip

Contents

Overview

Verification has evolved into a complex project that often spans internal and external teams, but the discontinuity associated with multiple, incompatible methodologies among those teams has limited productivity. The Universal Verification Methodology (UVM) 1.0 Class Reference addresses verification complexity and interoperability within companies and throughout the electronics industry for both novice and advanced teams while also providing consistency. While UVM is revolutionary, being the first verification methodology to be standardized, it is also evolutionary, as it is built on the Open Verification Methodology (OVM), which combined the Advanced Verification Methodology (AVM) with the Universal Reuse Methodology (URM) and concepts from the ***e*** Reuse Methodology (eRM). Furthermore, UVM also infuses concepts and code from the Verification Methodology Manual (VMM), plus the collective experience and knowledge of the 300+ members of the Accellera Verification IP Technical Subcommittee (VIP-TSC) to help standardize verification methodology.

Scope

The UVM application programming interface (API) defines a standard for the creation, integration, and extension of UVM Verification Components (UVCs) and verification environments that scale from block to system. The UVM 1.0 Class Reference is independent of any specific design processes and is complete for the construction of verification environments. The generator to connect register abstractions, many of which are captured using IP-XACT (IEEE Std 1685™), is not part of the standard, although a register package is.

Purpose

The purpose of the UVM 1.0 Class Reference is to enable verification interoperability throughout the electronics ecosystem. To further that goal, a reference implementation will be made available, along with the UVM 1.0 User's Guide. While these materials are neither required to implement UVM, nor considered part of the standard, they help provide consistency when the UVM 1.0 Class Reference is applied and further enable UVM to achieve its purpose.

Normative References

The following referenced documents are indispensable for the application of this specification (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEEE Std 1800™, IEEE Standard for SystemVerilog Unified Hardware Design, Specification and Verification Language.[1, 2]

Definitions, Acronyms, and Abbreviations

For the purposes of this document, the following terms and definitions apply. The *IEEE Standards Dictionary: Glossary of Terms & Definitions*[3] should be referenced for terms not defined in this chapter.

Definitions

agent: An abstract container used to emulate and verify DUT devices; agents encapsulate a **driver**, **sequencer**, and **monitor**.

blocking: An interface where tasks block execution until they complete. See also: **non blocking**.

component: A piece of VIP that provides functionality and interfaces. Also referred to as a *transactor*.

consumer: A verification component that receives **transaction**s from another **component**.

driver: A component responsible for executing or otherwise processing **transaction**s, usually interacting with the device under test (DUT) to do so.

environment: The container object that defines the **testbench** topology.

export: A transaction level modeling (TLM) interface that provides the implementation of methods used for communication. Used in UVM to connect to a port.

factory method: A classic software design pattern used to create generic code by deferring, until run time, the exact specification of the object to be created.

foreign methodology: A verification methodology that is different from the methodology being used for the majority of the verification environment.

generator: A verification component that provides transactions to another **component**. Also referred to as a *producer*.

monitor: A passive entity that samples DUT signals, but does not drive them.

non blocking: A call that returns immediately. See also: **blocking**.

---

[1]IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).

[2]The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

[3]The *IEEE Standards Dictionary: Glossary of Terms & Definitions* is available at http://shop.ieee.org/.

port: A TLM interface that defines the set of methods used for communication. Used in UVM to connect to an export.

primary (host) methodology: The methodology that manages the top-level operation of the verification environment and with which the user/integrator is presumably more familiar.

request: A **transaction** that provides information to initiate the processing of a particular operation.

response: A **transaction** that provides information about the completion or status of a particular operation.

scoreboard: The mechanism used to dynamically predict the response of the design and check the observed response against the predicted response. Usually refers to the entire dynamic response-checking structure.

sequence: An UVM object that procedurally defines a set of **transaction**s to be executed and/or controls the execution of other sequences.

sequencer: An advanced stimulus generator which executes **sequence**s that define the **transaction**s provided to the **driver** for execution.

test: Specific customization of an environment to exercise required functionality of the DUT.

testbench: The structural definition of a set of verification components used to verify a DUT. Also referred to as a *verification environment*.

transaction: A class instance that encapsulates information used to communicate between two or more **component**s.

transactor: See *component*.

virtual sequence: A conceptual term for a **sequence** that controls the execution of **sequence**s on other **sequencer**s.

Acronyms and Abbreviations

API	application programming interface
CDV	coverage-driven verification
CBCL	common base class library
CLI	command line interface
DUT	device under test
DUV	device under verification
EDA	electronic design automation
FIFO	first-in, first-out
HDL	hardware description language
HVL	high-level verification language

IP	intellectual property
OSCI	Open SystemC Initiative
TLM	transaction level modeling
UVC	UVM Verification Component
UVM	Universal Verification Methodology
VIP	verification intellectual property

UVM Class Reference

The UVM Class Library provides the building blocks needed to quickly develop well-constructed and reusable verification components and test environments in SystemVerilog.

This UVM Class Reference provides detailed reference information for each user-visible class in the UVM library. For additional information on using UVM, see the UVM User's Guide located in the top level directory within the UVM kit.

We divide the UVM classes and utilities into categories pertaining to their role or function. A more detailed overview of each category-- and the classes comprising them-- can be found in the menu at left.

Globals	This category defines a small list of types, variables, functions, and tasks defined in the *uvm_pkg* scope. These items are accessible from any scope that imports the *uvm_pkg*. See Types and Enumerations and Globals for details.
Base	This basic building blocks for all environments are components, which do the actual work, transactions, which convey information between components, and ports, which provide the interfaces used to convey transactions. The UVM's core *base* classes provide these building blocks. See Core Base Classes for more information.
Reporting	The *reporting* classes provide a facility for issuing reports (messages) with consistent formatting and configurable side effects, such as logging to a file or exiting simulation. Users can also filter out reports based on their verbosity , unique ID, or severity. See Reporting Classes for more information.
Factory	As the name implies, the UVM factory is used to manufacture (create) UVM objects and components. Users can configure the factory to produce an object of a given type on a global or instance basis. Use of the factory allows dynamically configurable component hierarchies and object substitutions without having to modify their code and without breaking encapsulation. See Factory Classes for details.
Configuration and Resources	The Configuration and Resource Classes are a set of classes which provide a configuration database. The configuration database is used to store and retrieve both configuration time and run time properties.
Sychronization	The UVM provides event and barrier synchronization classes for process synchronization. See Synchronization Classes for more information.
Containers	The Container Classes are type parameterized datastructures which provide queue and pool services. The class based queue and pool types allow for efficient sharing of the datastructures compared with their SystemVerilog built-in

	counterparts.
Policies	Each of UVM's policy classes perform a specific task for uvm_object-based objects: printing, comparing, recording, packing, and unpacking. They are implemented separately from *uvm_object* so that users can plug in different ways to print, compare, etc. without modifying the object class being operated on. The user can simply apply a different printer or compare "policy" to change how an object is printed or compared. See Policy Classes for more information.
TLM	The UVM TLM library defines several abstract, transaction-level interfaces and the ports and exports that facilitate their use. Each TLM interface consists of one or more methods used to transport data, typically whole transactions (objects) at a time. Component designs that use TLM ports and exports to communicate are inherently more reusable, interoperable, and modular. See TLM Interfaces for details.
Components	Components form the foundation of the UVM. They encapsulate behavior of drivers, scoreboards, and other objects in a testbench. The UVM library provides a set of predefined component types, all derived directly or indirectly from uvm_component. See Predefined Component Classes for more information.
Sequencers	The sequencer serves as an arbiter for controlling transaction flow from multiple stimulus generators. More specifically, the sequencer controls the flow of uvm_sequence_item-based transactions generated by one or more uvm_sequence #(REQ,RSP)-based sequences. See Sequencer Classes for more information.
Sequences	Sequences encapsulate user-defined procedures that generate multiple uvm_sequence_item-based transactions. Such sequences can be reused, extended, randomized, and combined sequentially and hierarchically in interesting ways to produce realistic stimulus to your DUT. See Sequence Classes for more information.
Macros	The UVM provides several macros to help increase user productivity. See the set of macro categories in the main menu for a complete list of macros for Reporting, Components, Objects, Sequences, Callbacks, TLM and Registers.
Register Layer	The Register abstraction classes, when properly extended, abstract the read/write operations to registers and memories in a design-under-verification. See Register Layer for more information.
Command Line Processor	The command line processor provides a general interface to the command line arguments that

were provided for the given simulation. The capabilities are detailed in the uvm_cmdline_processor section.

Summary

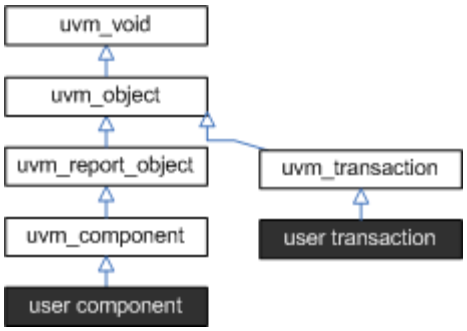
UVM Class Reference

The UVM Class Library provides the building blocks needed to quickly develop well-constructed and reusable verification components and test environments in SystemVerilog.

Core Base Classes

The UVM library defines a set of base classes and utilities that facilitate the design of modular, scalable, reusable verification environments.

The basic building blocks for all environments are components and the transactions they use to communicate. The UVM provides base classes for these, as shown below.



- uvm_object - All components and transactions derive from *uvm_object*, which defines an interface of core class-based operations: create, copy, compare, print, sprint, record, etc. It also defines interfaces for instance identification (name, type name, unique id, etc.) and random seeding.
- uvm_component - The *uvm_component* class is the root base class for all UVM components. Components are quasi-static objects that exist throughout simulation. This allows them to establish structural hierarchy much like *modules* and *program blocks*. Every component is uniquely addressable via a hierarchical path name, e.g. "env1.pci1.master3.driver". The *uvm_component* also defines a phased test flow that components follow during the course of simulation. Each phase-- *build*, *connect*, *run*, etc.-- is defined by a callback that is executed in precise order. Finally, the *uvm_component* also defines configuration, reporting, transaction recording, and factory interfaces.
- uvm_transaction - The *uvm_transaction* is the root base class for UVM transactions, which, unlike *uvm_components*, are transient in nature. It extends uvm_object to include a timing and recording interface. Simple transactions can derive directly from *uvm_transaction*, while sequence-enabled transactions derive from *uvm_sequence_item*.
- uvm_root - The *uvm_root* class is special *uvm_component* that serves as the top-level component for all UVM components, provides phasing control for all UVM components, and other global services.

Summary

Core Base Classes

The UVM library defines a set of base classes and utilities that facilitate the design of modular, scalable, reusable verification environments.

uvm_void

The *uvm_void* class is the base class for all UVM classes. It is an abstract class with no data members or functions. It allows for generic containers of objects to be created, similar to a void pointer in the C programming language. User classes derived directly from *uvm_void* inherit none of the UVM functionality, but such classes may be placed in *uvm_void*-typed containers along with other UVM objects.

Summary

uvm_void

The *uvm_void* class is the base class for all UVM classes.

uvm_object

The uvm_object class is the base class for all UVM data and hierarchical classes. Its primary role is to define a set of methods for such common operations as create, copy, compare, print, and record. Classes deriving from uvm_object must implement the pure virtual methods such as create and get_type_name.

Summary

uvm_object

The uvm_object class is the base class for all UVM data and hierarchical classes.

Class Hierarchy

uvm_void

uvm_object

Class Declaration

```
virtual class uvm_object extends uvm_void
```

new	Creates a new uvm_object with the given instance *name*.
Seeding	
use_uvm_seeding	This bit enables or disables the UVM seeding mechanism.
reseed	Calls *srandom* on the object to reseed the object using the UVM seeding mechanism, which sets the seed based on type name and instance name instead of based on instance position in a thread.
Identification	
set_name	Sets the instance name of this object, overwriting any previously given name.
get_name	Returns the name of the object, as provided by the *name* argument in the new constructor or set_name method.
get_full_name	Returns the full hierarchical name of this object.
get_inst_id	Returns the object's unique, numeric instance identifier.
get_inst_count	Returns the current value of the instance counter, which represents the total number of uvm_object-based objects that have been allocated in simulation.
get_type	Returns the type-proxy (wrapper) for this object.
get_object_type	Returns the type-proxy (wrapper) for this object.
get_type_name	This function returns the type name of the object, which is typically the type identifier enclosed in quotes.
Creation	
create	The create method allocates a new object of the same type as this object and returns it via a base uvm_object handle.
clone	The clone method creates and returns an exact copy of this object.
Printing	
print	The print method deep-prints this object's properties in a format and manner governed by the given *printer* argument; if the *printer* argument is not provided, the global uvm_default_printer is used.
sprint	The *sprint* method works just like the print method, except the output is returned in a string rather than displayed.
do_print	The *do_print* method is the user-definable hook called by print and sprint that allows users to customize what gets printed or sprinted beyond the field information

	provided by the `uvm_field_* macros, Utility and Field Macros for Components and Objects.
convert2string	This virtual function is a user-definable hook, called directly by the user, that allows users to provide object information in the form of a string.
Recording	
record	The record method deep-records this object's properties according to an optional *recorder* policy.
do_record	The do_record method is the user-definable hook called by the record method.
Copying	
copy	The copy method returns a deep copy of this object.
do_copy	The do_copy method is the user-definable hook called by the copy method.
Comparing	
compare	Deep compares members of this data object with those of the object provided in the *rhs* (right-hand side) argument, returning 1 on a match, 0 othewise.
do_compare	The do_compare method is the user-definable hook called by the compare method.
Packing	
pack pack_bytes pack_ints	The pack methods bitwise-concatenate this object's properties into an array of bits, bytes, or ints.
do_pack	The do_pack method is the user-definable hook called by the pack methods.
Unpacking	
unpack unpack_bytes unpack_ints	The unpack methods extract property values from an array of bits, bytes, or ints.
do_unpack	The do_unpack method is the user-definable hook called by the unpack method.
Configuration	
set_int_local set_string_local set_object_local	These methods provide write access to integral, string, and uvm_object-based properties indexed by a *field_name* string.

new

```
function new (string name = "")
```

Creates a new uvm_object with the given instance *name*. If *name* is not supplied, the object is unnamed.

Seeding

use_uvm_seeding

```
static bit use_uvm_seeding = 1
```

This bit enables or disables the UVM seeding mechanism. It globally affects the operation of the reseed method.

When enabled, UVM-based objects are seeded based on their type and full hierarchical name rather than allocation order. This improves random stability for objects whose instance names are unique across each type. The uvm_component class is an example of a type that has a unique instance name.

reseed

```
function void reseed ()
```

Calls *srandom* on the object to reseed the object using the UVM seeding mechanism, which sets the seed based on type name and instance name instead of based on instance position in a thread.

If the use_uvm_seeding static variable is set to 0, then reseed() does not perform any function.

IDENTIFICATION

set_name

```
virtual function void set_name (string name)
```

Sets the instance name of this object, overwriting any previously given name.

get_name

```
virtual function string get_name ()
```

Returns the name of the object, as provided by the *name* argument in the new constructor or set_name method.

get_full_name

```
virtual function string get_full_name ()
```

Returns the full hierarchical name of this object. The default implementation is the same as get_name, as uvm_objects do not inherently possess hierarchy.

Objects possessing hierarchy, such as uvm_components, override the default implementation. Other objects might be associated with component hierarchy but are not themselves components. For example, uvm_sequence #(REQ,RSP) classes are typically associated with a uvm_sequencer #(REQ,RSP). In this case, it is useful to override get_full_name to return the sequencer's full name concatenated with the sequence's name. This provides the sequence a full context, which is useful when debugging.

get_inst_id

```
virtual function int get_inst_id ()
```

Returns the object's unique, numeric instance identifier.

get_inst_count

```
static function int get_inst_count()
```

Returns the current value of the instance counter, which represents the total number of uvm_object-based objects that have been allocated in simulation. The instance counter is used to form a unique numeric instance identifier.

get_type

```
static function uvm_object_wrapper get_type ()
```

Returns the type-proxy (wrapper) for this object. The uvm_factory's type-based override and creation methods take arguments of uvm_object_wrapper. This method, if implemented, can be used as convenient means of supplying those arguments.

The default implementation of this method produces an error and returns null. To enable use of this method, a user's subtype must implement a version that returns the subtype's wrapper.

For example

```
class cmd extends uvm_object;
  typedef uvm_object_registry #(cmd) type_id;
  static function type_id get_type();
    return type_id::get();
  endfunction
endclass
```

Then, to use

```
factory.set_type_override(cmd::get_type(),subcmd::get_type());
```

This function is implemented by the `uvm_*_utils macros, if employed.

get_object_type

```
virtual function uvm_object_wrapper get_object_type ()
```

Returns the type-proxy (wrapper) for this object. The uvm_factory's type-based override and creation methods take arguments of uvm_object_wrapper. This method, if implemented, can be used as convenient means of supplying those arguments. This method is the same as the static get_type method, but uses an already allocated object to determine the type-proxy to access (instead of using the static object).

The default implementation of this method does a factory lookup of the proxy using the return value from get_type_name. If the type returned by get_type_name is not registered with the factory, then a null handle is returned.

For example

```
class cmd extends uvm_object;
  typedef uvm_object_registry #(cmd) type_id;
  static function type_id get_type();
    return type_id::get();
  endfunction
  virtual function type_id get_object_type();
    return type_id::get();
  endfunction
endclass
```

This function is implemented by the `uvm_*_utils macros, if employed.

get_type_name

```
virtual function string get_type_name ()
```

This function returns the type name of the object, which is typically the type identifier enclosed in quotes. It is used for various debugging functions in the library, and it is used by the factory for creating objects.

This function must be defined in every derived class.

A typical implementation is as follows

```
class mytype extends uvm_object;
  ...
  const static string type_name = "mytype";

  virtual function string get_type_name();
    return type_name;
  endfunction
```

We define the *type_name* static variable to enable access to the type name without need of an object of the class, i.e., to enable access via the scope operator, *mytype::type_name*.

Creation

create

```
virtual function uvm_object create (string name = "")
```

The create method allocates a new object of the same type as this object and returns it via a base uvm_object handle. Every class deriving from uvm_object, directly or indirectly, must implement the create method.

A typical implementation is as follows

```
class mytype extends uvm_object;
  ...
  virtual function uvm_object create(string name="");
    mytype t = new(name);
    return t;
```

```
  endfunction
```

clone

```
virtual function uvm_object clone ()
```

The clone method creates and returns an exact copy of this object.

The default implementation calls create followed by copy. As clone is virtual, derived classes may override this implementation if desired.

PRINTING

print

```
function void print (uvm_printer printer = null)
```

The print method deep-prints this object's properties in a format and manner governed by the given *printer* argument; if the *printer* argument is not provided, the global uvm_default_printer is used. See uvm_printer for more information on printer output formatting. See also uvm_line_printer, uvm_tree_printer, and uvm_table_printer for details on the pre-defined printer "policies," or formatters, provided by the UVM.

The *print* method is not virtual and must not be overloaded. To include custom information in the *print* and *sprint* operations, derived classes must override the do_print method and use the provided printer policy class to format the output.

sprint

```
function string sprint (uvm_printer printer = null)
```

The *sprint* method works just like the print method, except the output is returned in a string rather than displayed.

The *sprint* method is not virtual and must not be overloaded. To include additional fields in the *print* and *sprint* operation, derived classes must override the do_print method and use the provided printer policy class to format the output. The printer policy will manage all string concatenations and provide the string to *sprint* to return to the caller.

do_print

```
virtual function void do_print (uvm_printer printer)
```

The *do_print* method is the user-definable hook called by print and sprint that allows users to customize what gets printed or sprinted beyond the field information provided by the `uvm_field_* macros, Utility and Field Macros for Components and Objects.

The *printer* argument is the policy object that governs the format and content of the output. To ensure correct print and sprint operation, and to ensure a consistent output format, the *printer* must be used by all do_print implementations. That is, instead of using *$display* or string concatenations directly, a *do_print* implementation must call

through the *printer's* API to add information to be printed or sprinted.

An example implementation of *do_print* is as follows

```
class mytype extends uvm_object;
  data_obj data;
  int f1;
  virtual function void do_print (uvm_printer printer);
    super.do_print(printer);
    printer.print_int("f1", f1, $bits(f1), DEC);
    printer.print_object("data", data);
  endfunction
```

Then, to print and sprint the object, you could write

```
mytype t = new;
t.print();
uvm_report_info("Received",t.sprint());
```

See uvm_printer for information about the printer API.

convert2string

```
virtual function string convert2string()
```

This virtual function is a user-definable hook, called directly by the user, that allows users to provide object information in the form of a string. Unlike sprint, there is no requirement to use an uvm_printer policy object. As such, the format and content of the output is fully customizable, which may be suitable for applications not requiring the consistent formatting offered by the print/sprint/do_print API.

Fields declared in Utility Macros macros (`uvm_field_*), if used, will not automatically appear in calls to convert2string.

An example implementation of convert2string follows.

```
class base extends uvm_object;
  string field = "foo";
  virtual function string convert2string();
    convert2string = {"base_field=",field};
  endfunction
endclass

class obj2 extends uvm_object;
  string field = "bar";
  virtual function string convert2string();
    convert2string = {"child_field=",field};
  endfunction
endclass

class obj extends base;
  int addr = 'h123;
  int data = 'h456;
  bit write = 1;
  obj2 child = new;
  virtual function string convert2string();
     convert2string = {super.convert2string(),
       $psprintf(" write=%0d addr=%8h data=%8h ",write,addr,data),
       child.convert2string()};
  endfunction
endclass
```

Then, to display an object, you could write

```
obj o = new;
uvm_report_info("BusMaster",{"Sending:\n ",o.convert2string()});
```

The output will look similar to

```
UVM_INFO @ 0: reporter [BusMaster] Sending:
   base_field=foo write=1 addr=00000123 data=00000456 child_field=bar
```

RECORDING

record

```
function void record (uvm_recorder recorder = null)
```

The record method deep-records this object's properties according to an optional *recorder* policy. The method is not virtual and must not be overloaded. To include additional fields in the record operation, derived classes should override the do_record method.

The optional *recorder* argument specifies the recording policy, which governs how recording takes place. If a recorder policy is not provided explicitly, then the global uvm_default_recorder policy is used. See uvm_recorder for information.

A simulator's recording mechanism is vendor-specific. By providing access via a common interface, the uvm_recorder policy provides vendor-independent access to a simulator's recording capabilities.

do_record

```
virtual function void do_record (uvm_recorder recorder)
```

The do_record method is the user-definable hook called by the record method. A derived class should override this method to include its fields in a record operation.

The *recorder* argument is policy object for recording this object. A do_record implementation should call the appropriate recorder methods for each of its fields. Vendor-specific recording implementations are encapsulated in the *recorder* policy, thereby insulating user-code from vendor-specific behavior. See uvm_recorder for more information.

A typical implementation is as follows

```
class mytype extends uvm_object;
  data_obj data;
  int f1;
  function void do_record (uvm_recorder recorder);
    recorder.record_field_int("f1", f1, $bits(f1), DEC);
    recorder.record_object("data", data);
  endfunction
```

COPYING

copy

```
function void copy (uvm_object rhs)
```

The copy method returns a deep copy of this object.

The copy method is not virtual and should not be overloaded in derived classes. To copy the fields of a derived class, that class should override the do_copy method.

do_copy

```
virtual function void do_copy (uvm_object rhs)
```

The do_copy method is the user-definable hook called by the copy method. A derived class should override this method to include its fields in a copy operation.

A typical implementation is as follows

```
class mytype extends uvm_object;
  ...
  int f1;
  function void do_copy (uvm_object rhs);
    mytype rhs_;
    super.do_copy(rhs);
    $cast(rhs_,rhs);
    field_1 = rhs_.field_1;
  endfunction
```

The implementation must call *super.do_copy*, and it must $cast the rhs argument to the derived type before copying.

COMPARING

compare

```
function bit compare (uvm_object   rhs,
                      uvm_comparer comparer = null)
```

Deep compares members of this data object with those of the object provided in the *rhs* (right-hand side) argument, returning 1 on a match, 0 othewise.

The compare method is not virtual and should not be overloaded in derived classes. To compare the fields of a derived class, that class should override the do_compare method.

The optional *comparer* argument specifies the comparison policy. It allows you to control some aspects of the comparison operation. It also stores the results of the comparison, such as field-by-field miscompare information and the total number of miscompares. If a compare policy is not provided, then the global *uvm_default_comparer* policy is used. See uvm_comparer for more information.

do_compare

```
virtual function bit do_compare (uvm_object   rhs,
                                 uvm_comparer comparer)
```

The do_compare method is the user-definable hook called by the compare method. A derived class should override this method to include its fields in a compare operation. It should return 1 if the comparison succeeds, 0 otherwise.

A typical implementation is as follows

```
class mytype extends uvm_object;
  ...
  int f1;
  virtual function bit do_compare (uvm_object rhs,uvm_comparer comparer);
    mytype rhs_;
    do_compare = super.do_compare(rhs,comparer);
    $cast(rhs_,rhs);
    do_compare &= comparer.compare_field_int("f1", f1, rhs_.f1);
  endfunction
```

A derived class implementation must call *super.do_compare()* to ensure its base class' properties, if any, are included in the comparison. Also, the rhs argument is provided as a generic uvm_object. Thus, you must *$cast* it to the type of this object before comparing.

The actual comparison should be implemented using the uvm_comparer object rather than direct field-by-field comparison. This enables users of your class to customize how comparisons are performed and how much miscompare information is collected. See uvm_comparer for more details.

PACKING

pack

```
function int pack (  ref bit        bitstream[],
                   input uvm_packer packer       = null)
```

pack_bytes

```
function int pack_bytes (ref byte unsigned   bytestream[],
                         input uvm_packer packer        = null)
```

pack_ints

```
function int pack_ints (ref int unsigned   intstream[],
                        input uvm_packer packer       = null)
```

The pack methods bitwise-concatenate this object's properties into an array of bits, bytes, or ints. The methods are not virtual and must not be overloaded. To include additional fields in the pack operation, derived classes should override the do_pack method.

The optional *packer* argument specifies the packing policy, which governs the packing operation. If a packer policy is not provided, the global uvm_default_packer policy is used. See uvm_packer for more information.

The return value is the total number of bits packed into the given array. Use the array's built-in *size* method to get the number of bytes or ints consumed during the packing process.

do_pack

```
virtual function void do_pack (uvm_packer packer)
```

The do_pack method is the user-definable hook called by the pack methods. A derived class should override this method to include its fields in a pack operation.

The *packer* argument is the policy object for packing. The policy object should be used to pack objects.

A typical example of an object packing itself is as follows

```
class mysubtype extends mysupertype;
  ...
  shortint myshort;
  obj_type myobj;
  byte myarray[];
  ...
  function void do_pack (uvm_packer packer);
    super.do_pack(packer); // pack mysupertype properties
    packer.pack_field_int(myarray.size(), 32);
    foreach (myarray)
      packer.pack_field_int(myarray[index], 8);
    packer.pack_field_int(myshort, $bits(myshort));
    packer.pack_object(myobj);
  endfunction
```

The implementation must call *super.do_pack* so that base class properties are packed as well.

If your object contains dynamic data (object, string, queue, dynamic array, or associative array), and you intend to unpack into an equivalent data structure when unpacking, you must include meta-information about the dynamic data when packing as follows.

- For queues, dynamic arrays, or associative arrays, pack the number of elements in the array in the 32 bits immediately before packing individual elements, as shown above.
- For string data types, append a zero byte after packing the string contents.
- For objects, pack 4 bits immediately before packing the object. For null objects, pack 4'b0000. For non-null objects, pack 4'b0001.

When the `uvm_field_* macros are used, Utility and Field Macros for Components and Objects, the above meta information is included provided the uvm_packer::use_metadata variable is set for the packer.

Packing order does not need to match declaration order. However, unpacking order must match packing order.

UNPACKING

unpack

```
function int unpack (  ref bit        bitstream[],
                     input uvm_packer packer       = null)
```

unpack_bytes

```
function int unpack_bytes (ref byte unsigned   bytestream[],
                           input uvm_packer packer        = null)
```

unpack_ints

```
function int unpack_ints (ref int unsigned   intstream[],
                          input uvm_packer packer       = null)
```

The unpack methods extract property values from an array of bits, bytes, or ints. The method of unpacking *must* exactly correspond to the method of packing. This is assured if (a) the same *packer* policy is used to pack and unpack, and (b) the order of unpacking is the same as the order of packing used to create the input array.

The unpack methods are fixed (non-virtual) entry points that are directly callable by the user. To include additional fields in the unpack operation, derived classes should override the do_unpack method.

The optional *packer* argument specifies the packing policy, which governs both the pack and unpack operation. If a packer policy is not provided, then the global *uvm_default_packer* policy is used. See uvm_packer for more information.

The return value is the actual number of bits unpacked from the given array.

do_unpack

```
virtual function void do_unpack (uvm_packer packer)
```

The do_unpack method is the user-definable hook called by the unpack method. A derived class should override this method to include its fields in an unpack operation.

The *packer* argument is the policy object for both packing and unpacking. It must be the same packer used to pack the object into bits. Also, do_unpack must unpack fields in the same order in which they were packed. See uvm_packer for more information.

The following implementation corresponds to the example given in do_pack.

```
function void do_unpack (uvm_packer packer);
 int sz;
  super.do_unpack(packer); // unpack super's properties
  sz = packer.unpack_field_int(myarray.size(), 32);
  myarray.delete();
  for(int index=0; index<sz; index++)
    myarray[index] = packer.unpack_field_int(8);
  myshort = packer.unpack_field_int($bits(myshort));
  packer.unpack_object(myobj);
endfunction
```

If your object contains dynamic data (object, string, queue, dynamic array, or associative array), and you intend to unpack into an equivalent data structure, you must have included meta-information about the dynamic data when it was packed.

- For queues, dynamic arrays, or associative arrays, unpack the number of elements

in the array from the 32 bits immediately before unpacking individual elements, as shown above.

- For string data types, unpack into the new string until a null byte is encountered.
- For objects, unpack 4 bits into a byte or int variable. If the value is 0, the target object should be set to null and unpacking continues to the next property, if any. If the least significant bit is 1, then the target object should be allocated and its properties unpacked.

CONFIGURATION

set_int_local

```
virtual function void set_int_local (string          field_name,
                                     uvm_bitstream_t value,
                                     bit             recurse     = 1)
```

set_string_local

```
virtual function void set_string_local (string field_name,
                                        string value,
                                        bit    recurse     = 1)
```

set_object_local

```
virtual function void set_object_local (string     field_name,
                                        uvm_object value,
                                        bit        clone       = 1,
                                        bit        recurse     = 1 )
```

These methods provide write access to integral, string, and uvm_object-based properties indexed by a *field_name* string. The object designer choose which, if any, properties will be accessible, and overrides the appropriate methods depending on the properties' types. For objects, the optional *clone* argument specifies whether to clone the *value* argument before assignment.

The global uvm_is_match function is used to match the field names, so *field_name* may contain wildcards.

An example implementation of all three methods is as follows.

```
class mytype extends uvm_object;

  local int myint;
  local byte mybyte;
  local shortint myshort; // no access
  local string mystring;
  local obj_type myobj;

  // provide access to integral properties
  function void set_int_local(string field_name, uvm_bitstream_t value);
    if (uvm_is_match (field_name, "myint"))
      myint = value;
    else if (uvm_is_match (field_name, "mybyte"))
      mybyte = value;
  endfunction

  // provide access to string properties
  function void set_string_local(string field_name, string value);
    if (uvm_is_match (field_name, "mystring"))
      mystring = value;
  endfunction
```

```
  // provide access to sub-objects
  function void set_object_local(string field_name, uvm_object value,
                                 bit clone=1);
    if (uvm_is_match (field_name, "myobj")) begin
      if (value != null) begin
        obj_type tmp;
        // if provided value is not correct type, produce error
        if (!$cast(tmp, value) )
          /* error */
        else begin
          if(clone)
            $cast(myobj, tmp.clone());
          else
            myobj = tmp;
        end
      end
      else
        myobj = null; // value is null, so simply assign null to myobj
    end
  endfunction
  ...
```

Although the object designer implements these methods to provide outside access to one or more properties, they are intended for internal use (e.g., for command-line debugging and auto-configuration) and should not be called directly by the user.

uvm_transaction

The uvm_transaction class is the root base class for UVM transactions. Inheriting all the methods of uvm_object, uvm_transaction adds a timing and recording interface.

Summary

uvm_transaction

The uvm_transaction class is the root base class for UVM transactions.

Class Hierarchy

uvm_void

uvm_object

uvm_transaction

Class Declaration

```
virtual class uvm_transaction extends uvm_object
```

Methods

new	Creates a new transaction object.
accept_tr	Calling *accept_tr* indicates that the transaction has been accepted for processing by a consumer component, such as an uvm_driver.
do_accept_tr	This user-definable callback is called by accept_tr just before the accept event is triggered.
begin_tr	This function indicates that the transaction has been started and is not the child of another transaction.
begin_child_tr	This function indicates that the transaction has been started as a child of a parent transaction given by parent_handle.
do_begin_tr	This user-definable callback is called by begin_tr and begin_child_tr just before the begin event is triggered.
end_tr	This function indicates that the transaction execution has ended.
do_end_tr	This user-definable callback is called by end_tr just before the end event is triggered.
get_tr_handle	Returns the handle associated with the transaction, as set by a previous call to begin_child_tr or begin_tr with transaction recording enabled.
disable_recording	Turns off recording for the transaction stream.
enable_recording	Turns on recording to the stream specified by stream, whose interpretation is implementation specific.
is_recording_enabled	Returns 1 if recording is currently on, 0 otherwise.
is_active	Returns 1 if the transaction has been started but has not yet been ended.
get_event_pool	Returns the event pool associated with this transaction.
set_initiator	Sets initiator as the initiator of this transaction.
get_initiator	Returns the component that produced or started the transaction, as set by a previous call to set_initiator.
get_accept_time get_begin_time get_end_time	Returns the time at which this transaction was accepted, begun, or ended, as by a previous call to accept_tr, begin_tr, begin_child_tr, or end_tr.
set_transaction_id	Sets this transaction's numeric identifier to id.
get_transaction_id	Returns this transaction's numeric identifier, which is -1 if not set explicitly by *set_transaction_id*.

Variables	
events	The event pool instance for this transaction.
begin_event	The event that is triggered when transaction recording for this transaction begins.
end_event	The event that is triggered when transaction recording for this transaction ends.

Methods

new

```
function new (string        name      = "",
              uvm_component initiator = null)
```

Creates a new transaction object. The name is the instance name of the transaction. If not supplied, then the object is unnamed.

accept_tr

```
function void accept_tr (time accept_time = )
```

Calling *accept_tr* indicates that the transaction has been accepted for processing by a consumer component, such as an uvm_driver. With some protocols, the transaction may not be started immediately after it is accepted. For example, a bus driver may have to wait for a bus grant before starting the transaction.

This function performs the following actions

- The transaction's internal accept time is set to the current simulation time, or to accept_time if provided and non-zero. The *accept_time* may be any time, past or future.
- The transaction's internal accept event is triggered. Any processes waiting on the this event will resume in the next delta cycle.
- The do_accept_tr method is called to allow for any post-accept action in derived classes.

do_accept_tr

```
virtual protected function void do_accept_tr ()
```

This user-definable callback is called by accept_tr just before the accept event is triggered. Implementations should call *super.do_accept_tr* to ensure correct operation.

begin_tr

```
function integer begin_tr (time begin_time = )
```

This function indicates that the transaction has been started and is not the child of another transaction. Generally, a consumer component begins execution of the transactions it receives.

This function performs the following actions

- The transaction's internal start time is set to the current simulation time, or to begin_time if provided and non-zero. The begin_time may be any time, past or future, but should not be less than the accept time.
- If recording is enabled, then a new database-transaction is started with the same begin time as above. The record method inherited from uvm_object is then called, which records the current property values to this new transaction.
- The do_begin_tr method is called to allow for any post-begin action in derived classes.
- The transaction's internal begin event is triggered. Any processes waiting on this event will resume in the next delta cycle.

The return value is a transaction handle, which is valid (non-zero) only if recording is enabled. The meaning of the handle is implementation specific.

begin_child_tr

```
function integer begin_child_tr (time    begin_time    = 0,
                                 integer parent_handle = 0 )
```

This function indicates that the transaction has been started as a child of a parent transaction given by parent_handle. Generally, a consumer component begins execution of the transactions it receives.

The parent handle is obtained by a previous call to begin_tr or begin_child_tr. If the parent_handle is invalid (=0), then this function behaves the same as begin_tr.

This function performs the following actions

- The transaction's internal start time is set to the current simulation time, or to begin_time if provided and non-zero. The begin_time may be any time, past or future, but should not be less than the accept time.
- If recording is enabled, then a new database-transaction is started with the same begin time as above. The record method inherited from uvm_object is then called, which records the current property values to this new transaction. Finally, the newly started transaction is linked to the parent transaction given by parent_handle.
- The do_begin_tr method is called to allow for any post-begin action in derived classes.
- The transaction's internal begin event is triggered. Any processes waiting on this event will resume in the next delta cycle.

The return value is a transaction handle, which is valid (non-zero) only if recording is enabled. The meaning of the handle is implementation specific.

do_begin_tr

```
virtual protected function void do_begin_tr ()
```

This user-definable callback is called by begin_tr and begin_child_tr just before the begin event is triggered. Implementations should call *super.do_begin_tr* to ensure correct operation.

end_tr

```
function void end_tr (time end_time    = 0,
                      bit  free_handle = 1 )
```

This function indicates that the transaction execution has ended. Generally, a consumer component ends execution of the transactions it receives.

This function performs the following actions

- The transaction's internal end time is set to the current simulation time, or to *end_time* if provided and non-zero. The *end_time* may be any time, past or future, but should not be less than the begin time.
- If recording is enabled and a database-transaction is currently active, then the record method inherited from uvm_object is called, which records the final property values. The transaction is then ended. If *free_handle* is set, the transaction is released and can no longer be linked to (if supported by the implementation).
- The do_end_tr method is called to allow for any post-end action in derived classes.
- The transaction's internal end event is triggered. Any processes waiting on this event will resume in the next delta cycle.

do_end_tr

```
virtual protected function void do_end_tr ()
```

This user-definable callback is called by end_tr just before the end event is triggered. Implementations should call *super.do_end_tr* to ensure correct operation.

get_tr_handle

```
function integer get_tr_handle ()
```

Returns the handle associated with the transaction, as set by a previous call to begin_child_tr or begin_tr with transaction recording enabled.

disable_recording

```
function void disable_recording ()
```

Turns off recording for the transaction stream. This method does not effect a uvm_component's recording streams.

enable_recording

```
function void enable_recording (string stream)
```

Turns on recording to the stream specified by stream, whose interpretation is implementation specific.

If transaction recording is on, then a call to record is made when the transaction is started and when it is ended.

is_recording_enabled

```
function bit is_recording_enabled()
```

Returns 1 if recording is currently on, 0 otherwise.

is_active

```
function bit is_active ()
```

Returns 1 if the transaction has been started but has not yet been ended. Returns 0 if the transaction has not been started.

get_event_pool

```
function uvm_event_pool get_event_pool ()
```

Returns the event pool associated with this transaction.

By default, the event pool contains the events: begin, accept, and end. Events can also be added by derivative objects. An event pool is a specialization of an <uvm_pool #(T)>, e.g. a *uvm_pool#(uvm_event)*.

set_initiator

```
function void set_initiator (uvm_component initiator)
```

Sets initiator as the initiator of this transaction.

The initiator can be the component that produces the transaction. It can also be the component that started the transaction. This or any other usage is up to the transaction designer.

get_initiator

```
function uvm_component get_initiator ()
```

Returns the component that produced or started the transaction, as set by a previous call to set_initiator.

get_accept_time

```
function time get_accept_time ()
```

get_begin_time

```
function time get_begin_time ()
```

get_end_time

```
function time get_end_time ()
```

Returns the time at which this transaction was accepted, begun, or ended, as by a previous call to accept_tr, begin_tr, begin_child_tr, or end_tr.

set_transaction_id

```
function void set_transaction_id(integer id)
```

Sets this transaction's numeric identifier to id. If not set via this method, the transaction ID defaults to -1.

When using sequences to generate stimulus, the transaction ID is used along with the sequence ID to route responses in sequencers and to correlate responses to requests.

get_transaction_id

```
function integer get_transaction_id()
```

Returns this transaction's numeric identifier, which is -1 if not set explicitly by *set_transaction_id*.

When using a uvm_sequence #(REQ,RSP) to generate stimulus, the transaction ID is used along with the sequence ID to route responses in sequencers and to correlate responses to requests.

VARIABLES

events

```
const uvm_event_pool events = new
```

The event pool instance for this transaction.

begin_event

```
uvm_event begin_event
```

The event that is triggered when transaction recording for this transaction begins.

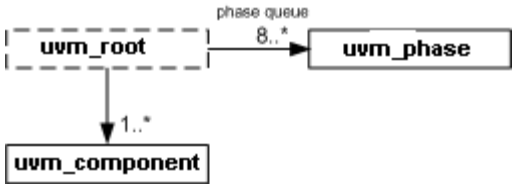
end_event

```
uvm_event end_event
```

The event that is triggered when transaction recording for this transaction ends.

uvm_root

The *uvm_root* class serves as the implicit top-level and phase controller for all UVM components. Users do not directly instantiate *uvm_root*. The UVM automatically creates a single instance of uvm_root that users can access via the global (uvm_pkg-scope) variable, *uvm_top*.



The *uvm_top* instance of *uvm_root* plays several key roles in the UVM.

Implicit top-level	The *uvm_top* serves as an implicit top-level component. Any component whose parent is specified as NULL becomes a child of *uvm_top*. Thus, all UVM components in simulation are descendants of *uvm_top*.
Phase control	*uvm_top* manages the phasing for all components. TBD
Search	Use *uvm_top* to search for components based on their hierarchical name. See find and find_all.
Report configuration	Use *uvm_top* to globally configure report verbosity, log files, and actions. For example, *uvm_top.set_report_verbosity_level_hier(UVM_FULL)* would set full verbosity for all components in simulation.
Global reporter	Because *uvm_top* is globally accessible (in uvm_pkg scope), UVM's reporting mechanism is accessible from anywhere outside *uvm_component*, such as in modules and sequences. See uvm_report_error, uvm_report_warning, and other global methods.

Summary

uvm_root

The *uvm_root* class serves as the implicit top-level and phase controller for all UVM components.

METHODS

run_test	Phases all components through all registered phases.

VARIABLES

top_levels	This variable is a list of all of the top level components in UVM.

METHODS

find find_all	Returns the component handle (find) or list of components handles (find_all) matching a given string.
print_topology	Print the verification environment's component topology.

VARIABLES

enable_print_topology	If set, then the entire testbench topology is printed just after completion of the end_of_elaboration phase.

finish_on_completion	If set, then run_test will call $finish after all phases are executed.
METHODS	
set_timeout	Specifies the timeout for task-based phases.
VARIABLES	
uvm_top	This is the top-level that governs phase execution and provides component search interface.

METHODS

run_test

```
virtual task run_test (string test_name = "")
```

Phases all components through all registered phases. If the optional test_name argument is provided, or if a command-line plusarg, +UVM_TESTNAME=TEST_NAME, is found, then the specified component is created just prior to phasing. The test may contain new verification components or the entire testbench, in which case the test and testbench can be chosen from the command line without forcing recompilation. If the global (package) variable, finish_on_completion, is set, then $finish is called after phasing completes.

VARIABLES

top_levels

```
uvm_component top_levels[$]
```

This variable is a list of all of the top level components in UVM. It includes the uvm_test_top component that is created by run_test as well as any other top level components that have been instantiated anywhere in the hierarchy.

METHODS

find

```
function uvm_component find (string comp_match)
```

find_all

```
function void find_all (      string        comp_match,
                         ref uvm_component comps[$],
                       input uvm_component comp       = null)
```

Returns the component handle (find) or list of components handles (find_all) matching a given string. The string may contain the wildcards,

and ?. Strings beginning with '.' are absolute path names. If optional comp arg is provided, then search begins from that component down (default=all components).

print_topology

```
function void print_topology (uvm_printer printer = null)
```

Print the verification environment's component topology. The *printer* is a uvm_printer object that controls the format of the topology printout; a *null* printer prints with the default output.

VARIABLES

enable_print_topology

```
bit enable_print_topology = 0
```

If set, then the entire testbench topology is printed just after completion of the end_of_elaboration phase.

finish_on_completion

```
bit finish_on_completion = 1
```

If set, then run_test will call $finish after all phases are executed.

METHODS

set_timeout

```
function void set_timeout(time timeout,
                          bit  overridable = 1)
```

Specifies the timeout for task-based phases. Default is 0, i.e. no timeout.

VARIABLES

uvm_top

```
const uvm_root uvm_top = uvm_root::get()
```

This is the top-level that governs phase execution and provides component search interface. See uvm_root for more information.

Port Base Classes

Contents

uvm_port_component_base

This class defines an interface for obtaining a port's connectivity lists after or during the end_of_elaboration phase. The sub-class, uvm_port_component #(PORT), implements this interface.

The connectivity lists are returned in the form of handles to objects of this type. This allowing traversal of any port's fan-out and fan-in network through recursive calls to get_connected_to and get_provided_to. Each port's full name and type name can be retrieved using get_full_name and get_type_name methods inherited from uvm_component.

Summary

uvm_port_component_base

This class defines an interface for obtaining a port's connectivity lists after or during the end_of_elaboration phase.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- **uvm_port_component_base**

Class Declaration

```
virtual class uvm_port_component_base extends
uvm_component
```

Methods

get_connected_to	For a port or export type, this function fills *list* with all of the ports, exports and implementations that this port is connected to.
get_provided_to	For an implementation or export type, this function fills *list* with all of the ports, exports and implementations that this port is provides its implementation to.

is_port is_export is_imp	These function determine the type of port.

METHODS

get_connected_to

```
pure virtual function void get_connected_to(ref uvm_port_list list)
```

For a port or export type, this function fills *list* with all of the ports, exports and implementations that this port is connected to.

get_provided_to

```
pure virtual function void get_provided_to(ref uvm_port_list list)
```

For an implementation or export type, this function fills *list* with all of the ports, exports and implementations that this port is provides its implementation to.

is_port

```
pure virtual function bit is_port()
```

is_export

```
pure virtual function bit is_export()
```

is_imp

```
pure virtual function bit is_imp()
```

These function determine the type of port. The functions are mutually exclusive; one will return 1 and the other two will return 0.

uvm_port_component #(PORT)

See description of uvm_port_component_base for information about this class

Summary

uvm_port_component #(PORT)

See description of uvm_port_component_base for information about this class

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- uvm_port_component_base
- **uvm_port_component#(PORT)**

Class Declaration

```
class uvm_port_component #(
   type PORT = uvm_object
) extends uvm_port_component_base
```

Methods

get_port	Retrieve the actual port object that this proxy refers to.

Methods

get_port

```
function PORT get_port()
```

Retrieve the actual port object that this proxy refers to.

uvm_port_base #(IF)

Transaction-level communication between components is handled via its ports, exports, and imps, all of which derive from this class.

The uvm_port_base extends IF, which is the type of the interface implemented by derived port, export, or implementation. IF is also a type parameter to uvm_port_base.

IF	The interface type implemented by the subtype to this base port

The UVM provides a complete set of ports, exports, and imps for the OSCI- standard TLM interfaces. They can be found in the ../src/tlm/ directory. For the TLM interfaces, the IF parameter is always uvm_tlm_if_base #(T1,T2).

Just before <uvm_component::end_of_elaboration>, an internal uvm_component::resolve_bindings process occurs, after which each port and export holds a list of all imps connected to it via hierarchical connections to other ports and exports. In effect, we are collapsing the port's fanout, which can span several levels up and down the component hierarchy, into a single array held local to the port. Once the list is determined, the port's min and max connection settings can be checked and enforced.

uvm_port_base possesses the properties of components in that they have a hierarchical instance path and parent. Because SystemVerilog does not support multiple inheritance, uvm_port_base can not extend both the interface it implements and uvm_component. Thus, uvm_port_base contains a local instance of uvm_component, to which it delegates

such commands as get_name, get_full_name, and get_parent.

Summary

uvm_port_base #(IF)

Transaction-level communication between components is handled via its ports, exports, and imps, all of which derive from this class.

Class Hierarchy

IF

uvm_port_base#(IF)

Class Declaration

```
virtual class uvm_port_base #(
   type IF = uvm_void
) extends IF
```

Methods

new	The first two arguments are the normal uvm_component constructor arguments.
get_name	Returns the leaf name of this port.
get_full_name	Returns the full hierarchical name of this port.
get_parent	Returns the handle to this port's parent, or null if it has no parent.
get_comp	Returns a handle to the internal proxy component representing this port.
get_type_name	Returns the type name to this port.
min_size	Returns the mininum number of implementation ports that must be connected to this port by the end_of_elaboration phase.
max_size	Returns the maximum number of implementation ports that must be connected to this port by the end_of_elaboration phase.
is_unbounded	Returns 1 if this port has no maximum on the number of implementation ports this port can connect to.
is_port is_export is_imp	Returns 1 if this port is of the type given by the method name, 0 otherwise.
size	Gets the number of implementation ports connected to this port.
set_default_index	Sets the default implementation port to use when calling an interface method.
connect	Connects this port to the given *provider* port.
debug_connected_to	The debug_connected_to method outputs a visual text display of the port/export/imp network to which this port connects (i.e., the port's fanout).
debug_provided_to	The debug_provided_to method outputs a visual display of the port/export network that ultimately connect to this port (i.e., the port's fanin).
resolve_bindings	This callback is called just before entering the end_of_elaboration phase.
get_if	Returns the implementation (imp) port at the given index from the array of imps this port is connected to.

Methods

new

```
function new (string          name,
              uvm_component   parent,
              uvm_port_type_e port_type,
              int             min_size   = 0,
              int             max_size   = 1 )
```

The first two arguments are the normal uvm_component constructor arguments.

The *port_type* can be one of UVM_PORT, UVM_EXPORT, or UVM_IMPLEMENTATION.

The *min_size* and *max_size* specify the minimum and maximum number of implementation (imp) ports that must be connected to this port base by the end of elaboration. Setting *max_size* to *UVM_UNBOUNDED_CONNECTIONS* sets no maximum, i.e., an unlimited number of connections are allowed.

By default, the parent/child relationship of any port being connected to this port is not checked. This can be overridden by configuring the port's *check_connection_relationships* bit via set_config_int. See connect for more information.

get_name

```
function string get_name()
```

Returns the leaf name of this port.

get_full_name

```
virtual function string get_full_name()
```

Returns the full hierarchical name of this port.

get_parent

```
virtual function uvm_component get_parent()
```

Returns the handle to this port's parent, or null if it has no parent.

get_comp

```
virtual function uvm_port_component_base get_comp()
```

Returns a handle to the internal proxy component representing this port.

Ports are considered components. However, they do not inherit uvm_component. Instead, they contain an instance of uvm_port_component #(PORT) that serves as a proxy to this port.

get_type_name

```
virtual function string get_type_name()
```

Returns the type name to this port. Derived port classes must implement this method to

return the concrete type. Otherwise, only a generic "uvm_port", "uvm_export" or "uvm_implementation" is returned.

min_size

Returns the mininum number of implementation ports that must be connected to this port by the end_of_elaboration phase.

max_size

Returns the maximum number of implementation ports that must be connected to this port by the end_of_elaboration phase.

is_unbounded

```
function bit is_unbounded ()
```

Returns 1 if this port has no maximum on the number of implementation ports this port can connect to. A port is unbounded when the *max_size* argument in the constructor is specified as *UVM_UNBOUNDED_CONNECTIONS*.

is_port

```
function bit is_port ()
```

is_export

```
function bit is_export ()
```

is_imp

```
function bit is_imp ()
```

Returns 1 if this port is of the type given by the method name, 0 otherwise.

size

```
function int size ()
```

Gets the number of implementation ports connected to this port. The value is not valid before the end_of_elaboration phase, as port connections have not yet been resolved.

set_default_index

```
function void set_default_index (int index)
```

Sets the default implementation port to use when calling an interface method. This

method should only be called on UVM_EXPORT types. The value must not be set before the end_of_elaboration phase, when port connections have not yet been resolved.

connect

```
virtual function void connect (this_type provider)
```

Connects this port to the given *provider* port. The ports must be compatible in the following ways

- Their type parameters must match
- The *provider*'s interface type (blocking, non-blocking, analysis, etc.) must be compatible. Each port has an interface mask that encodes the interface(s) it supports. If the bitwise AND of these masks is equal to the this port's mask, the requirement is met and the ports are compatible. For example, an uvm_blocking_put_port #(T) is compatible with an uvm_put_export #(T) and uvm_blocking_put_imp #(T) because the export and imp provide the interface required by the uvm_blocking_put_port.
- Ports of type UVM_EXPORT can only connect to other exports or imps.
- Ports of type UVM_IMPLEMENTATION can not be connected, as they are bound to the component that implements the interface at time of construction.

In addition to type-compatibility checks, the relationship between this port and the *provider* port will also be checked if the port's *check_connection_relationships* configuration has been set. (See new for more information.)

Relationships, when enabled, are checked are as follows

- If this port is an UVM_PORT type, the *provider* can be a parent port, or a sibling export or implementation port.
- If this port is an UVM_EXPORT type, the provider can be a child export or implementation port.

If any relationship check is violated, a warning is issued.

Note- the <uvm_component::connect> method is related to but not the same as this method. The component's connect method is a phase callback where port's connect method calls are made.

debug_connected_to

```
function void debug_connected_to (int level     = 0,
                                  int max_level = -1)
```

The debug_connected_to method outputs a visual text display of the port/export/imp network to which this port connects (i.e., the port's fanout).

This method must not be called before the end_of_elaboration phase, as port connections are not resolved until then.

debug_provided_to

```
function void debug_provided_to (int level     = 0,
                                 int max_level = -1)
```

The debug_provided_to method outputs a visual display of the port/export network that

ultimately connect to this port (i.e., the port's fanin).

This method must not be called before the end_of_elaboration phase, as port connections are not resolved until then.

resolve_bindings

```
virtual function void resolve_bindings()
```

This callback is called just before entering the end_of_elaboration phase. It recurses through each port's fanout to determine all the imp destina- tions. It then checks against the required min and max connections. After resolution, size returns a valid value and get_if can be used to access a particular imp.

This method is automatically called just before the start of the end_of_elaboration phase. Users should not need to call it directly.

get_if

```
function uvm_port_base #(IF) get_if(int index=0)
```

Returns the implementation (imp) port at the given index from the array of imps this port is connected to. Use size to get the valid range for index. This method can only be called at the end_of_elaboration phase or after, as port connections are not resolved before then.

Phasing

UVM implements an automated mechanism for phasing the execution of the various components in a testbench.

Contents

Pre-Defined Phases

This section describes the set of pre-defined phases provided as a standard part of the UVM library.

Summary

Global Variables	schedule which runs concurrently to the run_ph global run phase.
pre_reset_ph	Before reset is asserted.
reset_ph	Reset is asserted.
post_reset_ph	After reset is de-asserted.
pre_configure_ph	Before the DUT is configured by the SW.
configure_ph	The SW configures the DUT.
post_configure_ph	After the SW has configured the DUT.
pre_main_ph	Before the primary test stimulus starts.
main_ph	Primary test stimulus.
post_main_ph	After enough of the primary test stimulus.
pre_shutdown_ph	Before things settle down.
shutdown_ph	Letting things settle down.
post_shutdown_ph	After things have settled down.

Common Phases Global Variables

The common phases are the set of function and task phases that all uvm_components execute together. All uvm_components are always synchronized with respect to the common phases.

The common phases are executed in the sequence they are specified below.

build_ph

Create and configure of testbench structure

uvm_topdown_phase that calls the uvm_component::build_phase method.

Upon entry

- The top-level components have been instantiated under uvm_root.
- Current simulation time is still equal to 0 but some "delta cycles" may have occurred

Typical Uses

- Instantiate sub-components.
- Instantiate register model.
- Get configuration values for the component being built.
- Set configuration values for sub-components.

Exit Criteria

- All uvm_components have been instantiated.

connect_ph

Establish cross-component connections.

uvm_bottomup_phase that calls the uvm_component::connect_phase method.

Upon Entry

- All components have been instantiated.
- Current simulation time is still equal to 0 but some "delta cycles" may have

occurred.

Typical Uses

- Connect TLM ports and exports.
- Connect TLM initiator sockets and target sockets.
- Connect register model to adapter components.
- Setup explicit phase domains.

Exit Criteria

- All cross-component connections have been established.
- All independent phase domains are set.

end_of_elaboration_ph

Fine-tune the testbench.

uvm_bottomup_phase that calls the uvm_component::end_of_elaboration_phase method.

Upon Entry

- The verification environment has been completely assembled.
- Current simulation time is still equal to 0 but some "delta cycles" may have occurred.

Typical Uses

- Display environment topology.
- Open files.
- Define additional configuration settings for components.

Exit Criteria

- None.

start_of_simulation_ph

Get ready for DUT to be simulated.

uvm_bottomup_phase that calls the uvm_component::start_of_simulation_phase method.

Upon Entry

- Other simulation engines, debuggers, hardware assisted platforms and all other run-time tools have been started and synchronized.
- The verification environment has been completely configured and is ready to start.
- Current simulation time is still equal to 0 but some "delta cycles" may have occurred.

Typical Uses

- Display environment topology
- Set debugger breakpoint
- Set initial run-time configuration values.

Exit Criteria

- None.

run_ph

Stimulate the DUT.

This uvm_task_phase calls the uvm_component::run_phase virtual method. This phase runs in parallel to the runtime phases, <uvm_pre_reset_ph> through <uvm_post_shutdown_ph>. All components in the testbench are synchronized with respect to the run phase regardles of the phase domain they belong to.

Upon Entry

- Indicates that power has been applied.
- There should not have been any active clock edges before entry into this phase (e.g. x->1 transitions via initial blocks).
- Current simulation time is still equal to 0 but some "delta cycles" may have occurred.

Typical Uses

- Components implement behavior that is exhibited for the entire run-time, across the various run-time phases.
- Backward compatibility with OVM.

Exit Criteria

- The DUT no longer needs to be simulated, and
- The <uvm_post_shutdown_ph> is ready to end

The run phase terminates in one of four ways.

1. Explicit call to global_stop_request

When global_stop_request is called, an ordered shut-down for the run phase begins. First, all enabled components' uvm_component::stop tasks are called bottom-up, i.e., childrens' uvm_component::stop tasks are called before the parent's.

Stopping a component is enabled by its uvm_component::enable_stop_interrupt bit. Each component can implement uvm_component::stop to allow completion of in-progress transactions, flush queues, and other shut-down activities. Upon return from uvm_component::stop by all enabled components, the run phase becomes ready to end pending completion of the runtime phases (i.e. the <uvm_post_shutdown_ph> being ready to end.

If any component raised a phase objection in uvm_component::run_phase(), this stopping procedure is deferred until all outstanding objections have been dropped.

2. All run phase objections have been dropped after having been raised

When all objections on the run phase objection have been dropped by the uvm_component::run_phase() methods, global_stop_request is called automatically, thus kicking off the stopping procedure described above.

If no component ever raises a phase objection, this termination mechanism never happens.

3. Explicit call to uvm_component::kill or uvm_component::do_kill_all

When uvm_component::kill is called, that component's uvm_component::run_phase processes are killed immediately. The uvm_component::do_kill_all methods applies to the component and all its descendants.

Use of this method is not recommended. It is better to use the stopping mechanism, which affords a more ordered, safer shut-down. If an immediate termination is desired, a <uvm_component::jump> to the <uvm_extract_ph> phase is recommended as this will cause both the run phase and the parallel runtime phases to immediately end and go to extract.

4. Timeout

The phase ends if the timeout expires before an explicit call to global_stop_request or uvm_component::kill. By default, the timeout is set to 0, which is no timeout. You may override this via set_global_timeout.

If a timeout occurs in your simulation, or if simulation never ends despite completion of your test stimulus, then it usually indicates a missing call to global_stop_request.

extract_ph

Extract data from different points of the verficiation environment.

uvm_bottomup_phase that calls the uvm_component::extract_phase method.

Upon Entry

- The DUT no longer needs to be simulated.
- Simulation time will no longer advance.

Typical Uses

- Extract any remaining data and final state information from scoreboard and testbench components
- Probe the DUT (via zero-time hierarchical references and/or backdoor accesses) for final state information.
- Compute statistics and summaries.
- Display final state information
- Close files.

Exit Criteria

- All data has been collected and summarized.

check_ph

Check for any unexpected conditions in the verification environment.

uvm_bottomup_phase that calls the uvm_component::check_phase method.

Upon Entry

- All data has been collected.

Typical Uses

- Check that no unaccounted-for data remain.

Exit Criteria

- Test is known to have passed or failed.

report_ph

Report results of the test.

uvm_bottomup_phase that calls the uvm_component::report_phase method.

Upon Entry

- Test is known to have passed or failed.

Typical Uses

- Report test results.
- Write results to file.

Exit Criteria

- End of test.

final_ph

Tie up loose ends.

uvm_topdown_phase that calls the uvm_component::final_phase method.

Upon Entry

- All test-related activity has completed.

Typical Uses

- Close files.
- Terminate co-simulation engines.

Exit Criteria

- Ready to exit simulator.

Run-Time Schedule Global Variables

The run-time schedule is the pre-defined phase schedule which runs concurrently to the run_ph global run phase. By default, all uvm_components using the run-time schedule are synchronized with respect to the pre-defined phases in the schedule. It is possible for components to belong to different domains in which case their schedules can be unsynchronized.

pre_reset_ph

Before reset is asserted.

uvm_task_phase that calls the uvm_component::pre_reset_phase method. This phase starts at the same time as the <uvm_run_ph> unless a user defined phase is inserted in

front of this phase.

Upon Entry

- Indicates that power has been applied but not necessarily valid or stable.
- There should not have been any active clock edges before entry into this phase.

Typical Uses

- Wait for power good.
- Components connected to virtual interfaces should initialize their output to X's or Z's.
- Initialize the clock signals to a valid value
- Assign reset signals to X (power-on reset).
- Wait for reset signal to be asserted if not driven by the verification environment.

Exit Criteria

- Reset signal, if driven by the verification environment, is ready to be asserted.
- Reset signal, if not driven by the verification environment, is asserted.

reset_ph

Reset is asserted.

uvm_task_phase that calls the uvm_component::reset_phase method.

Upon Entry

- Indicates that the hardware reset signal is ready to be asserted.

Typical Uses

- Assert reset signals.
- Components connected to virtual interfaces should drive their output to their specified reset or idle value.
- Components and environments should initialize their state variables.
- Clock generators start generating active edges.
- De-assert the reset signal(s) just before exit.
- Wait for the reset signal(s) to be de-asserted.

Exit Criteria

- Reset signal has just been de-asserted.
- Main or base clock is working and stable.
- At least one active clock edge has occurred.
- Output signals and state variables have been initialized.

post_reset_ph

After reset is de-asserted.

uvm_task_phase that calls the uvm_component::post_reset_phase method.

Upon Entry

- Indicates that the DUT reset signal has been de-asserted.

Typical Uses

- Components should start behavior appropriate for reset being inactive. For example, components may start to transmit idle transactions or interface training and rate negotiation. This behavior typically continues beyond the end of this phase.

Exit Criteria

- The testbench and the DUT are in a known, active state.

pre_configure_ph

Before the DUT is configured by the SW.

uvm_task_phase that calls the uvm_component::pre_configure_phase method.

Upon Entry

- Indicates that the DUT has been completed reset and is ready to be configured.

Typical Uses

- Procedurally modify the DUT configuration information as described in the environment (and that will be eventually uploaded into the DUT).
- Wait for components required for DUT configuration to complete training and rate negotiation.

Exit Criteria

- DUT configuration information is defined.

configure_ph

The SW configures the DUT.

uvm_task_phase that calls the uvm_component::configure_phase method.

Upon Entry

- Indicates that the DUT is ready to be configured.

Typical Uses

- Components required for DUT configuration execute transactions normally.
- Set signals and program the DUT and memories (e.g. read/write operations and sequences) to match the desired configuration for the test and environment.

Exit Criteria

- The DUT has been configured and is ready to operate normally.

post_configure_ph

After the SW has configured the DUT.

uvm_task_phase that calls the uvm_component::post_configure_phase method.

Upon Entry

- Indicates that the configuration information has been fully uploaded.

Typical Uses

- Wait for configuration information to fully propagate and take effect.
- Wait for components to complete training and rate negotiation.
- Enable the DUT.
- Sample DUT configuration coverage.

Exit Criteria

- The DUT has been fully configured and enabled and is ready to start operating normally.

pre_main_ph

Before the primary test stimulus starts.

uvm_task_phase that calls the uvm_component::pre_main_phase method.

Upon Entry

- Indicates that the DUT has been fully configured.

Typical Uses

- Wait for components to complete training and rate negotiation.

Exit Criteria

- All components have completed training and rate negotiation.
- All components are ready to generate and/or observe normal stimulus.

main_ph

Primary test stimulus.

uvm_task_phase that calls the uvm_component::main_phase method.

Upon Entry

- The stimulus associated with the test objectives is ready to be applied.

Typical Uses

- Components execute transactions normally.
- Data stimulus sequences are started.
- Wait for a time-out or certain amount of time, or completion of stimulus sequences.

Exit Criteria

- Enough stimulus has been applied to meet the primary stimulus objective of the test.

post_main_ph

After enough of the primary test stimulus.

uvm_task_phase that calls the uvm_component::post_main_phase method.

Upon Entry

- The primary stimulus objective of the test has been met.

Typical Uses

- Included for symmetry.

Exit Criteria

- None.

pre_shutdown_ph

Before things settle down.

uvm_task_phase that calls the uvm_component::pre_shutdown_phase method.

Upon Entry

- None.

Typical Uses

- Included for symmetry.

Exit Criteria

- None.

shutdown_ph

Letting things settle down.

uvm_task_phase that calls the uvm_component::shutdown_phase method.

Upon Entry

- None.

Typical Uses

- Wait for all data to be drained out of the DUT.
- Extract data still buffered in the DUT, usually through read/write operations or sequences.

Exit Criteria

- All data has been drained or extracted from the DUT.
- All interfaces are idle.

post_shutdown_ph

After things have settled down.

uvm_task_phase that calls the uvm_component::post_shutdown_phase method. The end of this phase is synchronized to the end of the <uvm_run_ph> phase unless a user defined phase is added after this phase.

Upon Entry

- No more "data" stimulus is applied to the DUT.

Typical Uses

- Perform final checks that require run-time access to the DUT (e.g. read accounting registers or dump the content of memories).

Exit Criteria

- All run-time checks have been satisfied.
- The <uvm_run_ph> phase is ready to end.

User-Defined Phases

To defined your own custom phase, use the following pattern

1. extend the appropriate base class for your phase type

```
class my_PHASE_phase extends uvm_task_phase("PHASE");
class my_PHASE_phase extends uvm_topdown_phase("PHASE");
class my_PHASE_phase extends uvm_bottomup_phase("PHASE");
```

2. implement your exec_task or exec_func method

```
task exec_task(uvm_component comp, uvm_phase schedule);
function void exec_func(uvm_component comp, uvm_phase schedule);
```

3. the implementation usually calls the related method on the component

```
comp.PHASE_phase(uvm_phase phase);
```

4. after declaring your phase singleton class, instantiate one for global use

```
static my_``PHASE``_phase my_``PHASE``_ph = new();
```

5. insert the phase in a schedule using the uvm_phase::add_phase.method in side your VIP base class's definition of the <uvm_phase::define_phase_schedule> method.

Summary

User-Defined Phases

To defined your own custom phase, use the following pattern

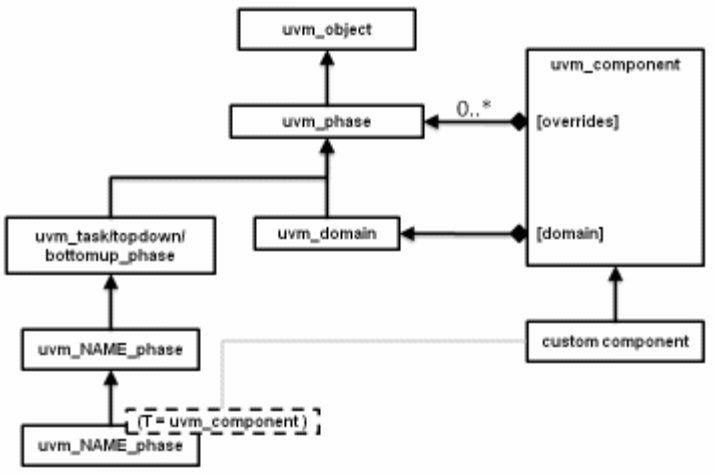
Phasing Implementation

The API described here provides a general purpose testbench phasing solution, consisting of a phaser machine, traversing a master schedule graph, which is built by the integrator from one or more instances of template schedules provided by UVM or by 3rd-party VIP, and which supports implicit or explicit synchronization, runtime control of threads and jumps.

Each schedule leaf node refers to a single phase that is compatible with that VIP's components and which executes the required behavior via a functor or delegate extending the phase into component context as required. Execution threads are tracked on a per-component basis and various thread semantics available to allow defined phase control and responsibility.

Class hierarchy

A single class represents both the definition, the state, and the context of a phase. It is instantiated once as a singleton IMP and one or more times as nodes in a graph which represents serial and parallel phase relationships and stores current state as the phaser progresses, and the phase implementation which specifies required component behavior (by extension into component context if non-default behavior required.)



The following classes related to phasing are defined herein

uvm_phase : The base class for defining a phase's behavior, state, context

uvm_bottomup_phase : A phase implemenation for bottom up function phases.

uvm_topdown_phase : A phase implemenation for topdown function phases.

uvm_task_phase : A phase implemenation for task phases.

Summary

Phasing Implementation

The API described here provides a general purpose testbench phasing solution, consisting of a phaser machine, traversing a master schedule graph, which is built by the integrator from one or more instances of template schedules provided by UVM or by 3rd-party VIP, and which supports implicit or explicit synchronization, runtime control of threads and jumps.

uvm_phase

This base class defines everything about a phase: behavior, state, and context

To define behavior, it is extended by UVM or the user to create singleton objects which capture the definition of what the phase does and how it does it. These are then cloned to produce multiple nodes which are hooked up in a graph structure to provide context: which phases follow which, and to hold the state of the phase throughout its lifetime. UVM provides default extensions of this class for the standard runtime phases. VIP Providers can likewise extend this class to define the phase functor for a particular component context as required.

Phase Definition

Singleton instances of those extensions are provided as package variables. These instances define the attributes of the phase (not what state it is in) They are then cloned into schedule nodes which point back to one of these implementations, and calls it's virtual task or function methods on each participating component. It is the base class for phase functors, for both predefined and user-defined phases. Per-component overrides can use a customized imp.

To create custom phases, do not extend uvm_phase directly: see the three predefined extended classes below which encapsulate behavior for different phase types: task, bottom-up function and top-down function.

Extend the appropriate one of these to create a uvm_YOURNAME_phase class (or YOURPREFIX_NAME_phase class) for each phase, containing the default implementation of the new phase, which must be a uvm_component-compatible delegate, and which may be a null implementation. Instantiate a singleton instance of that class for your code to use when a phase handle is required. If your custom phase depends on methods that are not in uvm_component, but are within an extended class, then extend the base YOURPREFIX_NAME_phase class with parameterized component class context as required, to create a specialized functor which calls your extended component class methods. This scheme ensures compile-safety for your extended component classes while providing homogeneous base types for APIs and underlying data structures.

Phase Context

A schedule is a coherent group of one or mode phase/state nodes linked together by a graph structure, allowing arbitrary linear/parallel relationships to be specified, and executed by stepping through them in the graph order. Each schedule node points to a phase and holds the execution state of that phase, and has optional links to other nodes for synchronization.

The main build operations are: construct, add phases, and instantiate hierarchically within another schedule.

Structure is a DAG (Directed Acyclic Graph). Each instance is a node connected to others to form the graph. Hierarchy is overlaid with m_parent. Each node in the graph

has zero or more successors, and zero or more predecessors. No nodes are completely isolated from others. Exactly one node has zero predecessors. This is the root node. Also the graph is acyclic, meaning for all nodes in the graph, by following the forward arrows you will never end up back where you started but you will eventually reach a node that has no successors.

Phase State

A given phase may appear multiple times in the complete phase graph, due to the multiple independent domain feature, and the ability for different VIP to customize their own phase schedules perhaps reusing existing phases. Each node instance in the graph maintains its own state of execution.

Phase Handle

Handles of this type uvm_phase are used frequently in the API, both by the user, to access phasing-specific API, and also as a parameter to some APIs. In many cases, the singleton package-global phase handles can be used (eg. connect_ph, run_ph) in APIs. For those APIs that need to look up that phase in the graph, this is done automatically.

Summary

uvm_phase

This base class defines everything about a phase: behavior, state, and context

Class Hierarchy

uvm_void

uvm_object

uvm_phase

Class Declaration

```
class uvm_phase extends uvm_object
```

Construction

new	Create a new phase node, with a name and a note of its type name - name of this phase type - task, topdown func or bottomup func
get_phase_type	Returns the phase type as defined by uvm_phase_type

State

get_state	Accessor to return current state of this phase
get_run_count	Accessor to return the integer number of times this phase has executed
find	Locate a phase node with the specified *name* and return its handle.
is	returns 1 if the containing uvm_phase refers to the same phase as the phase argument, 0 otherwise
is_before	Returns 1 if the containing uvm_phase refers to a phase that is earlier than the phase argument, 0 otherwise
is_after	returns 1 if the containing uvm_phase refers to a phase that is later than the phase argument, 0 otherwise

Callbacks

exec_func	Implements the functor/delegate functionality for a function phase type comp - the component to execute the functionality upon phase - the phase schedule that originated this phase call
exec_task	Implements the functor/delegate functionality for a task phase type comp - the component to execute the

	functionality upon phase - the phase schedule that originated this phase call
phase_started	Generic notification function called prior to exec_func()/exec_task() phase - the phase schedule that originated this phase call
phase_ended	Generic notification function called after exec_func()/exec_task() phase - the phase schedule that originated this phase call
SCHEDULE	
add_phase	Build up a schedule structure inserting phase by phase, specifying linkage
add_schedule	Build up schedule structure by adding another schedule flattened within it.
get_parent	Returns the parent schedule node, if any, for hierarchical graph traversal
get_schedule	Returns the topmost parent schedule node, if any, for hierarchical graph traversal
get_schedule_name	Accessor to return the schedule name associated with this schedule
SYNCHRONIZATION	
get_objection	Return the uvm_objection that gates the termination of the phase.
raise_objection	Raise an objection to ending this phase Provides components with greater control over the phase flow for processes which are not implicit objectors to the phase.
drop_objection	Drop an objection to ending this phase
sync	Synchonize two domains, fully or partially
unsync	Remove synchonization between two domains, fully or partially
wait_for_state	Wait until this phase compares with the given *state* and *op* operand.
JUMPING	
jump	Jump to a specified *phase*.
jump_all	Make all schedules jump to a specified *phase*, even if the jump target is local.
get_jump_target	Return handle to the target phase of the current jump, or null if no jump is in progress.

CONSTRUCTION

new

```
function new(string          name,
             uvm_phase_type  phase_type,
             uvm_phase       parent      = null)
```

Create a new phase node, with a name and a note of its type name - name of this phase type - task, topdown func or bottomup func

get_phase_type

```
function uvm_phase_type get_phase_type()
```

Returns the phase type as defined by uvm_phase_type

STATE

get_state

```
function uvm_phase_state get_state()
```

Accessor to return current state of this phase

get_run_count

```
function int get_run_count()
```

Accessor to return the integer number of times this phase has executed

find

```
function uvm_phase find(string name)
```

Locate a phase node with the specified *name* and return its handle. Look first within the current schedule, then current domain, then global

is

```
function bit is(uvm_phase phase)
```

returns 1 if the containing uvm_phase refers to the same phase as the phase argument, 0 otherwise

is_before

```
function bit is_before(uvm_phase phase)
```

Returns 1 if the containing uvm_phase refers to a phase that is earlier than the phase argument, 0 otherwise

is_after

```
function bit is_after(uvm_phase phase)
```

returns 1 if the containing uvm_phase refers to a phase that is later than the phase argument, 0 otherwise

CALLBACKS

exec_func

```
virtual function void exec_func(uvm_component comp,
```

```
                                    uvm_phase      phase)
```

Implements the functor/delegate functionality for a function phase type comp - the component to execute the functionality upon phase - the phase schedule that originated this phase call

exec_task

```
virtual task exec_task(uvm_component comp,
                       uvm_phase     phase)
```

Implements the functor/delegate functionality for a task phase type comp - the component to execute the functionality upon phase - the phase schedule that originated this phase call

phase_started

```
virtual function void phase_started(uvm_phase phase)
```

Generic notification function called prior to exec_func()/exec_task() phase - the phase schedule that originated this phase call

phase_ended

```
virtual function void phase_ended(uvm_phase phase)
```

Generic notification function called after exec_func()/exec_task() phase - the phase schedule that originated this phase call

SCHEDULE

add_phase

```
function void add_phase(uvm_phase phase,
                        uvm_phase with_phase   = null,
                        uvm_phase after_phase  = null,
                        uvm_phase before_phase = null )
```

Build up a schedule structure inserting phase by phase, specifying linkage

Phases can be added anywhere, in series or parallel with existing nodes

phase	handle of singleton derived imp containing actual functor. by default the new phase is appended to the schedule
with_phase	specify to add the new phase in parallel with this one
after_phase	specify to add the new phase as successor to this one
before_phase	specify to add the new phase as predecessor to this one

add_schedule

```
function void add_schedule(uvm_phase schedule,
```

```
                              uvm_phase with_phase   = null,
                              uvm_phase after_phase  = null,
                              uvm_phase before_phase = null )
```

Build up schedule structure by adding another schedule flattened within it.

Inserts a schedule structure hierarchically within the enclosing schedule's graph. It is essentially flattened graph-wise, but the hierarchy is preserved by the 'm_parent' handles which point to that schedule's begin node.

schedule	handle of new schedule to insert within this one
with_phase	specify to add the schedule in parallel with this phase node
after_phase	specify to add the schedule as successor to this phase node
before_phase	specify to add the schedule as predecessor to this phase node

get_parent

```
function uvm_phase get_parent()
```

Returns the parent schedule node, if any, for hierarchical graph traversal

get_schedule

```
function uvm_phase get_schedule()
```

Returns the topmost parent schedule node, if any, for hierarchical graph traversal

get_schedule_name

```
function string get_schedule_name()
```

Accessor to return the schedule name associated with this schedule

SYNCHRONIZATION

get_objection

```
function uvm_objection get_objection()
```

Return the uvm_objection that gates the termination of the phase.

raise_objection

```
virtual function void raise_objection (uvm_object obj,
                                       string     description = "",
                                       int        count       = 1 )
```

Raise an objection to ending this phase Provides components with greater control over the phase flow for processes which are not implicit objectors to the phase.

```
while(1) begin
  some_phase.raise_objection(this);
  ...
  some_phase.drop_objection(this);
end
...
```

drop_objection

```
virtual function void drop_objection (uvm_object obj,
                                      string     description = "",
                                      int        count       = 1 )
```

Drop an objection to ending this phase

The drop is expected to be matched with an earlier raise.

sync

```
function void sync(uvm_domain target,
                   uvm_phase  phase      = null,
                   uvm_phase  with_phase = null )
```

Synchonize two domains, fully or partially

target	handle of target domain to synchronize this one to
phase	optional single phase to synchronize, otherwise all
with_phase	optional different target-domain phase to synchronize with

unsync

```
function void unsync(uvm_domain target,
                     uvm_phase  phase      = null,
                     uvm_phase  with_phase = null )
```

Remove synchonization between two domains, fully or partially

target	handle of target domain to remove synchronization from
phase	optional single phase to un-synchronize, otherwise all
with_phase	optional different target-domain phase to un-synchronize with

wait_for_state

```
task wait_for_state(uvm_phase_state state,
                    uvm_wait_op     op     = UVM_EQ)
```

Wait until this phase compares with the given *state* and *op* operand. For UVM_EQ and UVM_NE operands, several uvm_phase_states can be supplied by ORing their enum constants, in which case the caller will wait until the phase state is any of (UVM_EQ) or none of (UVM_NE) the provided states.

To wait for the phase to be at the started state or after

```
wait_for_state(UVM_PHASE_STARTED, UVM_GT);
```

To wait for the phase to be either started or executing

```
wait_for_state(UVM_PHASE_STARTED | UVM_PHASE_EXECUTING, UVM_EQ);
```

JUMPING

jump

```
function void jump(uvm_phase phase)
```

Jump to a specified *phase*. If the destination *phase* is within the current phase schedule, a simple local jump takes place. If the jump-to *phase* is outside of the current schedule then the jump affects other schedules which share the phase.

jump_all

```
static function void jump_all(uvm_phase phase)
```

Make all schedules jump to a specified *phase*, even if the jump target is local. The jump happens to all phase schedules that contain the jump-to *phase*, i.e. a global jump.

get_jump_target

```
function uvm_phase get_jump_target()
```

Return handle to the target phase of the current jump, or null if no jump is in progress. Valid for use during the phase_ended() callback

uvm_domain

Phasing schedule node representing an independent branch of the schedule. Handle used to assign domains to components or hierarchies in the testbench

Summary

uvm_domain

Phasing schedule node representing an independent branch of the schedule.

CLASS HIERARCHY

uvm_void

uvm_object

uvm_phase

uvm_domain

CLASS DECLARATION

```
class uvm_domain extends uvm_phase
```

METHODS

get_common_domain	Get the common domain objection which consists of the common phases that all components executed together (build, connect, ..., report, final).

METHODS

get_common_domain

```
static function uvm_domain get_common_domain()
```

Get the common domain objection which consists of the common phases that all components executed together (build, connect, ..., report, final).

uvm_bottomup_phase

Virtual base class for function phases that operate bottom-up. The pure virtual function execute() is called for each component. This is the default traversal so is included only for naming.

A bottom-up function phase completes when the execute() method has been called and returned on all applicable components in the hierarchy.

Summary

uvm_bottomup_phase

Virtual base class for function phases that operate bottom-up.

CLASS HIERARCHY

uvm_void
uvm_object
uvm_phase
uvm_bottomup_phase

CLASS DECLARATION

```
virtual class uvm_bottomup_phase extends uvm_phase
```

METHODS

new	Create a new instance of a bottom-up phase.
traverse	Traverses the component tree in bottom-up order, calling execute for each component.
execute	Executes the bottom-up phase *phase* for the component *comp*.

Methods

new

```
function new(string name)
```

Create a new instance of a bottom-up phase.

traverse

```
virtual function void traverse(uvm_component   comp,
                               uvm_phase       phase,
                               uvm_phase_state state )
```

Traverses the component tree in bottom-up order, calling execute for each component.

execute

```
protected virtual function void execute(uvm_component comp,
                                        uvm_phase     phase)
```

Executes the bottom-up phase *phase* for the component *comp*.

uvm_topdown_phase

Virtual base class for function phases that operate top-down. The pure virtual function execute() is called for each component.

A top-down function phase completes when the execute() method has been called and returned on all applicable components in the hierarchy.

Summary

uvm_topdown_phase

Virtual base class for function phases that operate top-down.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_phase
- **uvm_topdown_phase**

Class Declaration

```
virtual class uvm_topdown_phase extends uvm_phase
```

Methods

new	Create a new instance of a top-down phase
traverse	Traverses the component tree in top-down order, calling execute

	for each component.
execute	Executes the top-down phase *phase* for the component *comp*.

METHODS

new

```
function new(string name)
```

Create a new instance of a top-down phase

traverse

```
virtual function void traverse(uvm_component   comp,
                               uvm_phase       phase,
                               uvm_phase_state state )
```

Traverses the component tree in top-down order, calling execute for each component.

execute

```
protected virtual function void execute(uvm_component comp,
                                        uvm_phase     phase)
```

Executes the top-down phase *phase* for the component *comp*.

uvm_task_phase

Base class for all task phases. It forks a call to uvm_phase::exec_task() for each component in the hierarchy.

A task phase completes when there are no raised objections to the end of phase. The completion of the task does not imply, nor is it required for, the end of phase. Once the phase completes, any remaining forked uvm_phase::exec_task() threads are forcibly and immediately killed.

The only way for a task phase to extend over time is if there is at least one component that raises an objection.

```
class my_comp extends uvm_component;
   task main_phase(uvm_phase phase);
      phase.raise_objection(this, "Applying stimulus")
      ...
      phase.drop_objection(this, "Applied enough stimulus")
   endtask
endclass
```

Summary

uvm_task_phase

Base class for all task phases.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_phase
- **uvm_task_phase**

Class Declaration

```
virtual class uvm_task_phase extends uvm_phase
```

Methods

new	Create a new instance of a task-based phase
traverse	Traverses the component tree in bottom-up order, calling execute for each component.
execute	Fork the task-based phase *phase* for the component *comp*.

Methods

new

```
function new(string name)
```

Create a new instance of a task-based phase

traverse

```
virtual function void traverse(uvm_component   comp,
                               uvm_phase       phase,
                               uvm_phase_state state )
```

Traverses the component tree in bottom-up order, calling execute for each component. The actual order for task-based phases doesn't really matter, as each component task is executed in a separate process whose starting order is not deterministic.

execute

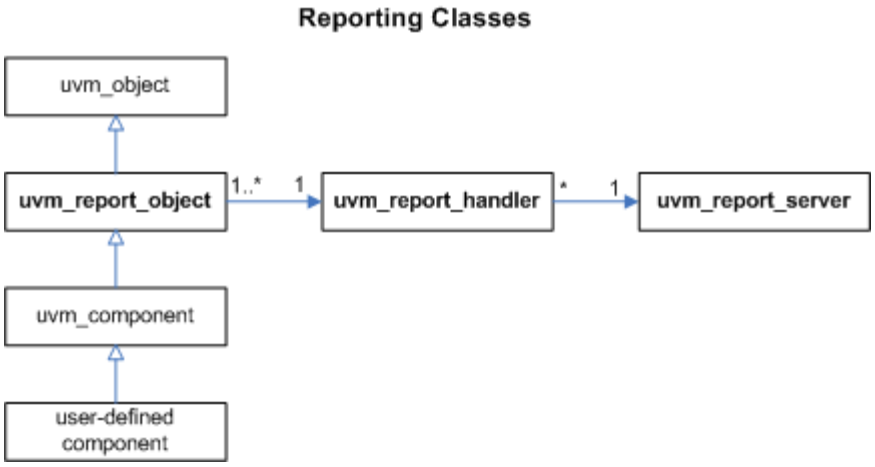
```
protected virtual function void execute(uvm_component comp,
                                        uvm_phase     phase)
```

Fork the task-based phase *phase* for the component *comp*.

Reporting Classes

The reporting classes provide a facility for issuing reports with consistent formatting. Users can configure what actions to take and what files to send output to based on report severity, ID, or both severity and ID. Users can also filter messages based on their verbosity settings.

The primary interface to the UVM reporting facility is the uvm_report_object from which all uvm_components extend. The uvm_report_object delegates most tasks to its internal uvm_report_handler. If the report handler determines the report is not filtered based the configured verbosity setting, it sends the report to the central uvm_report_server for formatting and processing.



Summary

Reporting Classes

The reporting classes provide a facility for issuing reports with consistent formatting.

uvm_report_object

The uvm_report_object provides an interface to the UVM reporting facility. Through this interface, components issue the various messages that occur during simulation. Users can configure what actions are taken and what file(s) are output for individual messages from a particular component or for all messages from all components in the environment. Defaults are applied where there is no explicit configuration.

Most methods in uvm_report_object are delegated to an internal instance of an uvm_report_handler, which stores the reporting configuration and determines whether an issued message should be displayed based on that configuration. Then, to display a message, the report handler delegates the actual formatting and production of messages to a central uvm_report_server.

A report consists of an id string, severity, verbosity level, and the textual message itself. They may optionally include the filename and line number from which the message came. If the verbosity level of a report is greater than the configured maximum verbosity level of its report object, it is ignored. If a report passes the verbosity filter in effect, the report's action is determined. If the action includes output to a file, the configured file descriptor(s) are determined.

Actions — can be set for (in increasing priority) severity, id, and (severity,id) pair. They include output to the screen UVM_DISPLAY, whether the message counters should be incremented UVM_COUNT, and whether a $finish should occur UVM_EXIT.

Default Actions — The following provides the default actions assigned to each severity. These can be overridden by any of the set_*_action methods.

```
UVM_INFO -        UVM_DISPLAY
UVM_WARNING -     UVM_DISPLAY
UVM_ERROR -       UVM_DISPLAY | UVM_COUNT
UVM_FATAL -       UVM_DISPLAY | UVM_EXIT
```

File descriptors — These can be set by (in increasing priority) default, severity level, an id, or (severity,id) pair. File descriptors are standard verilog file descriptors; they may refer to more than one file. It is the user's responsibility to open and close them.

Default file handle — The default file handle is 0, which means that reports are not sent to a file even if an UVM_LOG attribute is set in the action associated with the report. This can be overridden by any of the set_*_file methods.

Summary

uvm_report_object

The uvm_report_object provides an interface to the UVM reporting facility.

Class Hierarchy

- uvm_void
- uvm_object
- **uvm_report_object**

Class Declaration

```
class uvm_report_object extends uvm_object
```

new	Creates a new report object with the given name.

Reporting

uvm_report_info uvm_report_warning uvm_report_error uvm_report_fatal	These are the primary reporting methods in the UVM.

Callbacks

report_info_hook report_error_hook report_warning_hook report_fatal_hook report_hook	These hook methods can be defined in derived classes to perform additional actions when reports are issued.
report_header	Prints version and copyright information.
report_summarize	Outputs statistical information on the reports issued by the central report server.
die	This method is called by the report server if a report reaches the maximum quit count or has an UVM_EXIT action associated with it, e.g., as with fatal errors.

Configuration

set_report_verbosity_level	This method sets the maximum verbosity level for reports for this component.
set_report_id_verbosity set_report_severity_id_verbosity	These methods associate the specified verbosity with reports of the given *severity*, *id*, or *severity-id* pair.
set_report_severity_action set_report_id_action set_report_severity_id_action	These methods associate the specified action or actions with reports of the given *severity*, *id*, or *severity-id* pair.
set_report_severity_override set_report_severity_id_override	These methods provide the ability to upgrade or downgrade a message in terms of severity given *severity* and *id*.
set_report_default_file set_report_severity_file set_report_id_file set_report_severity_id_file	These methods configure the report handler to direct some or all of its output to the given file descriptor.
get_report_verbosity_level	Gets the verbosity level in effect for this object.
get_report_action	Gets the action associated with reports having the given *severity* and *id*.
get_report_file_handle	Gets the file descriptor associated with reports having the given *severity* and *id*.
uvm_report_enabled	Returns 1 if the configured verbosity for this severity/id is greater than *verbosity* and the action associated with the given *severity* and *id* is not UVM_NO_ACTION, else returns 0.
set_report_max_quit_count	Sets the maximum quit count in the report handler to *max_count*.

Setup

set_report_handler	Sets the report handler, overwriting the

	default instance.
get_report_handler	Returns the underlying report handler to which most reporting tasks are delegated.
reset_report_handler	Resets the underlying report handler to its default settings.
get_report_server	Returns the uvm_report_server instance associated with this report object.
dump_report_state	This method dumps the internal state of the report handler.

new

```
function new(string name = "")
```

Creates a new report object with the given name. This method also creates a new uvm_report_handler object to which most tasks are delegated.

Reporting

uvm_report_info

```
virtual function void uvm_report_info(string id,
                                      string message,
                                      int    verbosity = UVM_MEDIUM,
                                      string filename  = "",
                                      int    line      = 0          )
```

uvm_report_warning

```
virtual function void uvm_report_warning(string id,
                                         string message,
                                         int    verbosity = UVM_MEDIUM,
                                         string filename  = "",
                                         int    line      = 0          )
```

uvm_report_error

```
virtual function void uvm_report_error(string id,
                                       string message,
                                       int    verbosity = UVM_LOW,
                                       string filename  = "",
                                       int    line      = 0       )
```

uvm_report_fatal

```
virtual function void uvm_report_fatal(string id,
                                       string message,
                                       int    verbosity = UVM_NONE,
                                       string filename  = "",
                                       int    line      = 0       )
```

These are the primary reporting methods in the UVM. Using these instead of *$display* and other ad hoc approaches ensures consistent output and central control over where

output is directed and any actions that result. All reporting methods have the same arguments, although each has a different default verbosity:

id	a unique id for the report or report group that can be used for identification and therefore targeted filtering. You can configure an individual report's actions and output file(s) using this id string.
message	the message body, preformatted if necessary to a single string.
verbosity	the verbosity of the message, indicating its relative importance. If this number is less than or equal to the effective verbosity level, see set_report_verbosity_level, then the report is issued, subject to the configured action and file descriptor settings. Verbosity is ignored for warnings, errors, and fatals. However, if a warning, error or fatal is demoted to an info message using the uvm_report_catcher, then the verbosity is taken into account.
filename/line	(Optional) The location from which the report was issued. Use the predefined macros, `__FILE__ and `__LINE__. If specified, it is displayed in the output.

CALLBACKS

report_info_hook

```
virtual function bit report_info_hook(string id,
                                      string message,
                                      int    verbosity,
                                      string filename,
                                      int    line      )
```

report_error_hook

```
virtual function bit report_error_hook(string id,
                                       string message,
                                       int    verbosity,
                                       string filename,
                                       int    line      )
```

report_warning_hook

```
virtual function bit report_warning_hook(string id,
                                         string message,
                                         int    verbosity,
                                         string filename,
                                         int    line      )
```

report_fatal_hook

```
virtual function bit report_fatal_hook(string id,
                                       string message,
                                       int    verbosity,
                                       string filename,
                                       int    line      )
```

report_hook

```
virtual function bit report_hook(string id,
                                 string message,
                                 int    verbosity,
                                 string filename,
                                 int    line      )
```

These hook methods can be defined in derived classes to perform additional actions when reports are issued. They are called only if the UVM_CALL_HOOK bit is specified in the action associated with the report. The default implementations return 1, which allows the report to be processed. If an override returns 0, then the report is not processed.

First, the hook method associated with the report's severity is called with the same arguments as the given the report. If it returns 1, the catch-all method, report_hook, is then called. If the severity-specific hook returns 0, the catch-all hook is not called.

report_header

```
virtual function void report_header(UVM_FILE file = 0)
```

Prints version and copyright information. This information is sent to the command line if *file* is 0, or to the file descriptor *file* if it is not 0. The uvm_root::run_test task calls this method just before it component phasing begins.

report_summarize

```
virtual function void report_summarize(UVM_FILE file = 0)
```

Outputs statistical information on the reports issued by the central report server. This information will be sent to the command line if *file* is 0, or to the file descriptor *file* if it is not 0.

The run_test method in uvm_top calls this method.

die

```
virtual function void die()
```

This method is called by the report server if a report reaches the maximum quit count or has an UVM_EXIT action associated with it, e.g., as with fatal errors.

If this report object is an uvm_component and we're in a task-based phase (e.g. run), then die will issue a global_stop_request, which ends the phase and allows simulation to continue to the next phase.

If not a component, die calls report_summarize and terminates simulation with *$finish*.

CONFIGURATION

set_report_verbosity_level

```
function void set_report_verbosity_level (int verbosity_level)
```

This method sets the maximum verbosity level for reports for this component. Any report from this component whose verbosity exceeds this maximum will be ignored.

set_report_id_verbosity

```
function void set_report_id_verbosity (string id,
                                       int    verbosity)
```

set_report_severity_id_verbosity

```
function void set_report_severity_id_verbosity (uvm_severity severity,
                                                string       id,
                                                int          verbosity)
```

These methods associate the specified verbosity with reports of the given *severity*, *id*, or *severity-id* pair. An verbosity associated with a particular *severity-id* pair takes precedence over an verbosity associated with *id*, which take precedence over an an verbosity associated with a *severity*.

The *verbosity* argument can be any integer, but is most commonaly a predefined uvm_verbosity value, UVM_NONE, UVM_LOW, UVM_MEDIUM, UVM_HIGH, UVM_FULL.

set_report_severity_action

```
function void set_report_severity_action (uvm_severity severity,
                                          uvm_action   action   )
```

set_report_id_action

```
function void set_report_id_action (string     id,
                                    uvm_action action)
```

set_report_severity_id_action

```
function void set_report_severity_id_action (uvm_severity severity,
                                             string       id,
                                             uvm_action   action   )
```

These methods associate the specified action or actions with reports of the given *severity*, *id*, or *severity-id* pair. An action associated with a particular *severity-id* pair takes precedence over an action associated with *id*, which takes precedence over an an action associated with a *severity*.

The *action* argument can take the value UVM_NO_ACTION, or it can be a bitwise OR of any combination of UVM_DISPLAY, UVM_LOG, UVM_COUNT, UVM_STOP, UVM_EXIT, and UVM_CALL_HOOK.

set_report_severity_override

```
function void set_report_severity_override(uvm_severity cur_severity,
                                           uvm_severity new_severity )
```

set_report_severity_id_override

```
function void set_report_severity_id_override(uvm_severity cur_severity,
                                              string       id,
                                              uvm_severity new_severity )
```

These methods provide the ability to upgrade or downgrade a message in terms of severity given *severity* and *id*. An upgrade or downgrade for a specific *id* takes precedence over an upgrade or downgrade associated with a *severity*.

set_report_default_file

```
function void set_report_default_file (UVM_FILE file)
```

set_report_severity_file

```
function void set_report_severity_file (uvm_severity severity,
                                        UVM_FILE     file     )
```

set_report_id_file

```
function void set_report_id_file (string   id,
                                  UVM_FILE file)
```

set_report_severity_id_file

```
function void set_report_severity_id_file (uvm_severity severity,
                                           string       id,
                                           UVM_FILE     file     )
```

These methods configure the report handler to direct some or all of its output to the given file descriptor. The *file* argument must be a multi-channel descriptor (mcd) or file id compatible with $fdisplay.

A FILE descriptor can be associated with with reports of the given *severity*, *id*, or *severity-id* pair. A FILE associated with a particular *severity-id* pair takes precedence over a FILE associated with *id*, which take precedence over an a FILE associated with a *severity*, which takes precedence over the default FILE descriptor.

When a report is issued and its associated action has the UVM_LOG bit set, the report will be sent to its associated FILE descriptor. The user is responsible for opening and closing these files.

get_report_verbosity_level

```
function int get_report_verbosity_level(uvm_severity severity = UVM_INFO,
                                        string       id       = ""       )
```

Gets the verbosity level in effect for this object. Reports issued with verbosity greater than this will be filtered out. The severity and tag arguments check if the verbosity level has been modified for specific severity/tag combinations.

get_report_action

```
function int get_report_action(uvm_severity severity,
                               string       id       )
```

Gets the action associated with reports having the given *severity* and *id*.

get_report_file_handle

```
function int get_report_file_handle(uvm_severity severity,
                                    string       id       )
```

Gets the file descriptor associated with reports having the given *severity* and *id*.

uvm_report_enabled

```
function int uvm_report_enabled(int          verbosity,
                                uvm_severity severity   = UVM_INFO,
                                string       id         = ""       )
```

Returns 1 if the configured verbosity for this severity/id is greater than *verbosity* and the action associated with the given *severity* and *id* is not UVM_NO_ACTION, else returns 0.

See also get_report_verbosity_level and get_report_action, and the global version of uvm_report_enabled.

set_report_max_quit_count

```
function void set_report_max_quit_count(int max_count)
```

Sets the maximum quit count in the report handler to *max_count*. When the number of UVM_COUNT actions reaches *max_count*, the die method is called.

The default value of 0 indicates that there is no upper limit to the number of UVM_COUNT reports.

Setup

set_report_handler

```
function void set_report_handler(uvm_report_handler handler)
```

Sets the report handler, overwriting the default instance. This allows more than one component to share the same report handler.

get_report_handler

```
function uvm_report_handler get_report_handler()
```

Returns the underlying report handler to which most reporting tasks are delegated.

reset_report_handler

```
function void reset_report_handler
```

Resets the underlying report handler to its default settings. This clears any settings made with the set_report_* methods (see below).

get_report_server

```
function uvm_report_server get_report_server()
```

Returns the uvm_report_server instance associated with this report object.

dump_report_state

```
function void dump_report_state()
```

This method dumps the internal state of the report handler. This includes information about the maximum quit count, the maximum verbosity, and the action and files associated with severities, ids, and (severity, id) pairs.

uvm_report_handler

The uvm_report_handler is the class to which most methods in uvm_report_object delegate. It stores the maximum verbosity, actions, and files that affect the way reports are handled.

The report handler is not intended for direct use. See uvm_report_object for information on the UVM reporting mechanism.

The relationship between uvm_report_object (a base class for uvm_component) and uvm_report_handler is typically one to one, but it can be many to one if several uvm_report_objects are configured to use the same uvm_report_handler_object. See uvm_report_object::set_report_handler.

The relationship between uvm_report_handler and uvm_report_server is many to one.

Summary

uvm_report_handler

The uvm_report_handler is the class to which most methods in uvm_report_object delegate.

METHODS

new	Creates and initializes a new uvm_report_handler object.
run_hooks	The run_hooks method is called if the UVM_CALL_HOOK action is set for a report.
get_verbosity_level	Returns the verbosity associated with the given *severity* and *id*.
get_action	Returns the action associated with the given *severity* and *id*.
get_file_handle	Returns the file descriptor associated with the given *severity* and *id*.
report	This is the common handler method used by the four core reporting methods (e.g., uvm_report_error) in uvm_report_object.
format_action	Returns a string representation of the *action*, e.g., "DISPLAY".

METHODS

new

```
function new()
```

Creates and initializes a new uvm_report_handler object.

run_hooks

```
virtual function bit run_hooks(uvm_report_object client,
                               uvm_severity      severity,
                               string            id,
                               string            message,
                               int               verbosity,
```

```
                                string            filename,
                                int               line      )
```

The run_hooks method is called if the UVM_CALL_HOOK action is set for a report. It first calls the client's uvm_report_object::report_hook method, followed by the appropriate severity-specific hook method. If either returns 0, then the report is not processed.

get_verbosity_level

```
function int get_verbosity_level(uvm_severity severity = UVM_INFO,
                                  string       id       = ""       )
```

Returns the verbosity associated with the given *severity* and *id*.

First, if there is a verbosity associated with the *(severity,id)* pair, return that. Else, if there is an verbosity associated with the *id*, return that. Else, return the max verbosity setting.

get_action

```
function uvm_action get_action(uvm_severity severity,
                               string       id       )
```

Returns the action associated with the given *severity* and *id*.

First, if there is an action associated with the *(severity,id)* pair, return that. Else, if there is an action associated with the *id*, return that. Else, if there is an action associated with the *severity*, return that. Else, return the default action associated with the *severity*.

get_file_handle

```
function UVM_FILE get_file_handle(uvm_severity severity,
                                  string       id       )
```

Returns the file descriptor associated with the given *severity* and *id*.

First, if there is a file handle associated with the *(severity,id)* pair, return that. Else, if there is a file handle associated with the *id*, return that. Else, if there is an file handle associated with the *severity*, return that. Else, return the default file handle.

report

```
virtual function void report(uvm_severity      severity,
                             string            name,
                             string            id,
                             string            message,
                             int               verbosity_level,
                             string            filename,
                             int               line,
                             uvm_report_object client          )
```

This is the common handler method used by the four core reporting methods (e.g., uvm_report_error) in uvm_report_object.

format_action

```
function string format_action(uvm_action action)
```

Returns a string representation of the *action*, e.g., “DISPLAY”.

uvm_report_server

uvm_report_server is a global server that processes all of the reports generated by an uvm_report_handler. None of its methods are intended to be called by normal testbench code, although in some circumstances the virtual methods process_report and/or compose_uvm_info may be overloaded in a subclass.

Summary

uvm_report_server

uvm_report_server is a global server that processes all of the reports generated by an uvm_report_handler.

VARIABLES

id_count	An associative array holding the number of occurences for each unique report ID.

METHODS

new	Creates the central report server, if not already created.
set_server	Sets the global report server to use for reporting.
get_server	Gets the global report server.
set_max_quit_count get_max_quit_count	Get or set the maximum number of COUNT actions that can be tolerated before an UVM_EXIT action is taken.
set_quit_count get_quit_count incr_quit_count reset_quit_count	Set, get, increment, or reset to 0 the quit count, i.e., the number of COUNT actions issued.
is_quit_count_reached	If is_quit_count_reached returns 1, then the quit counter has reached the maximum.
set_severity_count get_severity_count incr_severity_count reset_severity_counts	Set, get, or increment the counter for the given severity, or reset all severity counters to 0.
set_id_count get_id_count incr_id_count	Set, get, or increment the counter for reports with the given id.
process_report	Calls compose_message to construct the actual message to be output.
compose_message	Constructs the actual string sent to the file or command line from the severity, component name, report id, and the message itself.
summarize	See uvm_report_object::report_summarize method.
dump_server_state	Dumps server state information.
get_server	Returns a handle to the central report server.

VARIABLES

id_count

```
protected int id_count[string]
```

An associative array holding the number of occurences for each unique report ID.

METHODS

new

```
function new()
```

Creates the central report server, if not already created. Else, does nothing. The constructor is protected to enforce a singleton.

set_server

```
static function void set_server(uvm_report_server server)
```

Sets the global report server to use for reporting. The report server is responsible for formatting messages.

get_server

```
static function uvm_report_server get_server()
```

Gets the global report server. The method will always return a valid handle to a report server.

set_max_quit_count

```
function void set_max_quit_count(int count,
                                 bit overridable = 1)
```

get_max_quit_count

```
function int get_max_quit_count()
```

Get or set the maximum number of COUNT actions that can be tolerated before an UVM_EXIT action is taken. The default is 0, which specifies no maximum.

set_quit_count

```
function void set_quit_count(int quit_count)
```

get_quit_count

```
function int get_quit_count()
```

incr_quit_count

```
function void incr_quit_count()
```

reset_quit_count

```
function void reset_quit_count()
```

Set, get, increment, or reset to 0 the quit count, i.e., the number of COUNT actions issued.

is_quit_count_reached

```
function bit is_quit_count_reached()
```

If is_quit_count_reached returns 1, then the quit counter has reached the maximum.

set_severity_count

```
function void set_severity_count(uvm_severity severity,
                                 int          count    )
```

get_severity_count

```
function int get_severity_count(uvm_severity severity)
```

incr_severity_count

```
function void incr_severity_count(uvm_severity severity)
```

reset_severity_counts

```
function void reset_severity_counts()
```

Set, get, or increment the counter for the given severity, or reset all severity counters to 0.

set_id_count

```
function void set_id_count(string id,
                           int    count)
```

get_id_count

```
function int get_id_count(string id)
```

incr_id_count

```
function void incr_id_count(string id)
```

Set, get, or increment the counter for reports with the given id.

process_report

```
virtual function void process_report(uvm_severity      severity,
                                     string            name,
                                     string            id,
                                     string            message,
                                     uvm_action        action,
                                     UVM_FILE          file,
                                     string            filename,
                                     int               line,
                                     string            composed_message,
                                     int               verbosity_level,
                                     uvm_report_object client           )
```

Calls compose_message to construct the actual message to be output. It then takes the appropriate action according to the value of action and file.

This method can be overloaded by expert users to customize the way the reporting system processes reports and the actions enabled for them.

compose_message

```
virtual function string compose_message(uvm_severity severity,
                                        string       name,
                                        string       id,
                                        string       message,
                                        string       filename,
                                        int          line     )
```

Constructs the actual string sent to the file or command line from the severity, component name, report id, and the message itself.

Expert users can overload this method to customize report formatting.

summarize

```
virtual function void summarize(UVM_FILE file = )
```

See uvm_report_object::report_summarize method.

dump_server_state

```
function void dump_server_state()
```

Dumps server state information.

get_server

```
function uvm_report_server get_server()
```

Returns a handle to the central report server.

uvm_report_catcher

The uvm_report_catcher is used to catch messages issued by the uvm report server. Catchers are uvm_callbacks#(uvm_report_object,uvm_report_catcher) objects, so all factilities in the uvm_callback and uvm_callbacks#(T,CB) classes are available for registering catchers and controlling catcher state. The uvm_callbacks#(uvm_report_object,uvm_report_catcher) class is aliased to *uvm_report_cb* to make it easier to use. Multiple report catchers can be registered with a report object. The catchers can be registered as default catchers which catch all reports on all uvm_report_object reporters, or catchers can be attached to specific report objects (i.e. components).

User extensions of uvm_report_catcher must implement the catch method in which the action to be taken on catching the report is specified. The catch method can return *CAUGHT*, in which case further processing of the report is immediately stopped, or return *THROW* in which case the (possibly modified) report is passed on to other registered catchers. The catchers are processed in the order in which they are registered.

On catching a report, the catch method can modify the severity, id, action, verbosity or the report string itself before the report is finally issued by the report server. The report can be immediately issued from within the catcher class by calling the issue method.

The catcher maintains a count of all reports with FATAL,ERROR or WARNING severity and a count of all reports with FATAL, ERROR or WARNING severity whose severity was lowered. These statistics are reported in the summary of the uvm_report_server.

This example shows the basic concept of creating a report catching callback and attaching it to all messages that get emitted:

```
class my_error_demoter extends uvm_report_catcher;
  function new(string name="my_error_demoter");
    super.new(name);
  endfunction
  //This example demotes "MY_ID" errors to an info message
  function action_e catch();
    if(get_severity() == UVM_ERROR && get_id() == "MY_ID")
      set_severity(UVM_INFO);
    return THROW;
  endfunction
endclass

my_error_demoter demoter = new;
initial begin
 // Catchers are callbacks on report objects (components are report
 // objects, so catchers can be attached to components).

 // To affect all reporters, use null for the object
 uvm_report_cb::add(null, demoter);

 // To affect some specific object use the specific reporter
 uvm_report_cb::add(mytest.myenv.myagent.mydriver, demoter);

 // To affect some set of components using the component name
 uvm_report_cb::add_by_name("*.*driver", demoter);
end
```

Summary

uvm_report_catcher

The uvm_report_catcher is used to catch messages issued by the uvm report server.

Class Declaration

```
typedef class uvm_report_catcher
```

new	Create a new report object.
Current Message State	
get_client	Returns the uvm_report_object that has generated the message that is currently being processes.
get_severity	Returns the uvm_severity of the message that is currently being processed.
get_verbosity	Returns the verbosity of the message that is currently being processed.
get_id	Returns the string id of the message that is currently being processed.
get_message	Returns the string message of the message that is currently being processed.
get_action	Returns the uvm_action of the message that is currently being processed.
get_fname	Returns the file name of the message.
get_line	Returns the line number of the message.
Change Message State	
set_severity	Change the severity of the message to *severity*.
set_verbosity	Change the verbosity of the message to *verbosity*.
set_id	Change the id of the message to *id*.
set_message	Change the text of the message to *message*.
set_action	Change the action of the message to *action*.
Debug	
get_report_catcher	Returns the first report catcher that has *name*.
print_catcher	Prints information about all of the report catchers that are registered.
Callback Interface	
catch	This is the method that is called for each registered report catcher.
Reporting	
uvm_report_fatal	Issues a fatal message using the current messages report object.
uvm_report_error	Issues a error message using the current messages report object.
uvm_report_warning	Issues a warning message using the current messages report object.
uvm_report_info	Issues a info message using the current messages report object.
issue	Immediately issues the message which is currently being processed.
summarize_report_catcher	This function is called automatically by uvm_report_server::summarize().

new

```
function new(string name = "uvm_report_catcher")
```

Create a new report object. The name argument is optional, but should generally be provided to aid in debugging.

Current Message State

get_client

```
function uvm_report_object get_client()
```

Returns the uvm_report_object that has generated the message that is currently being processes.

get_severity

```
function uvm_severity get_severity()
```

Returns the uvm_severity of the message that is currently being processed. If the severity was modified by a previously executed report object (which re-threw the message), then the returned severity is the modified value.

get_verbosity

```
function int get_verbosity()
```

Returns the verbosity of the message that is currently being processed. If the verbosity was modified by a previously executed report object (which re-threw the message), then the returned verbosity is the modified value.

get_id

```
function string get_id()
```

Returns the string id of the message that is currently being processed. If the id was modified by a previously executed report object (which re-threw the message), then the returned id is the modified value.

get_message

```
function string get_message()
```

Returns the string message of the message that is currently being processed. If the message was modified by a previously executed report object (which re-threw the message), then the returned message is the modified value.

get_action

```
function uvm_action get_action()
```

Returns the uvm_action of the message that is currently being processed. If the action was modified by a previously executed report object (which re-threw the message), then the returned action is the modified value.

get_fname

```
function string get_fname()
```

Returns the file name of the message.

get_line

```
function int get_line()
```

Returns the line number of the message.

Change Message State

set_severity

```
protected function void set_severity(uvm_severity severity)
```

Change the severity of the message to *severity*. Any other report catchers will see the modified value.

set_verbosity

```
protected function void set_verbosity(int verbosity)
```

Change the verbosity of the message to *verbosity*. Any other report catchers will see the modified value.

set_id

```
protected function void set_id(string id)
```

Change the id of the message to *id*. Any other report catchers will see the modified value.

set_message

```
protected function void set_message(string message)
```

Change the text of the message to *message*. Any other report catchers will see the modified value.

set_action

```
protected function void set_action(uvm_action action)
```

Change the action of the message to *action*. Any other report catchers will see the modified value.

Debug

get_report_catcher

```
static function uvm_report_catcher get_report_catcher(string name)
```

Returns the first report catcher that has *name*.

print_catcher

```
static function void print_catcher(UVM_FILE file = )
```

Prints information about all of the report catchers that are registered. For finer grained detail, the uvm_callbacks #(T,CB)::display method can be used by calling uvm_report_cb::display(uvm_report_object).

Callback Interface

catch

```
pure virtual function action_e catch()
```

This is the method that is called for each registered report catcher. There are no arguments to this function. The Current Message State interface methods can be used to access information about the current message being processed.

Reporting

uvm_report_fatal

```
protected function void uvm_report_fatal(string id,
                                         string message,
                                         int    verbosity,
                                         string fname      = "",
                                         int    line       = 0  )
```

Issues a fatal message using the current messages report object. This message will bypass any message catching callbacks.

uvm_report_error

```
protected function void uvm_report_error(string id,
                                         string message,
                                         int    verbosity,
                                         string fname      = "",
                                         int    line       = 0  )
```

Issues a error message using the current messages report object. This message will bypass any message catching callbacks.

uvm_report_warning

```
protected function void uvm_report_warning(string id,
                                           string message,
                                           int    verbosity,
                                           string fname      = "",
                                           int    line       = 0  )
```

Issues a warning message using the current messages report object. This message will bypass any message catching callbacks.

uvm_report_info

```
protected function void uvm_report_info(string id,
                                        string message,
                                        int    verbosity,
                                        string fname      = "",
                                        int    line       = 0  )
```

Issues a info message using the current messages report object. This message will bypass any message catching callbacks.

issue

```
protected function void issue()
```

Immediately issues the message which is currently being processed. This is useful if the message is being *CAUGHT* but should still be emitted.

Issuing a message will update the report_server stats, possibly multiple times if the message is not *CAUGHT*.

summarize_report_catcher

```
static function void summarize_report_catcher(UVM_FILE file)
```

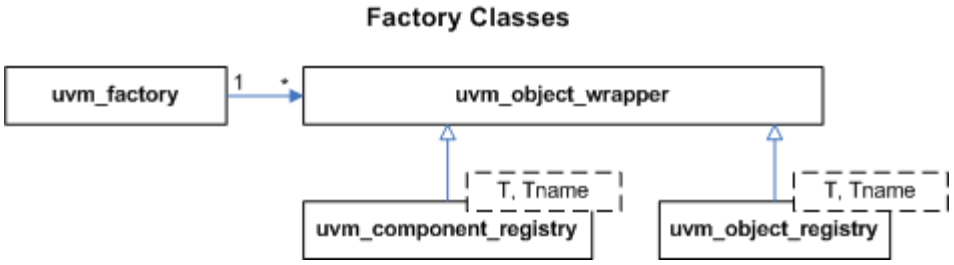
This function is called automatically by uvm_report_server::summarize(). It prints the statistics for the active catchers.

Factory Classes

As the name implies, the uvm_factory is used to manufacture (create) UVM objects and components. Only one instance of the factory is present in a given simulation.

User-defined object and component types are registered with the factory via typedef or macro invocation, as explained in uvm_factory::Usage. The factory generates and stores lightweight proxies to the user-defined objects and components: uvm_object_registry #(T,Tname) for objects and uvm_component_registry #(T,Tname) for components. Each proxy only knows how to create an instance of the object or component it represents, and so is very efficient in terms of memory usage.

When the user requests a new object or component from the factory (e.g. uvm_factory::create_object_by_type), the factory will determine what type of object to create based on its configuration, then ask that type's proxy to create an instance of the type, which is returned to the user.



Summary

Factory Classes

As the name implies, the uvm_factory is used to manufacture (create) UVM objects and components.

Factory Component and Object Wrappers

This section defines the proxy component and object classes used by the factory. To avoid the overhead of creating an instance of every component and object that get registered, the factory holds lightweight wrappers, or proxies. When a request for a new object is made, the factory calls upon the proxy to create the object it represents.

Contents

uvm_component_registry #(T,Tname)

The uvm_component_registry serves as a lightweight proxy for a component of type *T* and type name *Tname*, a string. The proxy enables efficient registration with the uvm_factory. Without it, registration would require an instance of the component itself.

See Usage section below for information on using uvm_component_registry.

Summary

uvm_component_registry #(T,Tname)

The uvm_component_registry serves as a lightweight proxy for a component of type *T* and type name *Tname*, a string.

Class Hierarchy

uvm_object_wrapper

uvm_component_registry#(T,Tname)

Class Declaration

```
class uvm_component_registry #(
   type   T     = uvm_component,
   string Tname = "<unknown>"
) extends uvm_object_wrapper
```

Methods

create_component	Creates a component of type T having the provided *name* and *parent*.
get_type_name	Returns the value given by the string parameter, *Tname*.
get	Returns the singleton instance of this type.
create	Returns an instance of the component type, *T*, represented by this proxy, subject to any factory overrides based on the context provided by the *parent*'s full name.
set_type_override	Configures the factory to create an object of the type

	represented by *override_type* whenever a request is made to create an object of the type, *T*, represented by this proxy, provided no instance override applies.
set_inst_override	Configures the factory to create a component of the type represented by *override_type* whenever a request is made to create an object of the type, *T*, represented by this proxy, with matching instance paths.

METHODS

create_component

```
virtual function uvm_component create_component (string        name,
                                                 uvm_component parent)
```

Creates a component of type T having the provided *name* and *parent*. This is an override of the method in uvm_object_wrapper. It is called by the factory after determining the type of object to create. You should not call this method directly. Call create instead.

get_type_name

```
virtual function string get_type_name()
```

Returns the value given by the string parameter, *Tname*. This method overrides the method in uvm_object_wrapper.

get

```
static function this_type get()
```

Returns the singleton instance of this type. Type-based factory operation depends on there being a single proxy instance for each registered type.

create

```
static function T create(string        name,
                         uvm_component parent,
                         string        contxt  = "")
```

Returns an instance of the component type, *T*, represented by this proxy, subject to any factory overrides based on the context provided by the *parent*'s full name. The *contxt* argument, if supplied, supercedes the *parent*'s context. The new instance will have the given leaf *name* and *parent*.

set_type_override

```
static function void set_type_override (uvm_object_wrapper override_type,
                                        bit                 replace        = 1
```

Configures the factory to create an object of the type represented by *override_type* whenever a request is made to create an object of the type, *T*, represented by this proxy, provided no instance override applies. The original type, *T*, is typically a super class of the override type.

set_inst_override

```
static function void set_inst_override(uvm_object_wrapper override_type,
                                       string             inst_path,
                                       uvm_component      parent         = nu
```

Configures the factory to create a component of the type represented by *override_type* whenever a request is made to create an object of the type, *T*, represented by this proxy, with matching instance paths. The original type, *T*, is typically a super class of the override type.

If *parent* is not specified, *inst_path* is interpreted as an absolute instance path, which enables instance overrides to be set from outside component classes. If *parent* is specified, *inst_path* is interpreted as being relative to the *parent*'s hierarchical instance path, i.e. *{parent.get_full_name(),".",inst_path}* is the instance path that is registered with the override. The *inst_path* may contain wildcards for matching against multiple contexts.

uvm_object_registry #(T,Tname)

The uvm_object_registry serves as a lightweight proxy for an uvm_object of type *T* and type name *Tname*, a string. The proxy enables efficient registration with the uvm_factory. Without it, registration would require an instance of the object itself.

See Usage section below for information on using uvm_component_registry.

Summary

uvm_object_registry #(T,Tname)

The uvm_object_registry serves as a lightweight proxy for an uvm_object of type *T* and type name *Tname*, a string.

Class Hierarchy

Class Declaration

```
class uvm_object_registry #(
   type   T     = uvm_object,
   string Tname = "<unknown>"
) extends uvm_object_wrapper
```

create_object	Creates an object of type *T* and returns it as a handle to an uvm_object.
get_type_name	Returns the value given by the string parameter, *Tname*.
get	Returns the singleton instance of this type.
create	Returns an instance of the object type, *T*, represented by this proxy, subject to any factory overrides based on the context provided by the *parent*'s full name.
set_type_override	Configures the factory to create an object of the type

	represented by *override_type* whenever a request is made to create an object of the type represented by this proxy, provided no instance override applies.
set_inst_override	Configures the factory to create an object of the type represented by *override_type* whenever a request is made to create an object of the type represented by this proxy, with matching instance paths.
USAGE	This section describes usage for the uvm_*_registry classes.

create_object

```
virtual function uvm_object create_object(string name = "")
```

Creates an object of type *T* and returns it as a handle to an uvm_object. This is an override of the method in uvm_object_wrapper. It is called by the factory after determining the type of object to create. You should not call this method directly. Call create instead.

get_type_name

```
virtual function string get_type_name()
```

Returns the value given by the string parameter, *Tname*. This method overrides the method in uvm_object_wrapper.

get

```
static function this_type get()
```

Returns the singleton instance of this type. Type-based factory operation depends on there being a single proxy instance for each registered type.

create

```
static function T create (string        name   = "",
                          uvm_component parent = null,
                          string        contxt = ""   )
```

Returns an instance of the object type, *T*, represented by this proxy, subject to any factory overrides based on the context provided by the *parent*'s full name. The *contxt* argument, if supplied, supercedes the *parent*'s context. The new instance will have the given leaf *name*, if provided.

set_type_override

```
static function void set_type_override (uvm_object_wrapper override_type,
                                        bit                 replace       = 1
```

Configures the factory to create an object of the type represented by *override_type* whenever a request is made to create an object of the type represented by this proxy, provided no instance override applies. The original type, *T*, is typically a super class of

the override type.

set_inst_override

```
static function void set_inst_override(uvm_object_wrapper override_type,
                                       string             inst_path,
                                       uvm_component      parent         = nu
```

Configures the factory to create an object of the type represented by *override_type* whenever a request is made to create an object of the type represented by this proxy, with matching instance paths. The original type, *T*, is typically a super class of the override type.

If *parent* is not specified, *inst_path* is interpreted as an absolute instance path, which enables instance overrides to be set from outside component classes. If *parent* is specified, *inst_path* is interpreted as being relative to the *parent*'s hierarchical instance path, i.e. *{parent.get_full_name(),".",inst_path}* is the instance path that is registered with the override. The *inst_path* may contain wildcards for matching against multiple contexts.

USAGE

This section describes usage for the uvm_*_registry classes.

The wrapper classes are used to register lightweight proxies of objects and components.

To register a particular component type, you need only typedef a specialization of its proxy class, which is typically done inside the class.

For example, to register an UVM component of type *mycomp*

```
class mycomp extends uvm_component;
  typedef uvm_component_registry #(mycomp,"mycomp") type_id;
endclass
```

However, because of differences between simulators, it is necessary to use a macro to ensure vendor interoperability with factory registration. To register an UVM component of type *mycomp* in a vendor-independent way, you would write instead:

```
class mycomp extends uvm_component;
  `uvm_component_utils(mycomp);
  ...
endclass
```

The `uvm_component_utils macro is for non-parameterized classes. In this example, the typedef underlying the macro specifies the *Tname* parameter as "mycomp", and *mycomp*'s get_type_name() is defined to return the same. With *Tname* defined, you can use the factory's name-based methods to set overrides and create objects and components of non-parameterized types.

For parameterized types, the type name changes with each specialization, so you can not specify a *Tname* inside a parameterized class and get the behavior you want; the same type name string would be registered for all specializations of the class! (The factory would produce warnings for each specialization beyond the first.) To avoid the warnings

and simulator interoperability issues with parameterized classes, you must register parameterized classes with a different macro.

For example, to register an UVM component of type driver #(T), you would write:

```
class driver #(type T=int) extends uvm_component;
  `uvm_component_param_utils(driver #(T));
  ...
endclass
```

The `uvm_component_param_utils and `uvm_object_param_utils macros are used to register parameterized classes with the factory. Unlike the the non-param versions, these macros do not specify the *Tname* parameter in the underlying uvm_component_registry typedef, and they do not define the get_type_name method for the user class. Consequently, you will not be able to use the factory's name-based methods for parameterized classes.

The primary purpose for adding the factory's type-based methods was to accommodate registration of parameterized types and eliminate the many sources of errors associated with string-based factory usage. Thus, use of name-based lookup in uvm_factory is no longer recommended.

UVM Factory

This page covers the classes that define the UVM factory facility.

Contents

uvm_factory

As the name implies, uvm_factory is used to manufacture (create) UVM objects and components. Only one instance of the factory is present in a given simulation (termed a singleton). Object and component types are registered with the factory using lightweight proxies to the actual objects and components being created. The uvm_object_registry #(T,Tname) and uvm_component_registry #(T,Tname) class are used to proxy uvm_objects and uvm_components.

The factory provides both name-based and type-based interfaces.

type-based	The type-based interface is far less prone to errors in usage. When errors do occur, they are caught at compile-time.
name-based	The name-based interface is dominated by string arguments that can be misspelled and provided in the wrong order. Errors in name-based requests might only be caught at the time of the call, if at all. Further, the name-based interface is not portable across simulators when used with parameterized classes.

See Usage section for details on configuring and using the factory.

Summary

uvm_factory

As the name implies, uvm_factory is used to manufacture (create) UVM objects and components.

Class Declaration

```
class uvm_factory
```

Registering Types

register	Registers the given proxy object, *obj*, with the factory.

Type & Instance Overrides

set_inst_override_by_type set_inst_override_by_name	Configures the factory to create an object of the override's type whenever a request is made to create an object of the original type using a context that matches *full_inst_path*.
set_type_override_by_type	

set_type_override_by_name	Configures the factory to create an object of the override's type whenever a request is made to create an object of the original type, provided no instance override applies.
CREATION	
create_object_by_type create_component_by_type create_object_by_name create_component_by_name	Creates and returns a component or object of the requested type, which may be specified by type or by name.
DEBUG	
debug_create_by_type debug_create_by_name	These methods perform the same search algorithm as the create_* methods, but they do not create new objects.
find_override_by_type find_override_by_name	These methods return the proxy to the object that would be created given the arguments.
print	Prints the state of the uvm_factory, including registered types, instance overrides, and type overrides.
USAGE	Using the factory involves three basic operations

REGISTERING TYPES

register

```
function void register (uvm_object_wrapper obj)
```

Registers the given proxy object, *obj*, with the factory. The proxy object is a lightweight substitute for the component or object it represents. When the factory needs to create an object of a given type, it calls the proxy's create_object or create_component method to do so.

When doing name-based operations, the factory calls the proxy's get_type_name method to match against the *requested_type_name* argument in subsequent calls to create_component_by_name and create_object_by_name. If the proxy object's get_type_name method returns the empty string, name-based lookup is effectively disabled.

TYPE & INSTANCE OVERRIDES

set_inst_override_by_type

```
function void set_inst_override_by_type (uvm_object_wrapper original_type,
                                         uvm_object_wrapper override_type,
                                         string             full_inst_path)
```

set_inst_override_by_name

```
function void set_inst_override_by_name (string original_type_name,
                                         string override_type_name,
                                         string full_inst_path     )
```

Configures the factory to create an object of the override's type whenever a request is made to create an object of the original type using a context that matches *full_inst_path*. The original type is typically a super class of the override type.

When overriding by type, the *original_type* and *override_type* are handles to the types' proxy objects. Preregistration is not required.

When overriding by name, the *original_type_name* typically refers to a preregistered type in the factory. It may, however, be any arbitrary string. Future calls to any of the create_* methods with the same string and matching instance path will produce the type represented by *override_type_name*, which must be preregistered with the factory.

The *full_inst_path* is matched against the contentation of {*parent_inst_path*, ".", *name*} provided in future create requests. The *full_inst_path* may include wildcards (* and ?) such that a single instance override can be applied in multiple contexts. A *full_inst_path* of "*" is effectively a type override, as it will match all contexts.

When the factory processes instance overrides, the instance queue is processed in order of override registrations, and the first override match prevails. Thus, more specific overrides should be registered first, followed by more general overrides.

set_type_override_by_type

```
function void set_type_override_by_type (uvm_object_wrapper original_type,
                                         uvm_object_wrapper override_type,
                                         bit               replace       =
```

set_type_override_by_name

```
function void set_type_override_by_name (string original_type_name,
                                         string override_type_name,
                                         bit    replace             = 1)
```

Configures the factory to create an object of the override's type whenever a request is made to create an object of the original type, provided no instance override applies. The original type is typically a super class of the override type.

When overriding by type, the *original_type* and *override_type* are handles to the types' proxy objects. Preregistration is not required.

When overriding by name, the *original_type_name* typically refers to a preregistered type in the factory. It may, however, be any arbitrary string. Future calls to any of the create_* methods with the same string and matching instance path will produce the type represented by *override_type_name*, which must be preregistered with the factory.

When *replace* is 1, a previous override on *original_type_name* is replaced, otherwise a previous override, if any, remains intact.

CREATION

create_object_by_type

```
function uvm_object create_object_by_type (uvm_object_wrapper requested_type,
                                           string              parent_inst_pat
                                           string              name
```

create_component_by_type

```
function uvm_component create_component_by_type (
   uvm_object_wrapper requested_type,
   string             parent_inst_path = "",
   string             name,
   uvm_component      parent
)
```

create_object_by_name

```
function uvm_object create_object_by_name (string requested_type_name,
                                           string parent_inst_path     = "",
                                           string name                = "" )
```

create_component_by_name

```
function uvm_component create_component_by_name (string        requested_type
                                                 string        parent_inst_pa
                                                 string        name,
                                                 uvm_component parent
```

Creates and returns a component or object of the requested type, which may be specified by type or by name. A requested component must be derived from the uvm_component base class, and a requested object must be derived from the uvm_object base class.

When requesting by type, the *requested_type* is a handle to the type's proxy object. Preregistration is not required.

When requesting by name, the *request_type_name* is a string representing the requested type, which must have been registered with the factory with that name prior to the request. If the factory does not recognize the *requested_type_name*, an error is produced and a null handle returned.

If the optional *parent_inst_path* is provided, then the concatenation, {*parent_inst_path*, ".",~name~}, forms an instance path (context) that is used to search for an instance override. The *parent_inst_path* is typically obtained by calling the uvm_component::get_full_name on the parent.

If no instance override is found, the factory then searches for a type override.

Once the final override is found, an instance of that component or object is returned in place of the requested type. New components will have the given *name* and *parent*. New objects will have the given *name*, if provided.

Override searches are recursively applied, with instance overrides taking precedence over type overrides. If *foo* overrides *bar*, and *xyz* overrides *foo*, then a request for *bar* will produce *xyz*. Recursive loops will result in an error, in which case the type returned will be that which formed the loop. Using the previous example, if *bar* overrides *xyz*, then

bar is returned after the error is issued.

Debug

debug_create_by_type

```
function void debug_create_by_type (uvm_object_wrapper requested_type,
                                    string             parent_inst_path = "",
                                    string             name             = "");
```

debug_create_by_name

```
function void debug_create_by_name (string requested_type_name,
                                    string parent_inst_path     = "",
                                    string name                 = "" )
```

These methods perform the same search algorithm as the create_* methods, but they do not create new objects. Instead, they provide detailed information about what type of object it would return, listing each override that was applied to arrive at the result. Interpretation of the arguments are exactly as with the create_* methods.

find_override_by_type

```
function uvm_object_wrapper find_override_by_type (
   uvm_object_wrapper requested_type,
   string             full_inst_path
)
```

find_override_by_name

```
function uvm_object_wrapper find_override_by_name (string requested_type_name
                                                   string full_inst_path
```

These methods return the proxy to the object that would be created given the arguments. The *full_inst_path* is typically derived from the parent's instance path and the leaf name of the object to be created, i.e. { parent.get_full_name(), ".", name }.

print

```
function void print (int all_types = 1)
```

Prints the state of the uvm_factory, including registered types, instance overrides, and type overrides.

When *all_types* is 0, only type and instance overrides are displayed. When *all_types* is 1 (default), all registered user-defined types are printed as well, provided they have names associated with them. When *all_types* is 2, the UVM types (prefixed with uvm_) are included in the list of registered types.

USAGE

Using the factory involves three basic operations

1. Registering objects and components types with the factory
2. Designing components to use the factory to create objects or components
3. Configuring the factory with type and instance overrides, both within and outside components

We'll briefly cover each of these steps here. More reference information can be found at Utility Macros, uvm_component_registry #(T,Tname), uvm_object_registry #(T,Tname), uvm_component.

1 -- Registering objects and component types with the factory

When defining uvm_object and uvm_component-based classes, simply invoke the appropriate macro. Use of macros are required to ensure portability across different vendors' simulators.

Objects that are not parameterized are declared as

```
class packet extends uvm_object;
  `uvm_object_utils(packet)
endclass

class packetD extends packet;
  `uvm_object_utils(packetD)
endclass
```

Objects that are parameterized are declared as

```
class packet #(type T=int, int WIDTH=32) extends uvm_object;
  `uvm_object_param_utils(packet #(T,WIDTH))
 endclass
```

Components that are not parameterized are declared as

```
class comp extends uvm_component;
  `uvm_component_utils(comp)
endclass
```

Components that are parameterized are declared as

```
class comp #(type T=int, int WIDTH=32) extends uvm_component;
  `uvm_component_param_utils(comp #(T,WIDTH))
endclass
```

The `uvm_*_utils macros for simple, non-parameterized classes will register the type with the factory and define the get_type, get_type_name, and create virtual methods inherited from uvm_object. It will also define a static type_name variable in the class, which will allow you to determine the type without having to allocate an instance.

The `uvm_*_param_utils macros for parameterized classes differ from `uvm_*_utils classes in the following ways:

- The get_type_name method and static type_name variable are not defined. You will need to implement these manually.
- A type name is not associated with the type when registeriing with the factory, so the factory's *_by_name operations will not work with parameterized classes.
- The factory's print, debug_create_by_type, and debug_create_by_name methods, which depend on type names to convey information, will list parameterized types as <unknown>.

It is worth noting that environments that exclusively use the type-based factory methods (*_by_type) do not require type registration. The factory's type-based methods will register the types involved "on the fly," when first used. However, registering with the `uvm_*_utils macros enables name-based factory usage and implements some useful utility functions.

2 -- Designing components that defer creation to the factory

Having registered your objects and components with the factory, you can now make requests for new objects and components via the factory. Using the factory instead of allocating them directly (via new) allows different objects to be substituted for the original without modifying the requesting class. The following code defines a driver class that is parameterized.

```
class driverB #(type T=uvm_object) extends uvm_driver;

  // parameterized classes must use the _param_utils version
  `uvm_component_param_utils(driverB #(T))

  // our packet type; this can be overridden via the factory
  T pkt;

  // standard component constructor
  function new(string name, uvm_component parent=null);
    super.new(name,parent);
  endfunction

  // get_type_name not implemented by macro for parameterized classes
  const static string type_name = {"driverB #(",T::type_name,")"};
  virtual function string get_type_name();
    return type_name;
  endfunction

  // using the factory allows pkt overrides from outside the class
  virtual function void build_phase(uvm_phase phase);
    pkt = packet::type_id::create("pkt",this);
  endfunction

  // print the packet so we can confirm its type when printing
  virtual function void do_print(uvm_printer printer);
    printer.print_object("pkt",pkt);
  endfunction

endclass
```

For purposes of illustrating type and instance overrides, we define two subtypes of the *driverB* class. The subtypes are also parameterized, so we must again provide an implementation for uvm_object::get_type_name, which we recommend writing in terms of a static string constant.

```
class driverD1 #(type T=uvm_object) extends driverB #(T);

  `uvm_component_param_utils(driverD1 #(T))

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

  const static string type_name = {"driverD1 #(",T::type_name,")"};
  virtual function string get_type_name();
    ...return type_name;
  endfunction

endclass
```

```
class driverD2 #(type T=uvm_object) extends driverB #(T);

  `uvm_component_param_utils(driverD2 #(T))

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

  const static string type_name = {"driverD2 #(",T::type_name,")"};
  virtual function string get_type_name();
    return type_name;
  endfunction

endclass

// typedef some specializations for convenience
typedef driverB  #(packet) B_driver;   // the base driver
typedef driverD1 #(packet) D1_driver;  // a derived driver
typedef driverD2 #(packet) D2_driver;  // another derived driver
```

Next, we'll define a agent component, which requires a utils macro for non-parameterized types. Before creating the drivers using the factory, we override *driver0*'s packet type to be *packetD*.

```
class agent extends uvm_agent;

  `uvm_component_utils(agent)
  ...
  B_driver driver0;
  B_driver driver1;

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

  virtual function void build_phase(uvm_phase phase);

    // override the packet type for driver0 and below
    packet::type_id::set_inst_override(packetD::get_type(),"driver0.*");

    // create using the factory; actual driver types may be different
    driver0 = B_driver::type_id::create("driver0",this);
    driver1 = B_driver::type_id::create("driver1",this);

  endfunction

endclass
```

Finally we define an environment class, also not parameterized. Its build method shows three methods for setting an instance override on a grandchild component with relative path name, *agent1.driver1*, all equivalent.

```
class env extends uvm_env;

  `uvm_component_utils(env)

  agent agent0;
  agent agent1;

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

  virtual function void build_phase(uvm_phase phase);

    // three methods to set an instance override for agent1.driver1
    // - via component convenience method...
    set_inst_override_by_type("agent1.driver1",
                              B_driver::get_type(),
                              D2_driver::get_type());

    // - via the component's proxy (same approach as create)...
    B_driver::type_id::set_inst_override(D2_driver::get_type(),
                                         "agent1.driver1",this);

    // - via a direct call to a factory method...
    factory.set_inst_override_by_type(B_driver::get_type(),
                                      D2_driver::get_type(),

{get_full_name(),".agent1.driver1"});

    // create agents using the factory; actual agent types may be different
```

```
    agent0 = agent::type_id::create("agent0",this);
    agent1 = agent::type_id::create("agent1",this);

  endfunction

  // at end_of_elaboration, print topology and factory state to verify
  virtual function void end_of_elaboration_phase(uvm_phase phase);
    uvm_top.print_topology();
  endfunction

  virtual task run_phase(uvm_phase phase);
    #100 global_stop_request();
  endfunction

endclass
```

3 -- Configuring the factory with type and instance overrides

In the previous step, we demonstrated setting instance overrides and creating components using the factory within component classes. Here, we will demonstrate setting overrides from outside components, as when initializing the environment prior to running the test.

```
module top;

  env env0;

  initial begin

    // Being registered first, the following overrides take precedence
    // over any overrides made within env0's construction & build.

    // Replace all base drivers with derived drivers...
    B_driver::type_id::set_type_override(D_driver::get_type());

    // ...except for agent0.driver0, whose type remains a base driver.
    //     (Both methods below have the equivalent result.)

    // - via the component's proxy (preferred)
    B_driver::type_id::set_inst_override(B_driver::get_type(),
                                         "env0.agent0.driver0");

    // - via a direct call to a factory method
    factory.set_inst_override_by_type(B_driver::get_type(),
                                      B_driver::get_type(),
                                   {get_full_name(),"env0.agent0.driver0"});

    // now, create the environment; our factory configuration will
    // govern what topology gets created
    env0 = new("env0");

    // run the test (will execute build phase)
    run_test();

  end

endmodule
```

When the above example is run, the resulting topology (displayed via a call to uvm_root::print_topology in env's uvm_component::end_of_elaboration_phase method) is similar to the following:

```
# UVM_INFO @ 0 [RNTST] Running test ...
# UVM_INFO @ 0 [UVMTOP] UVM testbench topology:
# ------------------------------------------------------------------------
# Name                     Type                Size                Value
# ------------------------------------------------------------------------
# env0                     env                 -                   env0@2
#   agent0                 agent               -                   agent0@4
#     driver0              driverB #(packet)   -                   driver0@8
#       pkt                packet              -                   pkt@21
#     driver1              driverD #(packet)   -                   driver1@14
#       pkt                packet              -                   pkt@23
#   agent1                 agent               -                   agent1@6
#     driver0              driverD #(packet)   -                   driver0@24
#       pkt                packet              -                   pkt@37
#     driver1              driverD2 #(packet)  -                   driver1@30
#       pkt                packet              -                   pkt@39
# ------------------------------------------------------------------------
```

uvm_object_wrapper

The uvm_object_wrapper provides an abstract interface for creating object and component proxies. Instances of these lightweight proxies, representing every uvm_object-based and uvm_component-based object available in the test environment, are registered with the uvm_factory. When the factory is called upon to create an object or component, it finds and delegates the request to the appropriate proxy.

Summary

uvm_object_wrapper

The uvm_object_wrapper provides an abstract interface for creating object and component proxies.

CLASS DECLARATION

```
virtual class uvm_object_wrapper
```

METHODS

create_object	Creates a new object with the optional *name*.
create_component	Creates a new component, passing to its constructor the given *name* and *parent*.
get_type_name	Derived classes implement this method to return the type name of the object created by create_component or create_object.

METHODS

create_object

```
virtual function uvm_object create_object (string name = "")
```

Creates a new object with the optional *name*. An object proxy (e.g., uvm_object_registry #(T,Tname)) implements this method to create an object of a specific type, T.

create_component

```
virtual function uvm_component create_component (string        name,
                                                 uvm_component parent)
```

Creates a new component, passing to its constructor the given *name* and *parent*. A component proxy (e.g. uvm_component_registry #(T,Tname)) implements this method to create a component of a specific type, T.

get_type_name

```
pure virtual function string get_type_name()
```

Derived classes implement this method to return the type name of the object created by create_component or create_object. The factory uses this name when matching against the requested type in name-based lookups.

Configuration and Resource Classes

The configuration and resources classes provide access to a centralized database where type specific information can be stored and recieved. The uvm_resource_db is the low level resource database which users can write to or read from. The uvm_config_db#(T) is layered on top of the resoure database and provides a typed intereface for configuration setting that is consistent with the uvm_component::Configuration Interface.

Information can be read from or written to the database at any time during simulation. A resource may be associated with a specific hierarchical scope of a uvm_component or it may be visible to all components regardless of their hierarchical position.

Summary

Configuration and Resource Classes

The configuration and resources classes provide access to a centralized database where type specific information can be stored and recieved.

Resources

A resource is a parameterized container that holds arbitrary data. Resources can be used to configure components, supply data to sequences, or enable sharing of information across disparate parts of a testbench. They are stored using scoping information so their visibility can be constrained to certain parts of the testbench. Resource containers can hold any type of data, constrained only by the data types available in SystemVerilog. Resources can contain scalar objects, class handles, queues, lists, or even virtual interfaces.

Resources are stored in a resource database so that each resource can be retrieved by name or by type. The databse has both a name table and a type table and each resource is entered into both. The database is globally accessible.

Each resource has a set of scopes over which it is visible. The set of scopes is represented as a regular expression. When a resource is looked up the scope of the entity doing the looking up is supplied to the lookup function. This is called the *current scope*. If the current scope is in the set of scopes over which a resource is visible then the resource can be retuned in the lookup.

Resources can be looked up by name or by type. To support type lookup each resource has a static type handle that uniquely identifies the type of each specialized resource container.

Mutliple resources that have the same name are stored in a queue. Each resource is pushed into a queue with the first one at the front of the queue and each subsequent one behind it. The same happens for multiple resources that have the same type. The resource queues are searched front to back, so those placed earlier in the queue have precedence over those placed later.

The precedence of resources with the same name or same type can be altered. One way is to set the *precedence* member of the resource container to any arbitrary value. The search algorithm will return the resource with the highest precedence. In the case where there are multiple resources that match the search criteria and have the same (highest) precedence, the earliest one located in the queue will be one returned. Another way to change the precedence is to use the set_priority function to move a resource to either the front or back of the queue.

The classes defined here form the low level layer of the resource database. The classes include the resource container and the database that holds the containers. The following set of classes are defined here:

uvm_resource_types: A class without methods or members, only typedefs and enums. These types and enums are used throughout the resources facility. Putting the types in a class keeps them confined to a specific name space.

uvm_resource_options: policy class for setting options, such as auditing, which effect resources.

uvm_resource_base: the base (untyped) resource class living in the resource database. This class includes the interface for locking, setting a resource as read-only, notification, scope management, altering search priority, and managing auditing.

uvm_resource#(T): parameterized resource container. This class includes the interfaces for reading and writing each resource. Because the class is parameterized, all the access functions are type sace.

uvm_resource_pool: the resource database. This is a singleton class object.

Contents

Resources	A resource is a parameterized container that holds arbitrary data.
uvm_resource_types	Provides typedefs and enums used throughout the resources facility.
uvm_resource_options	Provides a namespace for managing options for the resources facility.
uvm_resource_base	Non-parameterized base class for resources.
uvm_resource_pool	The global (singleton) resource database.
uvm_resource #(T)	Parameterized resource.

uvm_resource_types

Provides typedefs and enums used throughout the resources facility. This class has no members or methods, only typedefs. It's used in lieu of package-scope types. When needed, other classes can use these types by prefixing their usage with uvm_resource_types::. E.g.

```
uvm_resource_types::rsrc_q_t queue;
```

Summary

uvm_resource_types

Provides typedefs and enums used throughout the resources facility.

CLASS DECLARATION

```
class uvm_resource_types
```

uvm_resource_options

Provides a namespace for managing options for the resources facility. The only thing allowed in this class is static local data members and static functions for manipulating and retrieving the value of the data members. The static local data members represent options and settings that control the behavior of the resources facility.

Summary

uvm_resource_options

Provides a namespace for managing options for the resources facility.

METHODS

turn_on_auditing	Turn auditing on for the resource database.
turn_off_auditing	Turn auditing off for the resource database.
is_auditing	Returns 1 if the auditing facility is on and 0 if it is off.

METHODS

turn_on_auditing

```
static function void turn_on_auditing()
```

Turn auditing on for the resource database. This causes all reads and writes to the database to store information about the accesses.

turn_off_auditing

```
static function void turn_off_auditing()
```

Turn auditing off for the resource database. If auditing is it is not possible to get extra information about resource database accesses.

is_auditing

```
static function bit is_auditing()
```

Returns 1 if the auditing facility is on and 0 if it is off.

uvm_resource_base

Non-parameterized base class for resources. Supports interfaces for locking/unlocking, scope matching, and virtual functions for printing the resource and for printing the accessor list

Summary

uvm_resource_base

Non-parameterized base class for resources.

CLASS HIERARCHY

uvm_void

uvm_object

uvm_resource_base

CLASS DECLARATION

```
virtual class uvm_resource_base extends uvm_object
```

precedence	This variable is used to associate a precedence that a resource has with respect to other resources which match the same scope and name.
default_precedence	The default precedence for an resource that has been created.
new	constructor for uvm_resource_base.
get_type_handle	Pure virtual function that returns the type handle of the

	resource container.
Locking Interface	The task lock and the functions try_lock and unlock form a locking interface for resources.
lock	Retrieves a lock for this resource.
try_lock	Retrives the lock for this resource.
unlock	Releases the lock held by this semaphore.
Read-only Interface	
set_read_only	Establishes this resource as a read-only resource.
is_read_only	Retruns one if this resource has been set to read-only, zero otherwise
Notification	
wait_modified	This task blocks until the resource has been modified -- that is, a uvm_resource#(T)::write operation has been performed.
Scope Interface	Each resource has a name, a value and a set of scopes over which it is visible.
set_scope	Set the value of the regular expression that identifies the set of scopes over which this resource is visible.
get_scope	Retrieve the regular expression string that identifies the set of scopes over which this resource is visible.
match_scope	Using the regular expression facility, determine if this resource is visible in a scope.
Priority	Functions for manipulating the search priority of resources.
set priority	Change the search priority of the resource based on the value of the priority enum argument.
Utility Functions	
do_print	Implementation of do_print which is called by print().
Audit Trail	To find out what is happening as the simulation proceeds, an audit trail of each read and write is kept.
print_accessors	Dump the access records for this resource
init_access_record	Initalize a new access record

precedence

```
int unsigned precedence
```

This variable is used to associate a precedence that a resource has with respect to other resources which match the same scope and name. Resources are set to the default_precedence initially, and may be set to a higher or lower precedence as desired.

default_precedence

```
static int unsigned default_precedence = 1000
```

The default precedence for an resource that has been created. When two resources have the same precedence, the first resource found has precedence.

new

```
function new(string name = ""
             string s    = "*");
```

constructor for uvm_resource_base. The constructor takes two arguments, the name of the resource and a resgular expression which represents the set of scopes over which

this resource is visible.

get_type_handle

```
pure virtual function uvm_resource_base get_type_handle()
```

Pure virtual function that returns the type handle of the resource container.

Locking Interface

The task lock and the functions try_lock and unlock form a locking interface for resources. These can be used for thread-safe reads and writes. The interface methods write_with_lock and read_with_lock and their nonblocking counterparts in uvm_resource#(T) (a family of resource subclasses) obey the lock when reading and writing. See documentation in uvm_resource#(T) for more information on put/get. The lock interface is a wrapper around a local semaphore.

lock

```
task lock()
```

Retrieves a lock for this resource. The task blocks until the lock is obtained.

try_lock

```
function bit try_lock()
```

Retrives the lock for this resource. The function is nonblocking, so it will return immediately. If it was successfull in retrieving the lock then a one is returned, otherwise a zero is returned.

unlock

```
function void unlock()
```

Releases the lock held by this semaphore.

Read-only Interface

set_read_only

```
function void set_read_only()
```

Establishes this resource as a read-only resource. An attempt to call uvm_resource#(T)::write on the resource will cause an error.

is_read_only

```
function bit is_read_only()
```

Retruns one if this resource has been set to read-only, zero otherwise

Notification

wait_modified

```
task wait_modified()
```

This task blocks until the resource has been modified -- that is, a uvm_resource#(T)::write operation has been performed. When a uvm_resource#(T)::write is performed the modified bit is set which releases the block. Wait_modified() then clears the modified bit so it can be called repeatedly.

Scope Interface

Each resource has a name, a value and a set of scopes over which it is visible. A scope is a hierarchical entity or a context. A scope name is a multi-element string that identifies a scope. Each element refers to a scope context and the elements are separated by dots (.).

```
top.env.agent.monitor
```

Consider the example above of a scope name. It consists of four elements: "top", "env", "agent", and "monitor". The elements are strung together with a dot separating each element. *top.env.agent* is the parent of *top.env.agent.monitor*, *top.env* is the parent of *top.env.agent*, and so on. A set of scopes can be represented by a set of scope name strings. A very straightforward way to represent a set of strings is to use regular expressions. A regular expression is a special string that contains placeholders which can be substituted in various ways to generate or recognize a particular set of strings. Here are a few simple examples:

```
 top\..*                    all of the scopes whose top-level component
                          is top
top\.env\..*\.monitor   all of the scopes in env that end in monitor;
                          i.e. all the monitors two levels down from env
.*\.monitor                 all of the scopes that end in monitor; i.e.
                          all the monitors (assuming a naming convention
                          was used where all monitors are named "monitor")
top\.u[1-5]\.*           all of the scopes rooted and named u1, u2, u3,
```

u4, or u5, and any of their subscopes.

The examples above use posix regular expression notation. This is a very general and expressive notation. It is not always the case that so much expressiveness is required. Sometimes an expression syntax that is easy to read and easy to write is useful, even if the syntax is not as expressive as the full power of posix regular expressions. A popular substitute for regular expressions is globs. A glob is a simplified regular expression. It only has three metacharacters -- *, +, and ?. Character ranges are not allowed and dots are not a metacharacter in globs as they are in regular expressions. The following table shows glob metacharacters.

```
char meaning                  regular expression
                               equivalent
*        0 or more characters     .*
+        1 or more characters     .+
?        exactly one character    .
```

Of the examples above, the first three can easily be translated into globs. The last one cannot. It relies on notation that is not available in glob syntax.

```
regular expression     glob equivalent
---------------------  ------------------
top\..*                 top.*
top\.env\..*\.monitor      top.env.*.monitor
.*\.monitor                *.monitor
```

The resource facility supports both regular expression and glob syntax. Regular expressions are identified as such when they surrounded by '/' characters. For example, /^*top*\.*/ is interpreted as the regular expression ^*top*\.*, where the surrounding '/' characters have been removed. All other expressions are treated as glob expressions. They are converted from glob notation to regular expression notation internally. Regular expression compilation and matching as well as glob-to-regular expression conversion are handled by three DPI functions:

```
function int uvm_re_match(string re, string str);
function void uvm_dump_re_cache();
function string uvm_glob_to_re(string glob);
```

uvm_re_match both compiles and matches the regular expression. It uses internal caching of compiled information so that each match does not necessarily require a new compilation of the regular expression string. All of the matching is done using regular expressions, so globs are converted to regular expressions and then processed.

set_scope

```
function void set_scope(string s)
```

Set the value of the regular expression that identifies the set of scopes over which this resource is visible. If the supplied argument is a glob it will be converted to a regular expression before it is stored.

get_scope

```
function string get_scope()
```

Retrieve the regular expression string that identifies the set of scopes over which this resource is visible.

match_scope

```
function bit match_scope(string s)
```

Using the regular expression facility, determine if this resource is visible in a scope. Return one if it is, zero otherwise.

Priority

Functions for manipulating the search priority of resources. The function definitions here are pure virtual and are implemented in derived classes. The definitons serve as a priority management interface.

set priority

Change the search priority of the resource based on the value of the priority enum argument.

Utility Functions

do_print

```
function void do_print (uvm_printer printer)
```

Implementation of do_print which is called by print().

Audit Trail

To find out what is happening as the simulation proceeds, an audit trail of each read and write is kept. The read and write methods in uvm_resource#(T) each take an accessor argument. This is a handle to the object that performed that resource access.

```
function T read(uvm_object accessor = null);
function void write(T t, uvm_object accessor = null);
```

The accessor can by anything as long as it is derived from uvm_object. The accessor object can be a component or a sequence or whatever object from which a read or write was invoked. Typically the *this* handle is used as the accessor. For example:

```
uvm_resource#(int) rint;
int i;
...
rint.write(7, this);
i = rint.read(this);
```

The accessor's *get_full_name()* is stored as part of the audit trail. This way you can find out what object performed each resource access. Each audit record also includes the time of the access (simulation time) and the particular operation performed (read or write).

Auditting is controlled through the uvm_resource_options class.

print_accessors

```
virtual function void print_accessors()
```

Dump the access records for this resource

init_access_record

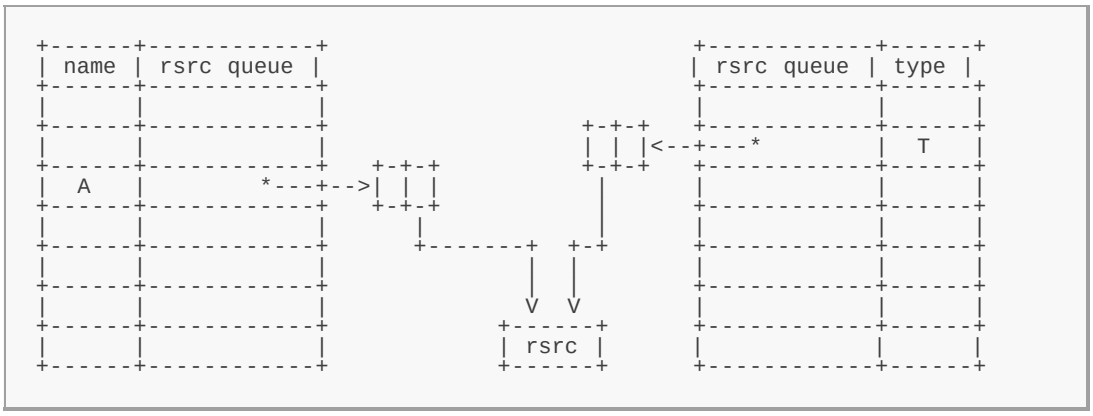
```
function void init_access_record (
   inout uvm_resource_types::access_t access_record
)
```

Initalize a new access record

uvm_resource_pool

The global (singleton) resource database.

Each resource is stored both by primary name and by type handle. The resource pool contains two associative arrays, one with name as the key and one with the type handle as the key. Each associative array contains a queue of resources. Each resource has a regular expression that represents the set of scopes over with it is visible.



The above diagrams illustrates how a resource whose name is A and type is T is stored in the pool. The pool contains an entry in the type map for type T and an entry in the name map for name A. The queues in each of the arrays each contain an entry for the resource A whose type is T. The name map can contain in its queue other resources whose name is A which may or may not have the same type as our resource A. Similarly, the type map can contain in its queue other resources whose type is T and whose name may or may not be A.

Resources are added to the pool by calling set; they are retrieved from the pool by calling get_by_name or get_by_type. When an object creates a new resource and calls set the resource is made available to be retrieved by other objects outside of itsef; an object gets a resource when it wants to access a resource not currently available in its scope.

The scope is stored in the resource itself (not in the pool) so whether you get by name or by type the resource's visibility is the same.

As an auditing capability, the pool contains a history of gets. A record of each get,

whether by get_by_type or get_by_name, is stored in the audit record. Both successful and failed gets are recorded. At the end of simulation, or any time for that matter, you can dump the history list. This will tell which resources were successfully located and which were not. You can use this information to determine if there is some error in name, type, or scope that has caused a resource to not be located or to be incorrrectly located (i.e. the wrong resource is located).

Summary

uvm_resource_pool

The global (singleton) resource database.

Class Declaration

```
class uvm_resource_pool
```

get	Returns the singleton handle to the resource pool
spell_check	Invokes the spell checker for a string s.
Set	
set	Add a new resource to the resource pool.
set_override	The resource provided as an argument will be entered into the pool and will override both by name and type.
set_name_override	The resource provided as an argument will entered into the pool using normal precedence in the type map and will override the name.
set_type_override	The resource provided as an argument will be entered into the pool using noraml precedence in the name map and will override the type.
Lookup	This group of functions is for finding resources in the resource database.
lookup_name	Lookup resources by *name*.
get_highest_precedence	Traverse a queue, *q*, of resources and return the one with the highest precedence.
get_by_name	Lookup a resource by *name* and *scope*.
lookup_type	Lookup resources by type.
get_by_type	Lookup a resource by *type_handle* and *scope*.
lookup_regex_names	This utility function answers the question, for a given *name* and *scope*, what are all of the resources with a matching name (where the resource name may be a regular expression) and a matching scope (where the resoucre scope may be a regular expression).
lookup_regex	Looks for all the resources whose name matches the regular expression argument and whose scope matches the current scope.
lookup_scope	This is a utility function that answers the question: For a given *scope*, what resources are visible to it?
Set Priority	Functions for altering the search priority of resources.
set_priority_type	Change the priority of the *rsrc* based on the value of *pri*, the priority enum argument.
set_priority_name	Change the priority of the *rsrc* based on the value of *pri*, the priority enum argument.
set_priority	Change the search priority of the *rsrc* based on the value of *pri*, the priority enum argument.
Debug	
find_unused_resources	Locate all the resources that have at least one write and no reads
print_resources	Print the resources that are in a single queue, *rq*.
dump	dump the entire resource pool.

get

```
static function uvm_resource_pool get()
```

Returns the singleton handle to the resource pool

spell_check

```
function bit spell_check(string s)
```

Invokes the spell checker for a string s. The universe of correctly spelled strings -- i.e. the dictionary -- is the name map.

Set

set

```
function void set (uvm_resource_base rsrc,
```

Add a new resource to the resource pool. The resource is inserted into both the name map and type map so it can be located by either.

An object creates a resources and *sets* it into the resource pool. Later, other objects that want to access the resource must *get* it from the pool

Overrides can be specified using this interface. Either a name override, a type override or both can be specified. If an override is specified then the resource is entered at the front of the queue instead of at the back. It is not recommended that users specify the override paramterer directly, rather they use the set_override, set_name_override, or set_type_override functions.

set_override

```
function void set_override(uvm_resource_base rsrc)
```

The resource provided as an argument will be entered into the pool and will override both by name and type.

set_name_override

```
function void set_name_override(uvm_resource_base rsrc)
```

The resource provided as an argument will entered into the pool using normal precedence in the type map and will override the name.

set_type_override

```
function void set_type_override(uvm_resource_base rsrc)
```

The resource provided as an argument will be entered into the pool using noraml precedence in the name map and will override the type.

Lookup

This group of functions is for finding resources in the resource database.

lookup_name and lookup_type locate the set of resources that matches the name or type (respectively) and is visible in the current scope. These functions return a queue of resources.

get_highest_precedence traverese a queue of resources and returns the one with the highest precedence -- i.e. the one whose precedence member has the highest value.

get_by_name and get_by_type use lookup_name and lookup_type (respectively) and get_highest_precedence to find the resource with the highest priority that matches the other search criteria.

lookup_name

```
function uvm_resource_types::rsrc_q_t lookup_name(string scope  = "",
                                                  string name,
                                                  bit    rpterr = 1  )
```

Lookup resources by *name*. Returns a queue of resources that match the *name* and *scope*. If no resources match the queue is returned empty. If *rpterr* is set then a warning is issued if no matches are found, and the spell checker is invoked on *name*.

get_highest_precedence

```
function uvm_resource_base get_highest_precedence(
   ref uvm_resource_types::rsrc_q_t q
)
```

Traverse a queue, *q*, of resources and return the one with the highest precedence. In the case where there exists more than one resource with the highest precedence value, the first one that has that precedence will be the one that is returned.

get_by_name

```
function uvm_resource_base get_by_name(string scope  = "",
                                       string name,
                                       bit    rpterr = 1  )
```

Lookup a resource by *name* and *scope*. Whether the get succeeds or fails, save a record of the get attempt. The *rpterr* flag indicates whether to report errors or not. Essentially, it serves as a verbose flag. If set then the spell checker will be invoked and warnings about multiple resources will be produced.

lookup_type

```
function uvm_resource_types::rsrc_q_t lookup_type(string            scope
                                                  uvm_resource_base type_hand
```

Lookup resources by type. Return a queue of resources that match the *type_handle* and *scope*. If no resources match then the returned queue is empty.

get_by_type

```
function uvm_resource_base get_by_type(string            scope       = "",
                                       uvm_resource_base type_handle       )
```

Lookup a resource by *type_handle* and *scope*. Insert a record into the get history list whether or not the get succeeded.

lookup_regex_names

```
function uvm_resource_types::rsrc_q_t lookup_regex_names(string scope,
                                                         string name  )
```

This utility function answers the question, for a given *name* and *scope*, what are all of the resources with a matching name (where the resource name may be a regular expression) and a matching scope (where the resoucre scope may be a regular expression). *name* and *scope* are explicit values.

lookup_regex

```
function uvm_resource_types::rsrc_q_t lookup_regex(string re,
                                                   scope)
```

Looks for all the resources whose name matches the regular expression argument and whose scope matches the current scope.

lookup_scope

```
function uvm_resource_types::rsrc_q_t lookup_scope(string scope)
```

This is a utility function that answers the question: For a given *scope*, what resources are visible to it? Locate all the resources that are visible to a particular scope. This operation could be quite expensive, as it has to traverse all of the resources in the database.

Set Priority

Functions for altering the search priority of resources. Resources are stored in queues in the type and name maps. When retrieving resoures, either by type or by name, the resource queue is search from front to back. The first one that matches the search criteria is the one that is returned. The *set_priority* functions let you change the order in which resources are searched. For any particular resource, you can set its priority to UVM_HIGH, in which case the resource is moved to the front of the queue, or to UVM_LOW in which case the resource is moved to the back of the queue.

set_priority_type

```
function void set_priority_type(                  uvm_resource_base rsrc,
                                uvm_resource_types::priority_e      pri  )
```

Change the priority of the *rsrc* based on the value of *pri*, the priority enum argument. This function changes the priority only in the type map, leavint the name map untouched.

set_priority_name

```
function void set_priority_name(                    uvm_resource_base rsrc,
                                uvm_resource_types::priority_e        pri  )
```

Change the priority of the *rsrc* based on the value of *pri*, the priority enum argument. This function changes the priority only in the name map, leaving the type map untouched.

set_priority

```
function void set_priority (                    uvm_resource_base rsrc,
                            uvm_resource_types::priority_e        pri  )
```

Change the search priority of the *rsrc* based on the value of *pri*, the priority enum argument. This function changes the priority in both the name and type maps.

Debug

find_unused_resources

```
function uvm_resource_types::rsrc_q_t find_unused_resources()
```

Locate all the resources that have at least one write and no reads

print_resources

```
function void print_resources(uvm_resource_types::rsrc_q_t rq,
                                                  bit      audit = 0)
```

Print the resources that are in a single queue, *rq*. This is a utility function that can be used to print any collection of resources stored in a queue. The *audit* flag determines whether or not the audit trail is printed for each resource along with the name, value, and scope regular expression.

dump

```
function void dump(bit audit = 0)
```

dump the entire resource pool. The resource pool is traversed and each resource is printed. The utility function print_resources() is used to initiate the printing. If the *audit* bit is set then the audit trail is dumped for each resource.

uvm_resource #(T)

Parameterized resource. Provides essential access methods to read from and write to the resource database. Also provides locking access methods including.

Summary

uvm_resource #(T)

Parameterized resource.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_resource_base
- **uvm_resource#(T)**

Class Declaration

```
class uvm_resource #(
   type  T = int
) extends uvm_resource_base
```

Type Interface	Resources can be identified by type using a static type handle.
get_type	Static function that returns the static type handle.
get_type_handle	Returns the static type handle of this resource in a polymorphic fashion.
Set/Get Interface	uvm_resource#(T) provides an interface for setting and getting a resources.
set	Simply put this resource into the global resource pool
set_override	Put a resource into the global resource pool as an override.
get_by_name	looks up a resource by *name* in the name map.
get_by_type	looks up a resource by *type_handle* in the type map.
Read/Write Interface	read and write provide a type-safe interface for getting and setting the object in the resource container.
read	Return the object stored in the resource container.
write	Modify the object stored in this resource container.
Priority	Functions for manipulating the search priority of resources.
set priority	Change the search priority of the resource based on the value of the priority enum argument, *pri*.
Locking Interface	This interface is optional, you can choose to lock a resource or not.
read_with_loc;	Locking version of read().
try_read_with_lock	Nonblocking form of read_with_lock().
write_with_lock	Locking form of write().
try_write_with_lock	Nonblocking form of write_with_lock().
get_highest_precedence	In a queue of resources, locate the first one with the highest precedence whose type is T.

Type Interface

Resources can be identified by type using a static type handle. The parent class provides the virtual function interface get_type_handle. Here we implement it by returning the static type handle.

get_type

```
static function this_type get_type()
```

Static function that returns the static type handle. The return type is this_type, which is the type of the parameterized class.

get_type_handle

```
function uvm_resource_base get_type_handle()
```

Returns the static type handle of this resource in a polymorphic fashion. The return type of get_type_handle() is uvm_resource_base. This function is not static and therefore can only be used by instances of a parameterized resource.

Set/Get Interface

uvm_resource#(T) provides an interface for setting and getting a resources. Specifically, a resource can insert itself into the resource pool. It doesn't make sense for a resource to get itself, since you can't call a funtion on a handle you don't have. However, a static get interface is provided as a convenience. This obviates the need for the user to get a handle to the global resource pool as this is done for him here.

set

```
function void set()
```

Simply put this resource into the global resource pool

set_override

```
function void set_override(
```

Put a resource into the global resource pool as an override. This means it gets put at the head of the list and is searched before other existing resources that occupy the same position in the name map or the type map. The default is to override both the name and type maps. However, using the *override* argument you can specify that either the name map or type map is overridden.

get_by_name

```
static function this_type get_by_name(string scope,
                                      string name,
                                      bit    rpterr = 1)
```

looks up a resource by *name* in the name map. The first resource with the specified name that is visible in the specified *scope* is returned, if one exists. The *rpterr* flag indicates whether or not an error should be reported if the search fails. If *rpterr* is set to one then a failure message is issued, including suggested spelling alternatives, based on resource names that exist in the database, gathered by the spell checker.

get_by_type

```
static function this_type get_by_type(string            scope       = "",
                                      uvm_resource_base type_handle      )
```

looks up a resource by *type_handle* in the type map. The first resource with the specified *type_handle* that is visible in the specified *scope* is returned, if one exists. Null is returned if there is no resource matching the specifications.

Read/Write Interface

read and write provide a type-safe interface for getting and setting the object in the resource container. The interface is type safe because the value argument for write and the return value of read are T, the type supplied in the class parameter. If either of these functions is used in an incorrect type context the compiler will complain.

read

```
function T read(uvm_object accessor = null)
```

Return the object stored in the resource container. If an *accessor* object is supplied then also update the accessor record for this resource.

write

```
function void write(T          t,
                    uvm_object accessor = null)
```

Modify the object stored in this resource container. If the resource is read-only then issue an error message and return without modifying the object in the container. If the resource is not read-only and an *accessor* object has been supplied then also update the accessor record. Lastly, replace the object value in the container with the value supplied as the argument, *t*, and release any processes blocked on uvm_resource_base::wait_modified.

Priority

Functions for manipulating the search priority of resources. These implementations of the interface defined in the base class delegate to the resource pool.

set priority

Change the search priority of the resource based on the value of the priority enum argument, *pri*.

Locking Interface

This interface is optional, you can choose to lock a resource or not. These methods are wrappers around the read/write interface. The difference between read/write interface and the locking interface is the use of a semaphore to guarantee exclusive access.

read_with_loc;

Locking version of read(). Like read(), this returns the contents of the resource container. In addtion it obeys the lock.

try_read_with_lock

```
function bit try_read_with_lock(output T          t,
                                input uvm_object accessor = null)
```

Nonblocking form of read_with_lock(). If the lock is availble it grabs the lock and returns one. If the lock is not available then it returns a 0. In either case the return is immediate with no blocking.

write_with_lock

```
task write_with_lock (input T          t,
                      uvm_object accessor = null)
```

Locking form of write(). Like write(), write_with_lock() sets the contents of the resource container. In addition it locks the resource before doing the write and unlocks it when the write is complete. If the lock is currently not available write_with_lock() will block until it is.

try_write_with_lock

```
function bit try_write_with_lock(input T          t,
                                 uvm_object accessor = null)
```

Nonblocking form of write_with_lock(). If the lock is available then the write() occurs immediately and a one is returned. If the lock is not available then the write does not occur and a zero is returned. IN either case try_write_with_lock() returns immediately with no blocking.

get_highest_precedence

```
static function this_type get_highest_precedence(
   ref uvm_resource_types::rsrc_q_t q
)
```

In a queue of resources, locate the first one with the highest precedence whose type is T. This function is static so that it can be called from anywhere.

uvm_resource_db

The uvm_resource_db#(T) class provides a convenience interface for the resources facility. In many cases basic operations such as creating and setting a resource or getting a resource could take multiple lines of code using the interfaces in uvm_resource_base or uvm_resource#(T). The convenience layer in uvm_resource_db#(T) reduces many of those operations to a single line of code.

All of the functions in uvm_resource_db#(T) are static, so they must be called using the :: operator. For example:

```
uvm_resource_db#(int)::set("A", "*", 17, this);
```

The parameter value "int" identifies the resource type as uvm_resource#(int). Thus, the type of the object in the resource container is int. This maintains the type-safety characteristics of resource operations.

Summary

uvm_resource_db

The uvm_resource_db#(T) class provides a convenience interface for the resources facility.

Class Declaration

```
class uvm_resource_db #(type T = uvm_object)
```

Methods

get_by_type	Get a resource by type.
get_by_name	Imports a resource by *name*.
set_default	add a new item into the resources database.
set	Create a new resource, write a *val* to it, and set it into the database using *name* and *scope* as the lookup parameters.
set_anonymous	Create a new resource, write a *val* to it, and set it into the database.
read_by_name	locate a resource by *name* and *scope* and read its value.
read_by_type	Read a value by type.
write_by_name	write a *val* into the resources database.
write_by_type	write a *val* into the resources database.
dump	Dump all the resources in the resource pool.

Methods

get_by_type

```
static function rsrc_t get_by_type(string scope)
```

Get a resource by type. The type is specified in the db class parameter so the only argument to this function is the *scope*.

get_by_name

```
static function rsrc_t get_by_name(string scope,
                                   string name,
                                   bit    rpterr = 1)
```

Imports a resource by *name*. The first argument is the *name* of the resource to be retrieved and the second argument is the current *scope*. The *rpterr* flag indicates whether or not to generate a warning if no matching resource is found.

set_default

```
static function rsrc_t set_default(string scope,
                                   string name  )
```

add a new item into the resources database. The item will not be written to so it will have its default value. The resource is created using *name* and *scope* as the lookup parameters.

set

```
static function void set(input string     scope,
                         input string     name,
                               T          val,
                         input uvm_object accessor = null)
```

Create a new resource, write a *val* to it, and set it into the database using *name* and *scope* as the lookup parameters. The *accessor* is used for auditting.

set_anonymous

```
static function void set_anonymous(input string     scope,
                                         T          val,
                                   input uvm_object accessor = null)
```

Create a new resource, write a *val* to it, and set it into the database. The resource has no name and therefore will not be entered into the name map. But is does have a *scope* for lookup purposes. The *accessor* is used for auditting.

read_by_name

```
static function bit read_by_name(input string     scope,
                                 input string     name,
                                   ref T          val,
                                 input uvm_object accessor = null)
```

locate a resource by *name* and *scope* and read its value. The value is returned through the ref argument *val*. The return value is a bit that indicates whether or not the read was successful. The *accessor* is used for auditting.

read_by_type

```
static function bit read_by_type(input string     scope,
                                   ref T          val,
                                 input uvm_object accessor = null)
```

Read a value by type. The value is returned through the ref argument *val*. The *scope* is used for the lookup. The return value is a bit that indicates whether or not the read is successful. The *accessor* is used for auditting.

write_by_name

```
static function bit write_by_name(input string     scope,
                                  input string     name,
                                        T          val,
                                  input uvm_object accessor = null)
```

write a *val* into the resources database. First, look up the resource by *name* and *scope*. If it is not located then add a new resource to the database and then write its value.

Because the *scope* is matched to a resource which may be a regular expression, and consequently may target other scopes beyond the *scope* argument. Care must be taken with this function. If a get_by_name match is found for *name* and *scope* then *val* will be written to that matching resource and thus may impact other scopes which also match the resource.

write_by_type

```
static function bit write_by_type(input string     scope,
                                  input T          val,
                                  input uvm_object accessor = null)
```

write a *val* into the resources database. First, look up the resource by type. If it is not located then add a new resource to the database and then write its value.

Because the *scope* is matched to a resource which may be a regular expression, and consequently may target other scopes beyond the *scope* argument. Care must be taken with this function. If a get_by_name match is found for *name* and *scope* then *val* will be written to that matching resource and thus may impact other scopes which also match the resource.

dump

```
static function void dump()
```

Dump all the resources in the resource pool. This is useful for debugging purposes. This function does not use the parameter T, so it will dump the same thing -- the entire database -- no matter the value of the parameter.

uvm_config_db#(T)

The uvm_config_db#(T) class provides a convenience interface on top of the uvm_resource_db to simplify the basic interface that is used for reading and writing into the resource database.

All of the functions in uvm_config_db#(T) are static, so they must be called using the :: operator. For example:

```
uvm_config_db#(int)::set(this, "*", "A");
```

The parameter value "int" identifies the configuration type as an int property.

The set and get methods provide the same api and semantics as the set/get_config_* functions in uvm_component.

Summary

uvm_config_db#(T)

The uvm_config_db#(T) class provides a convenience interface on top of the uvm_resource_db to simplify the basic interface that is used for reading and writing into the resource database.

Class Hierarchy

uvm_resource_db#(T)

uvm_config_db#(T)

Class Declaration

```
class uvm_config_db#(
    type   T =  int
) extends uvm_resource_db#(T)
```

Methods

get	Get the value *field_name* in *inst_name*, using component *cntxt* as the starting search point.
set	Create a new or update an existing configuration setting for *field_name* in *inst_name* from *cntxt*.
exists	Check if a value for *field_name* is available in *inst_name*, using component *cntxt* as the starting search point.
wait_modified	Wait for a configuration setting to be set for *field_name* in *cntxt* and *inst_name*.

Methods

get

```
static function bit get(    uvm_component cntxt,
                            string        inst_name,
                            string        field_name,
                        ref T             value      )
```

Get the value *field_name* in *inst_name*, using component *cntxt* as the starting search point. *inst_name* is an explicit instance name relative to *cntxt* and may be an empty string if the *cntxt* is the instance that the configuration object applies to. *field_name* is the specific field in the scope that is being searched for.

The basic get_config_* methods from uvm_component are mapped to this function as:

```
get_config_int(...) => uvm_config_db#(uvm_bitstream_t)::get(cntxt,...)
get_config_string(...) => uvm_config_db#(string)::get(cntxt,...)
get_config_object(...) => uvm_config_db#(uvm_object)::get(cntxt,...)
```

set

```
static function void set(uvm_component cntxt,
                         string        inst_name,
                         string        field_name,
                         T             value      )
```

Create a new or update an existing configuration setting for *field_name* in *inst_name* from *cntxt*. The setting is made at *cntxt*, with the full name of *cntxt* added to the *inst_name*. If *cntxt* is null then *inst_name* provides the complete scope information of the setting. *field_name* is the target field. Both *inst_name* and *field_name* may be glob style or regular expression style expressions.

If a setting is made at build time, the *cntxt* hierarchy is used to determine the setting's precedence in the database. Settings from hierarchically higher levels have higher precedence. Settings from the same level of hierarchy have a last setting wins semantic. A precedence setting of uvm_resource_base::default_precedence is used for uvm_top, and each hierarcical level below the top is decremented by 1.

After build time, all settings use the default precedence and thus have a last wins semantic. So, if at run time, a low level component makes a runtime setting of some field, that setting will have precedence over a setting from the test level that was made earlier in the simulation.

The basic set_config_* methods from uvm_component are mapped to this function as:

```
set_config_int(...) => uvm_config_db#(uvm_bitstream_t)::set(cntxt,...)
set_config_string(...) => uvm_config_db#(string)::set(cntxt,...)
set_config_object(...) => uvm_config_db#(uvm_object)::set(cntxt,...)
```

exists

```
static function bit exists(uvm_component cntxt,
                           string        inst_name,
                           string        field_name,
                           bit           spell_chk   = )
```

Check if a value for *field_name* is available in *inst_name*, using component *cntxt* as the starting search point. *inst_name* is an explicit instance name relative to *cntxt* and may be an empty string if the *cntxt* is the instance that the configuration object applies to. *field_name* is the specific field in the scope that is being searched for. The *spell_chk* arg can be set to 1 to turn spell checking on if it is expected that the field should exist in the database. The function returns 1 if a config parameter exists and 0 if it doesn't exist.

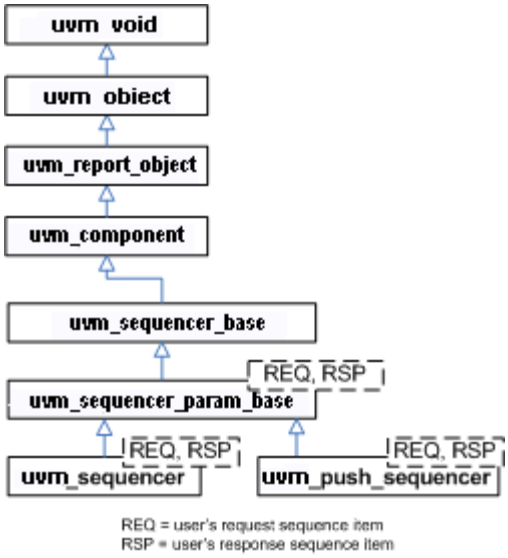
wait_modified

```
static task wait_modified(uvm_component cntxt,
                          string        inst_name,
                          string        field_name)
```

Wait for a configuration setting to be set for *field_name* in *cntxt* and *inst_name*. The task blocks until a new configuration setting is applied that effects the specified field.

Sequencer Classes

The sequencer serves as an arbiter for controlling transaction flow from multiple stimulus generators. More specifically, the sequencer controls the flow of uvm_sequence_item-based transactions generated by one or more uvm_sequence #(REQ,RSP)-based sequences.



There are two sequencer variants available.

- uvm_sequencer #(REQ,RSP) - Requests for new sequence items are initiated by the driver. Upon such requests, the sequencer selects a sequence from a list of available sequences to produce and deliver the next item to execute. This sequencer is typically connected to a user-extension of uvm_driver #(REQ,RSP).
- uvm_push_sequencer #(REQ,RSP) - Sequence items (from the currently running sequences) are pushed by the sequencer to the driver, which blocks item flow when it is not ready to accept new transactions. This sequencer is typically connected to a user-extension of uvm_push_driver #(REQ,RSP).

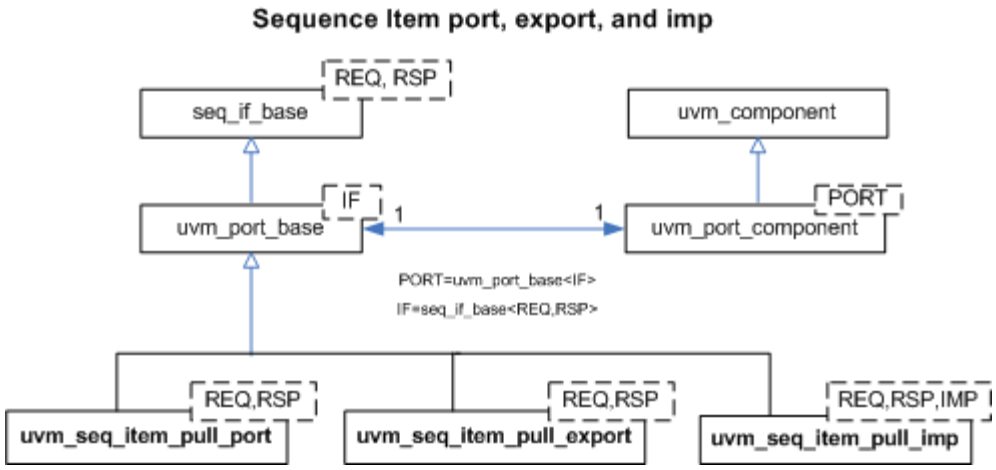
Sequencer-driver communication follows a *pull* or *push* semantic, depending on which sequencer type is used. However, sequence-sequencer communication is *always* initiated by the user-defined sequence, i.e. follows a push semantic.

See Sequence Classes for an overview on sequences and sequence items.

Sequence Item Ports

As with all UVM components, the sequencers and drivers described above use TLM Interfaces to communicate transactions.

The uvm_sequencer #(REQ,RSP) and uvm_driver #(REQ,RSP) pair also uses a *sequence item pull port* to achieve the special execution semantic needed by the sequencer-driver pair.



Sequencers and drivers use a *seq_item_port* specifically supports sequencer-driver communication. Connections to these ports are made in the same fashion as the TLM ports.

Summary

Sequencer Classes

The sequencer serves as an arbiter for controlling transaction flow from multiple stimulus generators.

uvm_sequencer_base

Controls the flow of sequences, which generate the stimulus (sequence item transactions) that is passed on to drivers for execution.

Summary

uvm_sequencer_base

Controls the flow of sequences, which generate the stimulus (sequence item transactions) that is passed on to drivers for execution.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- **uvm_sequencer_base**

Class Declaration

```
class uvm_sequencer_base extends uvm_component
```

Methods

new	Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: name is the name of the instance, and parent is the handle to the hierarchical parent.
is_child	Returns 1 if the child sequence is a child of the parent sequence, 0 otherwise.
user_priority_arbitration	When the sequencer arbitration mode is set to SEQ_ARB_USER (via the set_arbitration method), the sequencer will call this function each time that it needs to arbitrate among sequences.
execute_item	This task allows the user to supply an item or sequence to the sequencer and have it be executed procedurally.
start_phase_sequence	Start the default sequence for this phase, if any.
wait_for_grant	This task issues a request for the specified sequence.
wait_for_item_done	A sequence may optionally call wait_for_item_done.
is_blocked	Returns 1 if the sequence referred to by sequence_ptr is currently locked out of the sequencer.
has_lock	Returns 1 if the sequence refered to in the parameter currently has a lock on this sequencer, 0 otherwise.
lock	Requests a lock for the sequence specified by sequence_ptr.
grab	Requests a lock for the sequence specified by sequence_ptr.
unlock	Removes any locks and grabs obtained by the specified sequence_ptr.
ungrab	Removes any locks and grabs obtained by the specified sequence_ptr.
stop_sequences	Tells the sequencer to kill all sequences and child sequences currently operating on the sequencer, and remove all requests, locks and responses that are currently queued.

is_grabbed	Returns 1 if any sequence currently has a lock or grab on this sequencer, 0 otherwise.
current_grabber	Returns a reference to the sequence that currently has a lock or grab on the sequence.
has_do_available	Returns 1 if any sequence running on this sequencer is ready to supply a transaction, 0 otherwise.
set_arbitration	Specifies the arbitration mode for the sequencer.
get_arbitration	Return the current arbitration mode set for this sequencer.
wait_for_sequences	Waits for a sequence to have a new item available.
send_request	Derived classes implement this function to send a request item to the sequencer, which will forward it to the driver.

METHODS

new

```
function new (string        name,
              uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: name is the name of the instance, and parent is the handle to the hierarchical parent.

is_child

```
function bit is_child (uvm_sequence_base parent,
                       uvm_sequence_base child  )
```

Returns 1 if the child sequence is a child of the parent sequence, 0 otherwise.

user_priority_arbitration

```
virtual function integer user_priority_arbitration(integer avail_sequences[$]
```

When the sequencer arbitration mode is set to SEQ_ARB_USER (via the set_arbitration method), the sequencer will call this function each time that it needs to arbitrate among sequences.

Derived sequencers may override this method to perform a custom arbitration policy. The override must return one of the entries from the avail_sequences queue, which are indexes into an internal queue, arb_sequence_q. The

The default implementation behaves like SEQ_ARB_FIFO, which returns the entry at avail_sequences[0].

execute_item

```
virtual task execute_item(uvm_sequence_item item)
```

This task allows the user to supply an item or sequence to the sequencer and have it be executed procedurally. The parent sequence for the item or sequence is a temporary sequence that is automatically created. There is no capability to retrieve responses. The sequencer will drop responses to items done using this interface.

start_phase_sequence

```
virtual function void start_phase_sequence(uvm_phase phase)
```

Start the default sequence for this phase, if any. The default sequence is configured using resources using either a sequence instance or sequence object wrapper.

When setting the resource using *set*, the 1st argument specifies the context pointer, usually "this" for components or "null" when executed from outside the component hierarchy (i.e. in module). The 2nd argument is the instance string, which is a path name to the target sequencer, relative to the context pointer. The path must include the name of the phase with a "_phase" suffix. The 3rd argument is the resource name, which is "default_sequence". The 4th argument is either an object wrapper for the sequence type, or an instance of a sequence.

Configuration by instances allows pre-initialization, setting rand_mode, use of inline constraints, etc.

```
myseq_t myseq = new("myseq");
myseq.randomize() with { ... };
uvm_config_db #(uvm_sequence_base)::set(null, "top.agent.myseqr.main_phase",
                                        "default_sequence",
                                        myseq);
```

Configuration by type is shorter and can be substituted via the the factory.

```
uvm_config_db #(uvm_object_wrapper)::set(null,
"top.agent.myseqr.main_phase",
                                         "default_sequence",
                                         myseq_type::type_id::get());
```

The uvm_resource_db can similarly be used.

```
myseq_t myseq = new("myseq");
myseq.randomize() with { ... };
uvm_resource_db #(uvm_sequence_base)::set({get_full_name(),
".myseqr.main_phase",
                                          "default_sequence",
                                          myseq, this);
```

```
uvm_resource_db #(uvm_object_wrapper)::set({get_full_name(),
".myseqr.main_phase",
                                           "default_sequence",
                                           myseq_t::type_id::get(),
                                           this );
```

wait_for_grant

```
virtual task wait_for_grant(uvm_sequence_base sequence_ptr,
```

```
                              int               item_priority = -1,
                              bit               lock_request  = 0  )
```

This task issues a request for the specified sequence. If item_priority is not specified, then the current sequence priority will be used by the arbiter. If a lock_request is made, then the sequencer will issue a lock immediately before granting the sequence. (Note that the lock may be granted without the sequence being granted if is_relevant is not asserted).

When this method returns, the sequencer has granted the sequence, and the sequence must call send_request without inserting any simulation delay other than delta cycles. The driver is currently waiting for the next item to be sent via the send_request call.

wait_for_item_done

```
virtual task wait_for_item_done(uvm_sequence_base sequence_ptr,
                                int               transaction_id)
```

A sequence may optionally call wait_for_item_done. This task will block until the driver calls item_done() or put() on a transaction issued by the specified sequence. If no transaction_id parameter is specified, then the call will return the next time that the driver calls item_done() or put(). If a specific transaction_id is specified, then the call will only return when the driver indicates that it has completed that specific item.

Note that if a specific transaction_id has been specified, and the driver has already issued an item_done or put for that transaction, then the call will hang waiting for that specific transaction_id.

is_blocked

```
function bit is_blocked(uvm_sequence_base sequence_ptr)
```

Returns 1 if the sequence referred to by sequence_ptr is currently locked out of the sequencer. It will return 0 if the sequence is currently allowed to issue operations.

Note that even when a sequence is not blocked, it is possible for another sequence to issue a lock before this sequence is able to issue a request or lock.

has_lock

```
function bit has_lock(uvm_sequence_base sequence_ptr)
```

Returns 1 if the sequence refered to in the parameter currently has a lock on this sequencer, 0 otherwise.

Note that even if this sequence has a lock, a child sequence may also have a lock, in which case the sequence is still blocked from issueing operations on the sequencer

lock

```
virtual task lock(uvm_sequence_base sequence_ptr)
```

Requests a lock for the sequence specified by sequence_ptr.

A lock request will be arbitrated the same as any other request. A lock is granted after all earlier requests are completed and no other locks or grabs are blocking this sequence.

The lock call will return when the lock has been granted.

grab

```
virtual task grab(uvm_sequence_base sequence_ptr)
```

Requests a lock for the sequence specified by sequence_ptr.

A grab request is put in front of the arbitration queue. It will be arbitrated before any other requests. A grab is granted when no other grabs or locks are blocking this sequence.

The grab call will return when the grab has been granted.

unlock

```
virtual function void unlock(uvm_sequence_base sequence_ptr)
```

Removes any locks and grabs obtained by the specified sequence_ptr.

ungrab

```
virtual function void ungrab(uvm_sequence_base sequence_ptr)
```

Removes any locks and grabs obtained by the specified sequence_ptr.

stop_sequences

```
virtual function void stop_sequences()
```

Tells the sequencer to kill all sequences and child sequences currently operating on the sequencer, and remove all requests, locks and responses that are currently queued. This essentially resets the sequencer to an idle state.

is_grabbed

```
virtual function bit is_grabbed()
```

Returns 1 if any sequence currently has a lock or grab on this sequencer, 0 otherwise.

current_grabber

```
virtual function uvm_sequence_base current_grabber()
```

Returns a reference to the sequence that currently has a lock or grab on the sequence. If multiple hierarchical sequences have a lock, it returns the child that is currently allowed to perform operations on the sequencer.

has_do_available

```
virtual function bit has_do_available()
```

Returns 1 if any sequence running on this sequencer is ready to supply a transaction, 0 otherwise. A sequence is ready if it is not blocked (via *grab* or *lock* and *is_relevant* returns 1.

set_arbitration

```
function void set_arbitration(SEQ_ARB_TYPE val)
```

Specifies the arbitration mode for the sequencer. It is one of

SEQ_ARB_FIFO	Requests are granted in FIFO order (default)
SEQ_ARB_WEIGHTED	Requests are granted randomly by weight
SEQ_ARB_RANDOM	Requests are granted randomly
SEQ_ARB_STRICT_FIFO	Requests at highest priority granted in fifo order
SEQ_ARB_STRICT_RANDOM	Requests at highest priority granted in randomly
SEQ_ARB_USER	Arbitration is delegated to the user-defined function, user_priority_arbitration. That function will specify the next sequence to grant.

The default user function specifies FIFO order.

get_arbitration

```
function SEQ_ARB_TYPE get_arbitration()
```

Return the current arbitration mode set for this sequencer. See set_arbitration for a list of possible modes.

wait_for_sequences

```
virtual task wait_for_sequences()
```

Waits for a sequence to have a new item available. Uses uvm_wait_for_nba_region to give a sequence as much time as possible to deliver an item before advancing time.

send_request

```
virtual function void send_request(uvm_sequence_base sequence_ptr,
                                   uvm_sequence_item t,
                                   bit               rerandomize   = 0)
```

Derived classes implement this function to send a request item to the sequencer, which will forward it to the driver. If the rerandomize bit is set, the item will be randomized before being sent to the driver.

This function may only be called after a wait_for_grant call.

uvm_sequencer_param_base #(REQ,RSP)

Extends uvm_sequencer_base with an API depending on specific request (REQ) and response (RSP) types.

Summary

uvm_sequencer_param_base #(REQ,RSP)

Extends uvm_sequencer_base with an API depending on specific request (REQ) and response (RSP) types.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- uvm_sequencer_base
- **uvm_sequencer_param_base#(REQ,RSP)**

Class Declaration

```
class uvm_sequencer_param_base #(
   type REQ = uvm_sequence_item,
   type RSP = REQ
) extends uvm_sequencer_base
```

new	Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: name is the name of the instance, and parent is the handle to the hierarchical parent, if any.
send_request	The send_request function may only be called after a wait_for_grant call.
get_current_item	Returns the request_item currently being executed by the sequencer.

Requests

get_num_reqs_sent	Returns the number of requests that have been sent by this sequencer.
set_num_last_reqs	Sets the size of the last_requests buffer.
get_num_last_reqs	Returns the size of the last requests buffer, as set by set_num_last_reqs.
last_req	Returns the last request item by default.

Responses

rsp_export	Drivers or monitors can connect to this port to send responses to the sequencer.
get_num_rsps_received	Returns the number of responses received thus far by this sequencer.
set_num_last_rsps	Sets the size of the last_responses buffer.
get_num_last_rsps	Returns the max size of the last responses buffer, as set by set_num_last_rsps.
last_rsp	Returns the last response item by default.

new

```
function new (string        name,
```

```
                uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: name is the name of the instance, and parent is the handle to the hierarchical parent, if any.

send_request

```
virtual function void send_request(uvm_sequence_base sequence_ptr,
                                   uvm_sequence_item t,
                                   bit               rerandomize   = 0)
```

The send_request function may only be called after a wait_for_grant call. This call will send the request item, t, to the sequencer pointed to by sequence_ptr. The sequencer will forward it to the driver. If rerandomize is set, the item will be randomized before being sent to the driver.

get_current_item

```
function REQ get_current_item()
```

Returns the request_item currently being executed by the sequencer. If the sequencer is not currently executing an item, this method will return null.

The sequencer is executing an item from the time that get_next_item or peek is called until the time that get or item_done is called.

Note that a driver that only calls get() will never show a current item, since the item is completed at the same time as it is requsted.

Requests

get_num_reqs_sent

```
function int get_num_reqs_sent()
```

Returns the number of requests that have been sent by this sequencer.

set_num_last_reqs

```
function void set_num_last_reqs(int unsigned max)
```

Sets the size of the last_requests buffer. Note that the maximum buffer size is 1024. If max is greater than 1024, a warning is issued, and the buffer is set to 1024. The default value is 1.

get_num_last_reqs

```
function int unsigned get_num_last_reqs()
```

Returns the size of the last requests buffer, as set by set_num_last_reqs.

last_req

```
function REQ last_req(int unsigned n = 0)
```

Returns the last request item by default. If n is not 0, then it will get the n¿th before last request item. If n is greater than the last request buffer size, the function will return null.

Responses

rsp_export

Drivers or monitors can connect to this port to send responses to the sequencer. Alternatively, a driver can send responses via its seq_item_port.

```
seq_item_port.item_done(response)
seq_item_port.put(response)
rsp_port.write(response)   <--- via this export
```

The rsp_port in the driver and/or monitor must be connected to the rsp_export in this sequencer in order to send responses through the response analysis port.

get_num_rsps_received

```
function int get_num_rsps_received()
```

Returns the number of responses received thus far by this sequencer.

set_num_last_rsps

```
function void set_num_last_rsps(int unsigned max)
```

Sets the size of the last_responses buffer. The maximum buffer size is 1024. If max is greater than 1024, a warning is issued, and the buffer is set to 1024. The default value is 1.

get_num_last_rsps

```
function int unsigned get_num_last_rsps()
```

Returns the max size of the last responses buffer, as set by set_num_last_rsps.

last_rsp

```
function RSP last_rsp(int unsigned n = 0)
```

Returns the last response item by default. If n is not 0, then it will get the nth-before-

last response item. If n is greater than the last response buffer size, the function will return null.

uvm_sequencer #(REQ,RSP)

Summary

uvm_sequencer #(REQ,RSP)

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- uvm_sequencer_base
- uvm_sequencer_param_base#(REQ,RSP)
- **uvm_sequencer#(REQ,RSP)**

Class Declaration

```
class uvm_sequencer #(
   type REQ = uvm_sequence_item,
        RSP = REQ
) extends uvm_sequencer_param_base #(REQ, RSP)
```

Variables

seq_item_export	This export provides access to this sequencer's implementation of the sequencer interface, uvm_sqr_if_base #(REQ,RSP), which defines the following methods:

Methods

new	Standard component constructor that creates an instance of this class using the given *name* and *parent*, if any.
stop_sequences	Tells the sequencer to kill all sequences and child sequences currently operating on the sequencer, and remove all requests, locks and responses that are currently queued.
new	Standard component constructor that creates an instance of this class using the given *name* and *parent*, if any.
stop_sequences	Tells the sequencer to kill all sequences and child sequences currently operating on the sequencer, and remove all requests, locks and responses that are currently queued.

Variables

seq_item_export

```
uvm_seq_item_pull_imp #(REQ,
                        RSP,
                        this_type) seq_item_export
```

This export provides access to this sequencer's implementation of the sequencer interface, uvm_sqr_if_base #(REQ,RSP), which defines the following methods:

```
Requests:
 virtual task          get_next_item      (output REQ request);
 virtual task          try_next_item      (output REQ request);
 virtual task          get                (output REQ request);
 virtual task          peek               (output REQ request);
Responses:
 virtual function void item_done          (input RSP response=null);
 virtual task          put                (input RSP response);
Sync Control:
 virtual task          wait_for_sequences ();
 virtual function bit  has_do_available   ();
```

See uvm_sqr_if_base #(REQ,RSP) for information about this interface.

METHODS

new

```
function new (string        name,
              uvm_component parent = null)
```

Standard component constructor that creates an instance of this class using the given *name* and *parent*, if any.

stop_sequences

```
virtual function void stop_sequences()
```

Tells the sequencer to kill all sequences and child sequences currently operating on the sequencer, and remove all requests, locks and responses that are currently queued. This essentially resets the sequencer to an idle state.

new

```
function uvm_sequencer::new (string        name,
                             uvm_component parent = null)
```

Standard component constructor that creates an instance of this class using the given *name* and *parent*, if any.

stop_sequences

```
function void uvm_sequencer::stop_sequences()
```

Tells the sequencer to kill all sequences and child sequences currently operating on the sequencer, and remove all requests, locks and responses that are currently queued. This essentially resets the sequencer to an idle state.

uvm_push_sequencer #(REQ,RSP)

Summary

uvm_push_sequencer #(REQ,RSP)

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- uvm_sequencer_base
- uvm_sequencer_param_base#(REQ,RSP)
- **uvm_push_sequencer#(REQ,RSP)**

Class Declaration

```
class uvm_push_sequencer #(
   type REQ = uvm_sequence_item,
        RSP = REQ
) extends uvm_sequencer_param_base #(REQ, RSP)
```

Ports

req_port	The push sequencer requires access to a blocking put interface.

Methods

new	Standard component constructor that creates an instance of this class using the given *name* and *parent*, if any.
run_phase	The push sequencer continuously selects from its list of available sequences and sends the next item from the selected sequence out its req_port using req_port.put(item).

Ports

req_port

The push sequencer requires access to a blocking put interface. A continuous stream of sequence items are sent out this port, based on the list of available sequences loaded into this sequencer.

Methods

new

```
function new (string        name,
              uvm_component parent = null)
```

Standard component constructor that creates an instance of this class using the given *name* and *parent*, if any.

run_phase

```
task run_phase(uvm_phase phase)
```

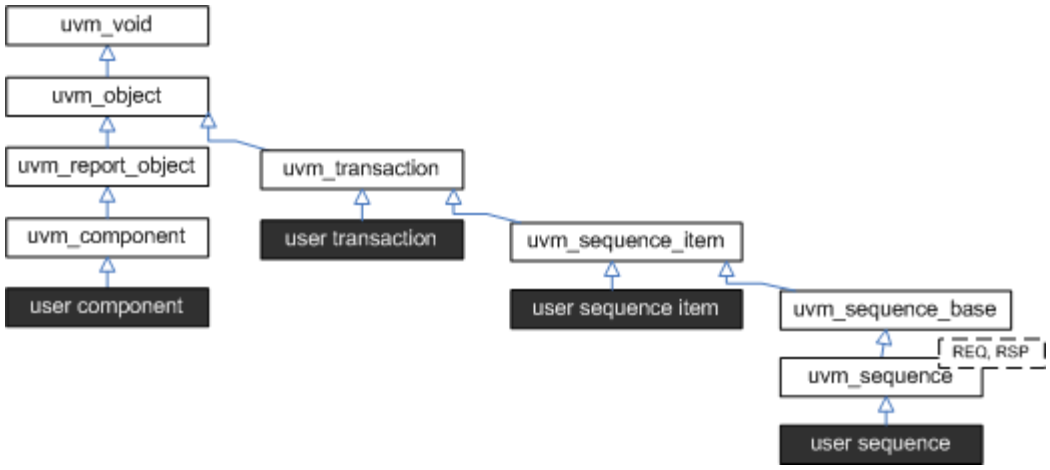
The push sequencer continuously selects from its list of available sequences and sends the next item from the selected sequence out its req_port using req_port.put(item). Typically, the req_port would be connected to the req_export on an instance of an uvm_push_driver #(REQ,RSP), which would be responsible for executing the item.

Sequence Classes

Sequences encapsulate user-defined procedures that generate multiple uvm_sequence_item-based transactions. Such sequences can be reused, extended, randomized, and combined sequentially and hierarchically in interesting ways to produce realistic stimulus to your DUT.

With *uvm_sequence* objects, users can encapsulate DUT initializaton code, bus-based stress tests, network protocol stacks-- anything procedural-- then have them all execute in specific or random order to more quickly reach corner cases and coverage goals.

The UVM sequence item and sequence class hierarchy is shown below.



- uvm_sequence_item - The *uvm_sequence_item* is the base class for user-defined transactions that leverage the stimulus generation and control capabilities of the sequence-sequencer mechanism.
- uvm_sequence #(REQ,RSP) - The *uvm_sequence* extends *uvm_sequence_item* to add the ability to generate streams of *uvm_sequence_items*, either directly or by recursively execting other *uvm_sequences*.

Summary

Sequence Classes

Sequences encapsulate user-defined procedures that generate multiple uvm_sequence_item-based transactions.

uvm_sequence_item

The base class for user-defined sequence items and also the base class for the uvm_sequence class. The uvm_sequence_item class provides the basic functionality for objects, both sequence items and sequences, to operate in the sequence mechanism.

Summary

uvm_sequence_item

The base class for user-defined sequence items and also the base class for the uvm_sequence class.

Class Hierarchy

uvm_void
uvm_object
uvm_transaction
uvm_sequence_item

Class Declaration

```
class uvm_sequence_item extends uvm_transaction
```

new	The constructor method for uvm_sequence_item.
get_sequence_id	private
set_use_sequence_info get_use_sequence_info	These methods are used to set and get the status of the use_sequence_info bit.
set_id_info	Copies the sequence_id and transaction_id from the referenced item into the calling item.
set_sequencer	Sets the default sequencer for the sequence to sequencer.
get_sequencer	Returns a reference to the default sequencer used by this sequence.
set_parent_sequence	Sets the parent sequence of this sequence_item.
get_parent_sequence	Returns a reference to the parent sequence of any sequence on which this method was called.
set_depth	The depth of any sequence is calculated automatically.
get_depth	Returns the depth of a sequence from it's parent.
is_item	This function may be called on any sequence_item or sequence.
start_item	start_item and finish_item together will initiate operation of either a sequence_item or sequence object.
finish_item	Finishes execution of a sequence item or sequence.
get_root_sequence_name	Provides the name of the root sequence (the top-most parent sequence).
get_root_sequence	Provides a reference to the root sequence (the top-most parent sequence).
get_sequence_path	Provides a string of names of each sequence in the full hierarchical path.
Reporting Interface	Sequence items and sequences will use the sequencer which they are associated with for reporting messages.
uvm_report_info uvm_report_warning uvm_report_error uvm_report_fatal	These are the primary reporting methods in the UVM.

new

```
function new (string name = "uvm_sequence_item")
```

The constructor method for uvm_sequence_item.

get_sequence_id

```
function int get_sequence_id()
```

private

Get_sequence_id is an internal method that is not intended for user code. The sequence_id is not a simple integer. The get_transaction_id is meant for users to identify specific transactions.

These methods allow access to the sequence_item sequence and transaction IDs. get_transaction_id and set_transaction_id are methods on the uvm_transaction base_class. These IDs are used to identify sequences to the sequencer, to route responses back to the sequence that issued a request, and to uniquely identify transactions.

The sequence_id is assigned automatically by a sequencer when a sequence initiates communication through any sequencer calls (i.e. `uvm_do_xxx, wait_for_grant). A sequence_id will remain unique for this sequence until it ends or it is killed. However, a single sequence may have multiple valid sequence ids at any point in time. Should a sequence start again after it has ended, it will be given a new unique sequence_id.

The transaction_id is assigned automatically by the sequence each time a transaction is sent to the sequencer with the transaction_id in its default (-1) value. If the user sets the transaction_id to any non-default value, that value will be maintained.

Responses are routed back to this sequences based on sequence_id. The sequence may use the transaction_id to correlate responses with their requests.

set_use_sequence_info

```
function void set_use_sequence_info(bit value)
```

get_use_sequence_info

```
function bit get_use_sequence_info()
```

These methods are used to set and get the status of the use_sequence_info bit. Use_sequence_info controls whether the sequence information (sequencer, parent_sequence, sequence_id, etc.) is printed, copied, or recorded. When use_sequence_info is the default value of 0, then the sequence information is not used. When use_sequence_info is set to 1, the sequence information will be used in printing and copying.

set_id_info

```
function void set_id_info(uvm_sequence_item item)
```

Copies the sequence_id and transaction_id from the referenced item into the calling item. This routine should always be used by drivers to initialize responses for future compatibility.

set_sequencer

```
virtual function void set_sequencer(uvm_sequencer_base sequencer)
```

Sets the default sequencer for the sequence to sequencer. It will take effect immediately, so it should not be called while the sequence is actively communicating with the sequencer.

get_sequencer

```
function uvm_sequencer_base get_sequencer()
```

Returns a reference to the default sequencer used by this sequence.

set_parent_sequence

```
function void set_parent_sequence(uvm_sequence_base parent)
```

Sets the parent sequence of this sequence_item. This is used to identify the source sequence of a sequence_item.

get_parent_sequence

```
function uvm_sequence_base get_parent_sequence()
```

Returns a reference to the parent sequence of any sequence on which this method was called. If this is a parent sequence, the method returns null.

set_depth

```
function void set_depth(int value)
```

The depth of any sequence is calculated automatically. However, the user may use set_depth to specify the depth of a particular sequence. This method will override the automatically calculated depth, even if it is incorrect.

get_depth

```
function int get_depth()
```

Returns the depth of a sequence from it's parent. A parent sequence will have a depth of 1, it's child will have a depth of 2, and it's grandchild will have a depth of 3.

is_item

```
virtual function bit is_item()
```

This function may be called on any sequence_item or sequence. It will return 1 for items and 0 for sequences (which derive from this class).

start_item

```
virtual task start_item (uvm_sequence_item  item_or_seq,
                         int                set_priority = -1,
                         uvm_sequencer_base sequencer    = null)
```

start_item and finish_item together will initiate operation of either a sequence_item or sequence object. If the object has not been initiated using create_item, then start_item will be initialized in start_item to use the default sequencer specified by m_sequencer. Randomization may be done between start_item and finish_item to ensure late generation

finish_item

```
virtual task finish_item (uvm_sequence_item item,
                          int               set_priority = -1)
```

Finishes execution of a sequence item or sequence. Finish_item must be called after start_item returns with no delays or delta-cycles. Randomization, or other functions may be called between the start_item and *finish_item* calls.

get_root_sequence_name

```
function string get_root_sequence_name()
```

Provides the name of the root sequence (the top-most parent sequence).

get_root_sequence

```
function uvm_sequence_base get_root_sequence()
```

Provides a reference to the root sequence (the top-most parent sequence).

get_sequence_path

```
function string get_sequence_path()
```

Provides a string of names of each sequence in the full hierarchical path. A ".” is used as the separator between each sequence.

Reporting Interface

Sequence items and sequences will use the sequencer which they are associated with for reporting messages. If no sequencer has been set for the item/sequence using set_sequencer (or start_item), then the global reporter will be used.

uvm_report_info

```
virtual function void uvm_report_info(string id,
                                       string message,
                                       int    verbosity = UVM_MEDIUM,
                                       string filename  = "",
                                       int    line      = 0          )
```

uvm_report_warning

```
virtual function void uvm_report_warning(string id,
                                         string message,
                                         int    verbosity = UVM_MEDIUM,
                                         string filename  = "",
                                         int    line      = 0          )
```

uvm_report_error

```
virtual function void uvm_report_error(string id,
                                       string message,
                                       int    verbosity = UVM_LOW,
                                       string filename  = "",
                                       int    line      = 0       )
```

uvm_report_fatal

```
virtual function void uvm_report_fatal(string id,
                                       string message,
                                       int    verbosity = UVM_NONE,
                                       string filename  = "",
                                       int    line      = 0        )
```

These are the primary reporting methods in the UVM. uvm_sequence_item derived types delegate these functions to their associated sequencer if they have one, or to the global reporter. See uvm_report_object::Reporting for details on the messaging functions.

uvm_sequence_base

The uvm_sequence_base class provides the interfaces needed to create streams of sequence items and/or other sequences.

A sequence is executed by calling its start method, either directly or indirectly via start_item/finish_item or invocation of any of the `uvm_do_* macros.

Executing sequences via start

A sequence's start method has a *parent_sequence* argument that controls whether pre_do, mid_do, and post_do are called **in the parent** sequence. It also has a *call_pre_post* argument that controls whether its pre_body and post_body methods are called.

When start is called directly, you can provide the appropriate arguments according to your application.

The sequence execution flow looks like

User code

```
sub_seq.randomize(...); // optional
sub_seq.start(seqr, parent_seq, priority, *call_pre_post*)
```

The following methods are called, in order

```
sub_seq.pre_body            (task)  if call_pre_post==1
  parent_seq.pre_do(0)      (task)  if parent_sequence!=null
  parent_seq.mid_do(this)   (func)  if parent_sequence!=null
sub_seq.body                (task)  YOUR STIMULUS CODE
  parent_seq.post_do(this)  (func)  if parent_sequence!=null
sub_seq.post_body           (task)  if call_pre_post==1
```

Executing sub-sequences via start_item/finish_item or `uvm_do macros

A sequence can also be indirectly started as a child in the body of a parent sequence. The child sequence's start method is called indirectly via calls to its start_item/finish_item methods or by invoking any of the `uvm_do macros. Child sequences can also be started by the predefined sequences, <uvm_random_sequence> and <uvm_exhaustive_sequence>. In all these cases, start is called with *call_pre_post* set to 0, preventing the started sequence's pre_body and post_body methods from being called. During execution of the child sequence, the parent's pre_do, mid_do, and post_do methods are called.

The sub-sequence execution flow looks like

User code

```
parent_seq.start_item(sub_seq, priority);
sub_seq.randomize(...);
parent_seq.finish_item(sub_seq);

or

`uvm_do_with_prior(seq_seq, { constraints }, priority)
```

The following methods are called, in order

```
parent_seq.pre_do(0)         (task)
parent_req.mid_do(sub_seq)  (func)
  sub_seq.body               (task)
parent_seq.post_do(sub_seq) (func)
```

Remember, it is the **parent** sequence's pre|mid|post_do that are called, not the sequence being executed.

Executing sequence items via start_item/finish_item or `uvm_do macros

Items are started in the body of a parent sequence via calls to start_item/finish_item or invocations of any of the `uvm_do macros. The pre_do, mid_do, and post_do methods of the parent sequence will be called as the item is executed.

The sequence-item execution flow looks like

User code

```
parent_seq.start_item(item, priority);
sub_seq.randomize(...) [with {constraints}];
parent_seq.finish_item(item);

or

`uvm_do_with_prior(item, constraints, priority)
```

The following methods are called, in order

```
sequencer.wait_for_grant(prior) (task) \ start_item   \
parent_seq.pre_do(1)             (task) /              \
                                                        `uvm_do* macros
parent_seq.mid_do(item)          (func) \              /
sequencer.send_request(item)    (func)  \finish_item /
sequencer.wait_for_item_done()  (task)  /
parent_seq.post_do(item)         (func) /
```

Summary

uvm_sequence_base

The uvm_sequence_base class provides the interfaces needed to create streams of sequence items and/or other sequences.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_transaction
- uvm_sequence_item
- **uvm_sequence_base**

Class Declaration

```
class uvm_sequence_base extends uvm_sequence_item
```

new	The constructor for

	uvm_sequence_base.
is_item	Returns 1 on items and 0 on sequences.
get_sequence_state	Returns the sequence state as an enumerated value.
wait_for_sequence_state	Waits until the sequence reaches the given *state*.

Sequence Execution

start	Executes this sequence, returning when the sequence has completed.
pre_body	This task is a user-definable callback that is called before the execution of body *only* when the sequence is started with start.
pre_do	This task is a user-definable callback task that is called *on the parent sequence*, if any.the sequence has issued a wait_for_grant() call and after the sequencer has selected this sequence, and before the item is randomized.
mid_do	This function is a user-definable callback function that is called after the sequence item has been randomized, and just before the item is sent to the driver.
body	This is the user-defined task where the main sequence code resides.
post_do	This function is a user-definable callback function that is called after the driver has indicated that it has completed the item, using either this item_done or put methods.
post_body	This task is a user-definable callback task that is called after the execution of body *only* when the sequence is started with start.
starting_phase	If non-null, specifies the phase in which this sequence was started.

Sequence Control

set_priority	The priority of a sequence may be changed at any point in time.
get_priority	This function returns the current priority of the sequence.
is_relevant	The default is_relevant implementation returns 1, indicating that the sequence is always relevant.
wait_for_relevant	This method is called by the sequencer when all available sequences are not relevant.
lock	Requests a lock on the specified sequencer.
grab	Requests a lock on the specified sequencer.
unlock	Removes any locks or grabs obtained by this sequence on the specified sequencer.

ungrab	Removes any locks or grabs obtained by this sequence on the specified sequencer.
is_blocked	Returns a bit indicating whether this sequence is currently prevented from running due to another lock or grab.
has_lock	Returns 1 if this sequence has a lock, 0 otherwise.
kill	This function will kill the sequence, and cause all current locks and requests in the sequence's default sequencer to be removed.
do_kill	This function is a user hook that is called whenever a sequence is terminated by using either sequence.kill() or sequencer.stop_sequences() (which effectively calls sequence.kill()).
Sequence Item Execution	
create_item	Create_item will create and initialize a sequence_item or sequence using the factory.
start_item	*start_item* and finish_item together will initiate operation of either a sequence item or sequence.
finish_item	finish_item, together with start_item together will initiate operation of either a sequence_item or sequence object.
wait_for_grant	This task issues a request to the current sequencer.
send_request	The send_request function may only be called after a wait_for_grant call.
wait_for_item_done	A sequence may optionally call wait_for_item_done.
Response API	
use_response_handler	When called with enable set to 1, responses will be sent to the response handler.
get_use_response_handler	Returns the state of the use_response_handler bit.
response_handler	When the use_reponse_handler bit is set to 1, this virtual task is called by the sequencer for each response that arrives for this sequence.
set_response_queue_error_report_disabled	By default, if the response_queue overflows, an error is reported.
get_response_queue_error_report_disabled	When this bit is 0 (default value), error reports are generated when the response queue overflows.
set_response_queue_depth	The default maximum depth of the response queue is 8.
get_response_queue_depth	Returns the current depth setting for the response queue.
clear_response_queue	Empties the response queue for this sequence.

new

```
function new (string name = "uvm_sequence")
```

The constructor for uvm_sequence_base.

is_item

```
virtual function bit is_item()
```

Returns 1 on items and 0 on sequences. As this object is a sequence, *is_item* will always return 0.

get_sequence_state

```
function uvm_sequence_state_enum get_sequence_state()
```

Returns the sequence state as an enumerated value. Can use to wait on the sequence reaching or changing from one or more states.

```
wait(get_sequence_state() & (STOPPED|FINISHED));
```

wait_for_sequence_state

```
task wait_for_sequence_state(uvm_sequence_state_enum state)
```

Waits until the sequence reaches the given *state*. If the sequence is already in this state, this method returns immediately. Convenience for wait (get_sequence_state == *state*);

Sequence Execution

start

```
virtual task start (uvm_sequencer_base sequencer,
                    uvm_sequence_base  parent_sequence = null,
                    integer            this_priority   = 100,
                    bit                call_pre_post   = 1    )
```

Executes this sequence, returning when the sequence has completed.

The *sequencer* argument specifies the sequencer on which to run this sequence. The sequencer must be compatible with the sequence.

If *parent_sequence* is null, then this sequence is a root parent, otherwise it is a child of *parent_sequence*. The *parent_sequence*'s pre_do, mid_do, and post_do methods will be called during the execution of this sequence.

By default, the *priority* of a sequence is 100. A different priority may be specified by *this_priority*. Higher numbers indicate higher priority.

If *call_pre_post* is set to 1 (default), then the pre_body and post_body tasks will be called before and after the sequence body is called.

pre_body

```
virtual task pre_body()
```

This task is a user-definable callback that is called before the execution of body *only* when the sequence is started with start. If start is called with *call_pre_post* set to 0, *pre_body* is not called. This method should not be called directly by the user.

pre_do

```
virtual task pre_do(bit is_item)
```

This task is a user-definable callback task that is called *on the parent sequence*, if any.the sequence has issued a wait_for_grant() call and after the sequencer has selected this sequence, and before the item is randomized.

Although pre_do is a task, consuming simulation cycles may result in unexpected behavior on the driver.

This method should not be called directly by the user.

mid_do

```
virtual function void mid_do(uvm_sequence_item this_item)
```

This function is a user-definable callback function that is called after the sequence item has been randomized, and just before the item is sent to the driver. This mehod should not be called directly by the user.

body

```
virtual task body()
```

This is the user-defined task where the main sequence code resides. This method should not be called directly by the user.

post_do

```
virtual function void post_do(uvm_sequence_item this_item)
```

This function is a user-definable callback function that is called after the driver has indicated that it has completed the item, using either this item_done or put methods. This method should not be called directly by the user.

post_body

```
virtual task post_body()
```

This task is a user-definable callback task that is called after the execution of body *only* when the sequence is started with start. If start is called with *call_pre_post* set to 0, *post_body* is not called. This task is a user-definable callback task that is called after the execution of the body, unless the sequence is started with call_pre_post=0. This method should not be called directly by the user.

starting_phase

```
uvm_phase starting_phase
```

If non-null, specifies the phase in which this sequence was started. The *starting_phase* is set automatically when this sequence is started as the default sequence. See uvm_sequencer_base::start_phase_sequence.

Sequence Control

set_priority

```
function void set_priority (int value)
```

The priority of a sequence may be changed at any point in time. When the priority of a sequence is changed, the new priority will be used by the sequencer the next time that it arbitrates between sequences.

The default priority value for a sequence is 100. Higher values result in higher priorities.

get_priority

```
function int get_priority()
```

This function returns the current priority of the sequence.

is_relevant

```
virtual function bit is_relevant()
```

The default is_relevant implementation returns 1, indicating that the sequence is always relevant.

Users may choose to override with their own virtual function to indicate to the sequencer that the sequence is not currently relevant after a request has been made.

When the sequencer arbitrates, it will call is_relevant on each requesting, unblocked sequence to see if it is relevant. If a 0 is returned, then the sequence will not be chosen.

If all requesting sequences are not relevant, then the sequencer will call wait_for_relevant on all sequences and re-arbitrate upon its return.

Any sequence that implements is_relevant must also implement wait_for_relevant so that the sequencer has a way to wait for a sequence to become relevant.

wait_for_relevant

```
virtual task wait_for_relevant()
```

This method is called by the sequencer when all available sequences are not relevant. When wait_for_relevant returns the sequencer attempt to re-arbitrate.

Returning from this call does not guarantee a sequence is relevant, although that would be the ideal. The method provide some delay to prevent an infinite loop.

If a sequence defines is_relevant so that it is not always relevant (by default, a sequence is always relevant), then the sequence must also supply a wait_for_relevant method.

lock

```
task lock(uvm_sequencer_base sequencer = null)
```

Requests a lock on the specified sequencer. If sequencer is null, the lock will be requested on the current default sequencer.

A lock request will be arbitrated the same as any other request. A lock is granted after all earlier requests are completed and no other locks or grabs are blocking this sequence.

The lock call will return when the lock has been granted.

grab

```
task grab(uvm_sequencer_base sequencer = null)
```

Requests a lock on the specified sequencer. If no argument is supplied, the lock will be requested on the current default sequencer.

A grab equest is put in front of the arbitration queue. It will be arbitrated before any other requests. A grab is granted when no other grabs or locks are blocking this sequence.

The grab call will return when the grab has been granted.

unlock

```
function void unlock(uvm_sequencer_base sequencer = null)
```

Removes any locks or grabs obtained by this sequence on the specified sequencer. If sequencer is null, then the unlock will be done on the current default sequencer.

ungrab

```
function void ungrab(uvm_sequencer_base sequencer = null)
```

Removes any locks or grabs obtained by this sequence on the specified sequencer. If sequencer is null, then the unlock will be done on the current default sequencer.

is_blocked

```
function bit is_blocked()
```

Returns a bit indicating whether this sequence is currently prevented from running due to another lock or grab. A 1 is returned if the sequence is currently blocked. A 0 is returned if no lock or grab prevents this sequence from executing. Note that even if a sequence is not blocked, it is possible for another sequence to issue a lock or grab before this sequence can issue a request.

has_lock

```
function bit has_lock()
```

Returns 1 if this sequence has a lock, 0 otherwise.

Note that even if this sequence has a lock, a child sequence may also have a lock, in which case the sequence is still blocked from issuing operations on the sequencer.

kill

```
function void kill()
```

This function will kill the sequence, and cause all current locks and requests in the sequence's default sequencer to be removed. The sequence state will change to STOPPED, and its post_body() method, if will not b

If a sequence has issued locks, grabs, or requests on sequencers other than the default sequencer, then care must be taken to unregister the sequence with the other sequencer(s) using the sequencer unregister_sequence() method.

do_kill

```
virtual function void do_kill()
```

This function is a user hook that is called whenever a sequence is terminated by using either sequence.kill() or sequencer.stop_sequences() (which effectively calls sequence.kill()).

Sequence Item Execution

create_item

```
protected function uvm_sequence_item create_item(
   uvm_object_wrapper type_var,
   uvm_sequencer_base l_sequencer,
   string             name
)
```

Create_item will create and initialize a sequence_item or sequence using the factory. The sequence_item or sequence will be initialized to communicate with the specified sequencer.

start_item

start_item and finish_item together will initiate operation of either a sequence item or sequence. If the item or sequence has not already been initialized using create_item, then it will be initialized here to use the default sequencer specified by m_sequencer. Randomization may be done between start_item and finish_item to ensure late generation

```
virtual task start_item(uvm_sequence_item item, int set_priority = -1);
```

finish_item

finish_item, together with start_item together will initiate operation of either a sequence_item or sequence object. Finish_item must be called after start_item with no delays or delta-cycles. Randomization, or other functions may be called between the start_item and finish_item calls.

```
virtual task finish_item(uvm_sequence_item item, int set_priority = -1);
```

wait_for_grant

```
virtual task wait_for_grant(int item_priority = -1,
                            bit lock_request  = 0  )
```

This task issues a request to the current sequencer. If item_priority is not specified, then the current sequence priority will be used by the arbiter. If a lock_request is made, then the sequencer will issue a lock immediately before granting the sequence. (Note that the lock may be granted without the sequence being granted if is_relevant is not asserted).

When this method returns, the sequencer has granted the sequence, and the sequence must call send_request without inserting any simulation delay other than delta cycles. The driver is currently waiting for the next item to be sent via the send_request call.

send_request

```
virtual function void send_request(uvm_sequence_item request,
                                   bit               rerandomize = 0)
```

The send_request function may only be called after a wait_for_grant call. This call will send the request item to the sequencer, which will forward it to the driver. If the rerandomize bit is set, the item will be randomized before being sent to the driver.

wait_for_item_done

```
virtual task wait_for_item_done(int transaction_id = -1)
```

A sequence may optionally call wait_for_item_done. This task will block until the driver

calls item_done or put. If no transaction_id parameter is specified, then the call will return the next time that the driver calls item_done or put. If a specific transaction_id is specified, then the call will return when the driver indicates completion of that specific item.

Note that if a specific transaction_id has been specified, and the driver has already issued an item_done or put for that transaction, then the call will hang, having missed the earlier notification.

Response API

use_response_handler

```
function void use_response_handler(bit enable)
```

When called with enable set to 1, responses will be sent to the response handler. Otherwise, responses must be retrieved using get_response.

By default, responses from the driver are retrieved in the sequence by calling get_response.

An alternative method is for the sequencer to call the response_handler function with each response.

get_use_response_handler

```
function bit get_use_response_handler()
```

Returns the state of the use_response_handler bit.

response_handler

```
virtual function void response_handler(uvm_sequence_item response)
```

When the use_reponse_handler bit is set to 1, this virtual task is called by the sequencer for each response that arrives for this sequence.

set_response_queue_error_report_disabled

```
function void set_response_queue_error_report_disabled(bit value)
```

By default, if the response_queue overflows, an error is reported. The response_queue will overflow if more responses are sent to this sequence from the driver than get_response calls are made. Setting value to 0 disables these errors, while setting it to 1 enables them.

get_response_queue_error_report_disabled

```
function bit get_response_queue_error_report_disabled()
```

When this bit is 0 (default value), error reports are generated when the response queue

overflows. When this bit is 1, no such error reports are generated.

set_response_queue_depth

```
function void set_response_queue_depth(int value)
```

The default maximum depth of the response queue is 8. These method is used to examine or change the maximum depth of the response queue.

Setting the response_queue_depth to -1 indicates an arbitrarily deep response queue. No checking is done.

get_response_queue_depth

```
function int get_response_queue_depth()
```

Returns the current depth setting for the response queue.

clear_response_queue

```
virtual function void clear_response_queue()
```

Empties the response queue for this sequence.

uvm_sequence #(REQ,RSP)

The uvm_sequence class provides the interfaces necessary in order to create streams of sequence items and/or other sequences.

Summary

uvm_sequence #(REQ,RSP)

The uvm_sequence class provides the interfaces necessary in order to create streams of sequence items and/or other sequences.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_transaction
- uvm_sequence_item
- uvm_sequence_base
- **uvm_sequence#(REQ,RSP)**

Class Declaration

```
virtual class uvm_sequence #(
   type REQ = uvm_sequence_item,
   type RSP = REQ
) extends uvm_sequence_base
```

Methods

new	Creates and initializes a new sequence object.
send_request	This method will send the request item to the sequencer, which will forward it to the driver.
get_current_item	Returns the request item currently being executed by the sequencer.
get_response	By default, sequences must retrieve responses by calling get_response.

Methods

new

```
function new (string name = "uvm_sequence")
```

Creates and initializes a new sequence object.

send_request

```
function void send_request(uvm_sequence_item request,
                           bit               rerandomize = 0)
```

This method will send the request item to the sequencer, which will forward it to the driver. If the rerandomize bit is set, the item will be randomized before being sent to

the driver. The send_request function may only be called after uvm_sequence_base::wait_for_grant returns.

get_current_item

```
function REQ get_current_item()
```

Returns the request item currently being executed by the sequencer. If the sequencer is not currently executing an item, this method will return null.

The sequencer is executing an item from the time that get_next_item or peek is called until the time that get or item_done is called.

Note that a driver that only calls get will never show a current item, since the item is completed at the same time as it is requested.

get_response

```
virtual task get_response(output RSP response,
                          input int transaction_id = -1)
```

By default, sequences must retrieve responses by calling get_response. If no transaction_id is specified, this task will return the next response sent to this sequence. If no response is available in the response queue, the method will block until a response is recieved.

If a transaction_id is parameter is specified, the task will block until a response with that transaction_id is received in the response queue.

The default size of the response queue is 8. The get_response method must be called soon enough to avoid an overflow of the response queue to prevent responses from being dropped.

If a response is dropped in the response queue, an error will be reported unless the error reporting is disabled via set_response_queue_error_report_disabled.

Synchronization Classes

The UVM provides event and barrier synchronization classes for managing concurrent processes.

- uvm_event - UVM's event class augments the SystemVerilog event datatype with such services as setting callbacks and data delivery.
- uvm_barrier - A barrier is used to prevent a pre-configured number of processes from continuing until all have reached a certain point in simulation.
- uvm_event_pool and uvm_barrier_pool - The event and barrier pool classes are specializations of uvm_object_string_pool #(T) used to store collections of uvm_events and uvm_barriers, respectively, indexed by string name. Each pool class contains a static, "global" pool instance for sharing across all processes.
- uvm_event_callback - The event callback is used to create callback objects that may be attached to uvm_events.

Summary

Synchronization Classes

uvm_event

The uvm_event class is a wrapper class around the SystemVerilog event construct. It provides some additional services such as setting callbacks and maintaining the number of waiters.

Summary

uvm_event

The uvm_event class is a wrapper class around the SystemVerilog event construct.

Class Hierarchy

uvm_void

uvm_object

uvm_event

Class Declaration

```
class uvm_event extends uvm_object
```

Methods

new	Creates a new event object.
wait_on	Waits for the event to be activated for the first time.
wait_off	If the event has already triggered and is "on", this task waits for the event to be turned "off" via a call to reset.
wait_trigger	Waits for the event to be triggered.
wait_ptrigger	Waits for a persistent trigger of the event.
wait_trigger_data	This method calls wait_trigger followed by get_trigger_data.
wait_ptrigger_data	This method calls wait_ptrigger followed by get_trigger_data.
trigger	Triggers the event, resuming all waiting processes.
get_trigger_data	Gets the data, if any, provided by the last call to trigger.
get_trigger_time	Gets the time that this event was last triggered.
is_on	Indicates whether the event has been triggered since it was last reset.
is_off	Indicates whether the event has been triggered or been reset.
reset	Resets the event to its off state.
add_callback	Registers a callback object, *cb*, with this event.
delete_callback	Unregisters the given callback, *cb*, from this event.
cancel	Decrements the number of waiters on the event.
get_num_waiters	Returns the number of processes waiting on the event.

Methods

new

```
function new (string name = "")
```

Creates a new event object.

wait_on

```
virtual task wait_on (bit delta = )
```

Waits for the event to be activated for the first time.

If the event has already been triggered, this task returns immediately. If *delta* is set, the caller will be forced to wait a single delta #0 before returning. This prevents the caller from returning before previously waiting processes have had a chance to resume.

Once an event has been triggered, it will be remain "on" until the event is reset.

wait_off

```
virtual task wait_off (bit delta = )
```

If the event has already triggered and is "on", this task waits for the event to be turned "off" via a call to reset.

If the event has not already been triggered, this task returns immediately. If *delta* is set, the caller will be forced to wait a single delta #0 before returning. This prevents the caller from returning before previously waiting processes have had a chance to resume.

wait_trigger

```
virtual task wait_trigger ()
```

Waits for the event to be triggered.

If one process calls wait_trigger in the same delta as another process calls trigger, a race condition occurs. If the call to wait occurs before the trigger, this method will return in this delta. If the wait occurs after the trigger, this method will not return until the next trigger, which may never occur and thus cause deadlock.

wait_ptrigger

```
virtual task wait_ptrigger ()
```

Waits for a persistent trigger of the event. Unlike wait_trigger, this views the trigger as persistent within a given time-slice and thus avoids certain race conditions. If this method is called after the trigger but within the same time-slice, the caller returns immediately.

wait_trigger_data

```
virtual task wait_trigger_data (output uvm_object data)
```

This method calls wait_trigger followed by get_trigger_data.

wait_ptrigger_data

```
virtual task wait_ptrigger_data (output uvm_object data)
```

This method calls wait_ptrigger followed by get_trigger_data.

trigger

```
virtual function void trigger (uvm_object data = null)
```

Triggers the event, resuming all waiting processes.

An optional *data* argument can be supplied with the enable to provide trigger-specific information.

get_trigger_data

```
virtual function uvm_object get_trigger_data ()
```

Gets the data, if any, provided by the last call to trigger.

get_trigger_time

```
virtual function time get_trigger_time ()
```

Gets the time that this event was last triggered. If the event has not been triggered, or the event has been reset, then the trigger time will be 0.

is_on

```
virtual function bit is_on ()
```

Indicates whether the event has been triggered since it was last reset.

A return of 1 indicates that the event has triggered.

is_off

```
virtual function bit is_off ()
```

Indicates whether the event has been triggered or been reset.

A return of 1 indicates that the event has not been triggered.

reset

```
virtual function void reset (bit wakeup = )
```

Resets the event to its off state. If *wakeup* is set, then all processes currently waiting for the event are activated before the reset.

No callbacks are called during a reset.

add_callback

```
virtual function void add_callback (uvm_event_callback cb,
                                    bit                  append = 1)
```

Registers a callback object, *cb*, with this event. The callback object may include pre_trigger and post_trigger functionality. If *append* is set to 1, the default, *cb* is added to the back of the callback list. Otherwise, *cb* is placed at the front of the callback list.

delete_callback

```
virtual function void delete_callback (uvm_event_callback cb)
```

Unregisters the given callback, *cb*, from this event.

cancel

```
virtual function void cancel ()
```

Decrements the number of waiters on the event.

This is used if a process that is waiting on an event is disabled or activated by some other means.

get_num_waiters

```
virtual function int get_num_waiters ()
```

Returns the number of processes waiting on the event.

uvm_event_callback

The uvm_event_callback class is an abstract class that is used to create callback objects which may be attached to uvm_events. To use, you derive a new class and override any or both pre_trigger and post_trigger.

Callbacks are an alternative to using processes that wait on events. When a callback is attached to an event, that callback object's callback function is called each time the event is triggered.

Summary

uvm_event_callback

The uvm_event_callback class is an abstract class that is used to create callback objects which may be attached to uvm_events.

Class Hierarchy

uvm_void
uvm_object
uvm_event_callback

Class Declaration

```
virtual class uvm_event_callback extends uvm_object
```

Methods

new	Creates a new callback object.
pre_trigger	This callback is called just before triggering the associated event.
post_trigger	This callback is called after triggering the associated event.

Methods

new

```
function new (string name = "")
```

Creates a new callback object.

pre_trigger

```
virtual function bit pre_trigger (uvm_event  e,
                                  uvm_object data = null)
```

This callback is called just before triggering the associated event. In a derived class, override this method to implement any pre-trigger functionality.

If your callback returns 1, then the event will not trigger and the post-trigger callback is not called. This provides a way for a callback to prevent the event from triggering.

In the function, *e* is the uvm_event that is being triggered, and *data* is the optional data

associated with the event trigger.

post_trigger

```
virtual function void post_trigger (uvm_event  e,
                                    uvm_object data = null)
```

This callback is called after triggering the associated event. In a derived class, override this method to implement any post-trigger functionality.

In the function, *e* is the uvm_event that is being triggered, and *data* is the optional data associated with the event trigger.

uvm_barrier

The uvm_barrier class provides a multiprocess synchronization mechanism. It enables a set of processes to block until the desired number of processes get to the synchronization point, at which time all of the processes are released.

Summary

uvm_barrier

The uvm_barrier class provides a multiprocess synchronization mechanism.

Class Hierarchy

uvm_void
uvm_object
uvm_barrier

Class Declaration

```
class uvm_barrier extends uvm_object
```

Methods

new	Creates a new barrier object.
wait_for	Waits for enough processes to reach the barrier before continuing.
reset	Resets the barrier.
set_auto_reset	Determines if the barrier should reset itself after the threshold is reached.
set_threshold	Sets the process threshold.
get_threshold	Gets the current threshold setting for the barrier.
get_num_waiters	Returns the number of processes currently waiting at the barrier.
cancel	Decrements the waiter count by one.

Methods

new

```
function new (string name      = "",
              int    threshold = 0  )
```

Creates a new barrier object.

wait_for

```
virtual task wait_for()
```

Waits for enough processes to reach the barrier before continuing.

The number of processes to wait for is set by the set_threshold method.

reset

```
virtual function void reset (bit wakeup = 1)
```

Resets the barrier. This sets the waiter count back to zero.

The threshold is unchanged. After reset, the barrier will force processes to wait for the threshold again.

If the *wakeup* bit is set, any currently waiting processes will be activated.

set_auto_reset

```
virtual function void set_auto_reset (bit value = 1)
```

Determines if the barrier should reset itself after the threshold is reached.

The default is on, so when a barrier hits its threshold it will reset, and new processes will block until the threshold is reached again.

If auto reset is off, then once the threshold is achieved, new processes pass through without being blocked until the barrier is reset.

set_threshold

```
virtual function void set_threshold (int threshold)
```

Sets the process threshold.

This determines how many processes must be waiting on the barrier before the processes may proceed.

Once the *threshold* is reached, all waiting processes are activated.

If *threshold* is set to a value less than the number of currently waiting processes, then the barrier is reset and waiting processes are activated.

get_threshold

```
virtual function int get_threshold ()
```

Gets the current threshold setting for the barrier.

get_num_waiters

```
virtual function int get_num_waiters ()
```

Returns the number of processes currently waiting at the barrier.

cancel

```
virtual function void cancel ()
```

Decrements the waiter count by one. This is used when a process that is waiting on the

barrier is killed or activated by some other means.

Objection Mechanism

The following classes define the objection mechanism and end-of-test functionality, which is based on uvm_objection.

Contents

uvm_objection

Objections provide a facility for coordinating status information between two or more participating components, objects, and even module-based IP. In particular, the *uvm_test_done* built-in objection provides a means for coordinating when to end a test, i.e. when to call global_stop_request to end the <uvm_component::run> phase. When all participating components have dropped their raised objections with *uvm_test_done*, an implicit call to *global_stop_request* is issued.

Tracing of objection activity can be turned on to follow the activity of the objection mechanism. It may be turned on for a specific objection instance with uvm_objection::trace_mode, or it can be set for all objections from the command line using the option +UVM_OBJECTION_TRACE.

Summary

uvm_objection

Objections provide a facility for coordinating status information between two or more participating components, objects, and even module-based IP.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- **uvm_objection**

Class Declaration

```
class uvm_objection extends uvm_report_object
```

Objection Control	
m_set_hier_mode	Hierarchical mode only needs to be set for intermediate components, not for uvm_root or a leaf component.
raise_objection	Raises the number of objections for the source *object* by *count*, which defaults to 1.
drop_objection	Drops the number of objections for the source *object* by *count*, which defaults to 1.
set_drain_time	Sets the drain time on the given *object* to *drain*.
Callback Hooks	
raised	Objection callback that is called when a raise_objection has reached *obj*.
dropped	Objection callback that is called when a drop_objection has reached *obj*.
all_dropped	Objection callback that is called when a drop_objection has reached *obj*, and the total count for *obj* goes to zero.
Objection Status	
get_objectors	Returns the current list of objecting objects (objects that raised an objection but have not dropped it).
wait_for	Waits for the raised, dropped, or all_dropped *event* to occur in the given *obj*.
get_objection_count	Returns the current number of objections raised by the given *object*.
get_objection_total	Returns the current number of objections raised by the given *object* and all descendants.
get_drain_time	Returns the current drain time set for the given *object* (default: 0 ns).
display_objections	Displays objection information about the given *object*.

new

```
function new(string name = "")
```

Creates a new objection instance. Accesses the command line argument +UVM_OBJECTION_TRACE to turn tracing on for all objection objects.

trace_mode

```
function bit trace_mode (int mode = -1)
```

Set or get the trace mode for the objection object. If no argument is specified (or an argument other than 0 or 1) the current trace mode is unaffected. A trace_mode of 0 turns tracing off. A trace mode of 1 turns tracing on. The return value is the mode prior to being reset.

Objection Control

m_set_hier_mode

```
function void m_set_hier_mode (uvm_object obj)
```

Hierarchical mode only needs to be set for intermediate components, not for uvm_root or a leaf component.

raise_objection

```
virtual function void raise_objection (uvm_object obj         = null,
                                       string     description = "",
                                       int        count       = 1    )
```

Raises the number of objections for the source *object* by *count*, which defaults to 1. The *object* is usually the *this* handle of the caller. If *object* is not specified or null, the implicit top-level component, *uvm_top*, is chosen.

Rasing an objection causes the following.

- The source and total objection counts for *object* are increased by *count*. *description* is a string that marks a specific objection and is used in tracing/debug.
- The objection's raised virtual method is called, which calls the uvm_component::raised method for all of the components up the hierarchy.

drop_objection

```
virtual function void drop_objection (uvm_object obj         = null,
                                      string     description = "",
                                      int        count       = 1    )
```

Drops the number of objections for the source *object* by *count*, which defaults to 1. The *object* is usually the *this* handle of the caller. If *object* is not specified or null, the implicit top-level component, *uvm_top*, is chosen.

Dropping an objection causes the following.

- The source and total objection counts for *object* are decreased by *count*. It is an error to drop the objection count for *object* below zero.
- The objection's dropped virtual method is called, which calls the uvm_component::dropped method for all of the components up the hierarchy.
- If the total objection count has not reached zero for *object*, then the drop is propagated up the object hierarchy as with raise_objection. Then, each object in the hierarchy will have updated their *source* counts--objections that they originated--and *total* counts--the total number of objections by them and all their descendants.

If the total objection count reaches zero, propagation up the hierarchy is deferred until a configurable drain-time has passed and the uvm_component::all_dropped callback for the current hierarchy level has returned. The following process occurs for each instance up the hierarchy from the source caller:

A process is forked in a non-blocking fashion, allowing the *drop* call to return. The forked process then does the following:

- If a drain time was set for the given *object*, the process waits for that amount of time.
- The objection's all_dropped virtual method is called, which calls the uvm_component::all_dropped method (if *object* is a component).
- The process then waits for the *all_dropped* callback to complete.
- After the drain time has elapsed and all_dropped callback has completed, propagation of the dropped objection to the parent proceeds as described in raise_objection, except as described below.

If a new objection for this *object* or any of its descendents is raised during the drain time or during execution of the all_dropped callback at any point, the hierarchical chain described above is terminated and the dropped callback does not go up the hierarchy.

The raised objection will propagate up the hierarchy, but the number of raised propagated up is reduced by the number of drops that were pending waiting for the all_dropped/drain time completion. Thus, if exactly one objection caused the count to go to zero, and during the drain exactly one new objection comes in, no raises or drops are propagted up the hierarchy,

As an optimization, if the *object* has no set drain-time and no registered callbacks, the forked process can be skipped and propagation proceeds immediately to the parent as described.

set_drain_time

Sets the drain time on the given *object* to *drain*.

The drain time is the amount of time to wait once all objections have been dropped before calling the all_dropped callback and propagating the objection to the parent.

If a new objection for this *object* or any of its descendents is raised during the drain time or during execution of the all_dropped callbacks, the drain_time/all_dropped execution is terminated.

Callback Hooks

raised

```
virtual function void raised (uvm_object obj,
                              uvm_object source_obj,
                              string     description,
                              int        count       )
```

Objection callback that is called when a raise_objection has reached *obj*. The default implementation calls uvm_component::raised.

dropped

```
virtual function void dropped (uvm_object obj,
                               uvm_object source_obj,
                               string     description,
                               int        count       )
```

Objection callback that is called when a drop_objection has reached *obj*. The default implementation calls uvm_component::dropped.

all_dropped

```
virtual task all_dropped (uvm_object obj,
                          uvm_object source_obj,
                          string     description,
                          int        count       )
```

Objection callback that is called when a drop_objection has reached *obj*, and the total count for *obj* goes to zero. This callback is executed after the drain time associated with *obj*. The default implementation calls uvm_component::all_dropped.

get_objectors

```
function void get_objectors(ref uvm_object list[$])
```

Returns the current list of objecting objects (objects that raised an objection but have not dropped it).

wait_for

```
task wait_for(uvm_objection_event objt_event,
              uvm_object          obj         = null)
```

Waits for the raised, dropped, or all_dropped *event* to occur in the given *obj*. The task returns after all corresponding callbacks have been executed.

get_objection_count

```
function int get_objection_count (uvm_object obj = null)
```

Returns the current number of objections raised by the given *object*.

get_objection_total

```
function int get_objection_total (uvm_object obj = null)
```

Returns the current number of objections raised by the given *object* and all descendants.

get_drain_time

```
function time get_drain_time (uvm_object obj = null)
```

Returns the current drain time set for the given *object* (default: 0 ns).

display_objections

```
function void display_objections(uvm_object obj         = null,
                                 bit         show_header = 1    )
```

Displays objection information about the given *object*. If *object* is not specified or *null*, the implicit top-level component, uvm_root, is chosen. The *show_header* argument allows control of whether a header is output.

uvm_test_done_objection

Provides built-in end-of-test coordination

Summary

uvm_test_done_objection

Provides built-in end-of-test coordination

Class Hierarchy

m_uvm_test_done_objection_base

uvm_test_done_objection

Class Declaration

```
class uvm_test_done_objection extends
m_uvm_test_done_objection_base
```

Methods

new	Creates the singleton test_done objection.
qualify	Checks that the given *object* is derived from either uvm_component or uvm_sequence_base.
stop_request	Calling this function triggers the process of shutting down the currently running task-based phase.

Variables

stop_timeout	These set watchdog timers for task-based phases and stop tasks.

Methods

all_dropped	This callback is called when the given *object's* objection count reaches zero; if the *object* is the implicit top-level, uvm_root then it means there are no more objections raised for the *uvm_test_done* objection.
raise_objection	Calls uvm_objection::raise_objection after calling qualify.
drop_objection	Calls uvm_objection::drop_objection after calling qualify.
force_stop	Forces the propagation of the all_dropped() callback, even if there are still outstanding objections.

Methods

new

Creates the singleton test_done objection. Users must not to call this method directly.

qualify

```
virtual function void qualify(uvm_object obj         = null,
                              bit        is_raise,
                              string     description        )
```

Checks that the given *object* is derived from either uvm_component or uvm_sequence_base.

stop_request

```
function void stop_request()
```

Calling this function triggers the process of shutting down the currently running task-based phase. This process involves calling all components' stop tasks for those components whose enable_stop_interrupt bit is set. Once all stop tasks return, or once the optional global_stop_timeout expires, all components' kill method is called, effectively ending the current phase. The uvm_top will then begin execution of the next phase, if any.

VARIABLES

stop_timeout

```
time stop_timeout = 0
```

These set watchdog timers for task-based phases and stop tasks. You can not disable the timeouts. When set to 0, a timeout of the maximum time possible is applied. A timeout at this value usually indicates a problem with your testbench. You should lower the timeout to prevent "never-ending" simulations.

METHODS

all_dropped

```
virtual task all_dropped (uvm_object obj,
                          uvm_object source_obj,
                          string     description,
                          int        count       )
```

This callback is called when the given *object's* objection count reaches zero; if the *object* is the implicit top-level, uvm_root then it means there are no more objections raised for the *uvm_test_done* objection. Thus, after calling uvm_objection::all_dropped, this method will call global_stop_request to stop the current task-based phase (e.g. run).

raise_objection

```
virtual function void raise_objection (uvm_object obj         = null,
                                       string     description = "",
                                       int        count       = 1    )
```

Calls uvm_objection::raise_objection after calling qualify. If the *object* is not provided or is *null*, then the implicit top-level component, *uvm_top*, is chosen.

drop_objection

```
virtual function void drop_objection (uvm_object obj         = null,
                                      string     description = "",
                                      int        count       = 1    )
```

Calls uvm_objection::drop_objection after calling qualify. If the *object* is not provided or is *null*, then the implicit top-level component, *uvm_top*, is chosen.

force_stop

```
virtual task force_stop(uvm_object obj = null)
```

Forces the propagation of the all_dropped() callback, even if there are still outstanding objections. The net effect of this action is to forcibly end the current phase.

uvm_callbacks_objection

The uvm_callbacks_objection is a specialized uvm_objection which contains callbacks for the raised and dropped events. Callbacks happend for the three standard callback activities, raised, dropped, and all_dropped.

The uvm_heartbeat mechanism use objections of this type for creating heartbeat conditions. Whenever the objection is raised or dropped, the component which did the raise/drop is considered to be alive.

Summary

uvm_callbacks_objection

The uvm_callbacks_objection is a specialized uvm_objection which contains callbacks for the raised and dropped events.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_objection
- **uvm_callbacks_objection**

Class Declaration

```
class uvm_callbacks_objection extends uvm_objection
```

Methods

raised	Executes the uvm_objection_callback::raised method in the user callback class whenever this objection is raised at the object *obj*.
dropped	Executes the uvm_objection_callback::dropped method in the user callback class whenever this objection is dropped at the object *obj*.
all_dropped	Executes the uvm_objection_callback::all_dropped task in the user callback class whenever the objection count for this objection in reference to *obj* goes to zero.

Methods

raised

```
virtual function void raised (uvm_object obj,
```

```
                                uvm_object source_obj,
                                string     description,
                                int        count       )
```

Executes the uvm_objection_callback::raised method in the user callback class whenever this objection is raised at the object *obj*.

dropped

```
virtual function void dropped (uvm_object obj,
                               uvm_object source_obj,
                               string     description,
                               int        count       )
```

Executes the uvm_objection_callback::dropped method in the user callback class whenever this objection is dropped at the object *obj*.

all_dropped

```
virtual task all_dropped (uvm_object obj,
                          uvm_object source_obj,
                          string     description,
                          int        count       )
```

Executes the uvm_objection_callback::all_dropped task in the user callback class whenever the objection count for this objection in reference to *obj* goes to zero.

uvm_objection_callback

The uvm_objection is the callback type that defines the callback implementations for an objection callback. A user uses the callback type uvm_objection_cbs_t to add callbacks to specific objections.

For example

```
class my_objection_cb extends uvm_objection_callback;
  function new(string name);
    super.new(name);
  endfunction

  virtual function void raised (uvm_objection objection, uvm_object obj,
      uvm_object source_obj, string description, int count);
    $display("%0t: Objection %s: Raised for %s", $time,
objection.get_name(),
        obj.get_full_name());
  endfunction
endclass
...
initial begin
  my_objection_cb cb = new("cb");
  uvm_objection_cbs_t::add(null, cb); //typewide callback
end
```

Summary

uvm_objection_callback

The uvm_objection is the callback type that defines the callback implementations

for an objection callback.

Class Hierarchy

uvm_void

uvm_object

uvm_callback

uvm_objection_callback

Class Declaration

```
class uvm_objection_callback extends uvm_callback
```

Methods

raised	Objection raised callback function.
dropped	Objection dropped callback function.
all_dropped	Objection all_dropped callback function.

Methods

raised

```
virtual function void raised (uvm_objection objection,
                              uvm_object    obj,
                              uvm_object    source_obj,
                              string        description,
                              int           count       )
```

Objection raised callback function. Called by uvm_callbacks_objection::raised.

dropped

```
virtual function void dropped (uvm_objection objection,
                               uvm_object    obj,
                               uvm_object    source_obj,
                               string        description,
                               int           count       )
```

Objection dropped callback function. Called by uvm_callbacks_objection::dropped.

all_dropped

```
virtual task all_dropped (uvm_objection objection,
                          uvm_object    obj,
                          uvm_object    source_obj,
                          string        description,
                          int           count       )
```

Objection all_dropped callback function. Called by uvm_callbacks_objection::all_dropped.

uvm_heartbeat

Heartbeats provide a way for environments to easily ensure that their descendants are alive. A uvm_heartbeat is associated with a specific objection object. A component that is being tracked by the heartbeat object must raise (or drop) the synchronizing objection during the heartbeat window. The synchronizing objection must be a uvm_callbacks_objection type.

The uvm_heartbeat object has a list of participating objects. The heartbeat can be configured so that all components (UVM_ALL_ACTIVE), exactly one (UVM_ONE_ACTIVE), or any component (UVM_ANY_ACTIVE) must trigger the objection in order to satisfy the heartbeat condition.

Summary

uvm_heartbeat

Heartbeats provide a way for environments to easily ensure that their descendants are alive.

METHODS

new	Creates a new heartbeat instance associated with *cntxt*.
set_mode	Sets or retrieves the heartbeat mode.
set_heartbeat	Sets up the heartbeat event and assigns a list of objects to watch.
add	Add a single component to the set of components to be monitored.
remove	Remove a single component to the set of components being monitored.
start	Starts the heartbeat monitor.
stop	Stops the heartbeat monitor.

METHODS

new

```
function new(string                   name,
             uvm_component            cntxt,
             uvm_callbacks_objection  objection = null)
```

Creates a new heartbeat instance associated with *cntxt*. The context is the hierarchical location that the heartbeat objections will flow through and be monitored at. The *objection* associated with the heartbeat is optional, if it is left null but it must be set before the heartbeat monitor will activate.

```
uvm_callbacks_objection myobjection = new("myobjection"); //some shared
objection
class myenv extends uvm_env;
   uvm_heartbeat hb = new("hb", this, myobjection);
   ...
endclass
```

set_mode

```
function uvm_heartbeat_modes set_mode (
   uvm_heartbeat_modes mode = UVM_NO_HB_MODE
)
```

Sets or retrieves the heartbeat mode. The current value for the heartbeat mode is returned. If an argument is specified to change the mode then the mode is changed to the new value.

set_heartbeat

```
function void set_heartbeat (     uvm_event     e,
                             ref uvm_component comps[$])
```

Sets up the heartbeat event and assigns a list of objects to watch. The monitoring is started as soon as this method is called. Once the monitoring has been started with a specific event, providing a new monitor event results in an error. To change trigger events, you must first stop the monitor and then start with a new event trigger.

If the trigger event *e* is null and there was no previously set trigger event, then the monitoring is not started. Monitoring can be started by explicitly calling start.

add

```
function void add (uvm_component comp)
```

Add a single component to the set of components to be monitored. This does not cause monitoring to be started. If monitoring is currently active then this component will be immediately added to the list of components and will be expected to participate in the currently active event window.

remove

```
function void remove (uvm_component comp)
```

Remove a single component to the set of components being monitored. Monitoring is not stopped, even if the last component has been removed (an explicit stop is required).

start

```
function void start (uvm_event e = null)
```

Starts the heartbeat monitor. If *e* is null then whatever event was previously set is used. If no event was previously set then a warning is issued. It is an error if the monitor is currently running and *e* is specifying a different trigger event from the current event.

stop

```
function void stop ()
```

Stops the heartbeat monitor. Current state information is reset so that if start is called

again the process will wait for the first event trigger to start the monitoring.

Container Classes

The container classes are type parameterized datastructures. The uvm_queue #(T) class implements a queue datastructure similar to the SystemVerilog queue construct. And the uvm_pool #(KEY,T) class implements a pool datastructure similar to the SystemVerilog associative array. The class based datastructures allow the objects to be shared by reference; for example, a copy of a uvm_pool #(KEY,T) object will copy just the class handle instead of the entire associative array.

Summary

Container Classes

The container classes are type parameterized datastructures.

Pool Classes

This section defines the <uvm_pool #(T)> class and derivative.

Contents

uvm_pool #(KEY,T)

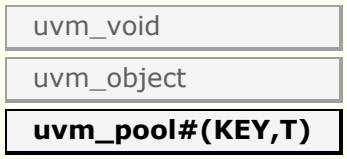
Implements a class-based dynamic associative array. Allows sparse arrays to be allocated on demand, and passed and stored by reference.

Summary

uvm_pool #(KEY,T)

Implements a class-based dynamic associative array.

Class Hierarchy



Class Declaration

```
class uvm_pool #(type KEY = int,
                       T   = uvm_void) extends uvm_object
```

Methods

new	Creates a new pool with the given *name*.
get_global_pool	Returns the singleton global pool for the item type, T.
get_global	Returns the specified item instance from the global item pool.
get	Returns the item with the given *key*.
add	Adds the given (*key*, *item*) pair to the pool.
num	Returns the number of uniquely keyed items stored in the pool.
delete	Removes the item with the given *key* from the pool.
exists	Returns 1 if a item with the given *key* exists in the pool, 0 otherwise.
first	Returns the key of the first item stored in the pool.
last	Returns the key of the last item stored in the pool.
next	Returns the key of the next item in the pool.
prev	Returns the key of the previous item in the pool.

METHODS

new

```
function new (string name = "")
```

Creates a new pool with the given *name*.

get_global_pool

```
static function this_type get_global_pool ()
```

Returns the singleton global pool for the item type, T.

This allows items to be shared amongst components throughout the verification environment.

get_global

```
static function T get_global (KEY key)
```

Returns the specified item instance from the global item pool.

get

```
virtual function T get (KEY key)
```

Returns the item with the given *key*.

If no item exists by that key, a new item is created with that key and returned.

add

```
virtual function void add (KEY key,
                           T   item)
```

Adds the given (*key*, *item*) pair to the pool. If an item already exists at the given *key* it is overwritten with the new *item*.

num

```
virtual function int num ()
```

Returns the number of uniquely keyed items stored in the pool.

delete

```
virtual function void delete (KEY key)
```

Removes the item with the given *key* from the pool.

exists

```
virtual function int exists (KEY key)
```

Returns 1 if a item with the given *key* exists in the pool, 0 otherwise.

first

```
virtual function int first (ref KEY key)
```

Returns the key of the first item stored in the pool.

If the pool is empty, then *key* is unchanged and 0 is returned.

If the pool is not empty, then *key* is key of the first item and 1 is returned.

last

```
virtual function int last (ref KEY key)
```

Returns the key of the last item stored in the pool.

If the pool is empty, then 0 is returned and *key* is unchanged.

If the pool is not empty, then *key* is set to the last key in the pool and 1 is returned.

next

```
virtual function int next (ref KEY key)
```

Returns the key of the next item in the pool.

If the input *key* is the last key in the pool, then *key* is left unchanged and 0 is returned.

If a next key is found, then *key* is updated with that key and 1 is returned.

prev

```
virtual function int prev (ref KEY key)
```

Returns the key of the previous item in the pool.

If the input *key* is the first key in the pool, then *key* is left unchanged and 0 is returned.

If a previous key is found, then *key* is updated with that key and 1 is returned.

uvm_object_string_pool #(T)

This provides a specialization of the generic uvm_pool #(KEY,T) class for an associative

array of uvm_object-based objects indexed by string. Specializations of this class include the *uvm_event_pool* (a uvm_object_string_pool storing uvm_events) and *uvm_barrier_pool* (a uvm_obejct_string_pool storing uvm_barriers).

Summary

uvm_object_string_pool #(T)

This provides a specialization of the generic uvm_pool #(KEY,T) class for an associative array of uvm_object-based objects indexed by string.

Class Hierarchy

uvm_pool#(string,T)

uvm_object_string_pool#(T)

Class Declaration

```
class uvm_object_string_pool #(
   type T = uvm_object
) extends uvm_pool #(string,T)
```

Methods

new	Creates a new pool with the given *name*.
get_type_name	Returns the type name of this object.
get_global_pool	Returns the singleton global pool for the item type, T.
get	Returns the object item at the given string *key*.
delete	Removes the item with the given string *key* from the pool.

Methods

new

```
function new (string name = "")
```

Creates a new pool with the given *name*.

get_type_name

```
virtual function string get_type_name()
```

Returns the type name of this object.

get_global_pool

```
static function this_type get_global_pool ()
```

Returns the singleton global pool for the item type, T.

This allows items to be shared amongst components throughout the verification environment.

get

```
virtual function T get (string key)
```

Returns the object item at the given string *key*.

If no item exists by the given *key*, a new item is created for that key and returned.

delete

```
virtual function void delete (string key)
```

Removes the item with the given string *key* from the pool.

uvm_queue #(T)

Implements a class-based dynamic queue. Allows queues to be allocated on demand, and passed and stored by reference.

Summary

uvm_queue #(T)

Implements a class-based dynamic queue.

Class Hierarchy

uvm_void

uvm_object

uvm_queue#(T)

Class Declaration

```
class uvm_queue #(type T = int) extends uvm_object
```

Methods

new	Creates a new queue with the given *name*.
get_global_queue	Returns the singleton global queue for the item type, T.
get_global	Returns the specified item instance from the global item queue.
get	Returns the item at the given *index*.
size	Returns the number of items stored in the queue.
insert	Inserts the item at the given *index* in the queue.
delete	Removes the item at the given *index* from the queue; if *index* is not provided, the entire contents of the queue are deleted.
pop_front	Returns the first element in the queue (index=0), or *null* if the queue is empty.
pop_back	Returns the last element in the queue (index=size()-1), or *null* if the queue is empty.
push_front	Inserts the given *item* at the front of the queue.
push_back	Inserts the given *item* at the back of the queue.

Methods

new

```
function new (string name = "")
```

Creates a new queue with the given *name*.

get_global_queue

```
static function this_type get_global_queue ()
```

Returns the singleton global queue for the item type, T.

This allows items to be shared amongst components throughout the verification

environment.

get_global

```
static function T get_global (int index)
```

Returns the specified item instance from the global item queue.

get

```
virtual function T get (int index)
```

Returns the item at the given *index*.

If no item exists by that key, a new item is created with that key and returned.

size

```
virtual function int size ()
```

Returns the number of items stored in the queue.

insert

```
virtual function void insert (int index,
                              T   item  )
```

Inserts the item at the given *index* in the queue.

delete

```
virtual function void delete (int index = -1)
```

Removes the item at the given *index* from the queue; if *index* is not provided, the entire contents of the queue are deleted.

pop_front

```
virtual function T pop_front()
```

Returns the first element in the queue (index=0), or *null* if the queue is empty.

pop_back

```
virtual function T pop_back()
```

Returns the last element in the queue (index=size()-1), or *null* if the queue is empty.

push_front

```
virtual function void push_front(T item)
```

Inserts the given *item* at the front of the queue.

push_back

```
virtual function void push_back(T item)
```

Inserts the given *item* at the back of the queue.

TLM Interfaces

The UVM TLM library defines several abstract, transaction-level interfaces and the ports and exports that facilitate their use. Each TLM interface consists of one or more methods used to transport data, typically whole transactions (objects) at a time. Component designs that use TLM ports and exports to communicate are inherently more reusable, interoperable, and modular.

The UVM TLM library specifies the required behavior (semantic) of each interface method. Classes (components) that implement a TLM interface must meet the specified semantic.

Summary

TLM Interfaces

The UVM TLM library defines several abstract, transaction-level interfaces and the ports and exports that facilitate their use.

TLM2	The TLM2 sockets provide blocking and nonblocking transaction-level interfaces with well-defined completion semantics.
TLM1	The TLM1 ports provide blocking and nonblocking pass-by-value transaction-level interfaces.
Sequencer Port	A push or pull port, with well-defined completion semantics.
Analysis	The *analysis* interface is used to perform non-blocking broadcasts of transactions to connected components.

TLM2

The TLM2 sockets provide blocking and nonblocking transaction-level interfaces with well-defined completion semantics.

TLM1

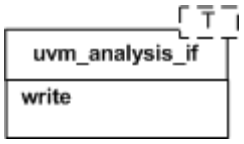
The TLM1 ports provide blocking and nonblocking pass-by-value transaction-level interfaces. The semantics of these interfaces are limited to message passing.

Sequencer Port

A push or pull port, with well-defined completion semantics. It is used to connect sequencers with drivers and layering sequences.

Analysis

The *analysis* interface is used to perform non-blocking broadcasts of transactions to connected components. It is typically used by such components as monitors to publish transactions observed on a bus to its subscribers, which are typically scoreboards and response/coverage collectors.


T
uvm_analysis_if
write

TLM1 Interfaces, Ports, Exports and Transport Interfaces

Each TLM1 interface is either blocking, non-blocking, or a combination of these two.

blocking	A blocking interface conveys transactions in blocking fashion; its methods do not return until the transaction has been successfully sent or retrieved. Because delivery may consume time to complete, the methods in such an interface are declared as tasks.
non-blocking	A non-blocking interface attempts to convey a transaction without consuming simulation time. Its methods are declared as functions. Because delivery may fail (e.g. the target component is busy and can not accept the request), the methods may return with failed status.
combination	A combination interface contains both the blocking and non-blocking variants. In SystemC, combination interfaces are defined through multiple inheritance. Because SystemVerilog does not support multiple inheritance, the UVM emulates hierarchical interfaces via a common base class and interface mask.

Like their SystemC counterparts, the UVM's TLM port and export implementations allow connections between ports whose interfaces are not an exact match. For example, an *uvm_blocking_get_port* can be connected to any port, export or imp port that provides *at the least* an implementation of the blocking_get interface, which includes the *uvm_get_** ports and exports, *uvm_blocking_get_peek_** ports and exports, and *uvm_get_peek_** ports and exports.

The sections below provide and overview of the unidirectional and bidirectional TLM interfaces, ports, and exports.

Summary

TLM1 Interfaces, Ports, Exports and Transport Interfaces

Each TLM1 interface is either blocking, non-blocking, or a combination of these two.

Unidirectional Interfaces & Ports	The unidirectional TLM interfaces consist of blocking, non-blocking, and combined blocking and non-blocking variants of the *put*, *get* and *peek* interfaces, plus a non-blocking *analysis* interface.
Put	The *put* interfaces are used to send, or *put*, transactions to other components.
Get and Peek	The *get* interfaces are used to retrieve transactions from other components.
Ports, Exports, and Imps	The UVM provides unidirectional ports, exports, and implementation ports for connecting your components via the TLM interfaces.
Bidirectional Interfaces & Ports	The bidirectional interfaces consist of blocking, non-blocking, and combined blocking and non-blocking variants of the *transport*, *master*, and *slave* interfaces.
Transport	The *transport* interface sends a request transaction and returns a response transaction in a single task call, thereby enforcing an in-order execution semantic.
Master and Slave	The primitive, unidirectional *put*, *get*, and *peek* interfaces are combined to form bidirectional master and slave interfaces.
Ports,	The UVM provides bidirectional ports, exports, and

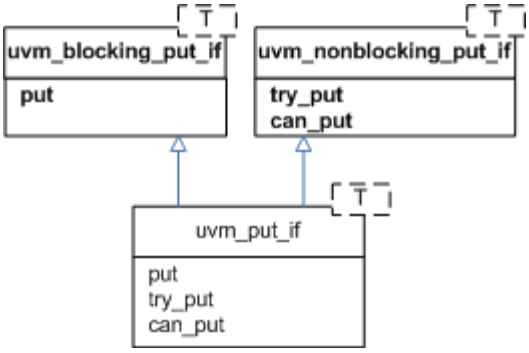
Exports, and Imps	implementation ports for connecting your components via the TLM interfaces.
USAGE	This example illustrates basic TLM connectivity using the blocking put inteface.

UNIDIRECTIONAL INTERFACES & PORTS

The unidirectional TLM interfaces consist of blocking, non-blocking, and combined blocking and non-blocking variants of the *put*, *get* and *peek* interfaces, plus a non-blocking *analysis* interface.

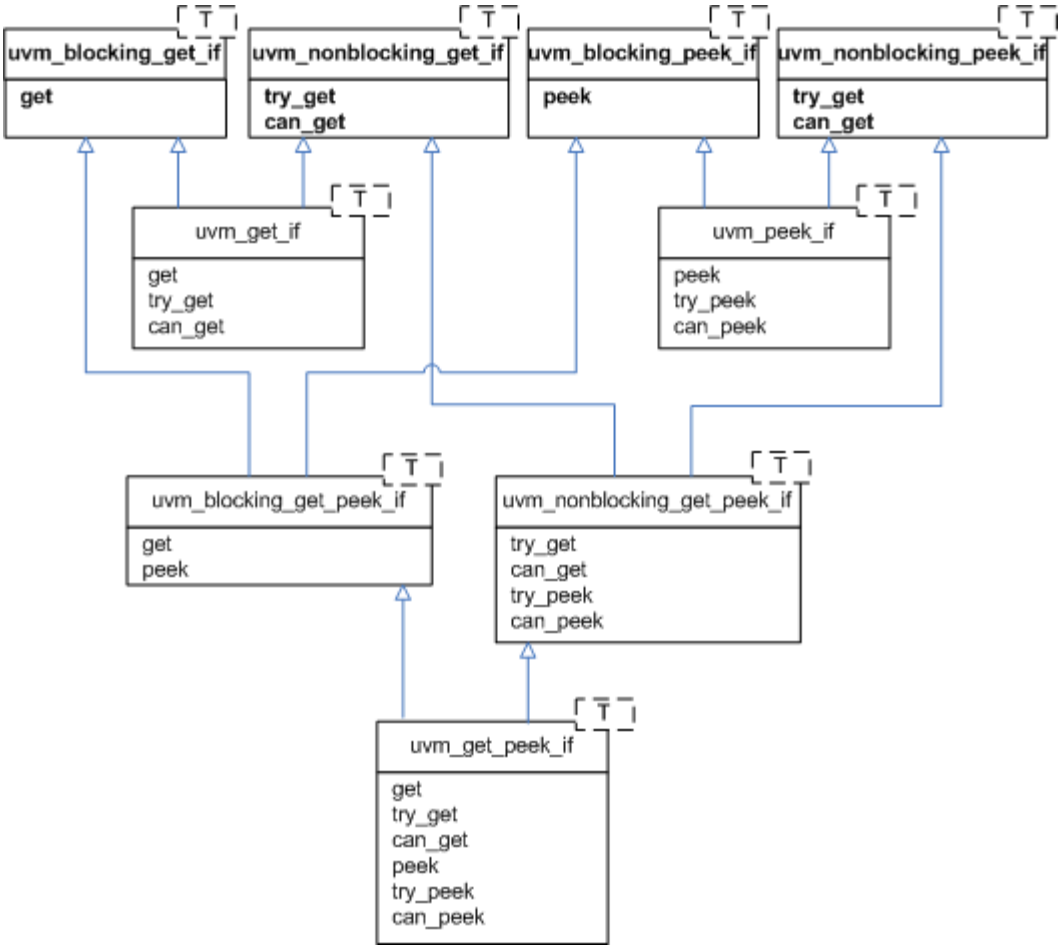
Put

The *put* interfaces are used to send, or *put*, transactions to other components. Successful completion of a put guarantees its delivery, not execution.



Get and Peek

The *get* interfaces are used to retrieve transactions from other components. The *peek* interfaces are used for the same purpose, except the retrieved transaction is not consumed; successive calls to *peek* will return the same object. Combined *get_peek* interfaces are also defined.



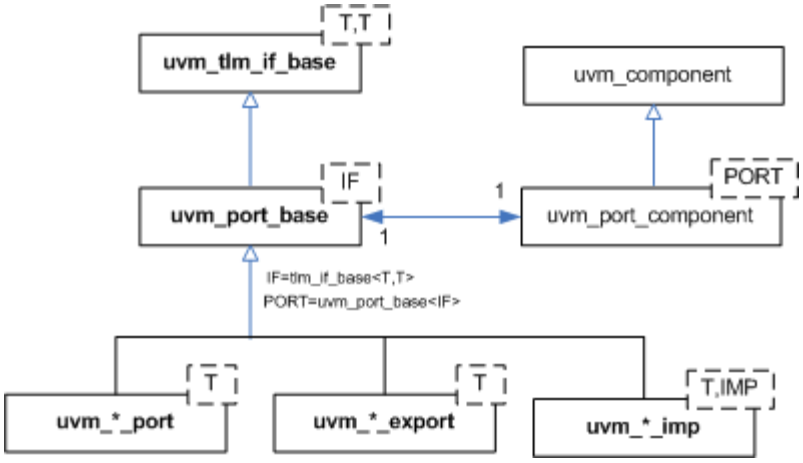
Ports, Exports, and Imps

The UVM provides unidirectional ports, exports, and implementation ports for connecting your components via the TLM interfaces.

Ports instantiated in components that *require*, or *use*, the associate interface to initiate transaction requests.

Exports instantiated by components that *forward* an implementation of the methods defined in the associated interface. The implementation is typically provided by an *imp* port in a child component.

Imps instantiated by components that *provide* or *implement* an implementation of the methods defined in the associated interface.



A summary of port, export, and imp declarations are

```
class uvm_*_export #(type T=int)
  extends uvm_port_base #(tlm_if_base #(T,T));

class uvm_*_port #(type T=int)
  extends uvm_port_base #(tlm_if_base #(T,T));

class uvm_*_imp #(type T=int)
  extends uvm_port_base #(tlm_if_base #(T,T));
```

where the asterisk can be any of

```
blocking_put
nonblocking_put
put

blocking_get
nonblocking_get
get

blocking_peek
nonblocking_peek
peek

blocking_get_peek
nonblocking_get_peek
get_peek

analysis
```

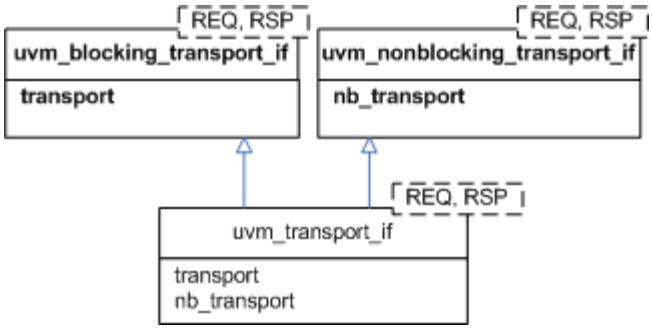
Bidirectional Interfaces & Ports

The bidirectional interfaces consist of blocking, non-blocking, and combined blocking and non-blocking variants of the *transport*, *master*, and *slave* interfaces.

Bidirectional interfaces involve both a transaction request and response.

Transport

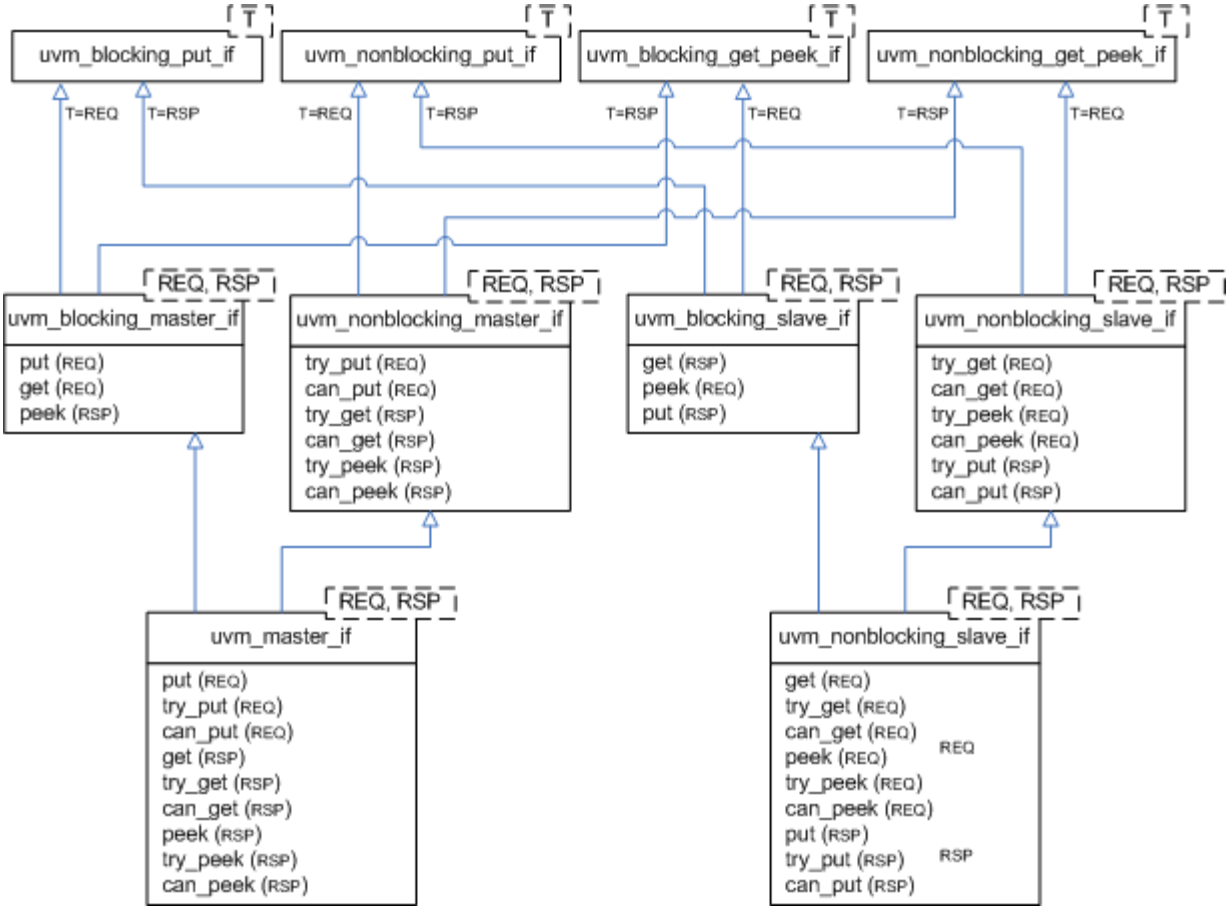
The *transport* interface sends a request transaction and returns a response transaction in a single task call, thereby enforcing an in-order execution semantic. The request and response transactions can be different types.



Master and Slave

The primitive, unidirectional *put*, *get*, and *peek* interfaces are combined to form bidirectional master and slave interfaces. The master puts requests and gets or peeks

responses. The slave gets or peeks requests and puts responses. Because the put and the get come from different function interface methods, the requests and responses are not coupled as they are with the *transport* interface.



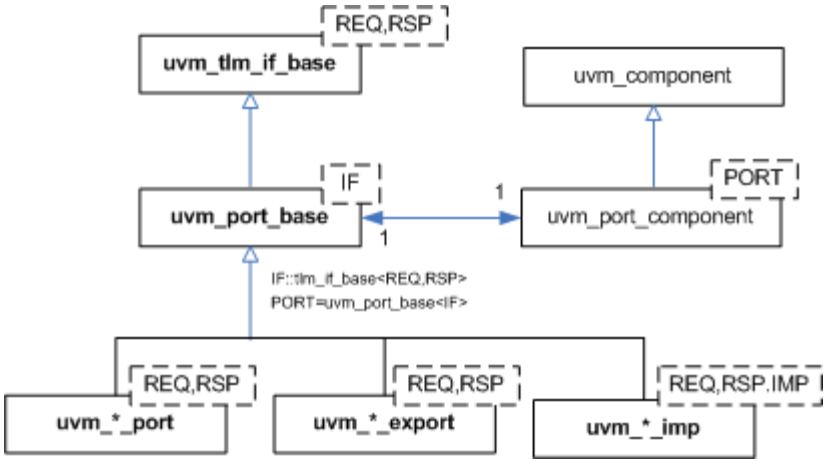
Ports, Exports, and Imps

The UVM provides bidirectional ports, exports, and implementation ports for connecting your components via the TLM interfaces.

Ports instantiated in components that *require*, or *use*, the associate interface to initiate transaction requests.

Exports instantiated by components that *forward* an implementation of the methods defined in the associated interface. The implementation is typically provided by an *imp* port in a child component.

Imps instantiated by components that *provide* or *implement* an implementation of the methods defined in the associated interface.



A summary of port, export, and imp declarations are

```
class uvm_*_port #(type REQ=int, RSP=int)
  extends uvm_port_base #(tlm_if_base #(REQ, RSP));

class uvm_*_export #(type REQ=int, RSP=int)
  extends uvm_port_base #(tlm_if_base #(REQ, RSP));

class uvm_*_imp #(type REQ=int, RSP=int)
  extends uvm_port_base #(tlm_if_base #(REQ, RSP));
```

where the asterisk can be any of

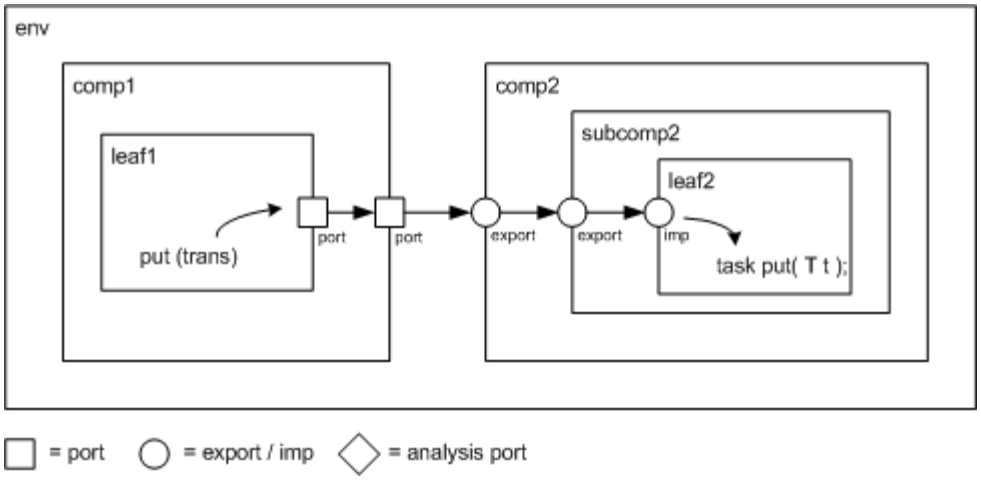
```
transport
blocking_transport
nonblocking_transport

blocking_master
nonblocking_master
master

blocking_slave
nonblocking_slave
slave
```

USAGE

This example illustrates basic TLM connectivity using the blocking put inteface.



port-to-port	leaf1's *out* port is connected to its parent's (comp1) *out* port
port-to-export	comp1's *out* port is connected to comp2's *in* export
export-to-export	comp2's *in* export is connected to its child's (subcomp2) *in* export
export-to-imp	subcomp2's *in* export is connected leaf2's *in* imp port.
imp-to-implementation	leaf2's *in* imp port is connected to its implementation, leaf2

Hierarchical port connections are resolved and optimized just before the <uvm_component::end_of_elaboration> phase. After optimization, calling any port's interface method (e.g. leaf1.out.put(trans)) incurs a single hop to get to the implementation (e.g. leaf2's put task), no matter how far up and down the hierarchy the implementation resides.

```
`include "uvm_pkg.sv"
import uvm_pkg::*;

class trans extends uvm_transaction;
  rand int addr;
  rand int data;
  rand bit write;
endclass

class leaf1 extends uvm_component;

  `uvm_component_utils(leaf1)

  uvm_blocking_put_port #(trans) out;

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

  virtual task run();
    trans t;
    t = new;
    t.randomize();
    out.put(t);
  endtask

endclass


class comp1 extends uvm_component;

  `uvm_component_utils(comp1)

  uvm_blocking_put_port #(trans) out;

  leaf1 leaf;

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

  virtual function void build();
    out = new("out",this);
    leaf = new("leaf1",this);
  endfunction

  // connect port to port
  virtual function void connect();
    leaf.out.connect(out);
  endfunction

endclass


class leaf2 extends uvm_component;

  `uvm_component_utils(leaf2)

  uvm_blocking_put_imp #(trans,leaf2) in;

  function new(string name, uvm_component parent=null);
    super.new(name,parent);
    // connect imp to implementation (this)
    in = new("in",this);
  endfunction
```

```
  virtual task put(trans t);
    $display("Got trans: addr=%0d, data=%0d, write=%0d",
         t.addr, t.data, t.write);
  endtask

endclass


class subcomp2 extends uvm_component;

  `uvm_component_utils(subcomp2)

  uvm_blocking_put_export #(trans) in;

  leaf2 leaf;

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

  virtual function void build();
    in = new("in",this);
    leaf = new("leaf2",this);
  endfunction

  // connect export to imp
  virtual function void connect();
    in.connect(leaf.in);
  endfunction

endclass


class comp2 extends uvm_component;

  `uvm_component_utils(comp2)

  uvm_blocking_put_export #(trans) in;

  subcomp2 subcomp;

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

  virtual function void build();
    in = new("in",this);
    subcomp = new("subcomp2",this);
  endfunction

  // connect export to export
  virtual function void connect();
    in.connect(subcomp.in);
  endfunction

endclass


class env extends uvm_component;

  `uvm_component_utils(comp1)

  comp1 comp1_i;
  comp2 comp2_i;

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

  virtual function void build();
    comp1_i = new("comp1",this);
    comp2_i = new("comp2",this);
  endfunction

  // connect port to export
  virtual function void connect();
    comp1_i.out.connect(comp2_i.in);
  endfunction

endclass


module top;
  env e = new("env");
  initial run_test();
  initial #10 uvm_top.stop_request();
endmodule
```

uvm_tlm_if_base #(T1,T2)

This class declares all of the methods of the TLM API.

Various subsets of these methods are combined to form primitive TLM interfaces, which are then paired in various ways to form more abstract "combination" TLM interfaces. Components that require a particular interface use ports to convey that requirement. Components that provide a particular interface use exports to convey its availability.

Communication between components is established by connecting ports to compatible exports, much like connecting module signal-level output ports to compatible input ports. The difference is that UVM ports and exports bind interfaces (groups of methods), not signals and wires. The methods of the interfaces so bound pass data as whole transactions (e.g. objects). The set of primitve and combination TLM interfaces afford many choices for designing components that communicate at the transaction level.

Summary

uvm_tlm_if_base #(T1,T2)

This class declares all of the methods of the TLM API.

Class Declaration

```
virtual class uvm_tlm_if_base #(type T1 = int,
                                type T2 = int )
```

Blocking put	
put	Sends a user-defined transaction of type T.
Blocking get	
get	Provides a new transaction of type T.
Blocking peek	
peek	Obtain a new transaction without consuming it.
Non-blocking put	
try_put	Sends a transaction of type T, if possible.
can_put	Returns 1 if the component is ready to accept the transaction; 0 otherwise.
Non-blocking get	
try_get	Provides a new transaction of type T.
can_get	Returns 1 if a new transaction can be provided immediately upon request, 0 otherwise.
Non-blocking peek	
try_peek	Provides a new transaction without consuming it.
can_peek	Returns 1 if a new transaction is available; 0 otherwise.
Blocking transport	
transport	Executes the given request and returns the response in the given output argument.
Non-blocking transport	
nb_transport	Executes the given request and returns the response in the given output argument.
Analysis	
write	Broadcasts a user-defined transaction of type T to any number of listeners.

BLOCKING PUT

put

```
virtual task put(input T1 t)
```

Sends a user-defined transaction of type T.

Components implementing the put method will block the calling thread if it cannot immediately accept delivery of the transaction.

BLOCKING GET

get

```
virtual task get(output T2 t)
```

Provides a new transaction of type T.

The calling thread is blocked if the requested transaction cannot be provided immediately. The new transaction is returned in the provided output argument.

The implementation of get must regard the transaction as consumed. Subsequent calls to get must return a different transaction instance.

BLOCKING PEEK

peek

```
virtual task peek(output T2 t)
```

Obtain a new transaction without consuming it.

If a transaction is available, then it is written to the provided output argument. If a transaction is not available, then the calling thread is blocked until one is available.

The returned transaction is not consumed. A subsequent peek or get will return the same transaction.

NON-BLOCKING PUT

try_put

```
virtual function bit try_put(input T1 t)
```

Sends a transaction of type T, if possible.

If the component is ready to accept the transaction argument, then it does so and returns 1, otherwise it returns 0.

can_put

```
virtual function bit can_put()
```

Returns 1 if the component is ready to accept the transaction; 0 otherwise.

Non-blocking get

try_get

```
virtual function bit try_get(output T2 t)
```

Provides a new transaction of type T.

If a transaction is immediately available, then it is written to the output argument and 1 is returned. Otherwise, the output argument is not modified and 0 is returned.

can_get

```
virtual function bit can_get()
```

Returns 1 if a new transaction can be provided immediately upon request, 0 otherwise.

Non-blocking peek

try_peek

```
virtual function bit try_peek(output T2 t)
```

Provides a new transaction without consuming it.

If available, a transaction is written to the output argument and 1 is returned. A subsequent peek or get will return the same transaction. If a transaction is not available, then the argument is unmodified and 0 is returned.

can_peek

```
virtual function bit can_peek()
```

Returns 1 if a new transaction is available; 0 otherwise.

Blocking transport

transport

```
virtual task transport(input T1 req ,
                       output T2  rsp)
```

Executes the given request and returns the response in the given output argument. The calling thread may block until the operation is complete.

NON-BLOCKING TRANSPORT

nb_transport

```
virtual function bit nb_transport( input T1 req,
                                   output T2 rsp )
```

Executes the given request and returns the response in the given output argument. Completion of this operation must occur without blocking.

If for any reason the operation could not be executed immediately, then a 0 must be returned; otherwise 1.

ANALYSIS

write

```
virtual function void write(input T1 t)
```

Broadcasts a user-defined transaction of type T to any number of listeners. The operation must complete without blocking.

TLM Port Classes

The following classes define the TLM port classes.

Contents

uvm_*_port #(T)

These unidirectional ports are instantiated by components that *require*, or *use*, the associated interface to convey transactions. A port can be connected to any compatible port, export, or imp port. Unless its *min_size* is 0, a port *must* be connected to at least one implementation of its assocated interface.

The asterisk in *uvm_*_port* is any of the following

```
blocking_put
nonblocking_put
put

blocking_get
nonblocking_get
get

blocking_peek
nonblocking_peek
peek

blocking_get_peek
nonblocking_get_peek
get_peek
```

Type parameters

T The type of transaction to be communicated by the export

Ports are connected to interface implementations directly via uvm_*_imp #(T,IMP) ports or indirectly via hierarchical connections to uvm_*_port #(T) and uvm_*_export #(T) ports.

Summary

uvm_*_port #(T)

These unidirectional ports are instantiated by components that *require*, or *use*, the associated interface to convey transactions.

METHODS

new The *name* and *parent* are the standard uvm_component constructor arguments.

METHODS

new

The *name* and *parent* are the standard uvm_component constructor arguments. The *min_size* and *max_size* specify the minimum and maximum number of interfaces that must have been connected to this port by the end of elaboration.

```
function new (string name,
              uvm_component parent,
              int min_size=1,
              int max_size=1)
```

uvm_*_port #(REQ,RSP)

These bidirectional ports are instantiated by components that *require*, or *use*, the associated interface to convey transactions. A port can be connected to any compatible port, export, or imp port. Unless its *min_size* is 0, a port *must* be connected to at least one implementation of its assocated interface.

The asterisk in *uvm_*_port* is any of the following

```
blocking_transport
nonblocking_transport
transport

blocking_master
nonblocking_master
master

blocking_slave
nonblocking_slave
slave
```

Ports are connected to interface implementations directly via uvm_*_imp #(REQ,RSP,IMP,REQ_IMP,RSP_IMP) ports or indirectly via hierarchical connections to uvm_*_port #(REQ,RSP) and uvm_*_export #(REQ,RSP) ports.

Type parameters

REQ	The type of request transaction to be communicated by the export
RSP	The type of response transaction to be communicated by the export

Summary

uvm_*_port #(REQ,RSP)

These bidirectional ports are instantiated by components that *require*, or *use*, the associated interface to convey transactions.

METHODS

new	The *name* and *parent* are the standard uvm_component constructor arguments.

METHODS

new

The *name* and *parent* are the standard uvm_component constructor arguments. The *min_size* and *max_size* specify the minimum and maximum number of interfaces that must have been supplied to this port by the end of elaboration.

function new (string name, uvm_component parent, int min_size=1, int max_size=1)

TLM Export Classes

The following classes define the TLM export classes.

Contents

uvm_*_export #(T)

The unidirectional uvm_*_export is a port that *forwards* or *promotes* an interface implementation from a child component to its parent. An export can be connected to any compatible child export or imp port. It must ultimately be connected to at least one implementation of its associated interface.

The interface type represented by the asterisk is any of the following

```
blocking_put
nonblocking_put
put

blocking_get
nonblocking_get
get

blocking_peek
nonblocking_peek
peek

blocking_get_peek
nonblocking_get_peek
get_peek
```

Type parameters

T The type of transaction to be communicated by the export

Exports are connected to interface implementations directly via uvm_*_imp #(T,IMP) ports or indirectly via other uvm_*_export #(T) exports.

Summary

uvm_*_export #(T)

The unidirectional uvm_*_export is a port that *forwards* or *promotes* an interface implementation from a child component to its parent.

Methods

new	The *name* and *parent* are the standard uvm_component

constructor arguments.

METHODS

new

The *name* and *parent* are the standard uvm_component constructor arguments. The *min_size* and *max_size* specify the minimum and maximum number of interfaces that must have been supplied to this port by the end of elaboration.

```
function new (string name,
              uvm_component parent,
              int min_size=1,
              int max_size=1)
```

uvm_*_export #(REQ,RSP)

The bidirectional uvm_*_export is a port that *forwards* or *promotes* an interface implementation from a child component to its parent. An export can be connected to any compatible child export or imp port. It must ultimately be connected to at least one implementation of its associated interface.

The interface type represented by the asterisk is any of the following

```
blocking_transport
nonblocking_transport
transport

blocking_master
nonblocking_master
master

blocking_slave
nonblocking_slave
slave
```

Type parameters

REQ	The type of request transaction to be communicated by the export
RSP	The type of response transaction to be communicated by the export

Exports are connected to interface implementations directly via uvm_*_imp #(REQ, RSP, IMP, REQ_IMP, RSP_IMP) ports or indirectly via other uvm_*_export #(REQ,RSP) exports.

Summary

uvm_*_export #(REQ,RSP)

The bidirectional uvm_*_export is a port that *forwards* or *promotes* an interface implementation from a child component to its parent.

METHODS

new	The *name* and *parent* are the standard uvm_component constructor arguments.

METHODS

new

The *name* and *parent* are the standard uvm_component constructor arguments. The *min_size* and *max_size* specify the minimum and maximum number of interfaces that must have been supplied to this port by the end of elaboration.

```
function new (string name,
              uvm_component parent,
              int min_size=1,
              int max_size=1)
```

uvm_*_imp ports

The following defines the TLM implementation (imp) classes.

Contents

uvm_*_imp #(T,IMP)

Unidirectional implementation (imp) port classes--An imp port provides access to an implementation of the associated interface to all connected *ports* and *exports*. Each imp port instance *must* be connected to the component instance that implements the associated interface, typically the imp port's parent. All other connections-- e.g. to other ports and exports-- are prohibited.

The asterisk in *uvm_*_imp* may be any of the following

```
blocking_put
nonblocking_put
put

blocking_get
nonblocking_get
get

blocking_peek
nonblocking_peek
peek

blocking_get_peek
nonblocking_get_peek
get_peek
```

Type parameters

T	The type of transaction to be communicated by the imp
IMP	The type of the component implementing the interface. That is, the class to which this imp will delegate.

The interface methods are implemented in a component of type *IMP*, a handle to which is passed in a constructor argument. The imp port delegates all interface calls to this component.

Summary

Unidirectional implementation (imp) port classes--An imp port provides access to an implementation of the associated interface to all connected *ports* and *exports*.

METHODS

new	Creates a new unidirectional imp port with the given *name* and *parent*.

METHODS

new

Creates a new unidirectional imp port with the given *name* and *parent*. The *parent* must implement the interface associated with this port. Its type must be the type specified in the imp's type-parameter, *IMP*.

```
function new (string name, IMP parent);
```

uvm_*_imp #(REQ, RSP, IMP, REQ_IMP, RSP_IMP)

Bidirectional implementation (imp) port classes--An imp port provides access to an implementation of the associated interface to all connected *ports* and *exports*. Each imp port instance *must* be connected to the component instance that implements the associated interface, typically the imp port's parent. All other connections-- e.g. to other ports and exports-- are prohibited.

The interface represented by the asterisk is any of the following

```
blocking_transport
nonblocking_transport
transport

blocking_master
nonblocking_master
master

blocking_slave
nonblocking_slave
slave
```

Type parameters

REQ	Request transaction type
RSP	Response transaction type
IMP	Component type that implements the interface methods, typically the the parent of this imp port.
REQ_IMP	Component type that implements the request side of the interface. Defaults to IMP. For master and slave imps only.
RSP_IMP	Component type that implements the response side of the interface. Defaults to IMP. For master and slave imps only.

The interface methods are implemented in a component of type *IMP*, a handle to which is passed in a constructor argument. The imp port delegates all interface calls to this component.

The master and slave imps have two modes of operation.

- A single component of type IMP implements the entire interface for both requests and responses.
- Two sibling components of type REQ_IMP and RSP_IMP implement the request and response interfaces, respectively. In this case, the IMP parent instantiates this imp port *and* the REQ_IMP and RSP_IMP components.

The second mode is needed when a component instantiates more than one imp port, as in the uvm_tlm_req_rsp_channel #(REQ,RSP) channel.

Summary

uvm_*_imp #(REQ, RSP, IMP, REQ_IMP, RSP_IMP)

Bidirectional implementation (imp) port classes--An imp port provides access to an implementation of the associated interface to all connected *ports* and *exports*.

METHODS

new	Creates a new bidirectional imp port with the given *name* and *parent*.

METHODS

new

Creates a new bidirectional imp port with the given *name* and *parent*. The *parent*, whose type is specified by *IMP* type parameter, must implement the interface associated with this port.

Transport imp constructor

```
function new(string name, IMP imp)
```

Master and slave imp constructor

The optional *req_imp* and *rsp_imp* arguments, available to master and slave imp ports, allow the requests and responses to be handled by different subcomponents. If they are specified, they must point to the underlying component that implements the request and response methods, respectively.

```
function new(string name, IMP imp,
                          REQ_IMP req_imp=imp, RSP_IMP rsp_imp=imp)
```

Analysis Ports

This section defines the port, export, and imp classes used for transaction analysis.

Contents

uvm_analysis_port

Broadcasts a value to all subscribers implementing a uvm_analysis_imp.

```
class mon extends uvm_component;
  uvm_analysis_port#(trans) ap;

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

  task run_phase(uvm_phase phase);
      trans t;
      ...
      ap.write(t);
      ...
  endfunction
endclass
```

Summary

uvm_analysis_port

Broadcasts a value to all subscribers implementing a uvm_analysis_imp.

Class Hierarchy

uvm_port_base#(uvm_tlm_if_base#(T,T))

uvm_analysis_port

Class Declaration

```
class uvm_analysis_port # (
    type  T = int
) extends uvm_port_base # (uvm_tlm_if_base #(T,T))
```

Methods

write	Send specified value to all connected interface

write

```
function void write (input T t)
```

Send specified value to all connected interface

uvm_analysis_imp

Receives all transactions broadcasted by a uvm_analysis_port. It serves as the termination point of an analysis port/export/imp connection. The component attached to the *imp* class--called a *subscriber*-- implements the analysis interface.

Will invoke the *write(T)* method in the parent component. The implementation of the *write(T)* method must not modify the value passed to it.

```
class sb extends uvm_component;
  uvm_analysis_imp#(trans, sb) ap;

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

  function void write(trans t);
     ...
  endfunction
endclass
```

Summary

uvm_analysis_imp

Receives all transactions broadcasted by a uvm_analysis_port.

CLASS HIERARCHY

uvm_port_base#(uvm_tlm_if_base#(T,T))

uvm_analysis_imp

CLASS DECLARATION

```
class uvm_analysis_imp #(
   type T   = int,
   type IMP = int
) extends uvm_port_base #(uvm_tlm_if_base #(T,T))
```

uvm_analysis_export

Exports a lower-level uvm_analysis_imp to its parent.

Summary

uvm_analysis_export

Exports a lower-level uvm_analysis_imp to its parent.

Class Hierarchy

uvm_port_base#(uvm_tlm_if_base#(T,T))

uvm_analysis_export

Class Declaration

```
class uvm_analysis_export #(
    type  T =  int
) extends uvm_port_base #(uvm_tlm_if_base #(T,T))
```

Methods

new	Instantiate the export.

Methods

new

```
function new (string        name,
              uvm_component parent = null)
```

Instantiate the export.

TLM FIFO Classes

This section defines TLM-based FIFO classes.

Contents

uvm_tlm_fifo

This class provides storage of transactions between two independently running processes. Transactions are put into the FIFO via the *put_export*. transactions are fetched from the FIFO in the order they arrived via the *get_peek_export*. The *put_export* and *get_peek_export* are inherited from the uvm_tlm_fifo_base #(T) super class, and the interface methods provided by these exports are defined by the uvm_tlm_if_base #(T1,T2) class.

Summary

uvm_tlm_fifo

This class provides storage of transactions between two independently running processes.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- uvm_tlm_fifo_base#(T)
- **uvm_tlm_fifo**

Class Declaration

```
class uvm_tlm_fifo #(
    type  T = int
) extends uvm_tlm_fifo_base #(T)
```

Methods

new	The *name* and *parent* are the normal uvm_component constructor arguments.
size	Returns the capacity of the FIFO-- that is, the number of entries the FIFO is capable of holding.
used	Returns the number of entries put into the FIFO.
is_empty	Returns 1 when there are no entries in the FIFO, 0 otherwise.
is_full	Returns 1 when the number of entries in the FIFO is equal to its size, 0 otherwise.
flush	Removes all entries from the FIFO, after which used returns 0

and is_empty returns 1.

METHODS

new

```
function new(string        name,
             uvm_component parent = null,
             int           size   = 1    )
```

The *name* and *parent* are the normal uvm_component constructor arguments. The *parent* should be null if the uvm_tlm_fifo is going to be used in a statically elaborated construct (e.g., a module). The *size* indicates the maximum size of the FIFO; a value of zero indicates no upper bound.

size

```
virtual function int size()
```

Returns the capacity of the FIFO-- that is, the number of entries the FIFO is capable of holding. A return value of 0 indicates the FIFO capacity has no limit.

used

```
virtual function int used()
```

Returns the number of entries put into the FIFO.

is_empty

```
virtual function bit is_empty()
```

Returns 1 when there are no entries in the FIFO, 0 otherwise.

is_full

```
virtual function bit is_full()
```

Returns 1 when the number of entries in the FIFO is equal to its size, 0 otherwise.

flush

```
virtual function void flush()
```

Removes all entries from the FIFO, after which used returns 0 and is_empty returns 1.

uvm_tlm_analysis_fifo

An analysis_fifo is a uvm_tlm_fifo with an unbounded size and a write interface. It can be used any place a uvm_analysis_imp is used. Typical usage is as a buffer between an uvm_analysis_port in an initiator component and TLM1 target component.

Summary

uvm_tlm_analysis_fifo

An analysis_fifo is a uvm_tlm_fifo with an unbounded size and a write interface.

Class Hierarchy

uvm_tlm_fifo#(T)

uvm_tlm_analysis_fifo

Class Declaration

```
class uvm_tlm_analysis_fifo #(
    type   T =  int
) extends uvm_tlm_fifo #(T)
```

Ports

analysis_export #(T)	The analysis_export provides the write method to all connected analysis ports and parent exports:

Methods

new	This is the standard uvm_component constructor.

Ports

analysis_export #(T)

The analysis_export provides the write method to all connected analysis ports and parent exports:

```
function void write (T t)
```

Access via ports bound to this export is the normal mechanism for writing to an analysis FIFO. See write method of uvm_tlm_if_base #(T1,T2) for more information.

Methods

new

```
function new(string name
             ,
             uvm_component parent = null)
```

This is the standard uvm_component constructor. *name* is the local name of this

component. The *parent* should be left unspecified when this component is instantiated in statically elaborated constructs and must be specified when this component is a child of another UVM component.

uvm_tlm_fifo_base #(T)

This class is the base for <uvm_tlm_fifo #(T)>. It defines the TLM exports through which all transaction-based FIFO operations occur. It also defines default implementations for each inteface method provided by these exports.

The interface methods provided by the put_export and the get_peek_export are defined and described by uvm_tlm_if_base #(T1,T2). See the TLM Overview section for a general discussion of TLM interface definition and usage.

Parameter type

T The type of transactions to be stored by this FIFO.

Summary

uvm_tlm_fifo_base #(T)

This class is the base for <uvm_tlm_fifo #(T)>.

Class Hierarchy

uvm_void
uvm_object
uvm_report_object
uvm_component
uvm_tlm_fifo_base#(T)

Class Declaration

```
virtual class uvm_tlm_fifo_base #(
    type  T = int
) extends uvm_component
```

Ports

put_export	The *put_export* provides both the blocking and non-blocking put interface methods to any attached port:
get_peek_export	The *get_peek_export* provides all the blocking and non-blocking get and peek interface methods:
put_ap	Transactions passed via *put* or *try_put* (via any port connected to the put_export) are sent out this port via its *write* method.
get_ap	Transactions passed via *get*, *try_get*, *peek*, or *try_peek* (via any port connected to the get_peek_export) are sent out this port via its *write* method.

Methods

new	The *name* and *parent* are the normal uvm_component constructor arguments.

Ports

put_export

The *put_export* provides both the blocking and non-blocking put interface methods to

any attached port:

```
task put (input T t)
function bit can_put ()
function bit try_put (input T t)
```

Any *put* port variant can connect and send transactions to the FIFO via this export, provided the transaction types match. See uvm_tlm_if_base #(T1,T2) for more information on each of the above interface methods.

get_peek_export

The *get_peek_export* provides all the blocking and non-blocking get and peek interface methods:

```
task get (output T t)
function bit can_get ()
function bit try_get (output T t)
task peek (output T t)
function bit can_peek ()
function bit try_peek (output T t)
```

Any *get* or *peek* port variant can connect to and retrieve transactions from the FIFO via this export, provided the transaction types match. See uvm_tlm_if_base #(T1,T2) for more information on each of the above interface methods.

put_ap

Transactions passed via *put* or *try_put* (via any port connected to the put_export) are sent out this port via its *write* method.

```
function void write (T t)
```

All connected analysis exports and imps will receive put transactions. See uvm_tlm_if_base #(T1,T2) for more information on the *write* interface method.

get_ap

Transactions passed via *get*, *try_get*, *peek*, or *try_peek* (via any port connected to the get_peek_export) are sent out this port via its *write* method.

```
function void write (T t)
```

All connected analysis exports and imps will receive get transactions. See uvm_tlm_if_base #(T1,T2) for more information on the *write* method.

new

```
function new(string        name,
             uvm_component parent = null)
```

The *name* and *parent* are the normal uvm_component constructor arguments. The *parent* should be null if the uvm_tlm_fifo is going to be used in a statically elaborated construct (e.g., a module). The *size* indicates the maximum size of the FIFO. A value of zero indicates no upper bound.

TLM Channel Classes

This section defines built-in TLM channel classes.

Contents

uvm_tlm_req_rsp_channel #(REQ,RSP)

The uvm_tlm_req_rsp_channel contains a request FIFO of type *REQ* and a response FIFO of type *RSP*. These FIFOs can be of any size. This channel is particularly useful for dealing with pipelined protocols where the request and response are not tightly coupled.

Type parameters

REQ	Type of the request transactions conveyed by this channel.
RSP	Type of the reponse transactions conveyed by this channel.

Summary

uvm_tlm_req_rsp_channel #(REQ,RSP)

The uvm_tlm_req_rsp_channel contains a request FIFO of type *REQ* and a response FIFO of type *RSP*.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- **uvm_tlm_req_rsp_channel#(REQ,RSP)**

Class Declaration

```
class uvm_tlm_req_rsp_channel #(
    type  REQ  =  int,
    type  RSP  =  REQ
) extends uvm_component
```

Ports

put_request_export	The put_export provides both the blocking and non-blocking put interface methods to the request FIFO:
get_peek_response_export	The get_peek_response_export provides all the blocking and non-blocking get and peek interface methods to the response FIFO:

get_peek_request_export	The get_peek_export provides all the blocking and non-blocking get and peek interface methods to the response FIFO:
put_response_export	The put_export provides both the blocking and non-blocking put interface methods to the response FIFO:
request_ap	Transactions passed via *put* or *try_put* (via any port connected to the put_request_export) are sent out this port via its write method.
response_ap	Transactions passed via *put* or *try_put* (via any port connected to the put_response_export) are sent out this port via its write method.
master_export	Exports a single interface that allows a master to put requests and get or peek responses.
slave_export	Exports a single interface that allows a slave to get or peek requests and to put responses.

METHODS

new	The *name* and *parent* are the standard uvm_component constructor arguments.

PORTS

put_request_export

The put_export provides both the blocking and non-blocking put interface methods to the request FIFO:

```
task put (input T t);
function bit can_put ();
function bit try_put (input T t);
```

Any put port variant can connect and send transactions to the request FIFO via this export, provided the transaction types match.

get_peek_response_export

The get_peek_response_export provides all the blocking and non-blocking get and peek interface methods to the response FIFO:

```
task get (output T t);
function bit can_get ();
function bit try_get (output T t);
task peek (output T t);
function bit can_peek ();
function bit try_peek (output T t);
```

Any get or peek port variant can connect to and retrieve transactions from the response FIFO via this export, provided the transaction types match.

get_peek_request_export

The get_peek_export provides all the blocking and non-blocking get and peek interface

methods to the response FIFO:

```
task get (output T t);
function bit can_get ();
function bit try_get (output T t);
task peek (output T t);
function bit can_peek ();
function bit try_peek (output T t);
```

Any get or peek port variant can connect to and retrieve transactions from the response FIFO via this export, provided the transaction types match.

put_response_export

The put_export provides both the blocking and non-blocking put interface methods to the response FIFO:

```
task put (input T t);
function bit can_put ();
function bit try_put (input T t);
```

Any put port variant can connect and send transactions to the response FIFO via this export, provided the transaction types match.

request_ap

Transactions passed via *put* or *try_put* (via any port connected to the put_request_export) are sent out this port via its write method.

```
function void write (T t);
```

All connected analysis exports and imps will receive these transactions.

response_ap

Transactions passed via *put* or *try_put* (via any port connected to the put_response_export) are sent out this port via its write method.

```
function void write (T t);
```

All connected analysis exports and imps will receive these transactions.

master_export

Exports a single interface that allows a master to put requests and get or peek responses. It is a combination of the put_request_export and get_peek_response_export.

slave_export

Exports a single interface that allows a slave to get or peek requests and to put responses. It is a combination of the get_peek_request_export and put_response_export.

Methods

new

```
function new (string        name,
              uvm_component parent             = null,
              int           request_fifo_size  = 1,
              int           response_fifo_size = 1    )
```

The *name* and *parent* are the standard uvm_component constructor arguments. The *parent* must be null if this component is defined within a static component such as a module, program block, or interface. The last two arguments specify the request and response FIFO sizes, which have default values of 1.

uvm_tlm_transport_channel #(REQ,RSP)

A uvm_tlm_transport_channel is a uvm_tlm_req_rsp_channel #(REQ,RSP) that implements the transport interface. It is useful when modeling a non-pipelined bus at the transaction level. Because the requests and responses have a tightly coupled one-to-one relationship, the request and response FIFO sizes are both set to one.

Summary

uvm_tlm_transport_channel #(REQ,RSP)

A uvm_tlm_transport_channel is a uvm_tlm_req_rsp_channel #(REQ,RSP) that implements the transport interface.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- uvm_tlm_req_rsp_channel#(REQ,RSP)
- **uvm_tlm_transport_channel#(REQ,RSP)**

Class Declaration

```
class uvm_tlm_transport_channel #(
    type   REQ  =  int,
    type   RSP  =  REQ
) extends uvm_tlm_req_rsp_channel #(REQ, RSP)
```

Ports

transport_export	The put_export provides both the blocking and non-blocking transport interface methods to the response FIFO:

METHODS

new	The *name* and *parent* are the standard uvm_component constructor arguments.

PORTS

transport_export

The put_export provides both the blocking and non-blocking transport interface methods to the response FIFO:

```
task transport(REQ request, output RSP response);
function bit nb_transport(REQ request, output RSP response);
```

Any transport port variant can connect to and send requests and retrieve responses via this export, provided the transaction types match. Upon return, the response argument carries the response to the request.

METHODS

new

```
function new (string        name,
              uvm_component parent = null)
```

The *name* and *parent* are the standard uvm_component constructor arguments. The *parent* must be null if this component is defined within a statically elaborated construct such as a module, program block, or interface.

TLM2 Interfaces, Ports, Exports and Transport Interfaces Subset

Sockets group together all the necessary core interfaces for transportation and binding, allowing more generic usage models than just TLM core interfaces.

A socket is like a port or export; in fact it is derived from the same base class as ports and export, namely uvm_port_base #(IF). However, unlike a port or export a socket provides both a forward and backward path. Thus you can enable asynchronous (pipelined) bi-directional communication by connecting sockets together. To enable this, a socket contains both a port and an export. Components that initiate transactions are called initiators, and components that receive transactions sent by an initiator are called targets. Initiators have initiator sockets and targets have target sockets. Initiator sockets can connect to target sockets. You cannot connect initiator sockets to other initiator sockets and you cannot connect target sockets to target sockets.

The UVM TLM2 subset provides the following two transport interfaces

Blocking (b_transport)	completes the entire transaction within a single method call
Non-blocking (nb_transport)	describes the progress of a transaction using multiple nb_transport() method calls going back-and-forth between initiator and target

In general,any component might modify a transaction object during its lifetime (subject to the rules of the protocol). Significant timing points during the lifetime of a transaction (for example: start-ofresponse- phase) are indicated by calling nb_transport() in either forward or backward direction, the specific timing point being given by the phase argument. Protocol-specific rules for reading or writing the attributes of a transaction can be expressed relative to the phase. The phase can be used for flow control, and for that reason might have a different value at each hop taken by a transaction; the phase is not an attribute of the transaction object.

A call to nb_transport() always represents a phase transition. However, the return from nb_transport() might or might not do so, the choice being indicated by the value returned from the function (UVM_TLM_ACCEPTED versus UVM_TLM_UPDATED). Generally, you indicate the completion of a transaction over a particular hop using the value of the phase argument. As a shortcut, a target might indicate the completion of the transaction by returning a special value of UVM_TLM_COMPLETED. However, this is an option, not a necessity.

The transaction object itself does not contain any timing information by design. Or even events and status information concerning the API. You can pass the delays as arguments to b_transport()/ nb_transport() and push the actual realization of any delay in the simulator kernel downstream and defer (for simulation speed).

Use Models

Since sockets are derived from uvm_port_base #(IF) they are created and connected in the same way as port, and exports. Create them in the build phase and connect them in the connect phase by calling connect(). Initiator and target termination sockets are on the ends of any connection. There can be an arbitrary number of passthrough sockets in the path between initator and target. Some socket types must be bound to imps implementations of the transport tasks and functions. Blocking terminator sockets must be bound to an implementation of b_transport(), for example. Nonblocking initiator sockets must be bound to an implementation of nb_transport_bw() and nonblocking target sockets must be bound to an implementation of nb_transport_fw(). Typically, the task or function is implemented in the component in which the socket is instantiated and

the component type and instance are provided to complete the binding.

Consider for example a consumer component with a blocking target socket.

Example

```
class consumer extends uvm_component;
   tlm2_b_target_socket #(consumer, trans) target_socket;
   function new(string name, uvm_component parent);
     super.new(name, parent);
   endfunction
   function void build();
     target_socket = new("target_socket", this, this);
   endfunction
   task b_transport(trans t, uvm_tlm_time delay);
     #5;
     uvm_report_info("consumer", t.convert2string());
   endtask
endclass
```

The interface task b_transport() is implemented in the consumer component. The consumer component type is used in the declaration of the target socket. This informs the socket object the type of the object that contains the interface task, in this case b_transport(). When the socket is instantiated "this" is passed in twice, once as the parent just like any other component instantiation and again to identify the object that holds the implementation of b_transport(). Finally, in order to complete the binding, an implementation of b_transport() must be present in the consumer component. Any component that has either a blocking termination socket, a nonblocking initiator socket, or a nonblocking termination socket must provide implementations of the relevant components. This includes initiator and target components as well as interconnect components that have these kinds of sockets. Components with passthrough sockets do not need to provide implementations of any sort. Of course, they must ultimately be connected to sockets that do that the necessary implementations.

In summary

Call to b_transport()	start-of-life of transaction
Return from b_transport()	end-of-life of transaction
Phase argument to nb_transport()	timing point within lifetime of transaction
Return value of nb_transport()	whether return path is being used (also shortcut to final phase)
Response status within transaction object	protocol-specific status, success/failure of transaction

On top of this, TLM-2.0 defines a generic payload and base protocol to enhance interoperability for models with a memory-mapped bus interface.

It is possible to use the interfaces described above with user-defined transaction types and protocols for the sake of interoperability. However, TLM-2.0 strongly recommends either using the base protocol off-the-shelf or creating models of specific protocols on top of the base protocol.

Summary

TLM2 Interfaces, Ports, Exports and Transport Interfaces Subset

Sockets group together all the necessary core interfaces for transportation and binding, allowing more generic usage models than just TLM core interfaces.

TLM Generic Payload & Extensions

The Generic Payload transaction represents a generic bus read/write access. It is used as the default transaction in TLM2 blocking and nonblocking transport interfaces.

Contents

GLOBALS

Defines, Constants, enums.

uvm_tlm_command_e

Command atribute type definition

UVM_TLM_READ_COMMAND	Bus read operation
UVM_TLM_WRITE_COMMAND	Bus write operation
UVM_TLM_IGNORE_COMMAND	No bus operation.

uvm_tlm_response_status_e

Respone status attribute type definition

UVM_TLM_OK_RESPONSE	Bus operation completed succesfully
UVM_TLM_INCOMPLETE_RESPONSE	Transaction was not delivered to target
UVM_TLM_GENERIC_ERROR_RESPONSE	Bus operation had an error
UVM_TLM_ADDRESS_ERROR_RESPONSE	Invalid address specified
UVM_TLM_COMMAND_ERROR_RESPONSE	Invalid command specified
UVM_TLM_BURST_ERROR_RESPONSE	Invalid burst specified
UVM_TLM_BYTE_ENABLE_ERROR_RESPONSE	Invalid byte enabling specified

uvm_tlm_generic_payload

This class provides a transaction definition commonly used in memory-mapped bus-based systems. It's intended to be a general purpose transaction class that lends itself to many applications. The class is derived from uvm_sequence_item which enables it to be generated in sequences and transported to drivers through sequencers.

Summary

uvm_tlm_generic_payload

This class provides a transaction definition commonly used in memory-mapped bus-based systems.

CLASS HIERARCHY

uvm_void
uvm_object
uvm_transaction
uvm_sequence_item
uvm_tlm_generic_payload

CLASS DECLARATION

```
class uvm_tlm_generic_payload extends uvm_sequence_item
```

m_address	Address for the bus operation.
m_command	Bus operation type.
m_data	Data read or to be written.
m_length	The number of bytes to be copied to or from the m_data array, inclusive of any bytes disabled by the m_byte_enable attribute.
m_response_status	Status of the bus operation.
m_dmi	DMI mode is not yet supported in the UVM TLM2 subset.
m_byte_enable	Indicates valid m_data array elements.
m_byte_enable_length	The number of elements in the m_byte_enable array.
m_streaming_width	Number of bytes transferred on each beat.
new	Create a new instance of the generic payload.
convert2string	Convert the contents of the class to a string suitable for printing.
ACCESSORS	The accessor functions let you set and get each of the members of the generic payload.
get_command	Get the value of the m_command variable
set_command	Set the value of the m_command variable
is_read	Returns true if the current value of the m_command variable is *UVM_TLM_READ_COMMAND*.
set_read	Set the current value of the m_command variable to *UVM_TLM_READ_COMMAND*.
is_write	Returns true if the current value of the m_command variable is *UVM_TLM_WRITE_COMMAND*.
set_write	Set the current value of the m_command variable

	to *UVM_TLM_WRITE_COMMAND*.
set_address	Set the value of the m_address variable
get_address	Get the value of the m_address variable
get_data	Return the value of the m_data array
set_data	Set the value of the m_data array
get_data_length	Return the current size of the m_data array
set_data_length	Set the value of the m_length
get_streaming_width	Get the value of the m_streaming_width array
set_streaming_width	Set the value of the m_streaming_width array
get_byte_enable	Return the value of the m_byte_enable array
set_byte_enable	Set the value of the m_byte_enable array
get_byte_enable_length	Return the current size of the m_byte_enable array
set_byte_enable_length	Set the size m_byte_enable_length of the m_byte_enable array i.e m_byte_enable.size()
set_dmi_allowed	DMI hint.
is_dmi_allowed	DMI hint.
get_response_status	Return the current value of the m_response_status variable
set_response_status	Set the current value of the m_response_status variable
is_response_ok	Return TRUE if the current value of the m_response_status variable is *UVM_TLM_OK_RESPONSE*
is_response_error	Return TRUE if the current value of the m_response_status variable is not *UVM_TLM_OK_RESPONSE*
get_response_string	Return the current value of the m_response_status variable as a string
Extensions Mechanism	
set_extension	Add an instance-specific extension.
get_num_extensions	Return the current number of instance specific extensions.
get_extension	Return the instance specific extension bound under the specified key.
clear_extension	Remove the instance-specific extension bound under the specified key.
clear_extensions	Remove all instance-specific extensions

m_address

```
rand bit [63:0] m_address
```

Address for the bus operation. Should be set or read using the set_address and get_address methods. The variable should be used only when constraining.

For a read command or a write command, the target shall interpret the current value of the address attribute as the start address in the system memory map of the contiguous block of data being read or written. The address associated with any given byte in the data array is dependent upon the address attribute, the array index, the streaming width attribute, the endianness and the width of the physical bus.

If the target is unable to execute the transaction with the given address attribute (because the address is out-of-range, for example) it shall generate a standard error response. The recommended response status is *UVM_TLM_ADDRESS_ERROR_RESPONSE*.

m_command

```
rand uvm_tlm_command_e m_command
```

Bus operation type. Should be set using the set_command, set_read or set_write

methods and read using the get_command, is_read or is_write methods. The variable should be used only when constraining.

If the target is unable to execute a read or write command, it shall generate a standard error response. The recommended response status is UVM_TLM_COMMAND_ERROR_RESPONSE.

On receipt of a generic payload transaction with the command attribute equal to UVM_TLM_IGNORE_COMMAND, the target shall not execute a write command or a read command not modify any data. The target may, however, use the value of any attribute in the generic payload, including any extensions.

The command attribute shall be set by the initiator, and shall not be overwritten by any interconnect

m_data

```
rand byte unsigned m_data[]
```

Data read or to be written. Should be set and read using the set_data or get_data methods The variable should be used only when constraining.

For a read command or a write command, the target shall copy data to or from the data array, respectively, honoring the semantics of the remaining attributes of the generic payload.

For a write command or UVM_TLM_IGNORE_COMMAND, the contents of the data array shall be set by the initiator, and shall not be overwritten by any interconnect component or target. For a read command, the contents of the data array shall be overwritten by the target (honoring the semantics of the byte enable) but by no other component.

Unlike the OSCI TLM-2.0 LRM, there is no requirement on the endiannes of multi-byte data in the generic payload to match the host endianness. Unlike C++, it is not possible in SystemVerilog to cast an arbitrary data type as an array of bytes. Therefore, matching the host endianness is not necessary. In constrast, arbitrary data types may be converted to and from a byte array using the streaming operator and uvm_object objects may be further converted using the uvm_object::pack_bytes() and uvm_object::unpack_bytes() methods. All that is required is that a consistent mechanism is used to fill the payload data array and later extract data from it.

Should a generic payload be transfered to/from a systemC model, it will be necessary for any multi-byte data in that generic payload to use/be interpreted using the host endianness. However, this process is currently outside the scope of this standard.

m_length

```
rand int unsigned m_length
```

The number of bytes to be copied to or from the m_data array, inclusive of any bytes disabled by the m_byte_enable attribute.

The data length attribute shall be set by the initiator, and shall not be overwritten by any interconnect component or target.

The data length attribute shall not be set to 0. In order to transfer zero bytes, the m_command attribute should be set to UVM_TLM_IGNORE_COMMAND.

m_response_status

```
rand uvm_tlm_response_status_e m_response_status
```

Status of the bus operation. Should be set using the set_response_status method and read using the get_response_status, get_response_string, is_response_ok or is_response_error methods. The variable should be used only when constraining.

The response status attribute shall be set to UVM_TLM_INCOMPLETE_RESPONSE by the initiator, and may be overwritten by the target. The response status attribute should not be overwritten by any interconnect component, because the default value UVM_TLM_INCOMPLETE_RESPONSE indicates that the transaction was not delivered to the target.

The target may set the response status attribute to UVM_TLM_OK_RESPONSE to indicate that it was able to execute the command successfully, or to one of the five error responses to indicate an error. The target should choose the appropriate error response depending on the cause of the error. If a target detects an error but is unable to select a specific error response, it may set the response status to UVM_TLM_GENERIC_ERROR_RESPONSE.

The target shall be responsible for setting the response status attribute at the appropriate point in the lifetime of the transaction. In the case of the blocking transport interface, this means before returning control from b_transport. In the case of the non-blocking transport interface and the base protocol, this means before sending the BEGIN_RESP phase or returning a value of UVM_TLM_COMPLETED.

It is recommended that the initiator should always check the response status attribute on receiving a transition to the BEGIN_RESP phase or after the completion of the transaction. An initiator may choose to ignore the response status if it is known in advance that the value will be UVM_TLM_OK_RESPONSE, perhaps because it is known in advance that the initiator is only connected to targets that always return UVM_TLM_OK_RESPONSE, but in general this will not be the case. In other words, the initiator ignores the response status at its own risk.

m_dmi

```
rand bit m_dmi
```

DMI mode is not yet supported in the UVM TLM2 subset. This variable is provided for completeness and interoperability with SystemC.

m_byte_enable

```
rand byte unsigned m_byte_enable[]
```

Indicates valid m_data array elements. Should be set and read using the set_byte_enable or get_byte_enable methods The variable should be used only when constraining.

The elements in the byte enable array shall be interpreted as follows. A value of 0 shall indicate that that corresponding byte is disabled, and a value of 1 shall indicate that the corresponding byte is enabled.

Byte enables may be used to create burst transfers where the address increment between each beat is greater than the number of significant bytes transferred on each beat, or to place words in selected byte lanes of a bus. At a more abstract level, byte enables may be used to create "lacy bursts" where the data array of the generic payload

has an arbitrary pattern of holes punched in it.

The byte enable mask may be defined by a small pattern applied repeatedly or by a large pattern covering the whole data array. The byte enable array may be empty, in which case byte enables shall not be used for the current transaction.

The byte enable array shall be set by the initiator and shall not be overwritten by any interconnect component or target.

If the byte enable pointer is non-null, the target shall either implement the semantics of the byte enable as defined below or shall generate a standard error response. The recommended response status is UVM_TLM_BYTE_ENABLE_ERROR_RESPONSE.

In the case of a write command, any interconnect component or target should ignore the values of any disabled bytes in the m_data array. In the case of a read command, any interconnect component or target should not modify the values of disabled bytes in the m_data array.

m_byte_enable_length

```
rand int unsigned m_byte_enable_length
```

The number of elements in the m_byte_enable array.

It shall be set by the initiator, and shall not be overwritten by any interconnect component or target.

m_streaming_width

```
rand int unsigned m_streaming_width
```

Number of bytes transferred on each beat. Should be set and read using the set_streaming_width or get_streaming_width methods The variable should be used only when constraining.

Streaming affects the way a component should interpret the data array. A stream consists of a sequence of data transfers occurring on successive notional beats, each beat having the same start address as given by the generic payload address attribute. The streaming width attribute shall determine the width of the stream, that is, the number of bytes transferred on each beat. In other words, streaming affects the local address associated with each byte in the data array. In all other respects, the organisation of the data array is unaffected by streaming.

The bytes within the data array have a corresponding sequence of local addresses within the component accessing the generic payload transaction. The lowest address is given by the value of the address attribute. The highest address is given by the formula address_attribute + streaming_width - 1. The address to or from which each byte is being copied in the target shall be set to the value of the address attribute at the start of each beat.

With respect to the interpretation of the data array, a single transaction with a streaming width shall be functionally equivalent to a sequence of transactions each having the same address as the original transaction, each having a data length attribute equal to the streaming width of the original, and each with a data array that is a different subset of the original data array on each beat. This subset effectively steps down the original data array maintaining the sequence of bytes.

A streaming width of 0 indicates that a streaming transfer is not required. it is equivalent to a streaming width value greater than or equal to the size of the m_data array.

Streaming may be used in conjunction with byte enables, in which case the streaming width would typically be equal to the byte enable length. It would also make sense to have the streaming width a multiple of the byte enable length. Having the byte enable length a multiple of the streaming width would imply that different bytes were enabled on each beat.

If the target is unable to execute the transaction with the given streaming width, it shall generate a standard error response. The recommended response status is TLM_BURST_ERROR_RESPONSE.

new

```
function new(string name = "")
```

Create a new instance of the generic payload. Initialize all the members to their default values.

convert2string

```
function string convert2string()
```

Convert the contents of the class to a string suitable for printing.

ACCESSORS

The accessor functions let you set and get each of the members of the generic payload. All of the accessor methods are virtual. This implies a slightly different use model for the generic payload than in SsytemC. The way the generic payload is defined in SystemC does not encourage you to create new transaction types derived from uvm_tlm_generic_payload. Instead, you would use the extensions mechanism. Thus in SystemC none of the accessors are virtual.

get_command

```
virtual function uvm_tlm_command_e get_command()
```

Get the value of the m_command variable

set_command

```
virtual function void set_command(uvm_tlm_command_e command)
```

Set the value of the m_command variable

is_read

```
virtual function bit is_read()
```

Returns true if the current value of the m_command variable is *UVM_TLM_READ_COMMAND*.

set_read

```
virtual function void set_read()
```

Set the current value of the m_command variable to *UVM_TLM_READ_COMMAND*.

is_write

```
virtual function bit is_write()
```

Returns true if the current value of the m_command variable is *UVM_TLM_WRITE_COMMAND*.

set_write

```
virtual function void set_write()
```

Set the current value of the m_command variable to *UVM_TLM_WRITE_COMMAND*.

set_address

```
virtual function void set_address(bit [63:0] addr)
```

Set the value of the m_address variable

get_address

```
virtual function bit [63:0] get_address()
```

Get the value of the m_address variable

get_data

```
virtual function void get_data (output byte unsigned p [])
```

Return the value of the m_data array

set_data

```
virtual function void set_data(ref byte unsigned p [])
```

Set the value of the m_data array

get_data_length

```
virtual function int unsigned get_data_length()
```

Return the current size of the m_data array

set_data_length

```
virtual function void set_data_length(int unsigned length)
```

Set the value of the m_length

get_streaming_width

```
virtual function int unsigned get_streaming_width()
```

Get the value of the m_streaming_width array

set_streaming_width

```
virtual function void set_streaming_width(int unsigned width)
```

Set the value of the m_streaming_width array

get_byte_enable

```
virtual function void get_byte_enable(output byte unsigned p[])
```

Return the value of the m_byte_enable array

set_byte_enable

```
virtual function void set_byte_enable(ref byte unsigned p[])
```

Set the value of the m_byte_enable array

get_byte_enable_length

```
virtual function int unsigned get_byte_enable_length()
```

Return the current size of the m_byte_enable array

set_byte_enable_length

```
virtual function void set_byte_enable_length(int unsigned length)
```

Set the size m_byte_enable_length of the m_byte_enable array i.e m_byte_enable.size()

set_dmi_allowed

```
virtual function void set_dmi_allowed(bit dmi)
```

DMI hint. Set the internal flag m_dmi to allow dmi access

is_dmi_allowed

```
virtual function bit is_dmi_allowed()
```

DMI hint. Query the internal flag m_dmi if allowed dmi access

get_response_status

```
virtual function uvm_tlm_response_status_e get_response_status()
```

Return the current value of the m_response_status variable

set_response_status

```
virtual function void set_response_status(uvm_tlm_response_status_e status)
```

Set the current value of the m_response_status variable

is_response_ok

```
virtual function bit is_response_ok()
```

Return TRUE if the current value of the m_response_status variable is *UVM_TLM_OK_RESPONSE*

is_response_error

```
virtual function bit is_response_error()
```

Return TRUE if the current value of the m_response_status variable is not *UVM_TLM_OK_RESPONSE*

get_response_string

```
virtual function string get_response_string()
```

Return the current value of the m_response_status variable as a string

Extensions Mechanism

set_extension

```
function uvm_tlm_extension_base set_extension(uvm_tlm_extension_base ext)
```

Add an instance-specific extension. The specified extension is bound to the generic payload by ts type handle.

get_num_extensions

```
function int get_num_extensions()
```

Return the current number of instance specific extensions.

get_extension

```
function uvm_tlm_extension_base get_extension(uvm_tlm_extension_base ext_hand
```

Return the instance specific extension bound under the specified key. If no extension is bound under that key, *null* is returned.

clear_extension

```
function void clear_extension(uvm_tlm_extension_base ext_handle)
```

Remove the instance-specific extension bound under the specified key.

clear_extensions

```
function void clear_extensions()
```

Remove all instance-specific extensions

uvm_tlm_gp

This typedef provides a short, more convenient name for the uvm_tlm_generic_payload type.

Summary

uvm_tlm_gp

This typedef provides a short, more convenient name for the uvm_tlm_generic_payload type.

CLASS DECLARATION

```
typedef uvm_tlm_generic_payload uvm_tlm_gp
```

uvm_tlm_extension_base

The class uvm_tlm_extension_base is the non-parameterized base class for all generic payload extensions. It includes the utility do_copy() and create(). The pure virtual

function get_type_handle() allows you to get a unique handles that represents the derived type. This is implemented in derived classes.

This class is never used directly by users. The uvm_tlm_extension class is used instead.

Summary

uvm_tlm_extension_base

The class uvm_tlm_extension_base is the non-parameterized base class for all generic payload extensions.

Class Hierarchy

uvm_void
uvm_object
uvm_tlm_extension_base

Class Declaration

```
virtual class uvm_tlm_extension_base extends uvm_object
```

Methods

new	
get_type_handle	An interface to polymorphically retrieve a handle that uniquely identifies the type of the sub-class
get_type_handle_name	An interface to polymorphically retrieve the name that uniquely identifies the type of the sub-class
create	

Methods

new

```
function new(string name = "")
```

get_type_handle

```
pure virtual function uvm_tlm_extension_base get_type_handle()
```

An interface to polymorphically retrieve a handle that uniquely identifies the type of the sub-class

get_type_handle_name

```
pure virtual function string get_type_handle_name()
```

An interface to polymorphically retrieve the name that uniquely identifies the type of the sub-class

create

```
virtual function uvm_object create (string name = "")
```

uvm_tlm_extension

TLM extension class. The class is parameterized with arbitrary type which represents the type of the extension. An instance of the generic payload can contain one extension object of each type; it cannot contain two instances of the same extension type.

The extension type can be identified using the ID() method.

To implement a generic payload extension, simply derive a new class from this class and specify the name of the derived class as the extension parameter.

```
class my_ID extends uvm_tlm_extension#(my_ID);
  int ID;

  `uvm_object_utils_begin(my_ID)
    `uvm_field_int(ID, UVM_ALL_ON)
  `uvm_object_utils_end

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

Summary

uvm_tlm_extension

TLM extension class.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_tlm_extension_base
- **uvm_tlm_extension**

Class Declaration

```
class uvm_tlm_extension #(
    type   T =  int
) extends uvm_tlm_extension_base
```

Methods

new	creates a new extension object.
ID()	Return the unique ID of this TLM extension type.

Methods

new

```
function new(string name = "")
```

creates a new extension object.

ID()

```
static function this_type ID()
```

Return the unique ID of this TLM extension type. This method is used to identify the type of the extension to retrieve from a uvm_tlm_generic_payload instance, using the uvm_tlm_generic_payload::get_extension() method.

tlm interfaces

Summary

GLOBALS

Global macro's & enums

uvm_tlm_phase_e

Nonblocking transport synchronization state values between an initiator and a target.

UNINITIALIZED_PHASE	Defaults for constructor
BEGIN_REQ	Beginning of request phase
END_REQ	End of request phase
BEGIN_RESP	Begining of response phase
END_RESP	End of response phase

uvm_tlm_sync_e

Pre-defined phase state values for the nonblocking transport Base Protocol between an initiator and a target.

UVM_TLM_ACCEPTED	Transaction has been accepted
UVM_TLM_UPDATED	Transaction has been modified
UVM_TLM_COMPLETED	Execution of transaction is complete

`UVM_TLM_TASK_ERROR

Defines Not-Yet-Implemented TLM tasks

`UVM_TLM_FUNCTION_ERROR

TLM IF CLASS

Base class type to define the transport functions.

uvm_tlm_if

Base class type to define the transport functions.

- nb_transport_fw
- nb_transport_bw
- b_transport

Summary

uvm_tlm_if

Base class type to define the transport functions.

CLASS DECLARATION

```
class uvm_tlm_if #(type T = uvm_tlm_generic_payload,
                   type P = uvm_tlm_phase_e          )
```

TLM TRANSPORT METHODS	Each of the interface methods take a handle to the transaction to be transported and a reference argument for the delay.
nb_transport_fw	Forward path call.
nb_transport_bw	Implementation of the backward path.
b_transport	Execute a blocking transaction.

TLM TRANSPORT METHODS

Each of the interface methods take a handle to the transaction to be transported and a reference argument for the delay. In addition, the nonblocking interfaces take a reference argument for the phase.

nb_transport_fw

```
virtual function uvm_tlm_sync_e nb_transport_fw(      T            t,
                                                ref P            p,
                                              input uvm_tlm_time delay)
```

Forward path call. The first call to this method for a transaction marks the initial timing point. Every call to this method may mark a timing point in the execution of the transaction. The timing annotation argument allows the timing points to be offset from the simulation times at which the forward path is used. The final timing point of a transaction may be marked by a call to nb_transport_bw or a return from this or subsequent call to nb_transport_fw.

See TLM2 Interfaces, Ports, Exports and Transport Interfaces Subset for more details on

the semantics and rules of the nonblocking transport interface.

nb_transport_bw

```
virtual function uvm_tlm_sync_e nb_transport_bw(      T            t,
                                                  ref P            p,
                                                input uvm_tlm_time delay)
```

Implementation of the backward path. This function MUST be implemented in the INITIATOR component class.

Every call to this method may mark a timing point, including the final timing point, in the execution of the transaction. The timing annotation argument allows the timing point to be offset from the simulation times at which the backward path is used. The final timing point of a transaction may be marked by a call to nb_transport_fw or a return from this or subsequent call to nb_transport_bw.

See TLM2 Interfaces, Ports, Exports and Transport Interfaces Subset for more details on the semantics and rules of the nonblocking transport interface.

Example

```
class master extends uvm_component;
```

uvm_tlm_nb_initiator_socket #(trans, uvm_tlm_phase_e, this_t) initiator_socket;

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

initiator_socket = new("initiator_socket", this, this);

```
    endfunction

    function uvm_tlm_sync_e nb_transport_bw(ref trans t,
                                    ref uvm_tlm_phase_e p,
                                    input uvm_tlm_time delay);
        transaction = t;
        state = p;
        return UVM_TLM_ACCEPTED;
    endfunction

    ...
endclass
```

b_transport

```
virtual task b_transport(T            t,
                         uvm_tlm_time delay)
```

Execute a blocking transaction. Once this method returns, the transaction is assumed to have been executed. Whether that execution is succesful or not must be indicated by the transaction itself.

The callee may modify or update the transaction object, subject to any constraints imposed by the transaction class. The initiator may re-use a transaction object from one

call to the next and across calls to b_transport().

The call to b_transport shall mark the first timing point of the transaction. The return from b_transport shall mark the final timing point of the transaction. The timing annotation argument allows the timing points to be offset from the simulation times at which the task call and return are executed.

TLM Sockets

Each uvm_tlm_*_socket class is derived from a corresponding uvm_tlm_*_socket_base class. The base class contains most of the implementation of the class, The derived classes (in this file) contain the connection semantics.

Sockets come in several flavors: Each socket is either an initiator or a target, a passthrough or a terminator. Further, any particular socket implements either the blocking interfaces or the nonblocking interfaces. Terminator sockets are used on initiators and targets as well as interconnect components as shown in the figure above. Passthrough sockets are used to enable connections to cross hierarchical boundaries.

There are eight socket types: the cross of blocking and nonblocking, passthrough and termination, target and initiator

Sockets are specified based on what they are (IS-A) and what they contains (HAS-A). IS-A and HAS-A are types of object relationships. IS-A refers to the inheritance relationship and HAS-A refers to the ownership relationship. For example if you say D is a B that means that D is derived from base B. If you say object A HAS-A B that means that B is a member of A.

Contents

uvm_tlm_b_initiator_socket

IS-A forward port; has no backward path except via the payload contents

Summary

uvm_tlm_b_initiator_socket

IS-A forward port; has no backward path except via the payload contents

Class Hierarchy

uvm_tlm_b_initiator_socket_base#(T)

uvm_tlm_b_initiator_socket

Class Declaration

```
class uvm_tlm_b_initiator_socket #(
   type T = uvm_tlm_generic_payload
) extends uvm_tlm_b_initiator_socket_base #(T)
```

Methods

new	Construct a new instance of this socket
Connect	Connect this socket to the specified uvm_tlm_b_target_socket

Methods

new

```
function new(string        name,
             uvm_component parent)
```

Construct a new instance of this socket

Connect

Connect this socket to the specified uvm_tlm_b_target_socket

uvm_tlm_b_target_socket

IS-A forward imp; has no backward path except via the payload contents.

The component instantiating this socket must implement a b_transport() method with the following signature

```
task b_transport(T t, uvm_tlm_time delay);
```

Summary

uvm_tlm_b_target_socket

IS-A forward imp; has no backward path except via the payload contents.

Class Hierarchy

uvm_tlm_b_target_socket_base#(T)

uvm_tlm_b_target_socket

Class Declaration

```
class uvm_tlm_b_target_socket #(
   type IMP = int,
   type T   = uvm_tlm_generic_payload
) extends uvm_tlm_b_target_socket_base #(T)
```

Methods

new	Construct a new instance of this socket *imp* is a reference to the class implementing the b_transport() method.
Connect	Connect this socket to the specified uvm_tlm_b_initiator_socket

Methods

new

```
function new (string        name,
              uvm_component parent,
              IMP           imp     = null)
```

Construct a new instance of this socket *imp* is a reference to the class implementing the b_transport() method. If not specified, it is assume to be the same as *parent*.

Connect

Connect this socket to the specified uvm_tlm_b_initiator_socket

uvm_tlm_nb_initiator_socket

IS-A forward port; HAS-A backward imp

The component instantiating this socket must implement a nb_transport_bw() method with the following signature

```
function uvm_tlm_sync_e nb_transport_bw(T t, ref P p, input uvm_tlm_time
delay);
```

Summary

uvm_tlm_nb_initiator_socket

IS-A forward port; HAS-A backward imp

Class Hierarchy

uvm_tlm_nb_initiator_socket_base#(T,P)

uvm_tlm_nb_initiator_socket

Class Declaration

```
class uvm_tlm_nb_initiator_socket #(
   type IMP = int,
   type T   = uvm_tlm_generic_payload,
   type P   = uvm_tlm_phase_e
) extends uvm_tlm_nb_initiator_socket_base #(T,P)
```

Methods

new	Construct a new instance of this socket *imp* is a reference to the class implementing the nb_transport_bw() method.
Connect	Connect this socket to the specified uvm_tlm_nb_target_socket

METHODS

new

```
function new(string        name,
             uvm_component parent,
             IMP           imp     = null)
```

Construct a new instance of this socket *imp* is a reference to the class implementing the nb_transport_bw() method. If not specified, it is assume to be the same as *parent*.

Connect

Connect this socket to the specified uvm_tlm_nb_target_socket

uvm_tlm_nb_target_socket

IS-A forward imp; HAS-A backward port

The component instantiating this socket must implement a nb_transport_fw() method with the following signature

```
function uvm_tlm_sync_e nb_transport_fw(T t, ref P p, input uvm_tlm_time
delay);
```

Summary

uvm_tlm_nb_target_socket

IS-A forward imp; HAS-A backward port

CLASS HIERARCHY

uvm_tlm_nb_target_socket_base#(T,P)

uvm_tlm_nb_target_socket

CLASS DECLARATION

```
class uvm_tlm_nb_target_socket #(
   type IMP = int,
   type T   = uvm_tlm_generic_payload,
   type P   = uvm_tlm_phase_e
) extends uvm_tlm_nb_target_socket_base #(T,P)
```

METHODS

new	Construct a new instance of this socket *imp* is a reference to the class implementing the nb_transport_fw() method.

connect	Connect this socket to the specified uvm_tlm_nb_initiator_socket

METHODS

new

```
function new (string        name,
              uvm_component parent,
              IMP           imp     = null)
```

Construct a new instance of this socket *imp* is a reference to the class implementing the nb_transport_fw() method. If not specified, it is assume to be the same as *parent*.

connect

```
function void connect(this_type provider)
```

Connect this socket to the specified uvm_tlm_nb_initiator_socket

uvm_tlm_b_passthrough_initiator_socket

IS-A forward port;

Summary

uvm_tlm_b_passthrough_initiator_socket

IS-A forward port;

CLASS HIERARCHY

uvm_tlm_b_passthrough_initiator_socket_base#(T)

uvm_tlm_b_passthrough_initiator_socket

CLASS DECLARATION

```
class uvm_tlm_b_passthrough_initiator_socket #(
    type  T =  uvm_tlm_generic_payload
) extends uvm_tlm_b_passthrough_initiator_socket_base
#(T)
```

uvm_tlm_b_passthrough_target_socket

IS-A forward export;

Summary

uvm_tlm_b_passthrough_target_socket

IS-A forward export;

Class Hierarchy

uvm_tlm_b_passthrough_target_socket_base#(T)

uvm_tlm_b_passthrough_target_socket

Class Declaration

```
class uvm_tlm_b_passthrough_target_socket #(
    type  T = uvm_tlm_generic_payload
) extends uvm_tlm_b_passthrough_target_socket_base #(T)
```

uvm_tlm_nb_passthrough_initiator_socket

IS-A forward port; HAS-A backward export

Summary

uvm_tlm_nb_passthrough_initiator_socket

IS-A forward port; HAS-A backward export

Class Hierarchy

uvm_tlm_nb_passthrough_initiator_socket_base#(T,P)

uvm_tlm_nb_passthrough_initiator_socket

Class Declaration

```
class uvm_tlm_nb_passthrough_initiator_socket #(
    type  T = uvm_tlm_generic_payload,
    type  P = uvm_tlm_phase_e
) extends uvm_tlm_nb_passthrough_initiator_socket_base
#(T,P)
```

uvm_tlm_nb_passthrough_target_socket

IS-A forward export; HAS-A backward port

Summary

uvm_tlm_nb_passthrough_target_socket

IS-A forward export; HAS-A backward port

Class Hierarchy

uvm_tlm_nb_passthrough_target_socket_base#(T,P)

uvm_tlm_nb_passthrough_target_socket

Class Declaration

```
class uvm_tlm_nb_passthrough_target_socket #(
    type  T = uvm_tlm_generic_payload,
    type  P = uvm_tlm_phase_e
) extends uvm_tlm_nb_passthrough_target_socket_base #(T,P)
```

Methods

connect	Connect this socket to the specified uvm_tlm_nb_initiator_socket

Methods

connect

```
function void connect(this_type provider)
```

Connect this socket to the specified uvm_tlm_nb_initiator_socket

TLM2 ports

The following defines TLM2 port classes.

Contents

uvm_tlm_b_transport_port

Class providing the blocking transport port, The port can be bound to one export. There is no backward path for the blocking transport.

Summary

uvm_tlm_b_transport_port

Class providing the blocking transport port, The port can be bound to one export.

Class Hierarchy

uvm_port_base#(uvm_tlm_if#(T))

uvm_tlm_b_transport_port

Class Declaration

```
class uvm_tlm_b_transport_port #(
   type T = uvm_tlm_generic_payload
) extends uvm_port_base #(uvm_tlm_if #(T))
```

uvm_tlm_nb_transport_fw_port

Class providing the non-blocking backward transport port. Transactions received from the producer, on the forward path, are sent back to the producer on the backward path using this non-blocking transport port. The port can be bound to one export.

Summary

uvm_tlm_nb_transport_fw_port

Class providing the non-blocking backward transport port.

Class Hierarchy

uvm_port_base#(uvm_tlm_if#(T,P))

uvm_tlm_nb_transport_fw_port

CLASS DECLARATION

```
class uvm_tlm_nb_transport_fw_port #(
   type T = uvm_tlm_generic_payload,
   type P = uvm_tlm_phase_e
) extends uvm_port_base #(uvm_tlm_if #(T,P))
```

uvm_tlm_nb_transport_bw_port

Class providing the non-blocking backward transport port. Transactions received from the producer, on the forward path, are sent back to the producer on the backward path using this non-blocking transport port The port can be bound to one export.

Summary

uvm_tlm_nb_transport_bw_port

Class providing the non-blocking backward transport port.

CLASS HIERARCHY

uvm_port_base#(uvm_tlm_if#(T,P))

uvm_tlm_nb_transport_bw_port

CLASS DECLARATION

```
class uvm_tlm_nb_transport_bw_port #(
   type T = uvm_tlm_generic_payload,
   type P = uvm_tlm_phase_e
) extends uvm_port_base #(uvm_tlm_if #(T,P))
```

METHODS

new

METHODS

new

TLM2 Export Classes

This section defines the export classes for connecting TLM2 interfaces.

Contents

uvm_tlm_b_transport_export

Blocking transport export class.

Summary

uvm_tlm_b_transport_export

Blocking transport export class.

Class Hierarchy

uvm_port_base#(uvm_tlm_if#(T))

uvm_tlm_b_transport_export

Class Declaration

```
class uvm_tlm_b_transport_export #(
   type T = uvm_tlm_generic_payload
) extends uvm_port_base #(uvm_tlm_if #(T))
```

uvm_tlm_nb_transport_fw_export

Non-blocking forward transport export class

Summary

uvm_tlm_nb_transport_fw_export

Non-blocking forward transport export class

Class Hierarchy

uvm_port_base#(uvm_tlm_if#(T,P))

uvm_tlm_nb_transport_fw_export

Class Declaration

```
class uvm_tlm_nb_transport_fw_export #(
   type T = uvm_tlm_generic_payload,
   type P = uvm_tlm_phase_e
) extends uvm_port_base #(uvm_tlm_if #(T,P))
```

uvm_tlm_nb_transport_bw_export

Non-blocking backward transport export class

Summary

uvm_tlm_nb_transport_bw_export

Non-blocking backward transport export class

Class Hierarchy

```
uvm_port_base#(uvm_tlm_if#(T,P))
```

uvm_tlm_nb_transport_bw_export

Class Declaration

```
class uvm_tlm_nb_transport_bw_export #(
   type T = uvm_tlm_generic_payload,
   type P = uvm_tlm_phase_e
) extends uvm_port_base #(uvm_tlm_if #(T,P))
```

Methods

Methods

new

TLM2 imps (interface implementations)

This section defines the implementation classes for connecting TLM2 interfaces.

TLM imps bind a TLM interface with the object that contains the interface implementation. In addition to the transaction type and the phase type, the imps are parameterized with the type of the object that will provide the implementation. Most often this will be the type of the component where the imp resides. The constructor of the imp takes as an argument an object of type IMP and installs it as the implementation object. Most often the imp constructor argument is "this".

Contents

IMP BINDING MACROS

`UVM_TLM_NB_TRANSPORT_FW_IMP

The macro wraps the forward path call function nb_transport_fw()

The first call to this method for a transaction marks the initial timing point. Every call to this method may mark a timing point in the execution of the transaction. The timing annotation argument allows the timing points to be offset from the simulation times at which the forward path is used. The final timing point of a transaction may be marked by a call to nb_transport_bw() within `UVM_TLM_NB_TRANSPORT_BW_IMP or a return from this or subsequent call to nb_transport_fw().

See TLM2 Interfaces, Ports, Exports and Transport Interfaces Subset for more details on the semantics and rules of the nonblocking transport interface.

`UVM_TLM_NB_TRANSPORT_BW_IMP

Implementation of the backward path. The macro wraps the function called

nb_transport_bw(). This function MUST be implemented in the INITIATOR component class.

Every call to this method may mark a timing point, including the final timing point, in the execution of the transaction. The timing annotation argument allows the timing point to be offset from the simulation times at which the backward path is used. The final timing point of a transaction may be marked by a call to nb_transport_fw() within `UVM_TLM_NB_TRANSPORT_FW_IMP or a return from this or subsequent call to nb_transport_bw().

See TLM2 Interfaces, Ports, Exports and Transport Interfaces Subset for more details on the semantics and rules of the nonblocking transport interface.

Example

```
class master extends uvm_component;
   uvm_tlm_nb_initiator_socket
         #(trans, uvm_tlm_phase_e, this_t) initiator_socket;

   function void build_phase(uvm_phase phase);
      initiator_socket = new("initiator_socket", this, this);
   endfunction

   function uvm_tlm_sync_e nb_transport_bw(trans t,
                                   ref uvm_tlm_phase_e p,
                                   input uvm_tlm_time delay);
       transaction = t;
       state = p;
       return UVM_TLM_ACCEPTED;
   endfunction

   ...
endclass
```

`UVM_TLM_B_TRANSPORT_IMP

The macro wraps the function b_transport() Execute a blocking transaction. Once this method returns, the transaction is assumed to have been executed. Whether that execution is succesful or not must be indicated by the transaction itself.

The callee may modify or update the transaction object, subject to any constraints imposed by the transaction class. The initiator may re-use a transaction object from one call to the next and across calls to b_transport().

The call to b_transport shall mark the first timing point of the transaction. The return from b_transport() shall mark the final timing point of the transaction. The timing annotation argument allows the timing points to be offset from the simulation times at which the task call and return are executed.

IMP BINDING CLASSES

uvm_tlm_b_transport_imp

Used like exports, except an addtional class parameter specifices the type of the implementation object. When the imp is instantiated the implementation object is bound.

Summary

uvm_tlm_b_transport_imp

Used like exports, except an addtional class parameter specifices the type of the implementation object.

Class Hierarchy

uvm_port_base#(uvm_tlm_if#(T))

uvm_tlm_b_transport_imp

Class Declaration

```
class uvm_tlm_b_transport_imp #(
   type T   = uvm_tlm_generic_payload,
   type IMP = int
) extends uvm_port_base #(uvm_tlm_if #(T))
```

uvm_tlm_nb_transport_fw_imp

Used like exports, except an addtional class parameter specifices the type of the implementation object. When the imp is instantiated the implementation object is bound.

Summary

uvm_tlm_nb_transport_fw_imp

Used like exports, except an addtional class parameter specifices the type of the implementation object.

Class Hierarchy

uvm_port_base#(uvm_tlm_if#(T,P))

uvm_tlm_nb_transport_fw_imp

Class Declaration

```
class uvm_tlm_nb_transport_fw_imp #(
   type T   = uvm_tlm_generic_payload,
   type P   = uvm_tlm_phase_e,
   type IMP = int
) extends uvm_port_base #(uvm_tlm_if #(T,P))
```

uvm_tlm_nb_transport_bw_imp

Used like exports, except an addtional class parameter specifices the type of the implementation object. When the imp is instantiated the implementation object is bound.

Summary

uvm_tlm_nb_transport_bw_imp

Used like exports, except an addtional class parameter specifices the type of the implementation object.

Class Hierarchy

uvm_port_base#(uvm_tlm_if#(T,P))

uvm_tlm_nb_transport_bw_imp

Class Declaration

```
class uvm_tlm_nb_transport_bw_imp #(
   type T   = uvm_tlm_generic_payload,
   type P   = uvm_tlm_phase_e,
   type IMP = int
) extends uvm_port_base #(uvm_tlm_if #(T,P))
```

Interface Masks

Each of the following macros is a mask that identifies which interfaces a particular port requires or export provides. The interfaces are identified by bit position and can be or'ed together for combination ports/exports. The mask is used to do run-time interface type checking of port/export connections.

Summary

Interface Masks

Each of the following macros is a mask that identifies which interfaces a particular port requires or export provides.

MACROS

`UVM_TLM_NB_FW_MASK	Define Non blocking Forward mask onehot assignment = 'b001
`UVM_TLM_NB_BW_MASK	Define Non blocking backward mask onehot assignment = 'b010
`UVM_TLM_B_MASK	Define blocking mask onehot assignment = 'b100

MACROS

`UVM_TLM_NB_FW_MASK

Define Non blocking Forward mask onehot assignment = 'b001

`UVM_TLM_NB_BW_MASK

Define Non blocking backward mask onehot assignment = 'b010

`UVM_TLM_B_MASK

Define blocking mask onehot assignment = 'b100

TLM Socket Base Classes

A collection of base classes, one for each socket type. The reason for having a base class for each socket is that all the socket (base) types must be known before connect is defined. Socket connection semantics are provided in the derived classes, which are user visible.

Termination Sockets	A termination socket must be the terminus of every TLM path. A transaction originates with an initator socket and ultimately ends up in a target socket. There may be zero or more passthrough sockets between initiator and target.
Passthrough Sockets	Passthrough initiators are ports and contain exports for instance IS-A port and HAS-A export. Passthrough targets are the opposite, they are exports and contain ports.

Contents

uvm_tlm_b_target_socket_base

IS-A forward imp; has no backward path except via the payload contents.

Summary

uvm_tlm_b_target_socket_base

IS-A forward imp; has no backward path except via the payload contents.

Class Hierarchy

uvm_port_base#(uvm_tlm_if#(T))

uvm_tlm_b_target_socket_base

Class Declaration

```
class uvm_tlm_b_target_socket_base #(
   type T = uvm_tlm_generic_payload
) extends uvm_port_base #(uvm_tlm_if #(T))
```

uvm_tlm_b_initiator_socket_base

IS-A forward port; has no backward path except via the payload contents

Summary

uvm_tlm_b_initiator_socket_base

IS-A forward port; has no backward path except via the payload contents

Class Hierarchy

```
uvm_port_base#(uvm_tlm_if#(T))
```

uvm_tlm_b_initiator_socket_base

Class Declaration

```
class uvm_tlm_b_initiator_socket_base #(
   type T = uvm_tlm_generic_payload
) extends uvm_port_base #(uvm_tlm_if #(T))
```

uvm_tlm_nb_target_socket_base

IS-A forward imp; HAS-A backward port

Summary

uvm_tlm_nb_target_socket_base

IS-A forward imp; HAS-A backward port

Class Hierarchy

```
uvm_port_base#(uvm_tlm_if#(T,P))
```

uvm_tlm_nb_target_socket_base

Class Declaration

```
class uvm_tlm_nb_target_socket_base #(
   type T = uvm_tlm_generic_payload,
   type P = uvm_tlm_phase_e
) extends uvm_port_base #(uvm_tlm_if #(T,P))
```

uvm_tlm_nb_initiator_socket_base

IS-A forward port; HAS-A backward imp

Summary

uvm_tlm_nb_initiator_socket_base

IS-A forward port; HAS-A backward imp

Class Hierarchy

uvm_port_base#(uvm_tlm_if#(T,P))

uvm_tlm_nb_initiator_socket_base

Class Declaration

```
class uvm_tlm_nb_initiator_socket_base #(
   type T = uvm_tlm_generic_payload,
   type P = uvm_tlm_phase_e
) extends uvm_port_base #(uvm_tlm_if #(T,P))
```

uvm_tlm_nb_passthrough_initiator_socket_base

IS-A forward port; HAS-A backward export

Summary

uvm_tlm_nb_passthrough_initiator_socket_base

IS-A forward port; HAS-A backward export

Class Hierarchy

uvm_port_base#(uvm_tlm_if#(T,P))

uvm_tlm_nb_passthrough_initiator_socket_base

Class Declaration

```
class uvm_tlm_nb_passthrough_initiator_socket_base #(
    type  T =  uvm_tlm_generic_payload,
    type  P =  uvm_tlm_phase_e
) extends uvm_port_base #(uvm_tlm_if #(T,P))
```

uvm_tlm_nb_passthrough_target_socket_base

IS-A forward export; HAS-A backward port

Summary

uvm_tlm_nb_passthrough_target_socket_base

IS-A forward export; HAS-A backward port

Class Hierarchy

uvm_port_base#(uvm_tlm_if#(T,P))

uvm_tlm_nb_passthrough_target_socket_base

Class Declaration

```
class uvm_tlm_nb_passthrough_target_socket_base #(
    type  T = uvm_tlm_generic_payload,
    type  P = uvm_tlm_phase_e
) extends uvm_port_base #(uvm_tlm_if #(T,P))
```

uvm_tlm_b_passthrough_initiator_socket_base

IS-A forward port

Summary

uvm_tlm_b_passthrough_initiator_socket_base

IS-A forward port

Class Hierarchy

uvm_port_base#(uvm_tlm_if#(T))

uvm_tlm_b_passthrough_initiator_socket_base

Class Declaration

```
class uvm_tlm_b_passthrough_initiator_socket_base #(
    type  T = uvm_tlm_generic_payload
) extends uvm_port_base #(uvm_tlm_if #(T))
```

uvm_tlm_b_passthrough_target_socket_base

IS-A forward export

Summary

uvm_tlm_b_passthrough_target_socket_base

IS-A forward export

Class Hierarchy

uvm_port_base#(uvm_tlm_if#(T))

uvm_tlm_b_passthrough_target_socket_base

Class Declaration

```
class uvm_tlm_b_passthrough_target_socket_base #(
    type  T = uvm_tlm_generic_payload
) extends uvm_port_base #(uvm_tlm_if #(T))
```

uvm_tlm_time

Canonical time type that can be used in different timescales

This time type is used to represent time values in a canonical form that can bridge initiators and targets located in different timescales and time precisions.

For a detailed explanation of the purpose for this class, see Why is this necessary.

Summary

uvm_tlm_time

Canonical time type that can be used in different timescales

Class Declaration

```
class uvm_tlm_time
```

set_time_resolution	Set the default canonical time resolution.
new	Create a new canonical time value.
get_name	Return the name of this instance
reset	Reset the value to 0
get_realtime	Return the current canonical time value, scaled for the caller's timescale
incr	Increment the time value by the specified number of scaled time unit
decr	Decrement the time value by the specified number of scaled time unit
get_abstime	Return the current canonical time value, in the number of specified time unit, reguardless of the current timescale of the caller.
set_abstime	Set the current canonical time value, to the number of specified time unit, reguardless of the current timescale of the caller.
Why is this necessary	Integers are not sufficient, on their own, to represent time without any ambiguity: you need to know the scale of that integer value.

set_time_resolution

```
static function void set_time_resolution(real res)
```

Set the default canonical time resolution.

Must be a power of 10. When co-simulating with SystemC, it is recommended that default canonical time resolution be set to the SystemC time resolution.

By default, the default resolution is 1.0e-12 (ps)

new

```
function new(string name = "uvm_tlm_time",
             real   res  = 0              )
```

Create a new canonical time value.

The new value is initialized to 0. If a resolution is not specified, the default resolution,

as specified by set_time_resolution(), is used.

get_name

```
function string get_name()
```

Return the name of this instance

reset

```
function void reset()
```

Reset the value to 0

get_realtime

```
function real get_realtime(time scaled,
                           real secs    = 1.0e-9)
```

Return the current canonical time value, scaled for the caller's timescale

scaled must be a time literal value that corresponds to the number of seconds specified in *secs* (1ns by default). It must be a time literal value that is greater or equal to the current timescale.

```
#(delay.get_realtime(1ns));
#(delay.get_realtime(1fs, 1.0e-15));
```

incr

```
function void incr(real t,
                   time scaled,
                   real secs    = 1.0e-9)
```

Increment the time value by the specified number of scaled time unit

t is a time value expressed in the scale and precision of the caller. *scaled* must be a time literal value that corresponds to the number of seconds specified in *secs* (1ns by default). It must be a time literal value that is greater or equal to the current timescale.

```
delay.incr(1.5ns, 1ns);
delay.incr(1.5ns, 1ps, 1.0e-12);
```

decr

```
function void decr(real t,
                   time scaled,
                   real secs   )
```

Decrement the time value by the specified number of scaled time unit

t is a time value expressed in the scale and precision of the caller. *scaled* must be a time literal value that corresponds to the number of seconds specified in *secs* (1ns by default). It must be a time literal value that is greater or equal to the current timescale.

```
delay.decr(200ps, 1ns);
```

get_abstime

```
function real get_abstime(real secs)
```

Return the current canonical time value, in the number of specified time unit, reguardless of the current timescale of the caller.

secs is the number of seconds in the desired time unit e.g. 1e-9 for nanoseconds.

```
$write("%.3f ps\n", delay.get_abstime(1e-12));
```

set_abstime

```
function void set_abstime(real t,
                          real secs)
```

Set the current canonical time value, to the number of specified time unit, reguardless of the current timescale of the caller.

secs is the number of seconds in the time unit in the value *t* e.g. 1e-9 for nanoseconds.

```
delay.set_abstime(1.5, 1e-12));
```

WHY IS THIS NECESSARY

Integers are not sufficient, on their own, to represent time without any ambiguity: you need to know the scale of that integer value. That scale is information conveyed outside of that integer. In SystemVerilog, it is based on the timescale that was active when the code was compiled. SystemVerilog properly scales time literals, but not integer values. That's because it does not know the difference between an integer that carries an integer value and an integer that carries a time value. The 'time' variables are simply 64-bit integers, they are not scaled back and forth to the underlying precision.

```
`timescale 1ns/1ps

module m();

time t;

initial
begin
   #1.5;
   $write("T=%f ns (1.5)\n", $realtime());
   t = 1.5;
   #t;
```

```
    $write("T=%f ns (3.0)\n", $realtime());
    #10ps;
    $write("T=%f ns (3.010)\n", $realtime());
    t = 10ps;
    #t;
    $write("T=%f ns (3.020)\n", $realtime());
end
endmodule
```

yields

```
T=1.500000 ns (1.5)
T=3.500000 ns (3.0)
T=3.510000 ns (3.010)
T=3.510000 ns (3.020)
```

Within SystemVerilog, we have to worry about

- different time scale
- different time precision

Because each endpoint in a socket could be coded in different packages and thus be executing under different timescale directives, a simple integer cannot be used to exchange time information across a socket.

For example

```
`timescale 1ns/1ps

package a_pkg;

class a;
   function void f(inout time t);
      t += 10ns;
   endfunction
endclass

endpackage


`timescale 1ps/1ps

program p;

import a_pkg::*;

time t = 0;

initial
begin
   a A = new;
   A.f(t);
   #t;
   $write("T=%0d ps (10,000)\n", $realtime());
end
endprogram
```

yeilds

```
T=10 ps (10,000)
```

Scaling is needed everytime you make a procedural call to code that may interpret a time value in a different timescale.

Using the uvm_tlm_time type

```
`timescale 1ns/1ps

   package a_pkg;

import uvm_pkg::*;

class a;
   function void f(uvm_tlm_time t);
      t.incr(10ns, 1ns);
   endfunction
endclass

endpackage


`timescale 1ps/1ps

program p;

import uvm_pkg::*;
import a_pkg::*;

uvm_tlm_time t = new;

initial
   begin
      a A = new;
      A.f(t);
      #(t.get_realtime(1ns));
      $write("T=%0d ps (10,000)\n", $realtime());
end
endprogram
```

yields

```
T=10000 ps (10,000)
```

A similar procedure is required when crossing any simulator or language boundary, such as interfacing between SystemVerilog and SystemC.

Sequence Item Pull Ports

This section defines the port, export, and imp port classes for communicating sequence items between uvm_sequencer #(REQ,RSP) and uvm_driver #(REQ,RSP).

Contents

uvm_seq_item_pull_port #(REQ,RSP)

UVM provides a port, export, and imp connector for use in sequencer-driver communication. All have standard port connector constructors, except that uvm_seq_item_pull_port's default min_size argument is 0; it can be left unconnected.

Summary

uvm_seq_item_pull_port #(REQ,RSP)

UVM provides a port, export, and imp connector for use in sequencer-driver communication.

Class Hierarchy

uvm_port_base#(uvm_sqr_if_base#(REQ,RSP))

uvm_seq_item_pull_port#(REQ,RSP)

Class Declaration

```
class uvm_seq_item_pull_port #(
    type  REQ  =  int,
    type  RSP  =  REQ
) extends uvm_port_base #(uvm_sqr_if_base #(REQ, RSP))
```

uvm_seq_item_pull_export #(REQ,RSP)

This export type is used in sequencer-driver communication. It has the standard constructor for exports.

Summary

uvm_seq_item_pull_export #(REQ,RSP)

This export type is used in sequencer-driver communication.

Class Hierarchy

uvm_port_base#(uvm_sqr_if_base#(REQ,RSP))

uvm_seq_item_pull_export#(REQ,RSP)

Class Declaration

```
class uvm_seq_item_pull_export #(
    type   REQ  =  int,
    type   RSP  =  REQ
) extends uvm_port_base #(uvm_sqr_if_base #(REQ, RSP))
```

uvm_seq_item_pull_imp #(REQ,RSP,IMP)

This imp type is used in sequencer-driver communication. It has the standard constructor for imp-type ports.

Summary

uvm_seq_item_pull_imp #(REQ,RSP,IMP)

This imp type is used in sequencer-driver communication.

Class Hierarchy

uvm_port_base#(uvm_sqr_if_base#(REQ,RSP))

uvm_seq_item_pull_imp#(REQ,RSP,IMP)

Class Declaration

```
class uvm_seq_item_pull_imp #(
    type  REQ  =  int,
    type  RSP  =  REQ,
    type  IMP  =  int
) extends uvm_port_base #(uvm_sqr_if_base #(REQ, RSP))
```

Methods

new

Methods

new

uvm_sqr_if_base #(REQ,RSP)

This class defines an interface for sequence drivers to communicate with sequencers. The driver requires the interface via a port, and the sequencer implements it and provides it via an export.

Summary

uvm_sqr_if_base #(REQ,RSP)

This class defines an interface for sequence drivers to communicate with sequencers.

CLASS DECLARATION

```
virtual class uvm_sqr_if_base #(type T1 = uvm_object,
                                     T2 = T1          )
```

METHODS

get_next_item	Retrieves the next available item from a sequence.
try_next_item	Retrieves the next available item from a sequence if one is available.
item_done	Indicates that the request is completed to the sequencer.
wait_for_sequences	Waits for a sequence to have a new item available.
has_do_available	Indicates whether a sequence item is available for immediate processing.
get	Retrieves the next available item from a sequence.
peek	Returns the current request item if one is in the sequencer fifo.
put	Sends a response back to the sequence that issued the request.

METHODS

get_next_item

```
virtual task get_next_item(output T1 t)
```

Retrieves the next available item from a sequence. The call will block until an item is available. The following steps occur on this call:

1. Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, wait for a requesting unlocked relevant sequence, then re-arbitrate.
2. The chosen sequence will return from wait_for_grant
3. The chosen sequence uvm_sequence_base::pre_do is called
4. The chosen sequence item is randomized
5. The chosen sequence uvm_sequence_base::post_do is called
6. Return with a reference to the item

Once get_next_item is called, item_done must be called to indicate the completion of the request to the sequencer. This will remove the request item from the sequencer fifo.

try_next_item

```
virtual task try_next_item(output T1 t)
```

Retrieves the next available item from a sequence if one is available. Otherwise, the function returns immediately with request set to null. The following steps occur on this call:

1. Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, return null.
2. The chosen sequence will return from wait_for_grant
3. The chosen sequence uvm_sequence_base::pre_do is called
4. The chosen sequence item is randomized
5. The chosen sequence uvm_sequence_base::post_do is called
6. Return with a reference to the item

Once try_next_item is called, item_done must be called to indicate the completion of the request to the sequencer. This will remove the request item from the sequencer fifo.

item_done

```
virtual function void item_done(input T2 t = null)
```

Indicates that the request is completed to the sequencer. Any uvm_sequence_base::wait_for_item_done calls made by a sequence for this item will return.

The current item is removed from the sequencer fifo.

If a response item is provided, then it will be sent back to the requesting sequence. The response item must have it's sequence ID and transaction ID set correctly, using the uvm_sequence_item::set_id_info method:

```
rsp.set_id_info(req);
```

Before item_done is called, any calls to peek will retrieve the current item that was obtained by get_next_item. After item_done is called, peek will cause the sequencer to arbitrate for a new item.

wait_for_sequences

```
virtual task wait_for_sequences()
```

Waits for a sequence to have a new item available. The default implementation in the sequencer delays <uvm_sequencer_base::pound_zero_count> delta cycles. User-derived sequencers may override its wait_for_sequences implementation to perform some other application-specific implementation.

has_do_available

```
virtual function bit has_do_available()
```

Indicates whether a sequence item is available for immediate processing. Implementations should return 1 if an item is available, 0 otherwise.

get

```
virtual task get(output T1 t)
```

Retrieves the next available item from a sequence. The call blocks until an item is available. The following steps occur on this call:

1. Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, wait for a requesting unlocked relevant sequence, then re-arbitrate.
2. The chosen sequence will return from uvm_sequence_base::wait_for_grant
3. The chosen sequence uvm_sequence_base::pre_do is called
4. The chosen sequence item is randomized
5. The chosen sequence uvm_sequence_base::post_do is called
6. Indicate item_done to the sequencer
7. Return with a reference to the item

When get is called, item_done may not be called. A new item can be obtained by calling get again, or a response may be sent using either put, or uvm_driver::rsp_port.write().

peek

```
virtual task peek(output T1 t)
```

Returns the current request item if one is in the sequencer fifo. If no item is in the fifo, then the call will block until the sequencer has a new request. The following steps will occur if the sequencer fifo is empty:

1. Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, wait for a requesting unlocked relevant sequence, then re-arbitrate.
2. The chosen sequence will return from uvm_sequence_base::wait_for_grant
3. The chosen sequence uvm_sequence_base::pre_do is called
4. The chosen sequence item is randomized
5. The chosen sequence uvm_sequence_base::post_do is called

Once a request item has been retrieved and is in the sequencer fifo, subsequent calls to peek will return the same item. The item will stay in the fifo until either get or item_done is called.

put

```
virtual task put(input T2 t)
```

Sends a response back to the sequence that issued the request. Before the response is put, it must have it's sequence ID and transaction ID set to match the request. This can

be done using the uvm_sequence_item::set_id_info call:

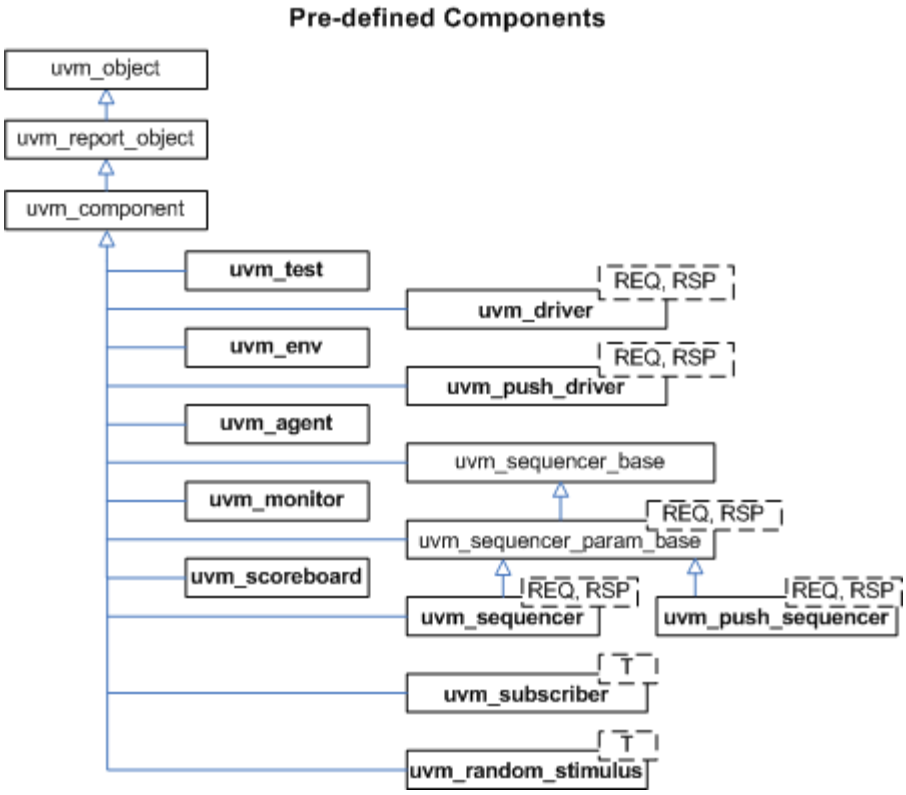
```
rsp.set_id_info(req);
```

This task will not block. The response will be put into the sequence response queue or it will be sent to the sequence response handler.

Predefined Component Classes

Components form the foundation of the UVM. They encapsulate behavior of drivers, scoreboards, and other objects in a testbench. The UVM library provides a set of predefined component types, all derived directly or indirectly from uvm_component.

Predefined Components



Summary

Predefined Component Classes

Components form the foundation of the UVM.

uvm_component

The uvm_component class is the root base class for UVM components. In addition to the features inherited from uvm_object and uvm_report_object, uvm_component provides the following interfaces:

Hierarchy	provides methods for searching and traversing the component hierarchy.
Phasing	defines a phased test flow that all components follow, with a group of standard phase methods and an API for custom phases and multiple independent phasing domains to mirror DUT behavior e.g. power
Configuration	provides methods for configuring component topology and other parameters ahead of and during component construction.
Reporting	provides a convenience interface to the uvm_report_handler. All messages, warnings, and errors are processed through this interface.
Transaction recording	provides methods for recording the transactions produced or consumed by the component to a transaction database (vendor specific).
Factory	provides a convenience interface to the uvm_factory. The factory is used to create new components and other objects based on type-wide and instance-specific configuration.

The uvm_component is automatically seeded during construction using UVM seeding, if enabled. All other objects must be manually reseeded, if appropriate. See uvm_object::reseed for more information.

Summary

uvm_component

The uvm_component class is the root base class for UVM components.

Class Hierarchy

uvm_void
uvm_object
uvm_report_object
uvm_component

Class Declaration

```
virtual class uvm_component extends uvm_report_object
```

new	Creates a new component with the given leaf instance *name* and handle to to its *parent*.
Hierarchy Interface	These methods provide user access to information about the component hierarchy, i.e., topology.
get_parent	Returns a handle to this component's parent, or null if it has no parent.
get_full_name	Returns the full hierarchical name of this object.
get_children	This function populates the end of the *children* array with the list of this component's children.

get_child get_next_child get_first_child	These methods are used to iterate through this component's children, if any.
get_num_children	Returns the number of this component's children.
has_child	Returns 1 if this component has a child with the given *name*, 0 otherwise.
set_name	Renames this component to *name* and recalculates all descendants' full names.
lookup	Looks for a component with the given hierarchical *name* relative to this component.
get_depth	Returns the component's depth from the root level.
Phasing Interface	These methods implement an interface which allows all components to step through a standard schedule of phases, or a customized schedule, and also an API to allow independent phase domains which can jump like state machines to reflect behavior e.g.
build_phase	The Pre-Defined Phases::build_ph phase implementation method.
connect_phase	The Pre-Defined Phases::connect_ph phase implementation method.
end_of_elaboration_phase	The Pre-Defined Phases::end_of_elaboration_ph phase implementation method.
start_of_simulation_phase	The Pre-Defined Phases::start_of_simulation_ph phase implementation method.
run_phase	The Pre-Defined Phases::run_ph phase implementation method.
pre_reset_phase	The Pre-Defined Phases::pre_reset_ph phase implementation method.
reset_phase	The Pre-Defined Phases::reset_ph phase implementation method.
post_reset_phase	The Pre-Defined Phases::post_reset_ph phase implementation method.
pre_configure_phase	The Pre-Defined Phases::pre_configure_ph phase implementation method.
configure_phase	The Pre-Defined Phases::configure_ph phase implementation method.
post_configure_phase	The Pre-Defined Phases::post_configure_ph phase implementation method.
pre_main_phase	The Pre-Defined Phases::pre_main_ph phase implementation method.
main_phase	The Pre-Defined Phases::main_ph phase implementation method.
post_main_phase	The Pre-Defined Phases::post_main_ph phase implementation method.
pre_shutdown_phase	The Pre-Defined Phases::pre_shutdown_ph phase implementation method.
shutdown_phase	The Pre-Defined Phases::shutdown_ph phase implementation method.
post_shutdown_phase	The Pre-Defined Phases::post_shutdown_ph phase implementation method.
extract_phase	The Pre-Defined Phases::extract_ph phase implementation method.
check_phase	The Pre-Defined Phases::check_ph phase implementation method.
report_phase	The Pre-Defined Phases::report_ph phase implementation method.
final_phase	The Pre-Defined Phases::final_ph phase implementation method.
phase_started	Invoked at the start of each phase.
phase_ended	Invoked at the end of each phase.
set_domain	Apply a phase domain to this component (by default, also to it's children).
get_domain	Return handle to the phase domain set on this component

get_schedule	Return handle to the phase schedule graph that applies to this component
define_phase_schedule	Builds and returns the required phase schedule subgraph for this component base
set_phase_imp	Override the default implementation for a phase on this component (tree) with a custom one, which must be created as a singleton object extending the default one and implementing required behavior in exec and traverse methods
suspend	Suspend this component.
resume	Resume this component.
status	Returns the status of this component.
kill	Kills the process tree associated with this component's currently running task-based phase, e.g., run.
do_kill_all	Recursively calls kill on this component and all its descendants, which abruptly ends the currently running task-based phase, e.g., run.
stop	The stop task is called when this component's enable_stop_interrupt bit is set and global_stop_request is called during a task-based phase, e.g., run.
enable_stop_interrupt	This bit allows a component to raise an objection to the stopping of the current phase.
resolve_bindings	Processes all port, export, and imp connections.
Configuration Interface	Components can be designed to be user-configurable in terms of its topology (the type and number of children it has), mode of operation, and run-time parameters (knobs).
set_config_int set_config_string set_config_object	Calling set_config_* causes configuration settings to be created and placed in a table internal to this component.
get_config_int get_config_string get_config_object	These methods retrieve configuration settings made by previous calls to their set_config_* counterparts.
check_config_usage	Check all configuration settings in a components configuration table to determine if the setting has been used, overridden or not used.
apply_config_settings	Searches for all config settings matching this component's instance path.
print_config_settings	Called without arguments, print_config_settings prints all configuration information for this component, as set by previous calls to set_config_*.
print_config	Print_config_settings prints all configuration information for this component, as set by previous calls to set_config_* and exports to the resources pool.
print_config_with_audit	Operates the same as print_config except that the audit bit is forced to 1.
print_config_matches	Setting this static variable causes get_config_* to print info about matching configuration settings as they are being applied.
Objection Interface	These methods provide object level hooks into the uvm_objection mechanism.
raised	The raised callback is called when a decendant of the component instance raises the specfied *objection*.
dropped	The dropped callback is called when a decendant of the component instance raises the specfied *objection*.
all_dropped	The all_dropped callback is called when a decendant of the component instance raises the specfied *objection*.

Factory Interface	The factory interface provides convenient access to a portion of UVM's uvm_factory interface.
create_component	A convenience function for uvm_factory::create_component_by_name, this method calls upon the factory to create a new child component whose type corresponds to the preregistered type name, *requested_type_name*, and instance name, *name*.
create_object	A convenience function for uvm_factory::create_object_by_name, this method calls upon the factory to create a new object whose type corresponds to the preregistered type name, *requested_type_name*, and instance name, *name*.
set_type_override_by_type	A convenience function for uvm_factory::set_type_override_by_type, this method registers a factory override for components and objects created at this level of hierarchy or below.
set_inst_override_by_type	A convenience function for uvm_factory::set_inst_override_by_type, this method registers a factory override for components and objects created at this level of hierarchy or below.
set_type_override	A convenience function for uvm_factory::set_type_override_by_name, this method configures the factory to create an object of type *override_type_name* whenever the factory is asked to produce a type represented by *original_type_name*.
set_inst_override	A convenience function for uvm_factory::set_inst_override_by_type, this method registers a factory override for components created at this level of hierarchy or below.
print_override_info	This factory debug method performs the same lookup process as create_object and create_component, but instead of creating an object, it prints information about what type of object would be created given the provided arguments.
Hierarchical Reporting Interface	This interface provides versions of the set_report_* methods in the uvm_report_object base class that are applied recursively to this component and all its children.
set_report_id_verbosity_hier set_report_severity_id_verbosity_hier	These methods recursively associate the specified verbosity with reports of the given *severity*, *id*, or *severity-id* pair.
set_report_severity_action_hier set_report_id_action_hier set_report_severity_id_action_hier	These methods recursively associate the specified action with reports of the given *severity*, *id*, or *severity-id* pair.
set_report_default_file_hier set_report_severity_file_hier set_report_id_file_hier set_report_severity_id_file_hier	These methods recursively associate the specified FILE descriptor with reports of the given *severity*, *id*, or *severity-id* pair.
set_report_verbosity_level_hier	This method recursively sets the maximum verbosity level for reports for this component and all those below it.
pre_abort	This callback is executed when the message system is executing a UVM_EXIT action.
Recording Interface	These methods comprise the component-based

	transaction recording interface.
accept_tr	This function marks the acceptance of a transaction, *tr*, by this component.
do_accept_tr	The accept_tr method calls this function to accommodate any user-defined post-accept action.
begin_tr	This function marks the start of a transaction, *tr*, by this component.
begin_child_tr	This function marks the start of a child transaction, *tr*, by this component.
do_begin_tr	The begin_tr and begin_child_tr methods call this function to accommodate any user-defined post-begin action.
end_tr	This function marks the end of a transaction, *tr*, by this component.
do_end_tr	The end_tr method calls this function to accommodate any user-defined post-end action.
record_error_tr	This function marks an error transaction by a component.
record_event_tr	This function marks an event transaction by a component.
print_enabled	This bit determines if this component should automatically be printed as a child of its parent object.
if overriding this method, always follow this pattern	only build a new schedule if one of that name does not yet exist under this domain to augment this base schedule, use result of super.define_phase_schedule(domain,MYNAME);

new

```
function new (string        name,
              uvm_component parent)
```

Creates a new component with the given leaf instance *name* and handle to to its *parent*. If the component is a top-level component (i.e. it is created in a static module or interface), *parent* should be null.

The component will be inserted as a child of the *parent* object, if any. If *parent* already has a child by the given *name*, an error is produced.

If *parent* is null, then the component will become a child of the implicit top-level component, *uvm_top*.

All classes derived from uvm_component must call super.new(name,parent).

Hierarchy Interface

These methods provide user access to information about the component hierarchy, i.e., topology.

get_parent

```
virtual function uvm_component get_parent ()
```

Returns a handle to this component's parent, or null if it has no parent.

get_full_name

```
virtual function string get_full_name ()
```

Returns the full hierarchical name of this object. The default implementation concatenates the hierarchical name of the parent, if any, with the leaf name of this object, as given by uvm_object::get_name.

get_children

```
function void get_children(ref uvm_component children[$])
```

This function populates the end of the *children* array with the list of this component's children.

```
uvm_component array[$];
my_comp.get_children(array);
foreach(array[i])
  do_something(array[i]);
```

get_child

```
function uvm_component get_child (string name)
```

get_next_child

```
function int get_next_child (ref string name)
```

get_first_child

```
function int get_first_child (ref string name)
```

These methods are used to iterate through this component's children, if any. For example, given a component with an object handle, *comp*, the following code calls uvm_object::print for each child:

```
string name;
uvm_component child;
if (comp.get_first_child(name))
  do begin
    child = comp.get_child(name);
    child.print();
  end while (comp.get_next_child(name));
```

get_num_children

```
function int get_num_children ()
```

Returns the number of this component's children.

has_child

```
function int has_child (string name)
```

Returns 1 if this component has a child with the given *name*, 0 otherwise.

set_name

```
virtual function void set_name (string name)
```

Renames this component to *name* and recalculates all descendants' full names.

lookup

```
function uvm_component lookup (string name)
```

Looks for a component with the given hierarchical *name* relative to this component. If the given *name* is preceded with a '.' (dot), then the search begins relative to the top level (absolute lookup). The handle of the matching component is returned, else null. The name must not contain wildcards.

get_depth

```
function int unsigned get_depth()
```

Returns the component's depth from the root level. uvm_top has a depth of 0. The test and any other top level components have a depth of 1, and so on.

Phasing Interface

These methods implement an interface which allows all components to step through a standard schedule of phases, or a customized schedule, and also an API to allow independent phase domains which can jump like state machines to reflect behavior e.g. power domains on the DUT in different portions of the testbench. The phase tasks and functions are the phase name with the _phase suffix. For example, the build phase function is build_phase.

All processes associated with a task-based phase are killed when the phase ends. See <uvm_phase::execute> for more details.

build_phase

```
virtual function void build_phase(uvm_phase phase)
```

The Pre-Defined Phases::build_ph phase implementation method.

Any override should call super.build_phase(phase) to execute the automatic configuration of fields registed in the component by calling apply_config_settings. To turn off automatic configuration for a component, do not call super.build_phase(phase).

This method should never be called directly.

connect_phase

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

The Pre-Defined Phases::connect_ph phase implementation method.

This method should never be called directly.

end_of_elaboration_phase

```
virtual function void end_of_elaboration_phase(uvm_phase phase)
```

The Pre-Defined Phases::end_of_elaboration_ph phase implementation method.

This method should never be called directly.

start_of_simulation_phase

```
virtual function void start_of_simulation_phase(uvm_phase phase)
```

The Pre-Defined Phases::start_of_simulation_ph phase implementation method.

This method should never be called directly.

run_phase

```
virtual task run_phase(uvm_phase phase)
```

The Pre-Defined Phases::run_ph phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. Unlike other task phases, it is not necessary to raise an objection to cause it to persist: it will persists until global_stop_request() is called. However, if a single phase objection is raised using *phase.raise_objection()*, then the phase will automatically ends once all objections are dropped using *phase.drop_objection()*.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

The run_phase task should never be called directly.

pre_reset_phase

```
virtual task pre_reset_phase(uvm_phase phase)
```

The Pre-Defined Phases::pre_reset_ph phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()*, or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

reset_phase

```
virtual task reset_phase(uvm_phase phase)
```

The Pre-Defined Phases::reset_ph phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()*, or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

post_reset_phase

```
virtual task post_reset_phase(uvm_phase phase)
```

The Pre-Defined Phases::post_reset_ph phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()*, or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

pre_configure_phase

```
virtual task pre_configure_phase(uvm_phase phase)
```

The Pre-Defined Phases::pre_configure_ph phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()*, or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

configure_phase

```
virtual task configure_phase(uvm_phase phase)
```

The Pre-Defined Phases::configure_ph phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()*, or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

post_configure_phase

```
virtual task post_configure_phase(uvm_phase phase)
```

The Pre-Defined Phases::post_configure_ph phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()*, or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

pre_main_phase

```
virtual task pre_main_phase(uvm_phase phase)
```

The Pre-Defined Phases::pre_main_ph phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()*, or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

main_phase

```
virtual task main_phase(uvm_phase phase)
```

The Pre-Defined Phases::main_ph phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()*, or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

post_main_phase

```
virtual task post_main_phase(uvm_phase phase)
```

The Pre-Defined Phases::post_main_ph phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()*, or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

pre_shutdown_phase

```
virtual task pre_shutdown_phase(uvm_phase phase)
```

The Pre-Defined Phases::pre_shutdown_ph phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()*, or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

shutdown_phase

```
virtual task shutdown_phase(uvm_phase phase)
```

The Pre-Defined Phases::shutdown_ph phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()*, or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

post_shutdown_phase

```
virtual task post_shutdown_phase(uvm_phase phase)
```

The Pre-Defined Phases::post_shutdown_ph phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to

persist. Once all components have dropped their respective objection using *phase.drop_objection()*, or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

extract_phase

```
virtual function void extract_phase(uvm_phase phase)
```

The Pre-Defined Phases::extract_ph phase implementation method.

This method should never be called directly.

check_phase

```
virtual function void check_phase(uvm_phase phase)
```

The Pre-Defined Phases::check_ph phase implementation method.

This method should never be called directly.

report_phase

```
virtual function void report_phase(uvm_phase phase)
```

The Pre-Defined Phases::report_ph phase implementation method.

This method should never be called directly.

final_phase

```
virtual function void final_phase(uvm_phase phase)
```

The Pre-Defined Phases::final_ph phase implementation method.

This method should never be called directly.

phase_started

```
virtual function void phase_started (uvm_phase phase)
```

Invoked at the start of each phase. The *phase* argument specifies the phase being started. Any threads spawned in this callback are not affected when the phase ends.

phase_ended

```
virtual function void phase_ended (uvm_phase phase)
```

Invoked at the end of each phase. The *phase* argument specifies the phase that is

ending. Any threads spawned in this callback are not affected when the phase ends.

set_domain

```
function void set_domain(uvm_domain domain,
                         int        hier    = 1)
```

Apply a phase domain to this component (by default, also to it's children). Get a copy of the schedule graph for this component base class as defined by virtual define_phase_schedule(), and add an instance of that to our domain branch in the master phasing schedule graph, if it does not already exist.

get_domain

```
function uvm_domain get_domain()
```

Return handle to the phase domain set on this component

get_schedule

```
function uvm_phase get_schedule()
```

Return handle to the phase schedule graph that applies to this component

define_phase_schedule

```
virtual protected function uvm_phase define_phase_schedule(uvm_domain domain,
                                                           string     name
```

Builds and returns the required phase schedule subgraph for this component base

Here we define the structure and organization of a schedule for this component base type (uvm_component). We give that schedule a name (default 'uvm') and return a handle to it to the caller (either the set_domain() method, or a subclass's define_phase_schedule() having called super.define_phase_schedule(), ready to be added into the main schedule graph.

Custom component base classes requiring a custom phasing schedule to augment or replace the default UVM schedule can override this method. They can inherit the parent schedule and build on it by calling super.define_phase_schedule(MYNAME) or they can create a new schedule from scratch by not calling the super method.

set_phase_imp

```
function void set_phase_imp(uvm_phase phase,
                            uvm_phase imp,
                            int       hier   = 1)
```

Override the default implementation for a phase on this component (tree) with a custom one, which must be created as a singleton object extending the default one and implementing required behavior in exec and traverse methods

The *hier* specifies whether to apply the custom functor to the whole tree or just this

component.

suspend

```
virtual task suspend ()
```

Suspend this component.

This method must be implemented by the user to suspend the component according to the protocol and functionality it implements. A suspended component can be subsequently resumed using resume().

resume

```
virtual task resume ()
```

Resume this component.

This method must be implemented by the user to resume a component that was previously suspended using suspend(). Some component may start in the suspended state and may need to be explicitly resumed.

status

```
function string status ()
```

Returns the status of this component.

Returns a string that describes the current status of the components. Possible values include, but are not limited to

"<unknown>"	Status is unknown (default)
"FINISHED"	Component has stopped on its own accord. May be resumed.
"RUNNING"	Component is running. May be suspended after normal completion of operation in progress.
"WAITING"	Component is waiting. May be suspended immediately.
"SUSPENDED"	Component is suspended. May be resumed.
"KILLED"	Component has been killed and is unable to operate any further. It cannot be resumed.

kill

```
virtual function void kill ()
```

Kills the process tree associated with this component's currently running task-based phase, e.g., run.

An alternative mechanism for stopping the run phase is the stop request. Calling global_stop_request causes all components' run_phase processes to be killed, but only after all components have had the opportunity to complete in progress transactions and shutdown cleanly via their stop tasks.

do_kill_all

```
virtual function void do_kill_all ()
```

Recursively calls kill on this component and all its descendants, which abruptly ends the currently running task-based phase, e.g., run. See run_phase for better options to ending a task-based phase.

stop

```
virtual task stop (string ph_name)
```

The stop task is called when this component's enable_stop_interrupt bit is set and global_stop_request is called during a task-based phase, e.g., run.

Before a phase is abruptly ended, e.g., when a test deems the simulation complete, some components may need extra time to shut down cleanly. Such components may implement stop to finish the currently executing transaction, flush the queue, or perform other cleanup. Upon return from its stop, a component signals it is ready to be stopped.

The stop method will not be called if enable_stop_interrupt is 0.

The default implementation of stop is empty, i.e., it will return immediately.

This method should never be called directly.

enable_stop_interrupt

```
int enable_stop_interrupt = 0
```

This bit allows a component to raise an objection to the stopping of the current phase. It affects only time consuming phases (such as the run phase).

When this bit is set, the stop task in the component is called as a result of a call to global_stop_request. Components that are sensitive to an immediate killing of its run-time processes should set this bit and implement the stop task to prepare for shutdown.

resolve_bindings

```
virtual function void resolve_bindings ()
```

Processes all port, export, and imp connections. Checks whether each port's min and max connection requirements are met.

It is called just before the end_of_elaboration phase.

Users should not call directly.

Configuration Interface

Components can be designed to be user-configurable in terms of its topology (the type and number of children it has), mode of operation, and run-time parameters (knobs). The configuration interface accommodates this common need, allowing component composition and state to be modified without having to derive new classes or new class hierarchies for every configuration scenario.

set_config_int

```
virtual function void set_config_int (string          inst_name,
                                      string          field_name,
                                      uvm_bitstream_t value      )
```

set_config_string

```
virtual function void set_config_string (string inst_name,
                                         string field_name,
                                         string value      )
```

set_config_object

```
virtual function void set_config_object (string     inst_name,
                                         string     field_name,
                                         uvm_object value,
                                         bit        clone       = 1)
```

Calling set_config_* causes configuration settings to be created and placed in a table internal to this component. There are similar global methods that store settings in a global table. Each setting stores the supplied *inst_name*, *field_name*, and *value* for later use by descendent components during their construction. (The global table applies to all components and takes precedence over the component tables.)

When a descendant component calls a get_config_* method, the *inst_name* and *field_name* provided in the get call are matched against all the configuration settings stored in the global table and then in each component in the parent hierarchy, top-down. Upon the first match, the value stored in the configuration setting is returned. Thus, precedence is global, following by the top-level component, and so on down to the descendent component's parent.

These methods work in conjunction with the get_config_* methods to provide a configuration setting mechanism for integral, string, and uvm_object-based types. Settings of other types, such as virtual interfaces and arrays, can be indirectly supported by defining a class that contains them.

Both *inst_name* and *field_name* may contain wildcards.

- For set_config_int, *value* is an integral value that can be anything from 1 bit to 4096 bits.
- For set_config_string, *value* is a string.
- For set_config_object, *value* must be an uvm_object-based object or null. Its clone argument specifies whether the object should be cloned. If set, the object is cloned both going into the table (during the set) and coming out of the table (during the get), so that multiple components matched to the same setting (by way of wildcards) do not end up sharing the same object.

The following message tags are used for configuration setting. You can use the standard uvm report messaging interface to control these messages. CFGNTS -- The configuration setting was not used by any component. This is a warning. CFGOVR -- The configuration setting was overridden by a setting above. CFGSET -- The configuration setting was used at least once.

See get_config_int, get_config_string, and get_config_object for information on getting the configurations set by these methods.

get_config_int

```
virtual function bit get_config_int (      string          field_name,
                                     inout uvm_bitstream_t value      )
```

get_config_string

```
virtual function bit get_config_string (      string field_name,
                                        inout string value      )
```

get_config_object

```
virtual function bit get_config_object (      string     field_name,
                                        inout uvm_object value,
                                        input bit        clone       = 1)
```

These methods retrieve configuration settings made by previous calls to their set_config_* counterparts. As the methods' names suggest, there is direct support for integral types, strings, and objects. Settings of other types can be indirectly supported by defining an object to contain them.

Configuration settings are stored in a global table and in each component instance. With each call to a get_config_* method, a top-down search is made for a setting that matches this component's full name and the given *field_name*. For example, say this component's full instance name is top.u1.u2. First, the global configuration table is searched. If that fails, then it searches the configuration table in component 'top', followed by top.u1.

The first instance/field that matches causes *value* to be written with the value of the configuration setting and 1 is returned. If no match is found, then *value* is unchanged and the 0 returned.

Calling the get_config_object method requires special handling. Because *value* is an output of type uvm_object, you must provide an uvm_object handle to assign to (not a derived class handle). After the call, you can then $cast to the actual type.

For example, the following code illustrates how a component designer might call upon the configuration mechanism to assign its *data* object property, whose type myobj_t derives from uvm_object.

```
class mycomponent extends uvm_component;

  local myobj_t data;

  function void build_phase(uvm_phase phase);
    uvm_object tmp;
    super.build_phase(phase);
    if(get_config_object("data", tmp))
      if (!$cast(data, tmp))
        $display("error! config setting for 'data' not of type myobj_t");
    endfunction
  ...
```

The above example overrides the build_phase method. If you want to retain any base functionality, you must call super.build_phase(uvm_phase phase).

The *clone* bit clones the data inbound. The get_config_object method can also clone the data outbound.

See Members for information on setting the global configuration table.

check_config_usage

```
function void check_config_usage (bit recurse = 1)
```

Check all configuration settings in a components configuration table to determine if the setting has been used, overridden or not used. When *recurse* is 1 (default), configuration for this and all child components are recursively checked. This function is automatically called in the check phase, but can be manually called at any time.

Additional detail is provided by the following message tags

- CFGOVR -- lists all configuration settings that have been overridden from above.
- CFGSET -- lists all configuration settings that have been set.

To get all configuration information prior to the run phase, do something like this in your top object:

```
function void start_of_simulation_phase(uvm_phase phase);
  set_report_id_action_hier("CFGOVR", UVM_DISPLAY);
  set_report_id_action_hier("CFGSET", UVM_DISPLAY);
  check_config_usage();
endfunction
```

apply_config_settings

```
virtual function void apply_config_settings (bit verbose = )
```

Searches for all config settings matching this component's instance path. For each match, the appropriate set_*_local method is called using the matching config setting's field_name and value. Provided the set_*_local method is implemented, the component property associated with the field_name is assigned the given value.

This function is called by uvm_component::build_phase.

The apply_config_settings method determines all the configuration settings targeting this component and calls the appropriate set_*_local method to set each one. To work, you must override one or more set_*_local methods to accommodate setting of your component's specific properties. Any properties registered with the optional `uvm_*_field macros do not require special handling by the set_*_local methods; the macros provide the set_*_local functionality for you.

If you do not want apply_config_settings to be called for a component, then the build_phase() method should be overloaded and you should not call super.build_phase(phase). Likewise, apply_config_settings can be overloaded to customize automated configuration.

When the *verbose* bit is set, all overrides are printed as they are applied. If the component's print_config_matches property is set, then apply_config_settings is automatically called with *verbose* = 1.

print_config_settings

```
function void print_config_settings (string        field   = "",
                                     uvm_component comp    = null,
                                     bit           recurse = 0    )
```

Called without arguments, print_config_settings prints all configuration information for this component, as set by previous calls to set_config_*. The settings are printing in the order of their precedence.

If *field* is specified and non-empty, then only configuration settings matching that field, if any, are printed. The field may not contain wildcards.

If *comp* is specified and non-null, then the configuration for that component is printed.

If *recurse* is set, then configuration information for all *comp*'s children and below are printed as well.

This function has been deprecated. Use print_config instead.

print_config

```
function void print_config(bit recurse = 0,
                           bit audit   = 0 )
```

Print_config_settings prints all configuration information for this component, as set by previous calls to set_config_* and exports to the resources pool. The settings are printing in the order of their precedence.

If *recurse* is set, then configuration information for all children and below are printed as well.

if *audit* is set then the audit trail for each resource is printed along with the resource name and value

print_config_with_audit

```
function void print_config_with_audit(bit recurse = 0)
```

Operates the same as print_config except that the audit bit is forced to 1. This interface makes user code a bit more readable as it avoids multiple arbitrary bit settings in the argument list.

If *recurse* is set, then configuration information for all children and below are printed as well.

print_config_matches

```
static bit print_config_matches = 0
```

Setting this static variable causes get_config_* to print info about matching configuration settings as they are being applied.

Objection Interface

These methods provide object level hooks into the uvm_objection mechanism.

raised

```
virtual function void raised (uvm objection objection,
```

```
                                uvm_object    source_obj,
                                string        description,
                                int           count       )
```

The raised callback is called when a decendant of the component instance raises the specfied *objection*. The *source_obj* is the object which originally raised the object. *count* is an optional count that was used to indicate a number of objections which were raised.

dropped

```
virtual function void dropped (uvm_objection objection,
                               uvm_object    source_obj,
                               string        description,
                               int           count       )
```

The dropped callback is called when a decendant of the component instance raises the specfied *objection*. The *source_obj* is the object which originally dropped the object. *count* is an optional count that was used to indicate a number of objections which were dropped.

all_dropped

```
virtual task all_dropped (uvm_objection objection,
                          uvm_object    source_obj,
                          string        description,
                          int           count       )
```

The all_dropped callback is called when a decendant of the component instance raises the specfied *objection*. The *source_obj* is the object which originally all_dropped the object. *count* is an optional count that was used to indicate a number of objections which were dropped. This callback is time-consuming and the all_dropped conditional will not be propagated up to the object's parent until the callback returns.

Factory Interface

The factory interface provides convenient access to a portion of UVM's uvm_factory interface. For creating new objects and components, the preferred method of accessing the factory is via the object or component wrapper (see uvm_component_registry #(T,Tname) and uvm_object_registry #(T,Tname)). The wrapper also provides functions for setting type and instance overrides.

create_component

```
function uvm_component create_component (string requested_type_name,
                                         string name                 )
```

A convenience function for uvm_factory::create_component_by_name, this method calls upon the factory to create a new child component whose type corresponds to the preregistered type name, *requested_type_name*, and instance name, *name*. This method is equivalent to:

```
factory.create_component_by_name(requested_type_name,
                                 get_full_name(), name, this);
```

If the factory determines that a type or instance override exists, the type of the component created may be different than the requested type. See set_type_override and set_inst_override. See also uvm_factory for details on factory operation.

create_object

```
function uvm_object create_object (string requested_type_name,
                                   string name                 = "")
```

A convenience function for uvm_factory::create_object_by_name, this method calls upon the factory to create a new object whose type corresponds to the preregistered type name, *requested_type_name*, and instance name, *name*. This method is equivalent to:

```
factory.create_object_by_name(requested_type_name,
                              get_full_name(), name);
```

If the factory determines that a type or instance override exists, the type of the object created may be different than the requested type. See uvm_factory for details on factory operation.

set_type_override_by_type

```
static function void set_type_override_by_type (
   uvm_object_wrapper original_type,
   uvm_object_wrapper override_type,
   bit                replace        = 1
)
```

A convenience function for uvm_factory::set_type_override_by_type, this method registers a factory override for components and objects created at this level of hierarchy or below. This method is equivalent to:

```
factory.set_type_override_by_type(original_type, override_type,replace);
```

The *relative_inst_path* is relative to this component and may include wildcards. The *original_type* represents the type that is being overridden. In subsequent calls to uvm_factory::create_object_by_type or uvm_factory::create_component_by_type, if the requested_type matches the *original_type* and the instance paths match, the factory will produce the *override_type*.

The original and override type arguments are lightweight proxies to the types they represent. See set_inst_override_by_type for information on usage.

set_inst_override_by_type

```
function void set_inst_override_by_type(string             relative_inst_path
                                        uvm_object_wrapper original_type,
                                        uvm_object_wrapper override_type
```

A convenience function for uvm_factory::set_inst_override_by_type, this method registers a factory override for components and objects created at this level of hierarchy or below. In typical usage, this method is equivalent to:

```
factory.set_inst_override_by_type({get_full_name(),".",
                                   relative_inst_path},
                                   original_type,
                                   override_type);
```

The *relative_inst_path* is relative to this component and may include wildcards. The *original_type* represents the type that is being overridden. In subsequent calls to uvm_factory::create_object_by_type or uvm_factory::create_component_by_type, if the requested_type matches the *original_type* and the instance paths match, the factory will produce the *override_type*.

The original and override types are lightweight proxies to the types they represent. They can be obtained by calling *type::get_type()*, if implemented by *type*, or by directly calling *type::type_id::get()*, where *type* is the user type and *type_id* is the name of the typedef to uvm_object_registry #(T,Tname) or uvm_component_registry #(T,Tname).

If you are employing the `uvm_*_utils macros, the typedef and the get_type method will be implemented for you. For details on the utils macros refer to Utility and Field Macros for Components and Objects.

The following example shows `uvm_*_utils usage

```
class comp extends uvm_component;
  `uvm_component_utils(comp)
  ...
endclass

class mycomp extends uvm_component;
  `uvm_component_utils(mycomp)
  ...
endclass

class block extends uvm_component;
  `uvm_component_utils(block)
  comp c_inst;
  virtual function void build_phase(uvm_phase phase);
    set_inst_override_by_type("c_inst",comp::get_type(),
                                       mycomp::get_type());
  endfunction
  ...
endclass
```

set_type_override

```
static function void set_type_override(string original_type_name,
                                       string override_type_name,
                                       bit    replace             = 1)
```

A convenience function for uvm_factory::set_type_override_by_name, this method configures the factory to create an object of type *override_type_name* whenever the factory is asked to produce a type represented by *original_type_name*. This method is equivalent to:

```
factory.set_type_override_by_name(original_type_name,
                                  override_type_name, replace);
```

The *original_type_name* typically refers to a preregistered type in the factory. It may, however, be any arbitrary string. Subsequent calls to create_component or create_object with the same string and matching instance path will produce the type

represented by override_type_name. The *override_type_name* must refer to a preregistered type in the factory.

set_inst_override

```
function void set_inst_override(string relative_inst_path,
                                string original_type_name,
                                string override_type_name )
```

A convenience function for uvm_factory::set_inst_override_by_type, this method registers a factory override for components created at this level of hierarchy or below. In typical usage, this method is equivalent to:

```
factory.set_inst_override_by_name({get_full_name(),".",
                                   relative_inst_path},
                                   original_type_name,
                                  override_type_name);
```

The *relative_inst_path* is relative to this component and may include wildcards. The *original_type_name* typically refers to a preregistered type in the factory. It may, however, be any arbitrary string. Subsequent calls to create_component or create_object with the same string and matching instance path will produce the type represented by *override_type_name*. The *override_type_name* must refer to a preregistered type in the factory.

print_override_info

```
function void print_override_info(string requested_type_name,
                                  string name                 = "")
```

This factory debug method performs the same lookup process as create_object and create_component, but instead of creating an object, it prints information about what type of object would be created given the provided arguments.

Hierarchical Reporting Interface

This interface provides versions of the set_report_* methods in the uvm_report_object base class that are applied recursively to this component and all its children.

When a report is issued and its associated action has the LOG bit set, the report will be sent to its associated FILE descriptor.

set_report_id_verbosity_hier

```
function void set_report_id_verbosity_hier (string id,
                                            int    verbosity)
```

set_report_severity_id_verbosity_hier

```
function void set_report_severity_id_verbosity_hier(uvm_severity severity,
                                                    string       id,
                                                    int          verbosity)
```

These methods recursively associate the specified verbosity with reports of the given *severity*, *id*, or *severity-id* pair. An verbosity associated with a particular severity-id pair takes precedence over an verbosity associated with id, which takes precedence over an an verbosity associated with a severity.

For a list of severities and their default verbosities, refer to uvm_report_handler.

set_report_severity_action_hier

```
function void set_report_severity_action_hier (uvm_severity severity,
                                               uvm_action   action   )
```

set_report_id_action_hier

```
function void set_report_id_action_hier (string     id,
                                         uvm_action action)
```

set_report_severity_id_action_hier

```
function void set_report_severity_id_action_hier(uvm_severity severity,
                                                 string       id,
                                                 uvm_action   action   )
```

These methods recursively associate the specified action with reports of the given *severity*, *id*, or *severity-id* pair. An action associated with a particular severity-id pair takes precedence over an action associated with id, which takes precedence over an an action associated with a severity.

For a list of severities and their default actions, refer to uvm_report_handler.

set_report_default_file_hier

```
function void set_report_default_file_hier (UVM_FILE file)
```

set_report_severity_file_hier

```
function void set_report_severity_file_hier (uvm_severity severity,
                                             UVM_FILE     file     )
```

set_report_id_file_hier

```
function void set_report_id_file_hier (string   id,
                                       UVM_FILE file)
```

set_report_severity_id_file_hier

```
function void set_report_severity_id_file_hier(uvm_severity severity,
                                               string       id,
                                               UVM_FILE     file     )
```

These methods recursively associate the specified FILE descriptor with reports of the

given *severity*, *id*, or *severity-id* pair. A FILE associated with a particular severity-id pair takes precedence over a FILE associated with id, which take precedence over an a FILE associated with a severity, which takes precedence over the default FILE descriptor.

For a list of severities and other information related to the report mechanism, refer to uvm_report_handler.

set_report_verbosity_level_hier

```
function void set_report_verbosity_level_hier (int verbosity)
```

This method recursively sets the maximum verbosity level for reports for this component and all those below it. Any report from this component subtree whose verbosity exceeds this maximum will be ignored.

See uvm_report_handler for a list of predefined message verbosity levels and their meaning.

pre_abort

```
virtual function void pre_abort
```

This callback is executed when the message system is executing a UVM_EXIT action. The exit action causes an immediate termination of the simulation, but the pre_abort callback hook gives components an opportunity to provide additional information to the user before the termination happens. For example, a test may want to executed the report function of a particular component even when an error condition has happened to force a premature termination you would write a function like:

```
function void mycomponent::pre_abort();
  report();
endfunction
```

Recording Interface

These methods comprise the component-based transaction recording interface. The methods can be used to record the transactions that this component "sees", i.e. produces or consumes.

The API and implementation are subject to change once a vendor-independent use-model is determined.

accept_tr

```
function void accept_tr (uvm_transaction tr,
                         time            accept_time = )
```

This function marks the acceptance of a transaction, *tr*, by this component. Specifically, it performs the following actions:

- Calls the *tr*'s uvm_transaction::accept_tr method, passing to it the *accept_time* argument.
- Calls this component's do_accept_tr method to allow for any post-begin action in

derived classes.

- Triggers the component's internal accept_tr event. Any processes waiting on this event will resume in the next delta cycle.

do_accept_tr

```
virtual protected function void do_accept_tr (uvm_transaction tr)
```

The accept_tr method calls this function to accommodate any user-defined post-accept action. Implementations should call super.do_accept_tr to ensure correct operation.

begin_tr

```
function integer begin_tr (uvm_transaction tr,
                           string          stream_name = "main",
                           string          label       = "",
                           string          desc        = "",
                           time            begin_time  = 0      )
```

This function marks the start of a transaction, *tr*, by this component. Specifically, it performs the following actions:

- Calls *tr*'s uvm_transaction::begin_tr method, passing to it the *begin_time* argument. The *begin_time* should be greater than or equal to the accept time. By default, when *begin_time* = 0, the current simulation time is used.

If recording is enabled (recording_detail != UVM_OFF), then a new database-transaction is started on the component's transaction stream given by the stream argument. No transaction properties are recorded at this time.

- Calls the component's do_begin_tr method to allow for any post-begin action in derived classes.
- Triggers the component's internal begin_tr event. Any processes waiting on this event will resume in the next delta cycle.

A handle to the transaction is returned. The meaning of this handle, as well as the interpretation of the arguments *stream_name*, *label*, and *desc* are vendor specific.

begin_child_tr

```
function integer begin_child_tr (uvm_transaction tr,
                                 integer         parent_handle = 0,
                                 string          stream_name   = "main",
                                 string          label         = "",
                                 string          desc          = "",
                                 time            begin_time    = 0      )
```

This function marks the start of a child transaction, *tr*, by this component. Its operation is identical to that of begin_tr, except that an association is made between this transaction and the provided parent transaction. This association is vendor-specific.

do_begin_tr

```
virtual protected function void do_begin_tr (uvm_transaction tr,
                                             string          stream_name,
                                             integer         tr_handle   )
```

The begin_tr and begin_child_tr methods call this function to accommodate any user-

defined post-begin action. Implementations should call super.do_begin_tr to ensure correct operation.

end_tr

```
function void end_tr (uvm_transaction tr,
                      time            end_time    = 0,
                      bit             free_handle = 1 )
```

This function marks the end of a transaction, *tr*, by this component. Specifically, it performs the following actions:

- Calls *tr*'s uvm_transaction::end_tr method, passing to it the *end_time* argument. The *end_time* must at least be greater than the begin time. By default, when *end_time* = 0, the current simulation time is used.

The transaction's properties are recorded to the database-transaction on which it was started, and then the transaction is ended. Only those properties handled by the transaction's do_record method (and optional `uvm_*_field macros) are recorded.

- Calls the component's do_end_tr method to accommodate any post-end action in derived classes.
- Triggers the component's internal end_tr event. Any processes waiting on this event will resume in the next delta cycle.

The *free_handle* bit indicates that this transaction is no longer needed. The implementation of free_handle is vendor-specific.

do_end_tr

```
virtual protected function void do_end_tr (uvm_transaction tr,
                                           integer         tr_handle)
```

The end_tr method calls this function to accommodate any user-defined post-end action. Implementations should call super.do_end_tr to ensure correct operation.

record_error_tr

```
function integer record_error_tr (string     stream_name = "main",
                                  uvm_object info        = null,
                                  string     label       = "error_tr",
                                  string     desc        = "",
                                  time       error_time  = 0,
                                  bit        keep_active = 0          )
```

This function marks an error transaction by a component. Properties of the given uvm_object, *info*, as implemented in its uvm_object::do_record method, are recorded to the transaction database.

An *error_time* of 0 indicates to use the current simulation time. The *keep_active* bit determines if the handle should remain active. If 0, then a zero-length error transaction is recorded. A handle to the database-transaction is returned.

Interpretation of this handle, as well as the strings *stream_name*, *label*, and *desc*, are vendor-specific.

record_event_tr

```
function integer record_event_tr (string     stream_name = "main",
                                  uvm_object info        = null,
                                  string     label       = "event_tr",
                                  string     desc        = "",
                                  time       event_time  = 0,
                                  bit        keep_active = 0          )
```

This function marks an event transaction by a component.

An *event_time* of 0 indicates to use the current simulation time.

A handle to the transaction is returned. The *keep_active* bit determines if the handle may be used for other vendor-specific purposes.

The strings for *stream_name*, *label*, and *desc* are vendor-specific identifiers for the transaction.

print_enabled

```
bit print_enabled = 1
```

This bit determines if this component should automatically be printed as a child of its parent object.

By default, all children are printed. However, this bit allows a parent component to disable the printing of specific children.

if overriding this method, always follow this pattern

only build a new schedule if one of that name does not yet exist under this domain to augment this base schedule, use result of
super.define_phase_schedule(domain,MYNAME);

Callbacks Classes

This section defines the classes used for callback registration, management, and user-defined callbacks.

Contents

uvm_callbacks #(T,CB)

The *uvm_callbacks* class provides a base class for implementing callbacks, which are typically used to modify or augment component behavior without changing the component class. To work effectively, the developer of the component class defines a set of "hook" methods that enable users to customize certain behaviors of the component in a manner that is controlled by the component developer. The integrity of the component's overall behavior is intact, while still allowing certain customizable actions by the user.

To enable compile-time type-safety, the class is parameterized on both the user-defined callback interface implementation as well as the object type associated with the callback. The object type-callback type pair are associated together using the `uvm_register_cb macro to define a valid pairing; valid pairings are checked when a user attempts to add a callback to an object.

To provide the most flexibility for end-user customization and reuse, it is recommended that the component developer also define a corresponding set of virtual method hooks in the component itself. This affords users the ability to customize via inheritance/factory overrides as well as callback object registration. The implementation of each virtual method would provide the default traversal algorithm for the particular callback being called. Being virtual, users can define subtypes that override the default algorithm, perform tasks before and/or after calling super.<method> to execute any registered callbacks, or to not call the base implementation, effectively disabling that particalar hook. A demonstration of this methodology is provided in an example included in the kit.

Summary

uvm_callbacks #(T,CB)

The *uvm_callbacks* class provides a base class for implementing callbacks, which are typically used to modify or augment component behavior without changing the component class.

T	This type parameter specifies the base object type with which the CB callback objects will be registered.

CB	This type parameter specifies the base callback type that will be managed by this callback class.
ADD/DELETE INTEFACE	
add	Registers the given callback object, *cb*, with the given *obj* handle.
add_by_name	Registers the given callback object, *cb*, with one or more uvm_components.
delete	Deletes the given callback object, *cb*, from the queue associated with the given *obj* handle.
delete_by_name	Removes the given callback object, *cb*, associated with one or more uvm_component callback queues.
ITERATOR INTERFACE	This set of functions provide an iterator interface for callback queues.
get_first	returns the first enabled callback of type CB which resides in the queue for *obj*.
get_last	returns the last enabled callback of type CB which resides in the queue for *obj*.
get_next	returns the next enabled callback of type CB which resides in the queue for *obj*, using *itr* as the starting point.
get_prev	returns the previous enabled callback of type CB which resides in the queue for *obj*, using *itr* as the starting point.
DEBUG	
display	This function displays callback information for *obj*.

T

This type parameter specifies the base object type with which the CB callback objects will be registered. This object must be a derivative of *uvm_object*.

CB

This type parameter specifies the base callback type that will be managed by this callback class. The callback type is typically a interface class, which defines one or more virtual method prototypes that users can override in subtypes. This type must be a derivative of uvm_callback.

ADD/DELETE INTEFACE

add

```
static function void add(T             obj,
                         uvm_callback  cb,
                         uvm_apprepend ordering = UVM_APPEND)
```

Registers the given callback object, *cb*, with the given *obj* handle. The *obj* handle can be null, which allows registration of callbacks without an object context. If *ordreing* is UVM_APPEND (default), the callback will be executed after previously added callbacks, else the callback will be executed ahead of previously added callbacks. The *cb* is the callback handle; it must be non-null, and if the callback has already been added to the object instance then a warning is issued. Note that the CB parameter is optional. For example, the following are equivalent:

```
uvm_callbacks#(my_comp)::add(comp_a, cb);
uvm_callbacks#(my_comp, my_callback)::add(comp_a,cb);
```

add_by_name

```
static function void add_by_name(string        name,
                                 uvm_callback  cb,
                                 uvm_component root,
                                 uvm_apprepend ordering = UVM_APPEND)
```

Registers the given callback object, *cb*, with one or more uvm_components. The components must already exist and must be type T or a derivative. As with add the CB parameter is optional. *root* specifies the location in the component hierarchy to start the search for *name*. See uvm_root::find_all for more details on searching by name.

delete

```
static function void delete(T            obj,
                            uvm_callback cb  )
```

Deletes the given callback object, *cb*, from the queue associated with the given *obj* handle. The *obj* handle can be null, which allows de-registration of callbacks without an object context. The *cb* is the callback handle; it must be non-null, and if the callback has already been removed from the object instance then a warning is issued. Note that the CB parameter is optional. For example, the following are equivalent:

```
uvm_callbacks#(my_comp)::delete(comp_a, cb);
uvm_callbacks#(my_comp, my_callback)::delete(comp_a,cb);
```

delete_by_name

```
static function void delete_by_name(string        name,
                                    uvm_callback  cb,
                                    uvm_component root )
```

Removes the given callback object, *cb*, associated with one or more uvm_component callback queues. As with delete the CB parameter is optional. *root* specifies the location in the component hierarchy to start the search for *name*. See uvm_root::find_all for more details on searching by name.

ITERATOR INTERFACE

This set of functions provide an iterator interface for callback queues. A facade class, uvm_callback_iter is also available, and is the generally preferred way to iterate over callback queues.

get_first

```
static function CB get_first (  ref int itr,
```

```
                                input T    obj )
```

returns the first enabled callback of type CB which resides in the queue for *obj*. If *obj* is null then the typewide queue for T is searched. *itr* is the iterator; it will be updated with a value that can be supplied to get_next to get the next callback object.

If the queue is empty then null is returned.

The iterator class uvm_callback_iter may be used as an alternative, simplified, iterator interface.

get_last

```
static function CB get_last (  ref int itr,
                               input T   obj )
```

returns the last enabled callback of type CB which resides in the queue for *obj*. If *obj* is null then the typewide queue for T is searched. *itr* is the iterator; it will be updated with a value that can be supplied to get_prev to get the previous callback object.

If the queue is empty then null is returned.

The iterator class uvm_callback_iter may be used as an alternative, simplified, iterator interface.

get_next

```
static function CB get_next (  ref int itr,
                               input T   obj )
```

returns the next enabled callback of type CB which resides in the queue for *obj*, using *itr* as the starting point. If *obj* is null then the typewide queue for T is searched. *itr* is the iterator; it will be updated with a value that can be supplied to get_next to get the next callback object.

If no more callbacks exist in the queue, then null is returned. get_next will continue to return null in this case until get_first or get_last has been used to reset the iterator.

The iterator class uvm_callback_iter may be used as an alternative, simplified, iterator interface.

get_prev

```
static function CB get_prev (  ref int itr,
                               input T   obj )
```

returns the previous enabled callback of type CB which resides in the queue for *obj*, using *itr* as the starting point. If *obj* is null then the typewide queue for T is searched. *itr* is the iterator; it will be updated with a value that can be supplied to get_prev to get the previous callback object.

If no more callbacks exist in the queue, then null is returned. get_prev will continue to return null in this case until get_first or get_last has been used to reset the iterator.

The iterator class uvm_callback_iter may be used as an alternative, simplified, iterator interface.

DEBUG

display

```
static function void display(T obj = null)
```

This function displays callback information for *obj*. If *obj* is null, then it displays callback information for all objects of type *T*, including typewide callbacks.

uvm_callback_iter

The *uvm_callback_iter* class is an iterator class for iterating over callback queues of a specific callback type. The typical usage of the class is:

```
uvm_callback_iter#(mycomp,mycb) iter = new(this);
for(mycb cb = iter.first(); cb != null; cb = iter.next())
   cb.dosomething();
```

The callback iteration macros, `uvm_do_callbacks and `uvm_do_callbacks_exit_on provide a simple method for iterating callbacks and executing the callback methods.

Summary

uvm_callback_iter

The *uvm_callback_iter* class is an iterator class for iterating over callback queues of a specific callback type.

CLASS DECLARATION

```
class uvm_callback_iter#(type T  = uvm_object,
                         type CB = uvm_callback)
```

METHODS

new	Creates a new callback iterator object.
first	Returns the first valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object.
last	Returns the last valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object.
next	Returns the next valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object.
prev	Returns the previous valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object.
get_cb	Returns the last callback accessed via a first() or next() call.

METHODS

new

```
function new(T obj)
```

Creates a new callback iterator object. It is required that the object context be provided.

first

```
function CB first()
```

Returns the first valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object. If the queue is empty then null is returned.

last

```
function CB last()
```

Returns the last valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object. If the queue is empty then null is returned.

next

```
function CB next()
```

Returns the next valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object. If there are no more valid callbacks in the queue, then null is returned.

prev

```
function CB prev()
```

Returns the previous valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object. If there are no more valid callbacks in the queue, then null is returned.

get_cb

```
function CB get_cb()
```

Returns the last callback accessed via a first() or next() call.

uvm_callback

The *uvm_callback* class is the base class for user-defined callback classes. Typically, the component developer defines an application-specific callback class that extends from this class. In it, he defines one or more virtual methods, called a *callback interface*, that represent the hooks available for user override.

Methods intended for optional override should not be declared *pure.* Usually, all the callback methods are defined with empty implementations so users have the option of overriding any or all of them.

The prototypes for each hook method are completely application specific with no restrictions.

Summary

uvm_callback

The *uvm_callback* class is the base class for user-defined callback classes.

Class Hierarchy

uvm_void

uvm_object

uvm_callback

Class Declaration

```
class uvm_callback extends uvm_object
```

Methods

new	Creates a new uvm_callback object, giving it an optional *name*.
callback_mode	Enable/disable callbacks (modeled like rand_mode and constraint_mode).
is_enabled	Returns 1 if the callback is enabled, 0 otherwise.
get_type_name	Returns the type name of this callback object.

Methods

new

```
function new(string name = "uvm_callback")
```

Creates a new uvm_callback object, giving it an optional *name*.

callback_mode

```
function bit callback_mode(int on = -1)
```

Enable/disable callbacks (modeled like rand_mode and constraint_mode).

is_enabled

```
function bit is_enabled()
```

Returns 1 if the callback is enabled, 0 otherwise.

get_type_name

```
virtual function string get_type_name()
```

Returns the type name of this callback object.

uvm_test

This class is the virtual base class for the user-defined tests.

The uvm_test virtual class should be used as the base class for user-defined tests. Doing so provides the ability to select which test to execute using the UVM_TESTNAME command line or argument to the uvm_root::run_test task.

For example

```
prompt> SIM_COMMAND +UVM_TESTNAME=test_bus_retry
```

The global run_test() task should be specified inside an initial block such as

```
initial run_test();
```

Multiple tests, identified by their type name, are compiled in and then selected for execution from the command line without need for recompilation. Random seed selection is also available on the command line.

If +UVM_TESTNAME=test_name is specified, then an object of type 'test_name' is created by factory and phasing begins. Here, it is presumed that the test will instantiate the test environment, or the test environment will have already been instantiated before the call to run_test().

If the specified test_name cannot be created by the uvm_factory, then a fatal error occurs. If run_test() is called without UVM_TESTNAME being specified, then all components constructed before the call to run_test will be cycled through their simulation phases.

Deriving from uvm_test will allow you to distinguish tests from other component types that inherit from uvm_component directly. Such tests will automatically inherit features that may be added to uvm_test in the future.

Summary

uvm_test

This class is the virtual base class for the user-defined tests.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- **uvm_test**

Class Declaration

```
virtual class uvm_test extends uvm_component
```

Methods

new	Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

METHODS

new

```
function new (string        name,
              uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

uvm_env

The base class for hierarchical containers of other components that together comprise a complete environment. The environment may initially consist of the entire testbench. Later, it can be reused as a sub-environment in even larger system-level environments.

Summary

uvm_env

The base class for hierarchical containers of other components that together comprise a complete environment.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- **uvm_env**

Class Declaration

```
virtual class uvm_env extends uvm_component
```

Methods

new	Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

Methods

new

```
function new (string        name   = "env",
              uvm_component parent = null  )
```

Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

uvm_agent

The uvm_agent virtual class should be used as the base class for the user- defined agents. Deriving from uvm_agent will allow you to distinguish agents from other component types also using its inheritance. Such agents will automatically inherit features that may be added to uvm_agent in the future.

While an agent's build function, inherited from uvm_component, can be implemented to define any agent topology, an agent typically contains three subcomponents: a driver, sequencer, and monitor. If the agent is active, subtypes should contain all three subcomponents. If the agent is passive, subtypes should contain only the monitor.

Summary

uvm_agent

The uvm_agent virtual class should be used as the base class for the user-defined agents.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- **uvm_agent**

Class Declaration

```
virtual class uvm_agent extends uvm_component
```

Methods

new	Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.
get_is_active	Returns UVM_ACTIVE is the agent is acting as an active agent and UVM_PASSIVE if it is acting as a passive agent.

Methods

new

```
function new (string        name,
              uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

The int configuration parameter is_active is used to identify whether this agent should be acting in active or passive mode. This parameter can be set by doing:

```
set_config_int("<path_to_agent>", "is_active", UVM_ACTIVE);
```

get_is_active

```
virtual function uvm_active_passive_enum get_is_active()
```

Returns UVM_ACTIVE is the agent is acting as an active agent and UVM_PASSIVE if it is acting as a passive agent. The default implementation is to just return the is_active flag, but the component developer may override this behavior if a more complex algorithm is needed to determine the active/passive nature of the agent.

uvm_monitor

This class should be used as the base class for user-defined monitors.

Deriving from uvm_monitor allows you to distinguish monitors from generic component types inheriting from uvm_component. Such monitors will automatically inherit features that may be added to uvm_monitor in the future.

Summary

uvm_monitor

This class should be used as the base class for user-defined monitors.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- **uvm_monitor**

Class Declaration

```
virtual class uvm_monitor extends uvm_component
```

Methods

new	Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

Methods

new

```
function new (string        name,
              uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

uvm_scoreboard

The uvm_scoreboard virtual class should be used as the base class for user-defined scoreboards.

Deriving from uvm_scoreboard will allow you to distinguish scoreboards from other component types inheriting directly from uvm_component. Such scoreboards will automatically inherit and benefit from features that may be added to uvm_scoreboard in the future.

Summary

uvm_scoreboard

The uvm_scoreboard virtual class should be used as the base class for user-defined scoreboards.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- **uvm_scoreboard**

Class Declaration

```
virtual class uvm_scoreboard extends uvm_component
```

Methods

new	Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

Methods

new

```
function new (string        name,
              uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

uvm_driver #(REQ,RSP)

The base class for drivers that initiate requests for new transactions via a uvm_seq_item_pull_port. The ports are typically connected to the exports of an appropriate sequencer component.

This driver operates in pull mode. Its ports are typically connected to the corresponding exports in a pull sequencer as follows:

```
driver.seq_item_port.connect(sequencer.seq_item_export);
driver.rsp_port.connect(sequencer.rsp_export);
```

The *rsp_port* needs connecting only if the driver will use it to write responses to the analysis export in the sequencer.

Summary

uvm_driver #(REQ,RSP)

The base class for drivers that initiate requests for new transactions via a uvm_seq_item_pull_port.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- **uvm_driver#(REQ,RSP)**

Class Declaration

```
class uvm_driver #(
   type REQ = uvm_sequence_item,
   type RSP = REQ
) extends uvm_component
```

Ports

seq_item_port	Derived driver classes should use this port to request items from the sequencer.
rsp_port	This port provides an alternate way of sending responses back to the originating sequencer.

Methods

new	Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

Ports

seq_item_port

Derived driver classes should use this port to request items from the sequencer. They may also use it to send responses back.

rsp_port

This port provides an alternate way of sending responses back to the originating sequencer. Which port to use depends on which export the sequencer provides for connection.

Methods

new

```
function new (string        name,
              uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

uvm_push_driver #(REQ,RSP)

Base class for a driver that passively receives transactions, i.e. does not initiate requests transactions. Also known as *push* mode. Its ports are typically connected to the corresponding ports in a push sequencer as follows:

```
push_sequencer.req_port.connect(push_driver.req_export);
push_driver.rsp_port.connect(push_sequencer.rsp_export);
```

The *rsp_port* needs connecting only if the driver will use it to write responses to the analysis export in the sequencer.

Summary

uvm_push_driver #(REQ,RSP)

Base class for a driver that passively receives transactions, i.e.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- **uvm_push_driver#(REQ,RSP)**

Class Declaration

```
class uvm_push_driver #(
   type REQ = uvm_sequence_item,
   type RSP = REQ
) extends uvm_component
```

Ports

req_export	This export provides the blocking put interface whose default implementation produces an error.
rsp_port	This analysis port is used to send response transactions back to the originating sequencer.

Methods

new	Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

Ports

req_export

This export provides the blocking put interface whose default implementation produces an error. Derived drivers must override *put* with an appropriate implementation (and not call super.put). Ports connected to this export will supply the driver with transactions.

rsp_port

This analysis port is used to send response transactions back to the originating sequencer.

METHODS

new

```
function new (string        name,
              uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

uvm_random_stimulus #(T)

A general purpose unidirectional random stimulus class.

The uvm_random_stimulus class generates streams of T transactions. These streams may be generated by the randomize method of T, or the randomize method of one of its subclasses. The stream may go indefinitely, until terminated by a call to stop_stimulus_generation, or we may specify the maximum number of transactions to be generated.

By using inheritance, we can add directed initialization or tidy up after random stimulus generation. Simply extend the class and define the run task, calling super.run() when you want to begin the random stimulus phase of simulation.

While very useful in its own right, this component can also be used as a template for defining other stimulus generators, or it can be extended to add additional stimulus generation methods and to simplify test writing.

Summary

uvm_random_stimulus #(T)

A general purpose unidirectional random stimulus class.

Class Hierarchy

uvm_void
uvm_object
uvm_report_object
uvm_component
uvm_random_stimulus#(T)

Class Declaration

```
class uvm_random_stimulus #(
   type T = uvm_transaction
) extends uvm_component
```

Ports

blocking_put_port	The blocking_put_port is used to send the generated stimulus to the rest of the testbench.

Methods

new	Creates a new instance of a specialization of this class.
generate_stimulus	Generate up to max_count transactions of type T.
stop_stimulus_generation	Stops the generation of stimulus.

Ports

blocking_put_port

The blocking_put_port is used to send the generated stimulus to the rest of the testbench.

METHODS

new

```
function new(string        name,
             uvm_component parent)
```

Creates a new instance of a specialization of this class. Also, displays the random state obtained from a get_randstate call. In subsequent simulations, set_randstate can be called with the same value to reproduce the same sequence of transactions.

generate_stimulus

```
virtual task generate_stimulus(T   t         = null,
                               int max_count = 0    )
```

Generate up to max_count transactions of type T. If t is not specified, a default instance of T is allocated and used. If t is specified, that transaction is used when randomizing. It must be a subclass of T.

max_count is the maximum number of transactions to be generated. A value of zero indicates no maximum - in this case, generate_stimulus will go on indefinitely unless stopped by some other process

The transactions are cloned before they are sent out over the blocking_put_port

stop_stimulus_generation

```
virtual function void stop_stimulus_generation
```

Stops the generation of stimulus. If a subclass of this method has forked additional processes, those processes will also need to be stopped in an overridden version of this method

uvm_subscriber

This class provides an analysis export for receiving transactions from a connected analysis export. Making such a connection "subscribes" this component to any transactions emitted by the connected analysis port.

Subtypes of this class must define the write method to process the incoming transactions. This class is particularly useful when designing a coverage collector that attaches to a monitor.

Summary

uvm_subscriber

This class provides an analysis export for receiving transactions from a connected analysis export.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- **uvm_subscriber**

Class Declaration

```
virtual class uvm_subscriber #(
   type T = int
) extends uvm_component
```

Ports

analysis_export	This export provides access to the write method, which derived subscribers must implement.

Methods

new	Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.
write	A pure virtual method that must be defined in each subclass.

Ports

analysis_export

This export provides access to the write method, which derived subscribers must implement.

Methods

new

```
function new (string        name,
              uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

write

```
pure virtual function void write(T t)
```

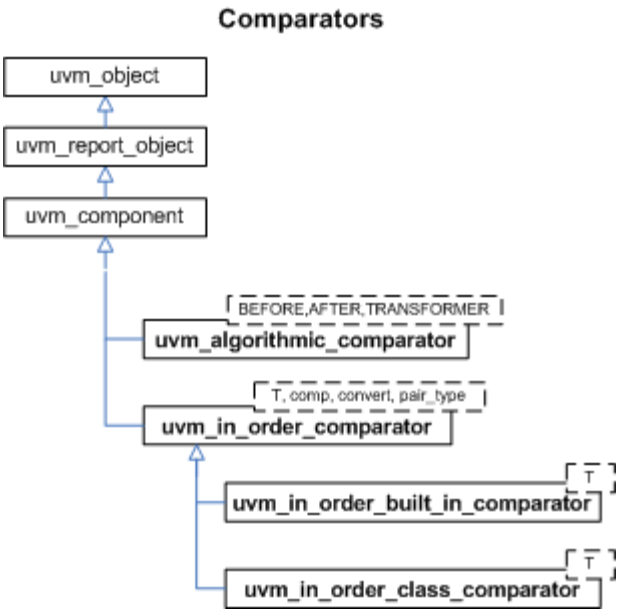
A pure virtual method that must be defined in each subclass. Access to this method by outside components should be done via the analysis_export.

COMPARATORS

A common function of testbenches is to compare streams of transactions for equivalence. For example, a testbench may compare a stream of transactions from a DUT with expected results.

The UVM library provides a base class called *uvm_in_order_comparator* and two derived classes: *uvm_in_order_built_in_comparator* for comparing streams of built-in types and *uvm_in_order_class_comparator* for comparing streams of class objects.

The *uvm_algorithmic_comparator* also compares two streams of transactions, but the transaction streams might be of different type objects. Thus, this comparator will employ a user-defined transformation function to convert one type to another before performing a comparison.



Summary

Comparators

A common function of testbenches is to compare streams of transactions for equivalence.

Comparators

The following classes define comparators for objects and built-in types.

Contents

uvm_in_order_comparator #(T,comp_type,convert,pair_type)

Compares two streams of data objects of the type parameter, T. These transactions may either be classes or built-in types. To be successfully compared, the two streams of data must be in the same order. Apart from that, there are no assumptions made about the relative timing of the two streams of data.

Type parameters

T	Specifies the type of transactions to be compared.
comp_type	A policy class to compare the two transaction streams. It must provide the method "function bit comp(T a, T b)" which returns *TRUE* if *a* and *b* are the same.
convert	A policy class to convert the transactions being compared to a string. It must provide the method "function string convert2string(T a)".
pair_type	A policy class to allow pairs of transactions to be handled as a single uvm_object type.

Built in types (such as ints, bits, logic, and structs) can be compared using the default values for comp_type, convert, and pair_type. For convenience, you can use the subtype, uvm_in_order_built_in_comparator #(T) for built-in types.

When T is a uvm_object, you can use the convenience subtype uvm_in_order_class_comparator #(T).

Comparisons are commutative, meaning it does not matter which data stream is connected to which export, before_export or after_export.

Comparisons are done in order and as soon as a transaction is received from both streams. Internal fifos are used to buffer incoming transactions on one stream until a transaction to compare arrives on the other stream.

Summary

uvm_in_order_comparator #(T,comp_type,convert,pair_type)

Compares two streams of data objects of the type parameter, T.

PORTS

before_export	The export to which one stream of data is written.
after_export	The export to which the other stream of data is written.
pair_ap	The comparator sends out pairs of transactions across this analysis port.

METHODS

flush	This method sets m_matches and m_mismatches back to zero.

PORTS

before_export

The export to which one stream of data is written. The port must be connected to an analysis port that will provide such data.

after_export

The export to which the other stream of data is written. The port must be connected to an analysis port that will provide such data.

pair_ap

The comparator sends out pairs of transactions across this analysis port. Both matched and unmatched pairs are published via a pair_type objects. Any connected analysis export(s) will receive these transaction pairs.

METHODS

flush

```
virtual function void flush()
```

This method sets m_matches and m_mismatches back to zero. The uvm_tlm_fifo::flush takes care of flushing the FIFOs.

uvm_in_order_built_in_comparator #(T)

This class uses the uvm_built_in_* comparison, converter, and pair classes. Use this class for built-in types (int, bit, string, etc.)

Summary

uvm_in_order_built_in_comparator #(T)

This class uses the uvm_built_in_* comparison, converter, and pair classes.

Class Hierarchy

uvm_in_order_comparator#(T)

uvm_in_order_built_in_comparator#(T)

Class Declaration

```
class uvm_in_order_built_in_comparator #(
      type      T  =    int
) extends uvm_in_order_comparator #(T)
```

uvm_in_order_class_comparator #(T)

This class uses the uvm_class_* comparison, converter, and pair classes. Use this class for comparing user-defined objects of type T, which must provide compare() and convert2string() method.

Summary

uvm_in_order_class_comparator #(T)

This class uses the uvm_class_* comparison, converter, and pair classes.

Class Hierarchy

uvm_in_order_comparator#(T,uvm_class_comp#(T),uvm_class_converter#(T),uvm_class_pair#(T,T))

uvm_in_order_class_comparator#(T)

Class Declaration

```
class uvm_in_order_class_comparator #(
      type      T  =    int
) extends uvm_in_order_comparator #( T , uvm_class_comp #( T ) , uvm_class_converter #(
T ) , uvm_class_pair #( T, T ) )
```

uvm_algorithmic_comparator.svh

Summary

uvm_algorithmic_comparator.svh

COMPARATORS A common function of testbenches is to compare streams of transactions for equivalence.

COMPARATORS

A common function of testbenches is to compare streams of transactions for equivalence. For example, a testbench may compare a stream of transactions from a DUT with expected results.

The UVM library provides a base class called uvm_in_order_comparator #(T,comp_type,convert,pair_type) and two derived classes, which are uvm_in_order_built_in_comparator #(T) for comparing streams of built-in types and uvm_in_order_class_comparator #(T) for comparing streams of class objects.

The uvm_algorithmic_comparator also compares two streams of transactions; however, the transaction streams might be of different type objects. This device will use a user-written transformation function to convert one type to another before performing a comparison.

uvm_algorithmic_comparator #(BEFORE,AFTER,TRANSFORMER)

Compares two streams of data objects of different types, *BEFORE* and *AFTER*.

The algorithmic comparator is a wrapper around uvm_in_order_class_comparator #(T). Like the in-order comparator, the algorithmic comparator compares two streams of transactions, the *BEFORE* stream and the *AFTER* stream. It is often the case when two streams of transactions need to be compared that the two streams are in different forms. That is, the type of the *BEFORE* transaction stream is different than the type of the *AFTER* transaction stream.

The uvm_algorithmic_comparator's *TRANSFORMER* type parameter specifies the class responsible for converting transactions of type *BEFORE* into those of type *AFTER*. This transformer class must provide a transform() method with the following prototype:

```
function AFTER transform (BEFORE b);
```

Matches and mistmatches are reported in terms of the *AFTER* transactions. For more information, see the uvm_in_order_comparator #(T,comp_type,convert,pair_type) class.

Summary

uvm_algorithmic_comparator #(BEFORE,AFTER,TRANSFORMER)

Compares two streams of data objects of different types, *BEFORE* and *AFTER*.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_report_object
- uvm_component
- **uvm_algorithmic_comparator#(BEFORE,AFTER,TRANSFORMER)**

Class Declaration

```
class uvm_algorithmic_comparator #(
   type BEFORE      = int,
   type AFTER       = int,
   type TRANSFORMER = int
) extends uvm_component
```

Ports

before_export	The export to which a data stream of type BEFORE is sent via a connected analysis port.
after_export	The export to which a data stream of type AFTER is sent via a connected analysis port.

Methods

new	Creates an instance of a specialization of this class.

Ports

before_export

The export to which a data stream of type BEFORE is sent via a connected analysis port. Publishers (monitors) can send in an ordered stream of transactions against which the transformed BEFORE transactions will (be compared.

after_export

The export to which a data stream of type AFTER is sent via a connected analysis port. Publishers (monitors) can send in an ordered stream of transactions to be transformed and compared to the AFTER transactions.

Methods

new

```
function new(string        name,
             uvm_component parent      = null,
             TRANSFORMER   transformer = null )
```

Creates an instance of a specialization of this class. In addition to the standard uvm_component constructor arguments, *name* and *parent*, the constructor takes a handle to a *transformer* object, which must already be allocated (no null handles) and must implement the transform() method.

uvm_pair classes

This section defines container classes for handling value pairs.

Contents

uvm_pair #(T1,T2)

Container holding handles to two objects whose types are specified by the type parameters, T1 and T2.

Summary

uvm_pair #(T1,T2)

Container holding handles to two objects whose types are specified by the type parameters, T1 and T2.

VARIABLES

T1 first	The handle to the first object in the pair
T2 second	The second variable in the pair

METHODS

new	Creates an instance of uvm_pair that holds two built-in type values.

VARIABLES

T1 first

```
T1 first
```

The handle to the first object in the pair

T2 second

```
T2 second
```

The second variable in the pair

METHODS

new

```
function new (string name = "",
              T1     f    = null,
              T2     s    = null )
```

Creates an instance of uvm_pair that holds two built-in type values. The optional name argument gives a name to the new pair object.

uvm_built_in_pair #(T1,T2)

Container holding two variables of built-in types (int, string, etc.). The types are specified by the type parameters, T1 and T2.

Summary

uvm_built_in_pair #(T1,T2)

Container holding two variables of built-in types (int, string, etc.)

CLASS HIERARCHY

uvm_void

uvm_object

uvm_transaction

uvm_built_in_pair#(T1,T2)

CLASS DECLARATION

```
class uvm_built_in_pair #(
   type T1 = int,
        T2 = T1
) extends uvm_transaction
```

METHODS

new	Creates an instance of uvm_pair that holds a handle to two elements, as provided by the first two arguments.

METHODS

new

```
function new (string name = "")
```

Creates an instance of uvm_pair that holds a handle to two elements, as provided by the first two arguments. The optional name argument gives a name to the new pair object.

Policy Classes

Policy classes are used to implement polymorphic operations that differ between built-in types and class-based types. Generic components can then be built that work with either classes or built-in types, depending on what policy class is used.

Contents

uvm_built_in_comp #(T)

This policy class is used to compare built-in types.

Provides a comp method that compares the built-in type, T, for which the == operator is defined.

Summary

uvm_built_in_comp #(T)

This policy class is used to compare built-in types.

Class Declaration

```
class uvm_built_in_comp #(type T = int)
```

uvm_built_in_converter #(T)

This policy class is used to convert built-in types to strings.

Provides a convert2string method that converts the built-in type, T, to a string using the %p format specifier.

Summary

uvm_built_in_converter #(T)

This policy class is used to convert built-in types to strings.

Class Declaration

```
class uvm_built_in_converter #(type T = int)
```

uvm_built_in_clone #(T)

This policy class is used to clone built-in types via the = operator.

Provides a clone method that returns a copy of the built-in type, T.

Summary

uvm_built_in_clone #(T)

This policy class is used to clone built-in types via the = operator.

Class Declaration

```
class uvm_built_in_clone #(type T = int)
```

uvm_class_comp #(T)

This policy class is used to compare two objects of the same type.

Provides a comp method that compares two objects of type T. The class T must provide the method "function bit compare(T rhs)", similar to the uvm_object::compare method.

Summary

uvm_class_comp #(T)

This policy class is used to compare two objects of the same type.

Class Declaration

```
class uvm_class_comp #(type T = int)
```

uvm_class_converter #(T)

This policy class is used to convert a class object to a string.

Provides a convert2string method that converts an instance of type T to a string. The class T must provide the method "function string convert2string()", similar to the uvm_object::convert2string method.

Summary

uvm_class_converter #(T)

This policy class is used to convert a class object to a string.

CLASS DECLARATION

```
class uvm_class_converter #(type T = int)
```

uvm_class_clone #(T)

This policy class is used to clone class objects.

Provides a clone method that returns a copy of the built-in type, T. The class T must implement the clone method, to which this class delegates the operation. If T is derived from uvm_object, then T must instead implement uvm_object::do_copy, either directly or indirectly through use of the `uvm_field macros.

Summary

uvm_class_clone #(T)

This policy class is used to clone class objects.

CLASS DECLARATION

```
class uvm_class_clone #(type T = int)
```

Report Macros

This set of macros provides wrappers around the uvm_report_* Reporting functions. The macros serve two essential purposes:

- To reduce the processing overhead associated with filtered out messages, a check is made against the report's verbosity setting and the action for the id/severity pair before any string formatting is performed. This affects only `uvm_info reports.
- The `__FILE__ and `__LINE__ information is automatically provided to the underlying uvm_report_* call. Having the file and line number from where a report was issued aides in debug. You can disable display of file and line information in reports by defining UVM_DISABLE_REPORT_FILE_LINE on the command line.

The macros also enforce a verbosity setting of UVM_NONE for warnings, errors and fatals so that they cannot be mistakingly turned off by setting the verbosity level too low (warning and errors can still be turned off by setting the actions appropriately).

To use the macros, replace the previous call to uvm_report_* with the corresponding macro.

```
//Previous calls to uvm_report_*
uvm_report_info("MYINFO1", $sformatf("val: %0d", val), UVM_LOW);
uvm_report_warning("MYWARN1", "This is a warning");
uvm_report_error("MYERR", "This is an error");
uvm_report_fatal("MYFATAL", "A fatal error has occurred");
```

The above code is replaced by

```
//New calls to `uvm_*
`uvm_info("MYINFO1", $sformatf("val: %0d", val), UVM_LOW)
`uvm_warning("MYWARN1", "This is a warning")
`uvm_error("MYERR", "This is an error")
`uvm_fatal("MYFATAL", "A fatal error has occurred")
```

Macros represent text substitutions, not statements, so they should not be terminated with semi-colons.

Summary

Report Macros

This set of macros provides wrappers around the uvm_report_* Reporting functions.

MACROS

`uvm_info	Calls uvm_report_info if *VERBOSITY* is lower than the configured verbosity of the associated reporter.
`uvm_warning	Calls uvm_report_warning with a verbosity of UVM_NONE.
`uvm_error	Calls uvm_report_error with a verbosity of UVM_NONE.
`uvm_fatal	Calls uvm_report_fatal with a verbosity of UVM_NONE.
`uvm_info_context	Operates identically to `uvm_info but requires that the context in which the message is printed be

	explicitly supplied as a macro argument.
`uvm_warning_context	Operates identically to `uvm_warning but requires that the context in which the message is printed be explicitly supplied as a macro argument.
`uvm_error_context	Operates identically to `uvm_error but requires that the context in which the message is printed be explicitly supplied as a macro argument.
`uvm_fatal_context	Operates identically to `uvm_fatal but requires that the context in which the message is printed be explicitly supplied as a macro argument.

MACROS

`uvm_info

Calls uvm_report_info if *VERBOSITY* is lower than the configured verbosity of the associated reporter. *ID* is given as the message tag and *MSG* is given as the message text. The file and line are also sent to the uvm_report_info call.

`uvm_warning

Calls uvm_report_warning with a verbosity of UVM_NONE. The message can not be turned off using the reporter's verbosity setting, but can be turned off by setting the action for the message. *ID* is given as the message tag and *MSG* is given as the message text. The file and line are also sent to the uvm_report_warning call.

`uvm_error

Calls uvm_report_error with a verbosity of UVM_NONE. The message can not be turned off using the reporter's verbosity setting, but can be turned off by setting the action for the message. *ID* is given as the message tag and *MSG* is given as the message text. The file and line are also sent to the uvm_report_error call.

`uvm_fatal

Calls uvm_report_fatal with a verbosity of UVM_NONE. The message can not be turned off using the reporter's verbosity setting, but can be turned off by setting the action for the message. *ID* is given as the message tag and *MSG* is given as the message text. The file and line are also sent to the uvm_report_fatal call.

`uvm_info_context

Operates identically to `uvm_info but requires that the context in which the message is printed be explicitly supplied as a macro argument.

`uvm_warning_context

Operates identically to `uvm_warning but requires that the context in which the message is printed be explicitly supplied as a macro argument.

`uvm_error_context

Operates identically to `uvm_error but requires that the context in which the message is printed be explicitly supplied as a macro argument.

`uvm_fatal_context

Operates identically to `uvm_fatal but requires that the context in which the message is printed be explicitly supplied as a macro argument.

Utility and Field Macros for Components and Objects

Summary

Utility and Field Macros for Components and Objects

UTILITY MACROS	The utility macros provide implementations of the uvm_object::create method, which is needed for cloning, and the uvm_object::get_type_name method, which is needed for a number of debugging features.
`uvm_field_utils_begin `uvm_field_utils_end	These macros form a block in which `uvm_field_* macros can be placed.
`uvm_object_utils `uvm_object_param_utils `uvm_object_utils_begin `uvm_object_param_utils_begin `uvm_object_utils_end	uvm_object-based class declarations may contain one of the above forms of utility macros.
`uvm_component_utils `uvm_component_param_utils `uvm_component_utils_begin `uvm_component_param_utils_begin `uvm_component_end	uvm_component-based class declarations may contain one of the above forms of utility macros.
`uvm_object_registry	Register a uvm_object-based class with the factory
`uvm_component_registry	Registers a uvm_component-based class with the factory
FIELD MACROS	The `uvm_field_* macros are invoked inside of the `uvm_*_utils_begin and `uvm_*_utils_end macro blocks to form "automatic" implementations of the core data methods: copy, compare, pack, unpack, record, print, and sprint.
`UVM_FIELD_* MACROS	Macros that implement data operations for scalar properties.
`uvm_field_int	Implements the data operations for any packed integral property.
`uvm_field_object	Implements the data operations for an uvm_object-based property.
`uvm_field_string	Implements the data operations for a string property.
`uvm_field_enum	Implements the data operations for an enumerated property.
`uvm_field_real	Implements the data operations for any real property.
`uvm_field_event	Implements the data operations for an event property.
`UVM_FIELD_SARRAY_* MACROS	Macros that implement data operations for one-dimensional static array properties.
`uvm_field_sarray_int	Implements the data operations for a one-dimensional static array of

	integrals.
`uvm_field_sarray_object	Implements the data operations for a one-dimensional static array of uvm_object-based objects.
`uvm_field_sarray_string	Implements the data operations for a one-dimensional static array of strings.
`uvm_field_sarray_enum	Implements the data operations for a one-dimensional static array of enums.
`UVM_FIELD_ARRAY_* MACROS	Macros that implement data operations for one-dimensional dynamic array properties.
`uvm_field_array_int	Implements the data operations for a one-dimensional dynamic array of integrals.
`uvm_field_array_object	Implements the data operations for a one-dimensional dynamic array of uvm_object-based objects.
`uvm_field_array_string	Implements the data operations for a one-dimensional dynamic array of strings.
`uvm_field_array_enum	Implements the data operations for a one-dimensional dynamic array of enums.
`UVM_FIELD_QUEUE_* MACROS	Macros that implement data operations for dynamic queues.
`uvm_field_queue_int	Implements the data operations for a queue of integrals.
`uvm_field_queue_object	Implements the data operations for a queue of uvm_object-based objects.
`uvm_field_queue_string	Implements the data operations for a queue of strings.
`uvm_field_queue_enum	Implements the data operations for a one-dimensional queue of enums.
`UVM_FIELD_AA_*_STRING MACROS	Macros that implement data operations for associative arrays indexed by *string*.
`uvm_field_aa_int_string	Implements the data operations for an associative array of integrals indexed by *string*.
`uvm_field_aa_object_string	Implements the data operations for an associative array of uvm_object-based objects indexed by *string*.
`uvm_field_aa_string_string	Implements the data operations for an associative array of strings indexed by *string*.
`UVM_FIELD_AA_*_INT MACROS	Macros that implement data operations for associative arrays indexed by an integral type.
`uvm_field_aa_object_int	Implements the data operations for an associative array of uvm_object-based objects indexed by the *int* data type.
`uvm_field_aa_int_int	Implements the data operations for an associative array of integral types indexed by the *int* data type.
`uvm_field_aa_int_int_unsigned	Implements the data operations for an associative array of integral types indexed by the *int unsigned* data type.
`uvm_field_aa_int_integer	Implements the data operations for an associative array of integral types indexed by the *integer* data type.
`uvm_field_aa_int_integer_unsigned	Implements the data operations for

	an associative array of integral types indexed by the *integer unsigned* data type.
`uvm_field_aa_int_byte	Implements the data operations for an associative array of integral types indexed by the *byte* data type.
`uvm_field_aa_int_byte_unsigned	Implements the data operations for an associative array of integral types indexed by the *byte unsigned* data type.
`uvm_field_aa_int_shortint	Implements the data operations for an associative array of integral types indexed by the *shortint* data type.
`uvm_field_aa_int_shortint_unsigned	Implements the data operations for an associative array of integral types indexed by the *shortint unsigned* data type.
`uvm_field_aa_int_longint	Implements the data operations for an associative array of integral types indexed by the *longint* data type.
`uvm_field_aa_int_longint_unsigned	Implements the data operations for an associative array of integral types indexed by the *longint unsigned* data type.
`uvm_field_aa_int_key	Implements the data operations for an associative array of integral types indexed by any integral key data type.
`uvm_field_aa_int_enumkey	Implements the data operations for an associative array of integral types indexed by any enumeration key data type.
Recording Macros	The recording macros assist users who implement the uvm_object::do_record method.
`uvm_record_attribute	Vendor-independent macro for recording attributes (fields) to a vendor-specific transaction database.
`uvm_record_field	Macro for recording name-value pairs into a transaction recording database.
Packing Macros	The packing macros assist users who implement the uvm_object::do_pack method.
Packing - With Size Info	
`uvm_pack_intN	Pack an integral variable.
`uvm_pack_enumN	Pack an integral variable.
`uvm_pack_sarrayN	Pack a static array of integrals.
`uvm_pack_arrayN	Pack a dynamic array of integrals.
`uvm_pack_queueN	Pack a queue of integrals.
Packing - No Size Info	
`uvm_pack_int	Pack an integral variable without having to also specify the bit size.
`uvm_pack_enum	Pack an enumeration value.
`uvm_pack_string	Pack a string variable.
`uvm_pack_real	Pack a variable of type real.
`uvm_pack_sarray	Pack a static array without having to also specify the bit size of its elements.
`uvm_pack_array	Pack a dynamic array without having to also specify the bit size of its elements.
`uvm_pack_queue	Pack a queue without having to also specify the bit size of its elements.
Unpacking Macros	The unpacking macros assist users

	who implement the uvm_object::do_unpack method.
Unpacking - With Size Info	
`uvm_unpack_intN	Unpack into an integral variable.
`uvm_unpack_enumN	Unpack enum of type *TYPE* into *VAR*.
`uvm_unpack_sarrayN	Unpack a static (fixed) array of integrals.
`uvm_unpack_arrayN	Unpack into a dynamic array of integrals.
`uvm_unpack_queueN	Unpack into a queue of integrals.
Unpacking - No Size Info	
`uvm_unpack_int	Unpack an integral variable without having to also specify the bit size.
`uvm_unpack_enum	Unpack an enumeration value, which requires its type be specified.
`uvm_unpack_string	Pack a string variable.
`uvm_unpack_real	Unpack a variable of type real.
`uvm_unpack_sarray	Unpack a static array without having to also specify the bit size of its elements.
`uvm_unpack_array	Unpack a dynamic array without having to also specify the bit size of its elements.
`uvm_unpack_queue	Unpack a queue without having to also specify the bit size of its elements.

Utility Macros

The utility macros provide implementations of the uvm_object::create method, which is needed for cloning, and the uvm_object::get_type_name method, which is needed for a number of debugging features. They also register the type with the uvm_factory, and they implement a *get_type* method, which is used when configuring the factory. And they implement the virtual uvm_object::get_object_type method for accessing the factory proxy of an allocated object.

Below is an example usage of the utility and field macros. By using the macros, you do not have to implement any of the data methods to get all of the capabilities of an uvm_object.

```
class mydata extends uvm_object;

  string str;
  mydata subdata;
  int field;
  myenum e1;
  int queue[$];

  `uvm_object_utils_begin(mydata) //requires ctor with default args
    `uvm_field_string(str, UVM_DEFAULT)
    `uvm_field_object(subdata, UVM_DEFAULT)
    `uvm_field_int(field, UVM_DEC) //use decimal radix
    `uvm_field_enum(myenum, e1, UVM_DEFAULT)
    `uvm_field_queue_int(queue, UVM_DEFAULT)
  `uvm_object_utils_end

endclass
```

`uvm_field_utils_begin

`uvm_field_utils_end

These macros form a block in which `uvm_field_* macros can be placed. Used as

```
`uvm_field_utils_begin(TYPE)
  `uvm_field_* macros here
`uvm_field_utils_end
```

These macros do NOT perform factory registration, implement get_type_name, nor implement the create method. Use this form when you need custom implementations of these two methods, or when you are setting up field macros for an abstract class (i.e. virtual class).

`uvm_object_utils

`uvm_object_param_utils

`uvm_object_utils_begin

`uvm_object_param_utils_begin

`uvm_object_utils_end

uvm_object-based class declarations may contain one of the above forms of utility macros.

For simple objects with no field macros, use

```
`uvm_object_utils(TYPE)
```

For simple objects with field macros, use

```
`uvm_object_utils_begin(TYPE)
  `uvm_field_* macro invocations here
`uvm_object_utils_end
```

For parameterized objects with no field macros, use

```
`uvm_object_param_utils(TYPE)
```

For parameterized objects, with field macros, use

```
`uvm_object_param_utils_begin(TYPE)
```

```
  `uvm_field_* macro invocations here
`uvm_object_utils_end
```

Simple (non-parameterized) objects use the uvm_object_utils* versions, which do the following:

- Implements get_type_name, which returns TYPE as a string
- Implements create, which allocates an object of type TYPE by calling its constructor with no arguments. TYPE's constructor, if defined, must have default values on all it arguments.
- Registers the TYPE with the factory, using the string TYPE as the factory lookup string for the type.
- Implements the static get_type() method which returns a factory proxy object for the type.
- Implements the virtual get_object_type() method which works just like the static get_type() method, but operates on an already allocated object.

Parameterized classes must use the uvm_object_param_utils* versions. They differ from `uvm_object_utils only in that they do not supply a type name when registering the object with the factory. As such, name-based lookup with the factory for parameterized classes is not possible.

The macros with _begin suffixes are the same as the non-suffixed versions except that they also start a block in which `uvm_field_* macros can be placed. The block must be terminated by `uvm_object_utils_end.

Objects deriving from uvm_sequence must use the `uvm_sequence_* macros instead of these macros. See <`uvm_sequence_utils> for details.

`uvm_component_utils

`uvm_component_param_utils

`uvm_component_utils_begin

`uvm_component_param_utils_begin

`uvm_component_end

uvm_component-based class declarations may contain one of the above forms of utility macros.

For simple components with no field macros, use

```
`uvm_component_utils(TYPE)
```

For simple components with field macros, use

```
`uvm_component_utils_begin(TYPE)
  `uvm_field_* macro invocations here
`uvm_component_utils_end
```

For parameterized components with no field macros, use

```
`uvm_component_param_utils(TYPE)
```

For parameterized components with field macros, use

```
`uvm_component_param_utils_begin(TYPE)
  `uvm_field_* macro invocations here
`uvm_component_utils_end
```

Simple (non-parameterized) components must use the uvm_components_utils* versions, which do the following:

- Implements get_type_name, which returns TYPE as a string.
- Implements create, which allocates a component of type TYPE using a two argument constructor. TYPE's constructor must have a name and a parent argument.
- Registers the TYPE with the factory, using the string TYPE as the factory lookup string for the type.
- Implements the static get_type() method which returns a factory proxy object for the type.
- Implements the virtual get_object_type() method which works just like the static get_type() method, but operates on an already allocated object.

Parameterized classes must use the uvm_object_param_utils* versions. They differ from `uvm_object_utils only in that they do not supply a type name when registering the object with the factory. As such, name-based lookup with the factory for parameterized classes is not possible.

The macros with _begin suffixes are the same as the non-suffixed versions except that they also start a block in which `uvm_field_* macros can be placed. The block must be terminated by `uvm_component_utils_end.

`uvm_object_registry

Register a uvm_object-based class with the factory

```
`uvm_object_registry(T,S)
```

Registers a uvm_object-based class *T* and lookup string *S* with the factory. *S* typically is the name of the class in quotes. The `uvm_object_utils family of macros uses this macro.

`uvm_component_registry

Registers a uvm_component-based class with the factory

```
`uvm_component_registry(T,S)
```

Registers a uvm_component-based class *T* and lookup string *S* with the factory. *S* typically is the name of the class in quotes. The `uvm_object_utils family of macros uses this macro.

Field Macros

The `uvm_field_* macros are invoked inside of the `uvm_*_utils_begin and `uvm_*_utils_end macro blocks to form "automatic" implementations of the core data methods: copy, compare, pack, unpack, record, print, and sprint. For example:

```
class my_trans extends uvm_transaction;
  string my_string;
  `uvm_object_utils_begin(my_trans)
    `uvm_field_string(my_string, UVM_ALL_ON)
  `uvm_object_utils_end
endclass
```

Each `uvm_field_* macro is named to correspond to a particular data type: integrals, strings, objects, queues, etc., and each has at least two arguments: *ARG* and *FLAG*.

ARG is the instance name of the variable, whose type must be compatible with the macro being invoked. In the example, class variable my_string is of type string, so we use the `uvm_field_string macro.

If *FLAG* is set to *UVM_ALL_ON*, as in the example, the ARG variable will be included in all data methods. The FLAG, if set to something other than *UVM_ALL_ON* or *UVM_DEFAULT*, specifies which data method implementations will NOT include the given variable. Thus, if *FLAG* is specified as *NO_COMPARE*, the ARG variable will not affect comparison operations, but it will be included in everything else.

All possible values for *FLAG* are listed and described below. Multiple flag values can be bitwise ORed together (in most cases they may be added together as well, but care must be taken when using the + operator to ensure that the same bit is not added more than once).

UVM_ALL_ON	Set all operations on (default).
UVM_DEFAULT	Use the default flag settings.
UVM_NOCOPY	Do not copy this field.
UVM_NOCOMPARE	Do not compare this field.
UVM_NOPRINT	Do not print this field.
UVM_NODEFPRINT	Do not print the field if it is the same as its
UVM_NOPACK	Do not pack or unpack this field.
UVM_PHYSICAL	Treat as a physical field. Use physical setting in policy class for this field.
UVM_ABSTRACT	Treat as an abstract field. Use the abstract setting in the policy class for this field.
UVM_READONLY	Do not allow setting of this field from the set_*_local methods.

A radix for printing and recording can be specified by OR'ing one of the following constants in the *FLAG* argument

UVM_BIN	Print / record the field in binary (base-2).
UVM_DEC	Print / record the field in decimal (base-10).
UVM_UNSIGNED	Print / record the field in unsigned decimal (base-10).
UVM_OCT	Print / record the field in octal (base-8).
UVM_HEX	Print / record the field in hexidecimal (base-16).
UVM_STRING	Print / record the field in string format.
UVM_TIME	Print / record the field in time format.

Radix settings for integral types. Hex is the default radix if none is specified.

`UVM_FIELD_* MACROS

Macros that implement data operations for scalar properties.

`uvm_field_int

Implements the data operations for any packed integral property.

```
`uvm_field_int(ARG,FLAG)
```

ARG is an integral property of the class, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_object

Implements the data operations for an uvm_object-based property.

```
`uvm_field_object(ARG,FLAG)
```

ARG is an object property of the class, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_string

Implements the data operations for a string property.

```
`uvm_field_string(ARG,FLAG)
```

ARG is a string property of the class, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_enum

Implements the data operations for an enumerated property.

```
`uvm_field_enum(T,ARG,FLAG)
```

T is an enumerated type, *ARG* is an instance of that type, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_real

Implements the data operations for any real property.

```
`uvm_field_real(ARG,FLAG)
```

ARG is an real property of the class, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_event

Implements the data operations for an event property.

```
`uvm_field_event(ARG,FLAG)
```

ARG is an event property of the class, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`UVM_FIELD_SARRAY_* MACROS

Macros that implement data operations for one-dimensional static array properties.

`uvm_field_sarray_int

Implements the data operations for a one-dimensional static array of integrals.

```
`uvm_field_sarray_int(ARG,FLAG)
```

ARG is a one-dimensional static array of integrals, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_sarray_object

Implements the data operations for a one-dimensional static array of uvm_object-based objects.

```
`uvm_field_sarray_object(ARG,FLAG)
```

ARG is a one-dimensional static array of uvm_object-based objects, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_sarray_string

Implements the data operations for a one-dimensional static array of strings.

```
`uvm_field_sarray_string(ARG,FLAG)
```

ARG is a one-dimensional static array of strings, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_sarray_enum

Implements the data operations for a one-dimensional static array of enums.

```
`uvm_field_sarray_enum(T,ARG,FLAG)
```

T is a one-dimensional dynamic array of enums type, *ARG* is an instance of that type, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`UVM_FIELD_ARRAY_* MACROS

Macros that implement data operations for one-dimensional dynamic array properties.

Implementation note

lines flagged with empty multi-line comments, /**/, are not needed or need to be different for fixed arrays, which can not be resized. Fixed arrays do not need to pack/unpack their size either, because their size is known; wouldn't hurt though if it allowed code consolidation. Unpacking would necessarily be different. */

`uvm_field_array_int

Implements the data operations for a one-dimensional dynamic array of integrals.

```
`uvm_field_array_int(ARG,FLAG)
```

ARG is a one-dimensional dynamic array of integrals, and *FLAG* is a bitwise OR of one or

more flag settings as described in Field Macros above.

`uvm_field_array_object

Implements the data operations for a one-dimensional dynamic array of uvm_object-based objects.

```
`uvm_field_array_object(ARG,FLAG)
```

ARG is a one-dimensional dynamic array of uvm_object-based objects, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_array_string

Implements the data operations for a one-dimensional dynamic array of strings.

```
`uvm_field_array_string(ARG,FLAG)
```

ARG is a one-dimensional dynamic array of strings, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_array_enum

Implements the data operations for a one-dimensional dynamic array of enums.

```
`uvm_field_array_enum(T,ARG,FLAG)
```

T is a one-dimensional dynamic array of enums type, *ARG* is an instance of that type, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`UVM_FIELD_QUEUE_* MACROS

Macros that implement data operations for dynamic queues.

`uvm_field_queue_int

Implements the data operations for a queue of integrals.

```
`uvm_field_queue_int(ARG,FLAG)
```

ARG is a one-dimensional queue of integrals, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_queue_object

Implements the data operations for a queue of uvm_object-based objects.

```
`uvm_field_queue_object(ARG,FLAG)
```

ARG is a one-dimensional queue of uvm_object-based objects, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_queue_string

Implements the data operations for a queue of strings.

```
`uvm_field_queue_string(ARG,FLAG)
```

ARG is a one-dimensional queue of strings, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_queue_enum

Implements the data operations for a one-dimensional queue of enums.

```
`uvm_field_queue_enum(T,ARG,FLAG)
```

T is a queue of enums type, *ARG* is an instance of that type, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`UVM_FIELD_AA_*_STRING MACROS

Macros that implement data operations for associative arrays indexed by *string*.

`uvm_field_aa_int_string

Implements the data operations for an associative array of integrals indexed by *string*.

```
`uvm_field_aa_int_string(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with string key, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_aa_object_string

Implements the data operations for an associative array of uvm_object-based objects indexed by *string*.

```
`uvm_field_aa_object_string(ARG,FLAG)
```

ARG is the name of a property that is an associative array of objects with string key, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_aa_string_string

Implements the data operations for an associative array of strings indexed by *string*.

```
`uvm_field_aa_string_string(ARG,FLAG)
```

ARG is the name of a property that is an associative array of strings with string key, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`UVM_FIELD_AA_*_INT MACROS

Macros that implement data operations for associative arrays indexed by an integral type.

`uvm_field_aa_object_int

Implements the data operations for an associative array of uvm_object-based objects indexed by the *int* data type.

```
`uvm_field_aa_object_int(ARG,FLAG)
```

ARG is the name of a property that is an associative array of objects with *int* key, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_aa_int_int

Implements the data operations for an associative array of integral types indexed by the *int* data type.

```
`uvm_field_aa_int_int(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *int* key, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_aa_int_int_unsigned

Implements the data operations for an associative array of integral types indexed by the *int unsigned* data type.

```
`uvm_field_aa_int_int_unsigned(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *int unsigned* key, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_aa_int_integer

Implements the data operations for an associative array of integral types indexed by the *integer* data type.

```
`uvm_field_aa_int_integer(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *integer* key, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_aa_int_integer_unsigned

Implements the data operations for an associative array of integral types indexed by the *integer unsigned* data type.

```
`uvm_field_aa_int_integer_unsigned(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *integer unsigned* key, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_aa_int_byte

Implements the data operations for an associative array of integral types indexed by the *byte* data type.

```
`uvm_field_aa_int_byte(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *byte* key, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_aa_int_byte_unsigned

Implements the data operations for an associative array of integral types indexed by the *byte unsigned* data type.

```
`uvm_field_aa_int_byte_unsigned(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *byte unsigned* key, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_aa_int_shortint

Implements the data operations for an associative array of integral types indexed by the *shortint* data type.

```
`uvm_field_aa_int_shortint(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *shortint* key, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_aa_int_shortint_unsigned

Implements the data operations for an associative array of integral types indexed by the *shortint unsigned* data type.

```
`uvm_field_aa_int_shortint_unsigned(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *shortint unsigned* key, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_aa_int_longint

Implements the data operations for an associative array of integral types indexed by the *longint* data type.

```
`uvm_field_aa_int_longint(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *longint* key, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_aa_int_longint_unsigned

Implements the data operations for an associative array of integral types indexed by the *longint unsigned* data type.

```
`uvm_field_aa_int_longint_unsigned(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *longint unsigned* key, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_aa_int_key

Implements the data operations for an associative array of integral types indexed by any integral key data type.

```
`uvm_field_aa_int_key(long unsigned,ARG,FLAG)
```

KEY is the data type of the integral key, *ARG* is the name of a property that is an associative array of integrals, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

`uvm_field_aa_int_enumkey

Implements the data operations for an associative array of integral types indexed by any enumeration key data type.

```
`uvm_field_aa_int_longint_unsigned(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *longint unsigned* key, and *FLAG* is a bitwise OR of one or more flag settings as described in Field Macros above.

Recording Macros

The recording macros assist users who implement the uvm_object::do_record method. They help ensure that the fields are recorded using a vendor- independent API. Unlike the uvm_recorder policy, fields recorded using the `uvm_record_field macro do not lose type information--they are passed directly to the vendor-specific API. This results in more efficient recording and no artificial limit on bit-widths. See your simulator vendor's documentation for more information on its transaction recording capabilities.

`uvm_record_attribute

Vendor-independent macro for recording attributes (fields) to a vendor-specific transaction database.

`uvm_record_field

Macro for recording name-value pairs into a transaction recording database. Requires a valid transaction handle, as provided by the uvm_transaction::begin_tr and uvm_component::begin_tr methods.

Packing Macros

The packing macros assist users who implement the uvm_object::do_pack method. They help ensure that the pack operation is the exact inverse of the unpack operation. See also Unpacking Macros.

```
virtual function void do_pack(uvm_packer packer);
  `uvm_pack_int(cmd)
  `uvm_pack_int(addr)
  `uvm_pack_array(data)
endfunction
```

The 'N' versions of these macros take a explicit size argument.

Packing - With Size Info

`uvm_pack_intN

Pack an integral variable.

```
`uvm_pack_intN(VAR,SIZE)
```

`uvm_pack_enumN

Pack an integral variable.

```
`uvm_pack_enumN(VAR,SIZE)
```

`uvm_pack_sarrayN

Pack a static array of integrals.

```
`uvm_pack_sarray(VAR,SIZE)
```

`uvm_pack_arrayN

Pack a dynamic array of integrals.

```
`uvm_pack_arrayN(VAR,SIZE)
```

`uvm_pack_queueN

Pack a queue of integrals.

```
`uvm_pack_queueN(VAR,SIZE)
```

Packing - No Size Info

`uvm_pack_int

Pack an integral variable without having to also specify the bit size.

```
`uvm_pack_int(VAR)
```

`uvm_pack_enum

Pack an enumeration value. Packing does not require its type be specified.

```
`uvm_pack_enum(VAR)
```

`uvm_pack_string

Pack a string variable.

```
`uvm_pack_string(VAR)
```

`uvm_pack_real

Pack a variable of type real.

```
`uvm_pack_real(VAR)
```

`uvm_pack_sarray

Pack a static array without having to also specify the bit size of its elements.

```
`uvm_pack_sarray(VAR)
```

`uvm_pack_array

Pack a dynamic array without having to also specify the bit size of its elements. Array size must be non-zero.

```
`uvm_pack_array(VAR)
```

`uvm_pack_queue

Pack a queue without having to also specify the bit size of its elements. Queue must not be empty.

```
`uvm_pack_queue(VAR)
```

Unpacking Macros

The unpacking macros assist users who implement the uvm_object::do_unpack method. They help ensure that the unpack operation is the exact inverse of the pack operation. See also Packing Macros.

```
virtual function void do_unpack(uvm_packer packer);
  `uvm_unpack_enum(cmd,cmd_t)
  `uvm_unpack_int(addr)
  `uvm_unpack_array(data)
endfunction
```

The 'N' versions of these macros take a explicit size argument.

Unpacking - With Size Info

`uvm_unpack_intN

Unpack into an integral variable.

```
`uvm_unpack_intN(VAR,SIZE)
```

`uvm_unpack_enumN

Unpack enum of type *TYPE* into *VAR*.

`uvm_unpack_enumN(VAR,SIZE,TYPE)

`uvm_unpack_sarrayN

Unpack a static (fixed) array of integrals.

```
`uvm_unpack_sarrayN(VAR,SIZE)
```

`uvm_unpack_arrayN

Unpack into a dynamic array of integrals.

```
`uvm_unpack_arrayN(VAR,SIZE)
```

`uvm_unpack_queueN

Unpack into a queue of integrals.

```
`uvm_unpack_queue(VAR,SIZE)
```

Unpacking - No Size Info

`uvm_unpack_int

Unpack an integral variable without having to also specify the bit size.

```
`uvm_unpack_int(VAR)
```

`uvm_unpack_enum

Unpack an enumeration value, which requires its type be specified.

```
`uvm_unpack_enum(VAR,TYPE)
```

`uvm_unpack_string

Pack a string variable.

```
`uvm_unpack_string(VAR)
```

`uvm_unpack_real

Unpack a variable of type real.

```
`uvm_unpack_real(VAR)
```

`uvm_unpack_sarray

Unpack a static array without having to also specify the bit size of its elements.

```
`uvm_unpack_sarray(VAR)
```

`uvm_unpack_array

Unpack a dynamic array without having to also specify the bit size of its elements. Array size must be non-zero.

```
`uvm_unpack_array(VAR)
```

`uvm_unpack_queue

Unpack a queue without having to also specify the bit size of its elements. Queue must not be empty.

```
`uvm_unpack_queue(VAR)
```

Sequence-Related Macros

Summary

Sequence-Related Macros

Sequence Action Macros	These macros are used to start sequences and sequence items on the default sequencer, <uvm_sequence_base::m_sequencer>.
`uvm_create	This action creates the item or sequence using the factory.
`uvm_do	This macro takes as an argument a uvm_sequence_item variable or object.
`uvm_do_pri	This is the same as `uvm_do except that the sequene item or sequence is executed with the priority specified in the argument
`uvm_do_with	This is the same as `uvm_do except that the constraint block in the 2nd argument is applied to the item or sequence in a randomize with statement before execution.
`uvm_do_pri_with	This is the same as `uvm_do_pri except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.
Sequence on Sequencer Action Macros	These macros are used to start sequences and sequence items on a specific sequencer.
`uvm_create_on	This is the same as `uvm_create except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *SEQUENCER_REF* argument.
`uvm_do_on	This is the same as `uvm_do except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *SEQUENCER_REF* argument.
`uvm_do_on_pri	This is the same as `uvm_do_pri except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *SEQUENCER_REF* argument.
`uvm_do_on_with	This is the same as `uvm_do_with except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *SEQUENCER_REF* argument.
`uvm_do_on_pri_with	This is the same as `uvm_do_pri_with except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *SEQUENCER_REF* argument.
Sequence Action Macros for Pre-Existing Sequences	These macros are used to start sequences and sequence items that have already been allocated, i.e.
`uvm_send	This macro processes the item or sequence that has been created using `uvm_create.
`uvm_send_pri	This is the same as `uvm_send except that the sequene item or sequence is

	executed with the priority specified in the argument.
`uvm_rand_send	This macro processes the item or sequence that has been already been allocated (possibly with `uvm_create).
`uvm_rand_send_pri	This is the same as `uvm_rand_send except that the sequene item or sequence is executed with the priority specified in the argument.
`uvm_rand_send_with	This is the same as `uvm_rand_send except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.
`uvm_rand_send_pri_with	This is the same as `uvm_rand_send_pri except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.
Sequence Library	
`uvm_add_to_sequence_library	Adds the given sequence *TYPE* to the given sequence library *LIBTYPE*
`uvm_sequence_library_utils	Declares the infrastructure needed to define extensions to the <uvm_sequence_library> class.
Sequencer Subtypes	
`uvm_declare_p_sequencer	

Sequence Action Macros

These macros are used to start sequences and sequence items on the default sequencer, <uvm_sequence_base::m_sequencer>. The default sequencer is set any number of ways.

- the sequencer handle provided in the uvm_sequence_base::start method
- the sequencer used by the parent sequence
- the sequencer that was set using the uvm_sequence_item::set_sequencer method

`uvm_create

This action creates the item or sequence using the factory. It intentionally does zero processing. After this action completes, the user can manually set values, manipulate rand_mode and constraint_mode, etc.

`uvm_do

This macro takes as an argument a uvm_sequence_item variable or object. uvm_sequence_item's are randomized *at the time* the sequencer grants the do request. This is called late-randomization or late-generation. In the case of a sequence a sub-sequence is spawned. In the case of an item, the item is sent to the driver through the associated sequencer.

`uvm_do_pri

This is the same as `uvm_do except that the sequene item or sequence is executed with

the priority specified in the argument

`uvm_do_with

This is the same as `uvm_do except that the constraint block in the 2nd argument is applied to the item or sequence in a randomize with statement before execution.

`uvm_do_pri_with

This is the same as `uvm_do_pri except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.

Sequence on Sequencer Action Macros

These macros are used to start sequences and sequence items on a specific sequencer. The sequence or item is created and executed on the given sequencer.

`uvm_create_on

This is the same as `uvm_create except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *SEQUENCER_REF* argument.

`uvm_do_on

This is the same as `uvm_do except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *SEQUENCER_REF* argument.

`uvm_do_on_pri

This is the same as `uvm_do_pri except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *SEQUENCER_REF* argument.

`uvm_do_on_with

This is the same as `uvm_do_with except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *SEQUENCER_REF* argument. The user must supply brackets around the constraints.

`uvm_do_on_pri_with

This is the same as `uvm_do_pri_with except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *SEQUENCER_REF* argument.

Sequence Action Macros for Pre-Existing Sequences

These macros are used to start sequences and sequence items that have already been allocated, i.e. do not need to be created.

`uvm_send

This macro processes the item or sequence that has been created using `uvm_create. The processing is done without randomization. Essentially, an `uvm_do without the create or randomization.

`uvm_send_pri

This is the same as `uvm_send except that the sequene item or sequence is executed with the priority specified in the argument.

`uvm_rand_send

This macro processes the item or sequence that has been already been allocated (possibly with `uvm_create). The processing is done with randomization. Essentially, an `uvm_do without the create.

`uvm_rand_send_pri

This is the same as `uvm_rand_send except that the sequene item or sequence is executed with the priority specified in the argument.

`uvm_rand_send_with

This is the same as `uvm_rand_send except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.

`uvm_rand_send_pri_with

This is the same as `uvm_rand_send_pri except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.

Sequence Library

`uvm_add_to_sequence_library

Adds the given sequence *TYPE* to the given sequence library *LIBTYPE*

Invoke any number of times within a sequence declaration to statically add that sequence to one or more sequence library types. The sequence will then be available for selection and execution in all instances of the given sequencer types.

```
class seqA extends uvm_sequence_base #(simple_item);

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

   `uvm_object_utils(seqA)

   `uvm_add_to_seq_lib(seqA, simple_seq_lib_RST)
   `uvm_add_to_seq_lib(seqA, simple_seq_lib_CFG)

   virtual task body(); \
     `uvm_info("SEQ_START", {"Executing sequence '", get_full_name(),
                             "' (",get_type_name(),")"},UVM_HIGH)
     #10;
   endtask

 endclass
```

`uvm_sequence_library_utils

Declares the infrastructure needed to define extensions to the <uvm_sequence_library> class. You define new sequence library subtypes to statically specify sequence membership from within sequence definitions. See also `uvm_add_to_sequence_library for more information.

Sequencer Subtypes

`uvm_declare_p_sequencer

```
`uvm_declare_p_sequencer(SEQUENCER)
```

This macro is used to declare a variable *p_sequencer* whose type is specified by *SEQUENCER*.

The example below shows using the the `uvm_declare_p_sequencer macro along with the uvm_object_utils macros to set up the sequence but not register the sequence in the sequencer's library.

```
class mysequence extends uvm_sequence#(mydata);
  `uvm_object_utils(mysequence)
  `uvm_declare_p_sequencer(some_seqr_type)
  task body;
    //Access some variable in the user's custom sequencer
    if(p_sequencer.some_variable) begin
      ...
    end
  endtask
endclass
```

uvm_callback_defines.svh

Summary

CALLBACK MACROS

`uvm_register_cb

Registers the given *CB* callback type with the given *T* object type. If a type-callback pair is not registered then a warning is issued if an attempt is made to use the pair (add, delete, etc.).

The registration will typically occur in the component that executes the given type of callback. For instance:

```
virtual class mycb extends uvm_callback;
  virtual function void doit();
endclass

class my_comp extends uvm_component;
  `uvm_register_cb(my_comp,mycb)
  ...
  task run_phase(uvm_phase phase);
    ...
    `uvm_do_callbacks(my_comp, mycb, doit())
  endtask
endclass
```

`uvm_set_super_type

Defines the super type of T to be ST. This allows for derived class objects to inherit typewide callbacks that are registered with the base class.

The registration will typically occur in the component that executes the given type of callback. For instance:

```
virtual class mycb extend uvm_callback;
  virtual function void doit();
endclass

class my_comp extends uvm_component;
  `uvm_register_cb(my_comp,mycb)
  ...
  task run_phase(uvm_phase phase);
    ...
    `uvm_do_callbacks(my_comp, mycb, doit())
  endtask
endclass

class my_derived_comp extends my_comp;
  `uvm_set_super_type(my_derived_comp,my_comp)
  ...
  task run_phase(uvm_phase phase);
    ...
    `uvm_do_callbacks(my_comp, mycb, doit())
  endtask
endclass
```

`uvm_do_callbacks

Calls the given *METHOD* of all callbacks of type *CB* registered with the calling object (i.e. *this* object), which is or is based on type *T*.

This macro executes all of the callbacks associated with the calling object (i.e. *this* object). The macro takes three arguments:

- CB is the class type of the callback objects to execute. The class type must have a function signature that matches the METHOD argument.
- T is the type associated with the callback. Typically, an instance of type T is passed as one the arguments in the *METHOD* call.
- METHOD is the method call to invoke, with all required arguments as if they were invoked directly.

For example, given the following callback class definition

```
virtual class mycb extends uvm_cb;
  pure function void my_function (mycomp comp, int addr, int data);
endclass
```

A component would invoke the macro as

```
task mycomp::run_phase(uvm_phase phase);
   int curr_addr, curr_data;
   ...
   `uvm_do_callbacks(mycb, mycomp, my_function(this, curr_addr, curr_data))
   ...
endtask
```

`uvm_do_obj_callbacks

Calls the given *METHOD* of all callbacks based on type *CB* registered with the given object, *OBJ*, which is or is based on type *T*.

This macro is identical to `uvm_do_callbacks macro, but it has an additional *OBJ* argument to allow the specification of an external object to associate the callback with. For example, if the callbacks are being applied in a sequence, *OBJ* could be specified as the associated sequencer or parent sequence.

```
...
`uvm_do_callbacks(mycb, mycomp, seqr, my_function(seqr, curr_addr,
curr_data))
...
```

`uvm_do_callbacks_exit_on

Calls the given *METHOD* of all callbacks of type *CB* registered with the calling object (i.e. *this* object), which is or is based on type *T*, returning upon the first callback returning the bit value given by *VAL*.

This macro executes all of the callbacks associated with the calling object (i.e. *this* object). The macro takes three arguments:

- CB is the class type of the callback objects to execute. The class type must have a function signature that matches the METHOD argument.
- T is the type associated with the callback. Typically, an instance of type T is passed as one the arguments in the *METHOD* call.
- METHOD is the method call to invoke, with all required arguments as if they were invoked directly.
- VAL, if 1, says return upon the first callback invocation that returns 1. If 0, says return upon the first callback invocation that returns 0.

For example, given the following callback class definition

```
virtual class mycb extends uvm_cb;
  pure function bit drop_trans (mycomp comp, my_trans trans);
endclass
```

A component would invoke the macro as

```
task mycomp::run_phase(uvm_phase phase);
   my_trans trans;
   forever begin
     get_port.get(trans);
     if(do_callbacks(trans) == 0)
       uvm_report_info("DROPPED",{"trans dropped:
%s",trans.convert2string()});
     else
       // execute transaction
   end
endtask
function bit do_callbacks(my_trans);
  // Returns 0 if drop happens and 1 otherwise
  `uvm_do_callbacks_exit_on(mycomp, mycb, extobj, drop_trans(this,trans), 1)
endfunction
```

`uvm_do_obj_callbacks_exit_on

Calls the given *METHOD* of all callbacks of type *CB* registered with the given object *OBJ*, which must be or be based on type *T*, and returns upon the first callback that returns

the bit value given by *VAL*. It is exactly the same as the `uvm_do_callbacks_exit_on but has a specific object instance (instead of the implicit this instance) as the third argument.

```
...
 // Exit with 0 if a callback returns a 1
  `uvm_do_callbacks_exit_on(mycomp, mycb, seqr, drop_trans(seqr,trans), 1)
...
```

TLM Implementation Port Declaration Macros

The TLM implemenation declaration macros provide a way for an implementer to provide multiple implemenation ports of the same implementation interface. When an implementation port is defined using the built-in set of imps, there must be exactly one implementation of the interface.

For example, if a component needs to provide a put implemenation then it would have an implementation port defined like:

```
class mycomp extends uvm_component;
  uvm_put_imp#(data_type, mycomp) put_imp;
  ...
  virtual task put (data_type t);
    ...
  endtask
endclass
```

There are times, however, when you need more than one implementation for for an interface. This set of declarations allow you to easily create a new implemenation class to allow for multiple implementations. Although the new implemenation class is a different class, it can be bound to the same types of exports and ports as the original class. Extending the put example above, lets say that mycomp needs to provide two put implementation ports. In that case, you would do something like:

```
//Define two new put interfaces which are compatible with uvm_put_ports
//and uvm_put_exports.

`uvm_put_imp_decl(_1)
`uvm_put_imp_decl(_2)

class my_put_imp#(type T=int) extends uvm_component;
   uvm_put_imp_1#(T) put_imp1;
   uvm_put_imp_2#(T) put_imp2;
   ...
   function void put_1 (input T t);
     //puts comming into put_imp1
     ...
   endfunction
   function void put_2(input T t);
     //puts comming into put_imp2
     ...
   endfunction
endclass
```

The important thing to note is that each `uvm_<interface>_imp_decl creates a new class of type uvm_<interface>_imp<suffix>, where suffix is the input argument to the macro. For this reason, you will typically want to put these macros in a seperate package to avoid collisions and to allow sharing of the definitions.

Summary

TLM Implementation Port Declaration Macros

The TLM implemenation declaration macros provide a way for an implementer to provide multiple implemenation ports of the same implementation interface.

Macros

`uvm_blocking_put_imp_decl	Define the class uvm_blocking_put_impSFX for providing blocking put implementations.
`uvm_nonblocking_put_imp_decl	Define the class

	uvm_nonblocking_put_impSFX for providing non-blocking put implementations.
`uvm_put_imp_decl	Define the class uvm_put_impSFX for providing both blocking and non-blocking put implementations.
`uvm_blocking_get_imp_decl	Define the class uvm_blocking_get_impSFX for providing blocking get implementations.
`uvm_nonblocking_get_imp_decl	Define the class uvm_nonblocking_get_impSFX for providing non-blocking get implementations.
`uvm_get_imp_decl	Define the class uvm_get_impSFX for providing both blocking and non-blocking get implementations.
`uvm_blocking_peek_imp_decl	Define the class uvm_blocking_peek_impSFX for providing blocking peek implementations.
`uvm_nonblocking_peek_imp_decl	Define the class uvm_nonblocking_peek_impSFX for providing non-blocking peek implementations.
`uvm_peek_imp_decl	Define the class uvm_peek_impSFX for providing both blocking and non-blocking peek implementations.
`uvm_blocking_get_peek_imp_decl	Define the class uvm_blocking_get_peek_impSFX for providing the blocking get_peek implemenation.
`uvm_nonblocking_get_peek_imp_decl	Define the class uvm_nonblocking_get_peek_impSFX for providing non-blocking get_peek implemenation.
`uvm_get_peek_imp_decl	Define the class uvm_get_peek_impSFX for providing both blocking and non-blocking get_peek implementations.
`uvm_blocking_master_imp_decl	Define the class uvm_blocking_master_impSFX for providing the blocking master implemenation.
`uvm_nonblocking_master_imp_decl	Define the class uvm_nonblocking_master_impSFX for providing the non-blocking master implemenation.
`uvm_master_imp_decl	Define the class uvm_master_impSFX for providing both blocking and non-blocking master implementations.
`uvm_blocking_slave_imp_decl	Define the class uvm_blocking_slave_impSFX for providing the blocking slave implemenation.
`uvm_nonblocking_slave_imp_decl	Define the class uvm_nonblocking_slave_impSFX for providing the non-blocking slave implemenation.
`uvm_slave_imp_decl	Define the class uvm_slave_impSFX for providing both blocking and non-blocking slave implementations.
`uvm_blocking_transport_imp_decl	Define the class uvm_blocking_transport_impSFX for providing the blocking transport implemenation.
`uvm_nonblocking_transport_imp_decl	Define the class uvm_nonblocking_transport_impSFX

	for providing the non-blocking transport implemenation.
`uvm_transport_imp_decl	Define the class uvm_transport_impSFX for providing both blocking and non-blocking transport implementations.
`uvm_analysis_imp_decl	Define the class uvm_analysis_impSFX for providing an analysis implementation.

MACROS

`uvm_blocking_put_imp_decl

Define the class uvm_blocking_put_impSFX for providing blocking put implementations. *SFX* is the suffix for the new class type.

`uvm_nonblocking_put_imp_decl

Define the class uvm_nonblocking_put_impSFX for providing non-blocking put implementations. *SFX* is the suffix for the new class type.

`uvm_put_imp_decl

Define the class uvm_put_impSFX for providing both blocking and non-blocking put implementations. *SFX* is the suffix for the new class type.

`uvm_blocking_get_imp_decl

Define the class uvm_blocking_get_impSFX for providing blocking get implementations. *SFX* is the suffix for the new class type.

`uvm_nonblocking_get_imp_decl

Define the class uvm_nonblocking_get_impSFX for providing non-blocking get implementations. *SFX* is the suffix for the new class type.

`uvm_get_imp_decl

Define the class uvm_get_impSFX for providing both blocking and non-blocking get implementations. *SFX* is the suffix for the new class type.

`uvm_blocking_peek_imp_decl

Define the class uvm_blocking_peek_impSFX for providing blocking peek implementations. *SFX* is the suffix for the new class type.

`uvm_nonblocking_peek_imp_decl

Define the class uvm_nonblocking_peek_impSFX for providing non-blocking peek implementations. *SFX* is the suffix for the new class type.

`uvm_peek_imp_decl

Define the class uvm_peek_impSFX for providing both blocking and non-blocking peek implementations. *SFX* is the suffix for the new class type.

`uvm_blocking_get_peek_imp_decl

Define the class uvm_blocking_get_peek_impSFX for providing the blocking get_peek implemenation.

`uvm_nonblocking_get_peek_imp_decl

Define the class uvm_nonblocking_get_peek_impSFX for providing non-blocking get_peek implemenation.

`uvm_get_peek_imp_decl

Define the class uvm_get_peek_impSFX for providing both blocking and non-blocking get_peek implementations. *SFX* is the suffix for the new class type.

`uvm_blocking_master_imp_decl

Define the class uvm_blocking_master_impSFX for providing the blocking master implemenation.

`uvm_nonblocking_master_imp_decl

Define the class uvm_nonblocking_master_impSFX for providing the non-blocking master implemenation.

`uvm_master_imp_decl

Define the class uvm_master_impSFX for providing both blocking and non-blocking master implementations. *SFX* is the suffix for the new class type.

`uvm_blocking_slave_imp_decl

Define the class uvm_blocking_slave_impSFX for providing the blocking slave implemenation.

`uvm_nonblocking_slave_imp_decl

Define the class uvm_nonblocking_slave_impSFX for providing the non-blocking slave implemenation.

`uvm_slave_imp_decl

Define the class uvm_slave_impSFX for providing both blocking and non-blocking slave implementations. *SFX* is the suffix for the new class type.

`uvm_blocking_transport_imp_decl

Define the class uvm_blocking_transport_impSFX for providing the blocking transport implemenation.

`uvm_nonblocking_transport_imp_decl

Define the class uvm_nonblocking_transport_impSFX for providing the non-blocking transport implemenation.

`uvm_transport_imp_decl

Define the class uvm_transport_impSFX for providing both blocking and non-blocking transport implementations. *SFX* is the suffix for the new class type.

`uvm_analysis_imp_decl

Define the class uvm_analysis_impSFX for providing an analysis implementation. *SFX* is the suffix for the new class type. The analysis implemenation is the write function. The `uvm_analysis_imp_decl allows for a scoreboard (or other analysis component) to support input from many places. For example:

```
`uvm_analysis_imp_decl(_ingress)
`uvm_analysis_imp_port(_egress)

class myscoreboard extends uvm_component;
  uvm_analysis_imp_ingress#(mydata, myscoreboard) ingress;
  uvm_analysis_imp_egress#(mydata, myscoreboard) egress;
  mydata ingress_list[$];
  ...

  function new(string name, uvm_component parent);
    super.new(name,parent);
    ingress = new("ingress", this);
    egress = new("egress", this);
  endfunction

  function void write_ingress(mydata t);
    ingress_list.push_back(t);
  endfunction

  function void write_egress(mydata t);
    find_match_in_ingress_list(t);
  endfunction

  function void find_match_in_ingress_list(mydata t);
    //implement scoreboarding for this particular dut
    ...
  endfunction
endclass
```

uvm_reg_defines.svh

Summary

uvm_reg_defines.svh

Register Defines

`UVM_REG_ADDR_WIDTH	Maximum address width in bits
`UVM_REG_DATA_WIDTH	Maximum data width in bits
`UVM_REG_BYTENABLE_WIDTH	Maximum number of byte enable bits
`UVM_REG_CVR_WIDTH	Maximum number of bits in a uvm_reg_cvr_t coverage model set.

Register Defines

`UVM_REG_ADDR_WIDTH

Maximum address width in bits

Default value is 64. Used to define the uvm_reg_addr_t type.

`UVM_REG_DATA_WIDTH

Maximum data width in bits

Default value is 64. Used to define the uvm_reg_data_t type.

`UVM_REG_BYTENABLE_WIDTH

Maximum number of byte enable bits

Default value is one per byte in `UVM_REG_DATA_WIDTH. Used to define the uvm_reg_byte_en_t type.

`UVM_REG_CVR_WIDTH

Maximum number of bits in a uvm_reg_cvr_t coverage model set.

Default value is 32.

Policy Classes

Each of UVM's policy classes perform a specific task for uvm_object-based objects: printing, comparing, recording, packing, and unpacking. They are implemented separately from *uvm_object* so that users can plug in different ways to print, compare, etc. without modifying the object class being operated on. The user can simply apply a different printer or compare "policy" to change how an object is printed or compared.

Each policy class includes several user-configurable parameters that control the operation. Users may also customize operations by deriving new policy subtypes from these base types. For example, the UVM provides four different *uvm_printer*-based policy classes, each of which print objects in a different format.

- uvm_printer - performs deep printing of *uvm_object*-based objects. The UVM provides several subtypes to *uvm_printer* that print objects in a specific format: uvm_table_printer, uvm_tree_printer, and uvm_line_printer. Each such printer has many configuration options that goven what and how object members are printed.
- uvm_comparer - performs deep comparison of *uvm_object*-based objects. Users may configure what is compared and how miscompares are reported.
- uvm_recorder - performs the task of recording *uvm_object*-based objects to a transaction data base. The implementation is vendor-specific.
- uvm_packer - used to pack (serialize) and unpack *uvm_object*-based properties into bit, byte, or int arrays and back again.

Summary

Policy Classes

Each of UVM's policy classes perform a specific task for uvm_object-based objects: printing, comparing, recording, packing, and unpacking.

uvm_printer

The uvm_printer class provides an interface for printing uvm_objects in various formats. Subtypes of uvm_printer implement different print formats, or policies.

A user-defined printer format can be created, or one of the following four built-in printers can be used:

- uvm_printer - provides base printer functionality; must be overridden.
- uvm_table_printer - prints the object in a tabular form.
- uvm_tree_printer - prints the object in a tree form.
- uvm_line_printer - prints the information on a single line, but uses the same object separators as the tree printer.

Printers have knobs that you use to control what and how information is printed. These knobs are contained in a separate knob class:

- uvm_printer_knobs - common printer settings

For convenience, global instances of each printer type are available for direct reference in your testbenches.

- uvm_default_tree_printer
- uvm_default_line_printer
- uvm_default_table_printer
- uvm_default_printer (set to default_table_printer by default)

When uvm_object::print and uvm_object::sprint are called without specifying a printer, the uvm_default_printer is used.

Contents

knobs

```
uvm_printer_knobs knobs = new
```

The knob object provides access to the variety of knobs associated with a specific printer instance.

Methods for printer usage

print_int

```
virtual function void print_int (string          name,
```

```
                                  uvm_bitstream_t value,
                                  int             size,
                                  uvm_radix_enum  radix           = UVM_NORADI
                                  byte            scope_separator = ".",
                                  string          type_name       = ""
```

Prints an integral field.

name	The name of the field.
value	The value of the field.
size	The number of bits of the field (maximum is 4096).
radix	The radix to use for printingthe printer knob for radix is used if no radix is specified.
scope_separator	is used to find the leaf name since many printers only print the leaf name of a field. Typical values for the separator are . (dot) or [(open bracket).

print_object

```
virtual function void print_object (string     name,
                                    uvm_object value,
                                    byte       scope_separator = ".")
```

Prints an object. Whether the object is recursed depends on a variety of knobs, such as the depth knob; if the current depth is at or below the depth setting, then the object is not recursed.

By default, the children of uvm_components are printed. To turn this behavior off, you must set the uvm_component::print_enabled bit to 0 for the specific children you do not want automatically printed.

print_string

```
virtual function void print_string (string name,
                                    string value,
                                    byte   scope_separator = ".")
```

Prints a string field.

print_time

```
virtual function void print_time (string name,
                                  time   value,
                                  byte   scope_separator = ".")
```

Prints a time value. name is the name of the field, and value is the value to print.

The print is subject to the *$timeformat* system task for formatting time values.

print_string

Prints a string field.

print_generic

```
virtual function void print_generic (string name,
                                     string type_name,
                                     int    size,
                                     string value,
                                     byte   scope_separator = ".")
```

Prints a field having the given *name*, *type_name*, *size*, and *value*.

Methods for printer subtyping

emit

```
virtual function string emit ()
```

Emits a string representing the contents of an object in a format defined by an extension of this object.

format_row

```
virtual function string format_row (uvm_printer_row_info row)
```

Hook for producing custom output of a single field (row).

format_row

Hook to override base header with a custom header.

format_header

Hook to override base footer with a custom footer.

adjust_name

```
virtual protected function string adjust_name (string id,
                                               byte   scope_separator = ".")
```

Prints a field's name, or *id*, which is the full instance name.

The intent of the separator is to mark where the leaf name starts if the printer if configured to print only the leaf name of the identifier.

print_array_header

```
virtual function void print_array_header(string name,
                                         int    size,
                                         string arraytype       = "array",
                                         byte   scope_separator = "."     )
```

Prints the header of an array. This function is called before each individual element is printed. print_array_footer is called to mark the completion of array printing.

print_array_range

```
virtual function void print_array_range (int min,
                                         int max )
```

Prints a range using ellipses for values. This method is used when honoring the array knobs for partial printing of large arrays, uvm_printer_knobs::begin_elements and uvm_printer_knobs::end_elements.

This function should be called after begin_elements have been printed and before end_elements have been printed.

print_array_footer

```
virtual function void print_array_footer (int size = )
```

Prints the header of a footer. This function marks the end of an array print. Generally, there is no output associated with the array footer, but this method lets the printer know that the array printing is complete.

uvm_table_printer

The table printer prints output in a tabular format.

The following shows sample output from the table printer.

```
----------------------------------------------------
Name         Type             Size        Value
----------------------------------------------------
c1           container        -           @1013
d1           mydata           -           @1022
v1           integral         32          'hcb8f1c97
e1           enum             32          THREE
str          string           2           hi
value        integral         12          'h2d
----------------------------------------------------
```

Summary

uvm_table_printer

The table printer prints output in a tabular format.

Class Hierarchy

uvm_printer

uvm_table_printer

Class Declaration

```
class uvm_table_printer extends uvm_printer
```

Variables

new	Creates a new instance of *uvm_table_printer*.

Methods

emit	Formats the collected information from prior calls to *print_** into table format.

Variables

new

```
function new()
```

Creates a new instance of *uvm_table_printer*.

Methods

emit

```
virtual function string emit()
```

Formats the collected information from prior calls to *print_** into table format.

uvm_tree_printer

By overriding various methods of the uvm_printer super class, the tree printer prints output in a tree format.

The following shows sample output from the tree printer.

```
c1: (container@1013) {
  d1: (mydata@1022) {
       v1: 'hcb8f1c97
       e1: THREE
       str: hi
  }
  value: 'h2d
}
```

Summary

uvm_tree_printer

By overriding various methods of the uvm_printer super class, the tree printer prints output in a tree format.

Class Hierarchy

uvm_printer

uvm_tree_printer

CLASS DECLARATION

```
class uvm_tree_printer extends uvm_printer
```

VARIABLES

new	Creates a new instance of *uvm_tree_printer*.

METHODS

emit	Formats the collected information from prior calls to *print_** into hierarchical tree format.

VARIABLES

new

```
function new()
```

Creates a new instance of *uvm_tree_printer*.

METHODS

emit

```
virtual function string emit()
```

Formats the collected information from prior calls to *print_** into hierarchical tree format.

uvm_line_printer

The line printer prints output in a line format.

The following shows sample output from the line printer.

```
c1: (container@1013) { d1: (mydata@1022) { v1: 'hcb8f1c97 e1: THREE str: hi
} value: 'h2d }
```

Summary

uvm_line_printer

The line printer prints output in a line format.

CLASS HIERARCHY

```
uvm_printer
uvm_tree_printer
```

uvm_line_printer

CLASS DECLARATION

```
class uvm_line_printer extends uvm_tree_printer
```

VARIABLES

new	Creates a new instance of *uvm_line_printer*.

VARIABLES

new

```
function new()
```

Creates a new instance of *uvm_line_printer*. It differs from the uvm_tree_printer only in that the output contains no line-feeds and indentation.

uvm_printer_knobs

The *uvm_printer_knobs* class defines the printer settings available to all printer subtypes.

Summary

uvm_printer_knobs

The *uvm_printer_knobs* class defines the printer settings available to all printer subtypes.

CLASS DECLARATION

```
class uvm_printer_knobs
```

VARIABLES

header	Indicates whether the <print_header> function should be called when printing an object.
footer	Indicates whether the <print_footer> function should be called when printing an object.
full_name	Indicates whether <adjust_name> should print the full name of an identifier or just the leaf name.
identifier	Indicates whether <adjust_name> should print the identifier.
type_name	Controls whether to print a field's type name.
size	Controls whether to print a field's size.
depth	Indicates how deep to recurse when printing objects.
reference	Controls whether to print a unique reference ID for object handles.
begin_elements	Defines the number of elements at the head of a list to print.
end_elements	This defines the number of elements at the end of a list that should be printed.
prefix	Specifies the string prepended to each output line
indent	This knob specifies the number of spaces to use for level indentation.
show_root	This setting indicates whether or not the initial object that is printed (when current depth is 0) prints the full path

	name.
mcd	This is a file descriptor, or multi-channel descriptor, that specifies where the print output should be directed.
separator	For tree printers only, determines the opening and closing separators used for nested objects.
show_radix	Indicates whether the radix string ('h, and so on) should be prepended to an integral value when one is printed.
default_radix	This knob sets the default radix to use for integral values when no radix enum is explicitly supplied to the print_int() method.
dec_radix	This string should be prepended to the value of an integral type when a radix of UVM_DEC is used for the radix of the integral object.
bin_radix	This string should be prepended to the value of an integral type when a radix of UVM_BIN is used for the radix of the integral object.
oct_radix	This string should be prepended to the value of an integral type when a radix of UVM_OCT is used for the radix of the integral object.
unsigned_radix	This is the string which should be prepended to the value of an integral type when a radix of UVM_UNSIGNED is used for the radix of the integral object.
hex_radix	This string should be prepended to the value of an integral type when a radix of UVM_HEX is used for the radix of the integral object.

METHODS

get_radix_str	Converts the radix from an enumerated to a printable radix according to the radix printing knobs (bin_radix, and so on).

VARIABLES

header

```
bit header = 1
```

Indicates whether the <print_header> function should be called when printing an object.

footer

```
bit footer = 1
```

Indicates whether the <print_footer> function should be called when printing an object.

full_name

```
bit full_name = 0
```

Indicates whether <adjust_name> should print the full name of an identifier or just the leaf name.

identifier

```
bit identifier = 1
```

Indicates whether <adjust_name> should print the identifier. This is useful in cases where you just want the values of an object, but no identifiers.

type_name

```
bit type_name = 1
```

Controls whether to print a field's type name.

size

```
bit size = 1
```

Controls whether to print a field's size.

depth

```
int depth = -1
```

Indicates how deep to recurse when printing objects. A depth of -1 means to print everything.

reference

```
bit reference = 1
```

Controls whether to print a unique reference ID for object handles. The behavior of this knob is simulator-dependent.

begin_elements

```
int begin_elements = 5
```

Defines the number of elements at the head of a list to print. Use -1 for no max.

end_elements

```
int end_elements = 5
```

This defines the number of elements at the end of a list that should be printed.

prefix

```
string prefix = ""
```

Specifies the string prepended to each output line

indent

```
int indent = 2
```

This knob specifies the number of spaces to use for level indentation. The default level indentation is two spaces.

show_root

```
bit show_root = 0
```

This setting indicates whether or not the initial object that is printed (when current depth is 0) prints the full path name. By default, the first object is treated like all other objects and only the leaf name is printed.

mcd

```
int mcd = UVM_STDOUT
```

This is a file descriptor, or multi-channel descriptor, that specifies where the print output should be directed.

By default, the output goes to the standard output of the simulator.

separator

```
string separator = "{}"
```

For tree printers only, determines the opening and closing separators used for nested objects.

show_radix

```
bit show_radix = 1
```

Indicates whether the radix string (`h, and so on) should be prepended to an integral value when one is printed.

default_radix

```
uvm_radix_enum default_radix = UVM_HEX
```

This knob sets the default radix to use for integral values when no radix enum is explicitly supplied to the print_int() method.

dec_radix

```
string dec_radix = "'d"
```

This string should be prepended to the value of an integral type when a radix of UVM_DEC is used for the radix of the integral object.

When a negative number is printed, the radix is not printed since only signed decimal values can print as negative.

bin_radix

```
string bin_radix = "'b"
```

This string should be prepended to the value of an integral type when a radix of UVM_BIN is used for the radix of the integral object.

oct_radix

```
string oct_radix = "'o"
```

This string should be prepended to the value of an integral type when a radix of UVM_OCT is used for the radix of the integral object.

unsigned_radix

```
string unsigned_radix = "'d"
```

This is the string which should be prepended to the value of an integral type when a radix of UVM_UNSIGNED is used for the radix of the integral object.

hex_radix

```
string hex_radix = "'h"
```

This string should be prepended to the value of an integral type when a radix of UVM_HEX is used for the radix of the integral object.

METHODS

get_radix_str

```
function string get_radix_str(uvm_radix_enum radix)
```

Converts the radix from an enumerated to a printable radix according to the radix printing knobs (bin_radix, and so on).

uvm_comparer

The uvm_comparer class provides a policy object for doing comparisons. The policies determine how miscompares are treated and counted. Results of a comparison are stored in the comparer object. The uvm_object::compare and uvm_object::do_compare methods are passed an uvm_comparer policy object.

Summary

uvm_comparer

The uvm_comparer class provides a policy object for doing comparisons.

Class Declaration

```
class uvm_comparer
```

Variables

policy	Determines whether comparison is UVM_DEEP, UVM_REFERENCE, or UVM_SHALLOW.
show_max	Sets the maximum number of messages to send to the messager for miscompares of an object.
verbosity	Sets the verbosity for printed messages.
sev	Sets the severity for printed messages.
miscompares	This string is reset to an empty string when a comparison is started.
physical	This bit provides a filtering mechanism for fields.
abstract	This bit provides a filtering mechanism for fields.
check_type	This bit determines whether the type, given by uvm_object::get_type_name, is used to verify that the types of two objects are the same.
result	This bit stores the number of miscompares for a given compare operation.

Methods

compare_field	Compares two integral values.
compare_field_int	This method is the same as compare_field except that the arguments are small integers, less than or equal to 64 bits.
compare_field_real	This method is the same as compare_field except that the arguments are real numbers.
compare_object	Compares two class objects using the policy knob to determine whether the comparison should be deep, shallow, or reference.
compare_string	Compares two string variables.
print_msg	Causes the error count to be incremented and the message, *msg*, to be appended to the miscompares string (a newline is used to separate messages).

Variables

policy

```
uvm_recursion_policy_enum policy = UVM_DEFAULT_POLICY
```

Determines whether comparison is UVM_DEEP, UVM_REFERENCE, or UVM_SHALLOW.

show_max

```
int unsigned show_max = 1
```

Sets the maximum number of messages to send to the messager for miscompares of an object.

verbosity

```
int unsigned verbosity = UVM_LOW
```

Sets the verbosity for printed messages.

The verbosity setting is used by the messaging mechanism to determine whether messages should be suppressed or shown.

sev

```
uvm_severity sev = UVM_INFO
```

Sets the severity for printed messages.

The severity setting is used by the messaging mechanism for printing and filtering messages.

miscompares

```
string miscompares = ""
```

This string is reset to an empty string when a comparison is started.

The string holds the last set of miscompares that occurred during a comparison.

physical

```
bit physical = 1
```

This bit provides a filtering mechanism for fields.

The abstract and physical settings allow an object to distinguish between two different classes of fields.

It is up to you, in the uvm_object::do_compare method, to test the setting of this field if you want to use the physical trait as a filter.

abstract

```
bit abstract = 1
```

This bit provides a filtering mechanism for fields.

The abstract and physical settings allow an object to distinguish between two different classes of fields.

It is up to you, in the uvm_object::do_compare method, to test the setting of this field if you want to use the abstract trait as a filter.

check_type

```
bit check_type = 1
```

This bit determines whether the type, given by uvm_object::get_type_name, is used to verify that the types of two objects are the same.

This bit is used by the compare_object method. In some cases it is useful to set this to 0 when the two operands are related by inheritance but are different types.

result

```
int unsigned result = 0
```

This bit stores the number of miscompares for a given compare operation. You can use the result to determine the number of miscompares that were found.

Methods

compare_field

```
virtual function bit compare_field (string          name,
                                    uvm_bitstream_t lhs,
                                    uvm_bitstream_t rhs,
                                    int             size,
                                    uvm_radix_enum  radix = UVM_NORADIX)
```

Compares two integral values.

The *name* input is used for purposes of storing and printing a miscompare.

The left-hand-side *lhs* and right-hand-side *rhs* objects are the two objects used for comparison.

The size variable indicates the number of bits to compare; size must be less than or equal to 4096.

The radix is used for reporting purposes, the default radix is hex.

compare_field_int

```
virtual function bit compare_field_int (string         name,
                                        logic[63:0]    lhs,
                                        logic[63:0]    rhs,
                                        int            size,
                                        uvm_radix_enum radix = UVM_NORADIX)
```

This method is the same as compare_field except that the arguments are small integers, less than or equal to 64 bits. It is automatically called by compare_field if the operand size is less than or equal to 64.

compare_field_real

```
virtual function bit compare_field_real (string name,
                                         real   lhs,
                                         real   rhs  )
```

This method is the same as compare_field except that the arguments are real numbers.

compare_object

```
virtual function bit compare_object (string     name,
                                     uvm_object lhs,
                                     uvm_object rhs  )
```

Compares two class objects using the policy knob to determine whether the comparison should be deep, shallow, or reference.

The name input is used for purposes of storing and printing a miscompare.

The *lhs* and *rhs* objects are the two objects used for comparison.

The *check_type* determines whether or not to verify the object types match (the return from *lhs.get_type_name()* matches *rhs.get_type_name()*).

compare_string

```
virtual function bit compare_string (string name,
                                     string lhs,
                                     string rhs  )
```

Compares two string variables.

The *name* input is used for purposes of storing and printing a miscompare.

The *lhs* and *rhs* objects are the two objects used for comparison.

print_msg

```
function void print_msg (string msg)
```

Causes the error count to be incremented and the message, *msg*, to be appended to the miscompares string (a newline is used to separate messages).

If the message count is less than the show_max setting, then the message is printed to standard-out using the current verbosity and severity settings. See the verbosity and sev variables for more information.

uvm_recorder

The uvm_recorder class provides a policy object for recording uvm_objects. The policies determine how recording should be done.

A default recorder instance, uvm_default_recorder, is used when the uvm_object::record is called without specifying a recorder.

Summary

uvm_recorder

The uvm_recorder class provides a policy object for recording uvm_objects.

CLASS DECLARATION

```
class uvm_recorder
```

VARIABLES

tr_handle	This is an integral handle to a transaction object.
default_radix	This is the default radix setting if record_field is called without a radix.
physical	This bit provides a filtering mechanism for fields.
abstract	This bit provides a filtering mechanism for fields.
identifier	This bit is used to specify whether or not an object's reference should be recorded when the object is recorded.
recursion_policy	Sets the recursion policy for recording objects.

METHODS

record_field	Records an integral field (less than or equal to 4096 bits).
record_field_real	Records an real field.
record_object	Records an object field.
record_string	Records a string field.
record_time	Records a time value.
record_generic	Records the *name-value* pair, where *value* has been converted to a string.

VARIABLES

tr_handle

```
integer tr_handle = 0
```

This is an integral handle to a transaction object. Its use is vendor specific.

A handle of 0 indicates there is no active transaction object.

default_radix

```
uvm_radix_enum default_radix = UVM_HEX
```

This is the default radix setting if record_field is called without a radix.

physical

```
bit physical = 1
```

This bit provides a filtering mechanism for fields.

The abstract and physical settings allow an object to distinguish between two different classes of fields.

It is up to you, in the uvm_object::do_record method, to test the setting of this field if you want to use the physical trait as a filter.

abstract

```
bit abstract = 1
```

This bit provides a filtering mechanism for fields.

The abstract and physical settings allow an object to distinguish between two different classes of fields.

It is up to you, in the uvm_object::do_record method, to test the setting of this field if you want to use the abstract trait as a filter.

identifier

```
bit identifier = 1
```

This bit is used to specify whether or not an object's reference should be recorded when the object is recorded.

recursion_policy

```
uvm_recursion_policy_enum policy = UVM_DEFAULT_POLICY
```

Sets the recursion policy for recording objects.

The default policy is deep (which means to recurse an object).

METHODS

record_field

```
virtual function void record_field (string          name,
                                    uvm_bitstream_t value,
                                    int             size,
                                    uvm_radix_enum  radix  = UVM_NORADIX)
```

Records an integral field (less than or equal to 4096 bits). *name* is the name of the field.

value is the value of the field to record. *size* is the number of bits of the field which apply. *radix* is the uvm_radix_enum to use.

record_field_real

```
virtual function void record_field_real (string name,
                                         real   value)
```

Records an real field. *value* is the value of the field to record.

record_object

```
virtual function void record_object (string     name,
                                     uvm_object value)
```

Records an object field. *name* is the name of the recorded field.

This method uses the recursion_policy to determine whether or not to recurse into the object.

record_string

```
virtual function void record_string (string name,
                                     string value)
```

Records a string field. *name* is the name of the recorded field.

record_time

```
virtual function void record_time (string name,
                                   time   value)
```

Records a time value. *name* is the name to record to the database.

record_generic

```
virtual function void record_generic (string name,
                                      string value)
```

Records the *name*-*value* pair, where *value* has been converted to a string. For example:

```
recorder.record_generic("myvar",$sformatf("%0d",myvar));
```

uvm_packer

The uvm_packer class provides a policy object for packing and unpacking uvm_objects. The policies determine how packing and unpacking should be done. Packing an object causes the object to be placed into a bit (byte or int) array. If the `uvm_field_* macro are used to implement pack and unpack, by default no metadata information is stored for the packing of dynamic objects (strings, arrays, class objects).

Summary

uvm_packer

The uvm_packer class provides a policy object for packing and unpacking uvm_objects.

PACKING

pack_field	Packs an integral value (less than or equal to 4096 bits) into the packed array.
pack_field_int	Packs the integral value (less than or equal to 64 bits) into the pack array.
pack_string	Packs a string value into the pack array.
pack_time	Packs a time *value* as 64 bits into the pack array.
pack_real	Packs a real *value* as 64 bits into the pack array.
pack_object	Packs an object value into the pack array.

UNPACKING

is_null	This method is used during unpack operations to peek at the next 4-bit chunk of the pack data and determine if it is 0.
unpack_field_int	Unpacks bits from the pack array and returns the bit-stream that was unpacked.
unpack_field	Unpacks bits from the pack array and returns the bit-stream that was unpacked.
unpack_string	Unpacks a string.
unpack_time	Unpacks the next 64 bits of the pack array and places them into a time variable.
unpack_real	Unpacks the next 64 bits of the pack array and places them into a real variable.
unpack_object	Unpacks an object and stores the result into *value*.
get_packed_size	Returns the number of bits that were packed.

VARIABLES

physical	This bit provides a filtering mechanism for fields.
abstract	This bit provides a filtering mechanism for fields.
use_metadata	This flag indicates whether to encode metadata when packing dynamic data, or to decode metadata when unpacking.
big_endian	This bit determines the order that integral data is packed (using pack_field, pack_field_int, pack_time, or pack_real) and how the data is unpacked from the pack array (using unpack_field, unpack_field_int, unpack_time, or unpack_real).

PACKING

pack_field

```
virtual function void pack_field (uvm_bitstream_t value,
```

```
                                    int              size  )
```

Packs an integral value (less than or equal to 4096 bits) into the packed array. *size* is the number of bits of *value* to pack.

pack_field_int

```
virtual function void pack_field_int (logic[63:0] value,
                                      int         size  )
```

Packs the integral value (less than or equal to 64 bits) into the pack array. The *size* is the number of bits to pack, usually obtained by *$bits*. This optimized version of pack_field is useful for sizes up to 64 bits.

pack_string

```
virtual function void pack_string (string value)
```

Packs a string value into the pack array.

When the metadata flag is set, the packed string is terminated by a null character to mark the end of the string.

This is useful for mixed language communication where unpacking may occur outside of SystemVerilog UVM.

pack_time

```
virtual function void pack_time (time value)
```

Packs a time *value* as 64 bits into the pack array.

pack_real

```
virtual function void pack_real (real value)
```

Packs a real *value* as 64 bits into the pack array.

The real *value* is converted to a 6-bit scalar value using the function $real2bits before it is packed into the array.

pack_object

```
virtual function void pack_object (uvm_object value)
```

Packs an object value into the pack array.

A 4-bit header is inserted ahead of the string to indicate the number of bits that was packed. If a null object was packed, then this header will be 0.

This is useful for mixed-language communication where unpacking may occur outside of SystemVerilog UVM.

UNPACKING

is_null

```
virtual function bit is_null ()
```

This method is used during unpack operations to peek at the next 4-bit chunk of the pack data and determine if it is 0.

If the next four bits are all 0, then the return value is a 1; otherwise it is 0.

This is useful when unpacking objects, to decide whether a new object needs to be allocated or not.

unpack_field_int

```
virtual function logic[63:0] unpack_field_int (int size)
```

Unpacks bits from the pack array and returns the bit-stream that was unpacked.

size is the number of bits to unpack; the maximum is 64 bits. This is a more efficient variant than unpack_field when unpacking into smaller vectors.

unpack_field

```
virtual function uvm_bitstream_t unpack_field (int size)
```

Unpacks bits from the pack array and returns the bit-stream that was unpacked. *size* is the number of bits to unpack; the maximum is 4096 bits.

unpack_string

```
virtual function string unpack_string (int num_chars = -1)
```

Unpacks a string.

num_chars bytes are unpacked into a string. If num_chars is -1 then unpacking stops on at the first null character that is encountered.

unpack_time

```
virtual function time unpack_time ()
```

Unpacks the next 64 bits of the pack array and places them into a time variable.

unpack_real

```
virtual function real unpack_real ()
```

Unpacks the next 64 bits of the pack array and places them into a real variable.

The 64 bits of packed data are converted to a real using the $bits2real system function.

unpack_object

```
virtual function void unpack_object (uvm_object value)
```

Unpacks an object and stores the result into *value*.

value must be an allocated object that has enough space for the data being unpacked. The first four bits of packed data are used to determine if a null object was packed into the array.

The is_null function can be used to peek at the next four bits in the pack array before calling this method.

get_packed_size

```
virtual function int get_packed_size()
```

Returns the number of bits that were packed.

VARIABLES

physical

```
bit physical = 1
```

This bit provides a filtering mechanism for fields.

The abstract and physical settings allow an object to distinguish between two different classes of fields. It is up to you, in the uvm_object::do_pack and uvm_object::do_unpack methods, to test the setting of this field if you want to use it as a filter.

abstract

```
bit abstract = 0
```

This bit provides a filtering mechanism for fields.

The abstract and physical settings allow an object to distinguish between two different classes of fields. It is up to you, in the uvm_object::do_pack and uvm_object::do_unpack routines, to test the setting of this field if you want to use it as a filter.

use_metadata

```
bit use_metadata = 0
```

This flag indicates whether to encode metadata when packing dynamic data, or to decode metadata when unpacking. Implementations of uvm_object::do_pack and

uvm_object::do_unpack should regard this bit when performing their respective operation. When set, metadata should be encoded as follows:

- For strings, pack an additional null byte after the string is packed.
- For objects, pack 4 bits prior to packing the object itself. Use 4'b0000 to indicate the object being packed is null, otherwise pack 4'b0001 (the remaining 3 bits are reserved).
- For queues, dynamic arrays, and associative arrays, pack 32 bits indicating the size of the array prior to to packing individual elements.

big_endian

```
bit big_endian = 1
```

This bit determines the order that integral data is packed (using pack_field, pack_field_int, pack_time, or pack_real) and how the data is unpacked from the pack array (using unpack_field, unpack_field_int, unpack_time, or unpack_real). When the bit is set, data is associated msb to lsb; otherwise, it is associated lsb to msb.

The following code illustrates how data can be associated msb to lsb and lsb to msb:

```
class mydata extends uvm_object;

  logic[15:0] value = 'h1234;

  function void do_pack (uvm_packer packer);
    packer.pack_field_int(value, 16);
  endfunction

  function void do_unpack (uvm_packer packer);
    value = packer.unpack_field_int(16);
  endfunction
endclass

mydata d = new;
bit bits[];

initial begin
  d.pack(bits);  // 'b0001001000110100
  uvm_default_packer.big_endian = 0;
  d.pack(bits);  // 'b0010110001001000
end
```

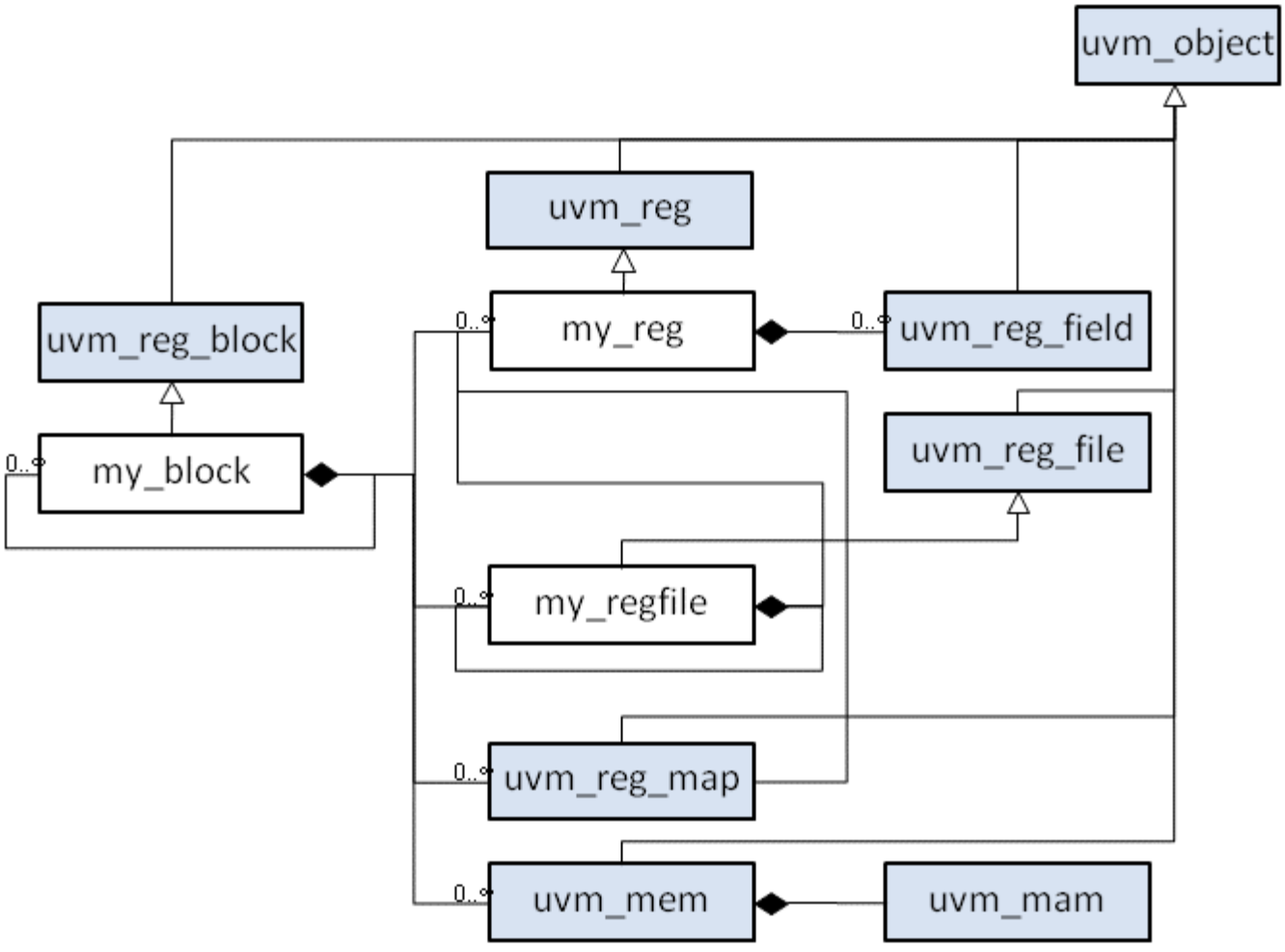
Register Layer

The UVM register layer defines several base classes that, when properly extended, abstract the read/write operations to registers and memories in a design-under-verification.

A register model is typically composed of a hierarchy of blocks that usually map to the design hierarchy. Blocks contain registers, register files and memories.

The UVM register layer classes are not usable as-is. They only provide generic and introspection capabilities. They must be specialized via extensions to provide an abstract view that corresponds to the actual registers and memories in a design. Due to the large number of registers in a design and the numerous small details involved in properly configuring the UVM register layer classes, this specialization is normally done by a model generator. Model generators work from a specification of the registers and memories in a design and are thus able to provide an up-to-date, correct-by-construction register model. Model generators are outside the scope of the UVM library.

The class diagram of a register layer model is shown below.



Summary

Register Layer

The UVM register layer defines several base classes that, when properly extended, abstract the read/write operations to registers and memories in a design-under-verification.

Global Declarations for the Register Layer

This section defines globally available types, enums, and utility classes.

Contents

TYPES

uvm_reg_data_t

2-state data value with `UVM_REG_DATA_WIDTH bits

uvm_reg_data_logic_t

4-state data value with `UVM_REG_DATA_WIDTH bits

uvm_reg_addr_t

2-state address value with `UVM_REG_ADDR_WIDTH bits

uvm_reg_addr_logic_t

4-state address value with `UVM_REG_ADDR_WIDTH bits

uvm_reg_byte_en_t

2-state byte_enable value with `UVM_REG_BYTENABLE_WIDTH bits

uvm_reg_cvr_t

Coverage model value set with `UVM_REG_CVR_WIDTH bits.

Symbolic values for individual coverage models are defined by the uvm_coverage_model_e type.

The following bits in the set are assigned as follows

0-7	UVM pre-defined coverage models
8-15	Coverage models defined by EDA vendors, implemented in a register model generator.
16-23	User-defined coverage models
24..	Reserved

uvm_hdl_path_slice

Slice of an HDL path

Struct that specifies the HDL variable that corresponds to all or a portion of a register.

path	Path to the HDL variable.
offset	Offset of the LSB in the register that this variable implements
size	Number of bits (toward the MSB) that this variable implements

If the HDL variable implements all of the register, *offset* and *size* are specified as -1. For example:

```
r1.add_hdl_path('{ '{"r1", -1, -1} });
```

ENUMERATIONS

uvm_status_e

Return status for register operations

UVM_IS_OK	Operation completed successfully

UVM_NOT_OK	Operation completed with error
UVM_HAS_X	Operation completed successfully bit had unknown bits.

uvm_path_e

Path used for register operation

UVM_FRONTDOOR	Use the front door
UVM_BACKDOOR	Use the back door
UVM_PREDICT	Operation derived from observations by a bus monitor via the uvm_reg_predictor class.
UVM_DEFAULT_PATH	Operation specified by the context

uvm_check_e

Read-only or read-and-check

UVM_NO_CHECK	Read only
UVM_CHECK	Read and check

uvm_endianness_e

Specifies byte ordering

UVM_NO_ENDIAN	Byte ordering not applicable
UVM_LITTLE_ENDIAN	Least-significant bytes first in consecutive addresses
UVM_BIG_ENDIAN	Most-significant bytes first in consecutive addresses
UVM_LITTLE_FIFO	Least-significant bytes first at the same address
UVM_BIG_FIFO	Most-significant bytes first at the same address

uvm_elem_kind_e

Type of element being read or written

UVM_REG	Register
UVM_FIELD	Field
UVM_MEM	Memory location

uvm_access_e

Type of operation begin performed

UVM_READ	Read operation
UVM_WRITE	Write operation

uvm_hier_e

Whether to provide the requested information from a hierarchical context.

UVM_NO_HIER	Provide info from the local context
UVM_HIER	Provide info based on the hierarchical context

uvm_predict_e

How the mirror is to be updated

UVM_PREDICT_DIRECT	Predicted value is as-is
UVM_PREDICT_READ	Predict based on the specified value having been read
UVM_PREDICT_WRITE	Predict based on the specified value having been written

uvm_coverage_model_e

Coverage models available or desired. Multiple models may be specified by bitwise OR'ing individual model identifiers.

UVM_NO_COVERAGE	None
UVM_CVR_REG_BITS	Individual register bits
UVM_CVR_ADDR_MAP	Individual register and memory addresses
UVM_CVR_FIELD_VALS	Field values
UVM_CVR_ALL	All coverage models

uvm_reg_mem_tests_e

Select which pre-defined test sequence to execute.

Multiple test sequences may be selected by bitwise OR'ing their respective symbolic values.

UVM_DO_REG_HW_RESET	Run uvm_reg_hw_reset_seq
UVM_DO_REG_BIT_BASH	Run uvm_reg_bit_bash_seq
UVM_DO_REG_ACCESS	Run uvm_reg_access_seq
UVM_DO_MEM_ACCESS	Run uvm_mem_access_seq
UVM_DO_SHARED_ACCESS	Run uvm_reg_mem_shared_access_seq
UVM_DO_MEM_WALK	Run uvm_mem_walk_seq
UVM_DO_ALL_REG_MEM_TESTS	Run all of the above

Test sequences, when selected, are executed in the order in which they are specified above.

Utility Classes

uvm_hdl_path_concat

Concatenation of HDL variables

An dArray of uvm_hdl_path_slice specifing a concatenation of HDL variables that implement a register in the HDL.

Slices must be specified in most-to-least significant order. Slices must not overlap. Gaps may exists in the concatentation if portions of the registers are not implemented.

For example, the following register

```
       1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
Bits:  5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
      +-+---+-------------+---+-------+
      |A|xxx|      B      |xxx|   C   |
      +-+---+-------------+---+-------+
```

If the register is implementd using a single HDL variable, The array should specify a single slice with its *offset* and *size* specified as -1. For example:

```
concat.set('{ '{"r1", -1, -1} });
```

Summary

uvm_hdl_path_concat

Concatenation of HDL variables

Class Declaration

```
class uvm_hdl_path_concat
```

Variables

slices	Array of individual slices, stored in most-to-least significant order

Methods

set	Initialize the concatenation using an array literal
add_slice	Append the specified *slice* literal to the path concatenation
add_path	Append the specified *path* to the path concatenation, for the specified number of bits at the specified *offset*.

Variables

slices

```
uvm_hdl_path_slice slices[]
```

Array of individual slices, stored in most-to-least significant order

Methods

set

```
function void set(uvm_hdl_path_slice t[])
```

Initialize the concatenation using an array literal

add_slice

```
function void add_slice(uvm_hdl_path_slice slice)
```

Append the specified *slice* literal to the path concatenation

add_path

```
function void add_path(    string   path,
                       int unsigned offset = -1,
                       int unsigned size   = -1 )
```

Append the specified *path* to the path concatenation, for the specified number of bits at the specified *offset*.

uvm_utils

This class contains useful template functions.

Summary

uvm_utils

This class contains useful template functions.

Class Declaration

```
class uvm_utils #(type   TYPE  = int,
                  string FIELD = "config")
```

Methods

find_all	Recursively finds all component instances of the parameter type *TYPE*, starting with the component given by *start*.
get_config	This method gets the object config of type *TYPE* associated with component *comp*.

Methods

find_all

```
static function types_t find_all(uvm_component start)
```

Recursively finds all component instances of the parameter type *TYPE*, starting with the

component given by *start*. Uses uvm_root::find_all.

get_config

```
static function TYPE get_config(uvm_component comp,
                                bit           is_fatal)
```

This method gets the object config of type *TYPE* associated with component *comp*. We check for the two kinds of error which may occur with this kind of operation.

uvm_reg_block

Block abstraction base class

A block represents a design hierarchy. It can contain registers, register files, memories and sub-blocks.

A block has one or more address maps, each corresponding to a physical interface on the block.

Summary

uvm_reg_block

Block abstraction base class

Class Hierarchy

uvm_void

uvm_object

uvm_reg_block

Class Declaration

```
virtual class uvm_reg_block extends uvm_object
```

default_path	Default access path for the registers and memories in this block.

Initialization

new	Create a new instance and type-specific configuration
configure	Instance-specific configuration
create_map	Create an address map in this block
check_data_width	Check that the specified data width (in bits) is less than or equal to the value of `UVM_REG_DATA_WIDTH
set_default_map	Defines the default address map
default_map	Default address map
lock_model	Lock a model and build the address map.
is_locked	Return TRUE if the model is locked.

Introspection

get_name	Get the simple name
get_full_name	Get the hierarchical name
get_parent	Get the parent block
get_root_blocks	Get the all root blocks
find_blocks	Find the blocks whose hierarchical names match the specified *name* glob.
find_block	Find the first block whose hierarchical names match the specified *name* glob.
get_blocks	Get the sub-blocks
get_maps	Get the address maps
get_registers	Get the registers
get_fields	Get the fields
get_virtual_registers	Get the virtual registers
get_virtual_fields	Get the virtual fields
get_block_by_name	Finds a sub-block with the specified simple name.
get_map_by_name	Finds an address map with the specified simple name.
get_reg_by_name	Finds a register with the specified simple name.
get_field_by_name	Finds a field with the specified simple name.
get_mem_by_name	Finds a memory with the specified simple name.
get_vreg_by_name	Finds a virtual register with the specified simple

	name.
get_vfield_by_name	Finds a virtual field with the specified simple name.
COVERAGE	
build_coverage	Check if all of the specified coverage model must be built.
add_coverage	Specify that additional coverage models are available.
has_coverage	Check if block has coverage model(s)
set_coverage	Turns on coverage measurement.
get_coverage	Check if coverage measurement is on.
sample	Functional coverage measurement method
sample_values	Functional coverage measurement method for field values
ACCESS	
get_default_path	Default access path
reset	Reset the mirror for this block.
needs_update	Check if DUT registers need to be written
update	Batch update of register.
mirror	Update the mirrored values
write_reg_by_name	Write the named register
read_reg_by_name	Read the named register
write_mem_by_name	Write the named memory
read_mem_by_name	Read the named memory
BACKDOOR	
get_backdoor	Get the user-defined backdoor for all registers in this block
set_backdoor	Set the user-defined backdoor for all registers in this block
clear_hdl_path	Delete HDL paths
add_hdl_path	Add an HDL path
has_hdl_path	Check if a HDL path is specified
get_hdl_path	Get the incremental HDL path(s)
get_full_hdl_path	Get the full hierarchical HDL path(s)
set_default_hdl_path	Set the default design abstraction
get_default_hdl_path	Get the default design abstraction
set_hdl_path_root	Specify a root HDL path
is_hdl_path_root	Check if this block has an absolute path

default_path

```
uvm_path_e default_path = UVM_DEFAULT_PATH
```

Default access path for the registers and memories in this block.

INITIALIZATION

new

```
function new(string name         = "",
             int    has_coverage = UVM_NO_COVERAGE)
```

Create a new instance and type-specific configuration

Creates an instance of a block abstraction class with the specified name.

has_coverage specifies which functional coverage models are present in the extension of the block abstraction class. Multiple functional coverage models may be specified by adding their symbolic names, as defined by the uvm_coverage_model_e type.

configure

```
function void configure(uvm_reg_block parent   = null,
                        string        hdl_path = ""   )
```

Instance-specific configuration

Specify the parent block of this block. A block without parent is a root block.

If the block file corresponds to a hierarchical RTL structure, it's contribution to the HDL path is specified as the *hdl_path*. Otherwise, the block does not correspond to a hierarchical RTL structure (e.g. it is physically flattened) and does not contribute to the hierarchical HDL path of any contained registers or memories.

create_map

```
virtual function uvm_reg_map create_map(     string           name,
                                             uvm_reg_addr_t   base_addr,
                                         int unsigned         n_bytes,
                                             uvm_endianness_e endian,
                                             bit              byte_addressing
```

Create an address map in this block

Create an address map with the specified *name*. The base address is usually 0. *n_bytes* specifies the number of bytes in the datapath that accesses this address map. *endian* specifies the endianness, should a register or sub-map with a greater number of bytes be accessed.

```
APB = create_map("APB", 0, 1, UVM_LITTLE_ENDIAN);
```

check_data_width

```
protected static function bit check_data_width(int unsigned width)
```

Check that the specified data width (in bits) is less than or equal to the value of `UVM_REG_DATA_WIDTH

This method is designed to be called by a static initializer

```
class my_blk extends uvm_reg_block;
  local static bit m_data_width = check_data_width(356);
  ...
endclass
```

set_default_map

```
function void set_default_map (uvm_reg_map map)
```

Defines the default address map

Set the specified address map as the default_map for this block. The address map must be a map of this address block.

default_map

```
uvm_reg_map default_map
```

Default address map

Default address map for this block, to be used when no address map is specified for a register operation and that register is accessible from more than one address map.

It is also the implciit address map for a block with a single, unamed address map because it has only one physical interface.

lock_model

```
virtual function void lock_model()
```

Lock a model and build the address map.

Recursively lock an entire register model and build the address maps to enable the uvm_reg_map::get_reg_by_offset() and uvm_reg_map::get_mem_by_offset() methods.

Once locked, no further structural changes, such as adding registers or memories, can be made.

It is not possible to unlock a model.

is_locked

```
function bit is_locked()
```

Return TRUE if the model is locked.

Introspection

get_name

Get the simple name

Return the simple object name of this block.

get_full_name

```
virtual function string get_full_name()
```

Get the hierarchical name

Return the hierarchal name of this block. The base of the hierarchical name is the root block.

get_parent

```
virtual function uvm_reg_block get_parent()
```

Get the parent block

If this a top-level block, returns *null*.

get_root_blocks

```
static function void get_root_blocks(ref uvm_reg_block blks[$])
```

Get the all root blocks

Returns an array of all root blocks in the simulation.

find_blocks

```
static function int find_blocks(input string        name,
                                ref uvm_reg_block blks[$],
                                input uvm_reg_block root     = null,
                                input uvm_object    accessor = null )
```

Find the blocks whose hierarchical names match the specified *name* glob. If a *root* block is specified, the name of the blocks are relative to that block, otherwise they are absolute.

Returns the number of blocks found.

find_block

```
static function uvm_reg_block find_block(input string        name,
                                         input uvm_reg_block root     = null,
                                         input uvm_object    accessor = null
```

Find the first block whose hierarchical names match the specified *name* glob. If a *root* block is specified, the name of the blocks are relative to that block, otherwise they are absolute.

Returns the first block found or *null* otherwise. A warning is issued if more than one block is found.

get_blocks

```
virtual function void get_blocks (  ref uvm_reg_block blks[$],
                                  input uvm_hier_e    hier     = UVM_HIER)
```

Get the sub-blocks

Get the blocks instantiated in this blocks. If *hier* is TRUE, recursively includes any sub-blocks.

get_maps

```
virtual function void get_maps (ref uvm_reg_map maps[$])
```

Get the address maps

Get the address maps instantiated in this block.

get_registers

```
virtual function void get_registers (  ref uvm_reg    regs[$],
                                     input uvm_hier_e hier     = UVM_HIER)
```

Get the registers

Get the registers instantiated in this block. If *hier* is TRUE, recursively includes the registers in the sub-blocks.

Note that registers may be located in different and/or multiple address maps. To get the registers in a specific address map, use the uvm_reg_map::get_registers() method.

get_fields

```
virtual function void get_fields (  ref uvm_reg_field fields[$],
                                  input uvm_hier_e    hier       = UVM_HIER)
```

Get the fields

Get the fields in the registers instantiated in this block. If *hier* is TRUE, recursively includes the fields of the registers in the sub-blocks.

get_virtual_registers

```
virtual function void get_virtual_registers(  ref uvm_vreg   regs[$],
                                            input uvm_hier_e hier     = UVM_H
```

Get the virtual registers

Get the virtual registers instantiated in this block. If *hier* is TRUE, recursively includes the virtual registers in the sub-blocks.

get_virtual_fields

```
virtual function void get_virtual_fields (  ref uvm_vreg_field fields[$],
                                          input uvm_hier_e     hier       = U
```

Get the virtual fields

Get the virtual fields from the virtual registers instantiated in this block. If *hier* is TRUE, recursively includes the virtual fields in the virtual registers in the sub-blocks.

get_block_by_name

```
virtual function uvm_reg_block get_block_by_name (string name)
```

Finds a sub-block with the specified simple name.

The name is the simple name of the block, not a hierarchical name. relative to this block. If no block with that name is found in this block, the sub-blocks are searched for a block of that name and the first one to be found is returned.

If no blocks are found, returns *null*.

get_map_by_name

```
virtual function uvm_reg_map get_map_by_name (string name)
```

Finds an address map with the specified simple name.

The name is the simple name of the address map, not a hierarchical name. relative to this block. If no map with that name is found in this block, the sub-blocks are searched for a map of that name and the first one to be found is returned.

If no address maps are found, returns *null*.

get_reg_by_name

```
virtual function uvm_reg get_reg_by_name (string name)
```

Finds a register with the specified simple name.

The name is the simple name of the register, not a hierarchical name. relative to this block. If no register with that name is found in this block, the sub-blocks are searched for a register of that name and the first one to be found is returned.

If no registers are found, returns *null*.

get_field_by_name

```
virtual function uvm_reg_field get_field_by_name (string name)
```

Finds a field with the specified simple name.

The name is the simple name of the field, not a hierarchical name. relative to this block. If no field with that name is found in this block, the sub-blocks are searched for a field of that name and the first one to be found is returned.

If no fields are found, returns *null*.

get_mem_by_name

```
virtual function uvm_mem get_mem_by_name (string name)
```

Finds a memory with the specified simple name.

The name is the simple name of the memory, not a hierarchical name. relative to this block. If no memory with that name is found in this block, the sub-blocks are searched for a memory of that name and the first one to be found is returned.

If no memories are found, returns *null*.

get_vreg_by_name

```
virtual function uvm_vreg get_vreg_by_name (string name)
```

Finds a virtual register with the specified simple name.

The name is the simple name of the virtual register, not a hierarchical name. relative to this block. If no virtual register with that name is found in this block, the sub-blocks are searched for a virtual register of that name and the first one to be found is returned.

If no virtual registers are found, returns *null*.

get_vfield_by_name

```
virtual function uvm_vreg_field get_vfield_by_name (string name)
```

Finds a virtual field with the specified simple name.

The name is the simple name of the virtual field, not a hierarchical name. relative to this block. If no virtual field with that name is found in this block, the sub-blocks are searched for a virtual field of that name and the first one to be found is returned.

If no virtual fields are found, returns *null*.

COVERAGE

build_coverage

```
protected function uvm_reg_cvr_t build_coverage(uvm_reg_cvr_t models)
```

Check if all of the specified coverage model must be built.

Check which of the specified coverage model must be built in this instance of the block abstraction class, as specified by calls to uvm_reg::include_coverage().

Models are specified by adding the symbolic value of individual coverage model as defined in uvm_coverage_model_e. Returns the sum of all coverage models to be built in the block model.

add_coverage

```
virtual protected function void add_coverage(uvm_reg_cvr_t models)
```

Specify that additional coverage models are available.

Add the specified coverage model to the coverage models available in this class. Models are specified by adding the symbolic value of individual coverage model as defined in uvm_coverage_model_e.

This method shall be called only in the constructor of subsequently derived classes.

has_coverage

```
virtual function bit has_coverage(uvm_reg_cvr_t models)
```

Check if block has coverage model(s)

Returns TRUE if the block abstraction class contains a coverage model for all of the models specified. Models are specified by adding the symbolic value of individual coverage model as defined in uvm_coverage_model_e.

set_coverage

```
virtual function uvm_reg_cvr_t set_coverage(uvm_reg_cvr_t is_on)
```

Turns on coverage measurement.

Turns the collection of functional coverage measurements on or off for this block and all blocks, registers, fields and memories within it. The functional coverage measurement is turned on for every coverage model specified using uvm_coverage_model_e symbolic identifers. 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 are present in the various abstraction classes, then enabled during construction. See the uvm_reg_block::has_coverage() method to identify the available functional coverage models.

get_coverage

```
virtual function bit get_coverage(uvm_reg_cvr_t is_on = UVM_CVR_ALL)
```

Check if coverage measurement is on.

Returns TRUE if measurement for all of the specified functional coverage models are currently on. Multiple functional coverage models can be specified by adding the functional coverage model identifiers.

See uvm_reg_block::set_coverage() for more details.

sample

```
protected virtual function void sample(uvm_reg_addr_t offset,
                                       bit            is_read,
                                       uvm_reg_map    map     )
```

Functional coverage measurement method

This method is invoked by the block abstraction class whenever an address within one of its address map is succesfully read or written. The specified offset is the offset within the block, not an absolute address.

Empty by default, this method may be extended by the abstraction class generator to perform the required sampling in any provided functional coverage model.

sample_values

```
virtual function void sample_values()
```

Functional coverage measurement method for field values

This method is invoked by the user or by the uvm_reg_block::sample_values() method of the parent block to trigger the sampling of the current field values in the block-level functional coverage model. It recursively invokes the uvm_reg_block::sample_values() and uvm_reg::sample_values() methods in the blocks and registers in this block.

This method may be extended by the abstraction class generator to perform the required sampling in any provided field-value functional coverage model. If this method is extended, it MUST call super.sample_values().

Access

get_default_path

```
virtual function uvm_path_e get_default_path()
```

Default access path

Returns the default access path for this block.

reset

```
virtual function void reset(string kind = "HARD")
```

Reset the mirror for this block.

Sets the mirror value of all registers in the block and sub-blocks to the reset value corresponding to the specified reset event. See uvm_reg_field::reset() for more details. Does not actually set the value of the registers in the design, only the values mirrored in their corresponding mirror.

needs_update

```
virtual function bit needs_update()
```

Check if DUT registers need to be written

If a mirror value has been modified in the abstraction model without actually updating the actual register (either through randomization or via the uvm_reg::set() method, the mirror and state of the registers are outdated. The corresponding registers in the DUT need to be updated.

This method returns TRUE if the state of at lest one register in the block or sub-blocks needs to be updated to match the mirrored values. The mirror values, or actual content of registers, are not modified. For additional information, see uvm_reg_block::update() method.

update

```
virtual task update(output uvm_status_e      status,
                    input uvm_path_e         path      = UVM_DEFAULT_PATH,
                    input uvm_sequence_base  parent    = null,
                    input int                prior     = -1,
                    input uvm_object         extension = null,
                    input string             fname     = "",
                    input int                lineno    = 0               )
```

Batch update of register.

Using the minimum number of write operations, updates the registers in the design to match the mirrored values in this block and sub-blocks. The update can be performed using the physical interfaces (front-door access) or back-door accesses. This method performs the reverse operation of uvm_reg_block::mirror().

mirror

```
virtual task mirror(output uvm_status_e      status,
                    input uvm_check_e        check     = UVM_NO_CHECK,
                    input uvm_path_e         path      = UVM_DEFAULT_PATH,
                    input uvm_sequence_base  parent    = null,
                    input int                prior     = -1,
                    input uvm_object         extension = null,
                    input string             fname     = "",
                    input int                lineno    = 0               )
```

Update the mirrored values

Read all of the registers in this block and sub-blocks and update their 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 accesses. If the *check* argument is specified as UVM_CHECK, 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 uvm_reg_block::update().

write_reg_by_name

```
virtual task write_reg_by_name(output uvm_status_e      status,
                               input string             name,
                               input uvm_reg_data_t     data,
                               input uvm_path_e         path      = UVM_DEFAU
                               input uvm_reg_map        map       = null,
                               input uvm_sequence_base  parent    = null,
                               input int                prior     = -1,
                               input uvm_object         extension = null,
                               input string             fname     = "",
                               input int                lineno    = 0
```

Write the named register

Equivalent to get_reg_by_name() followed by uvm_reg::write()

read_reg_by_name

```
virtual task read_reg_by_name(output uvm_status_e      status,
                              input string             name,
                             output uvm_reg_data_t     data,
                              input uvm_path_e         path      = UVM_DEFAUL
                              input uvm_reg_map        map       = null,
                              input uvm_sequence_base  parent    = null,
```

```
                                input int                prior     = -1,
                                input uvm_object         extension = null,
                                input string             fname     = "",
                                input int                lineno    = 0
```

Read the named register

Equivalent to get_reg_by_name() followed by uvm_reg::read()

write_mem_by_name

```
virtual task write_mem_by_name(output uvm_status_e       status,
                               input string              name,
                               input uvm_reg_addr_t      offset,
                               input uvm_reg_data_t      data,
                               input uvm_path_e          path      = UVM_DEFAU
                               input uvm_reg_map         map       = null,
                               input uvm_sequence_base   parent    = null,
                               input int                 prior     = -1,
                               input uvm_object          extension = null,
                               input string              fname     = "",
                               input int                 lineno    = 0
```

Write the named memory

Equivalent to get_mem_by_name() followed by uvm_mem::write()

read_mem_by_name

```
virtual task read_mem_by_name(output uvm_status_e       status,
                              input string              name,
                              input uvm_reg_addr_t      offset,
                             output uvm_reg_data_t      data,
                              input uvm_path_e          path      = UVM_DEFAUL
                              input uvm_reg_map         map       = null,
                              input uvm_sequence_base   parent    = null,
                              input int                 prior     = -1,
                              input uvm_object          extension = null,
                              input string              fname     = "",
                              input int                 lineno    = 0
```

Read the named memory

Equivalent to get_mem_by_name() followed by uvm_mem::read()

Backdoor

get_backdoor

```
function uvm_reg_backdoor get_backdoor(bit inherited = 1)
```

Get the user-defined backdoor for all registers in this block

Return the user-defined backdoor for all register in this block and all sub-blocks -- unless overriden by a backdoor set in a lower-level block or in the register itself.

If *inherited* is TRUE, returns the backdoor of the parent block if none have been specified

for this block.

set_backdoor

```
function void set_backdoor (uvm_reg_backdoor bkdr,
                            string           fname  = "",
                            int              lineno = 0 )
```

Set the user-defined backdoor for all registers in this block

Defines the backdoor mechanism for all registers instantiated in this block and sub-blocks, unless overriden by a definition in a lower-level block or register.

clear_hdl_path

```
function void clear_hdl_path (string kind = "RTL")
```

Delete HDL paths

Remove any previously specified HDL path to the block instance for the specified design abstraction.

add_hdl_path

```
function void add_hdl_path (string path,
                            string kind  = "RTL")
```

Add an HDL path

Add the specified HDL path to the block instance for the specified design abstraction. This method may be called more than once for the same design abstraction if the block is physically duplicated in the design abstraction

has_hdl_path

```
function bit has_hdl_path (string kind = "")
```

Check if a HDL path is specified

Returns TRUE if the block instance has a HDL path defined for the specified design abstraction. If no design abstraction is specified, uses the default design abstraction specified for this block or the nearest block ancestor with a specified default design abstraction.

get_hdl_path

```
function void get_hdl_path (  ref string paths[$],
                            input string kind      = "")
```

Get the incremental HDL path(s)

Returns the HDL path(s) defined for the specified design abstraction in the block instance. Returns only the component of the HDL paths that corresponds to the block, not a full hierarchical path

If no design asbtraction is specified, the default design abstraction for this block is used.

get_full_hdl_path

```
function void get_full_hdl_path (  ref string paths[$],
                                 input string kind      = "",
                                       string separator = ".")
```

Get the full hierarchical HDL path(s)

Returns the full hierarchical HDL path(s) defined for the specified design abstraction in the block instance. There may be more than one path returned even if only one path was defined for the block instance, if any of the parent components have more than one path defined for the same design abstraction

If no design asbtraction is specified, the default design abstraction for each ancestor block is used to get each incremental path.

set_default_hdl_path

```
function void set_default_hdl_path (string kind)
```

Set the default design abstraction

Set the default design abstraction for this block instance.

get_default_hdl_path

```
function string get_default_hdl_path ()
```

Get the default design abstraction

Returns the default design abstraction for this block instance. If a default design abstraction has not been explicitly set for this block instance, returns the default design absraction for the nearest block ancestor. Returns “” if no default design abstraction has been specified.

set_hdl_path_root

```
function void set_hdl_path_root (string path,
                                 string kind  = "RTL")
```

Specify a root HDL path

Set the specified path as the absolute HDL path to the block instance for the specified design abstraction. This absolute root path is preppended to all hierarchical paths under this block. The HDL path of any ancestor block is ignored. This method overrides any incremental path for the same design abstraction specified using add_hdl_path.

is_hdl_path_root

```
function bit is_hdl_path_root (string kind = "")
```

Check if this block has an absolute path

Returns TRUE if an absolute HDL path to the block instance for the specified design abstraction has been defined. If no design asbtraction is specified, the default design abstraction for this block is used.

uvm_reg_map

```
Address map abstraction class
```

This class represents an address map. An address map is a collection of registers and memories accessible via a specific physical interface. Address maps can be composed into higher-level address maps.

Address maps are created using the uvm_reg_block::create_map() method.

Summary

uvm_reg_map

Class Hierarchy

uvm_void

uvm_object

uvm_reg_map

Class Declaration

```
class uvm_reg_map extends uvm_object
```

Initialization

new	Create a new instance
configure	Instance-specific configuration
add_reg	Add a register
add_mem	Add a memory
add_submap	Add an address map
set_sequencer	Set the sequencer and adapter associated with this map.
set_submap_offset	Set the offset of the given *submap* to *offset*.
get_submap_offset	Return the offset of the given *submap*.
set_base_addr	Set the base address of this map.
reset	Reset the mirror for all registers in this address map.

Introspection

get_name	Get the simple name
get_full_name	Get the hierarchical name
get_root_map	Get the externally-visible address map
get_parent	Get the parent block
get_parent_map	Get the higher-level address map
get_base_addr	Get the base offset address for this map.
get_n_bytes	Get the width in bytes of the bus associated with this map.
get_base_addr	Gets the endianness of the bus associated with this map.
get_sequencer	Gets the sequencer for the bus associated with this map.
get_adapter	Gets the bus adapter for the bus associated with this map.
get_submaps	Get the address sub-maps
get_registers	Get the registers
get_fields	Get the fields
get_virtual_registers	Get the virtual registers
get_virtual_fields	Get the virtual fields
get_physical_addresses	Translate a local address into external addresses

get_reg_by_offset	Get register mapped at offset
get_mem_by_offset	Get memory mapped at offset

Bus Access

set_auto_predict	Sets the auto-predict mode for his map.
get_auto_predict	Gets the auto-predict mode setting for this map.
do_bus_write	Perform a bus write operation.
do_bus_read	Perform a bus read operation.
do_write	Perform a write operation.
do_read	Perform a read operation.

Initialization

new

```
function new(string name = "uvm_reg_map")
```

Create a new instance

configure

```
function void configure(    uvm_reg_block     parent,
                            uvm_reg_addr_t    base_addr,
                        int unsigned          n_bytes,
                            uvm_endianness_e  endian,
                            bit               byte_addressing = 1)
```

Instance-specific configuration

Configures this map with the following properties.

parent	the block in which this map is created and applied
base_addr	the base address for this map. All registers, memories, and sub-blocks will be at offsets to this address
n_bytes	the byte-width of the bus on which this map is used
endian	the endian format. See uvm_endianness_e for possible values
byte_addressing	specifies whether the address increment is on a per-byte basis. For example, consecutive memory locations with ~n_bytes~=4 (32-bit bus) are 4 apart: 0, 4, 8, and so on. Default is TRUE.

add_reg

```
virtual function void add_reg (uvm_reg           rg,
                               uvm_reg_addr_t    offset,
                               string            rights    = "RW",
                               bit               unmapped  = 0,
                               uvm_reg_frontdoor frontdoor = null )
```

Add a register

Add the specified register instance to this address map. The register is located at the specified base address and has the specified access rights ("RW", "RO" or "WO"). The number of consecutive physical addresses occupied by the register depends on the width

of the register and the number of bytes in the physical interface corresponding to this address map.

If *unmapped* is TRUE, the register does not occupy any physical addresses and the base address is ignored. Unmapped registers require a user-defined *frontdoor* to be specified.

A register may be added to multiple address maps if it is accessible from multiple physical interfaces. A register may only be added to an address map whose parent block is the same as the register's parent block.

add_mem

```
virtual function void add_mem (uvm_mem           mem,
                               uvm_reg_addr_t    offset,
                               string            rights    = "RW",
                               bit               unmapped  = 0,
                               uvm_reg_frontdoor frontdoor = null )
```

Add a memory

Add the specified memory instance to this address map. The memory is located at the specified base address and has the specified access rights ("RW", "RO" or "WO"). The number of consecutive physical addresses occupied by the memory depends on the width and size of the memory and the number of bytes in the physical interface corresponding to this address map.

If *unmapped* is TRUE, the memory does not occupy any physical addresses and the base address is ignored. Unmapped memorys require a user-defined *frontdoor* to be specified.

A memory may be added to multiple address maps if it is accessible from multiple physical interfaces. A memory may only be added to an address map whose parent block is the same as the memory's parent block.

add_submap

```
virtual function void add_submap (uvm_reg_map    child_map,
                                  uvm_reg_addr_t offset    )
```

Add an address map

Add the specified address map instance to this address map. The address map is located at the specified base address. The number of consecutive physical addresses occupied by the submap depends on the number of bytes in the physical interface that corresponds to the submap, the number of addresses used in the submap and the number of bytes in the physical interface corresponding to this address map.

An address map may be added to multiple address maps if it is accessible from multiple physical interfaces. An address map may only be added to an address map in the grand-parent block of the address submap.

set_sequencer

```
virtual function void set_sequencer (uvm_sequencer_base sequencer,
                                     uvm_reg_adapter    adapter    = null)
```

Set the sequencer and adapter associated with this map. This method *must* be called before starting any sequences based on uvm_reg_sequence.

set_submap_offset

```
virtual function void set_submap_offset (uvm_reg_map    submap,
                                         uvm_reg_addr_t offset )
```

Set the offset of the given *submap* to *offset*.

get_submap_offset

```
virtual function uvm_reg_addr_t get_submap_offset (uvm_reg_map submap)
```

Return the offset of the given *submap*.

set_base_addr

```
virtual function void set_base_addr (uvm_reg_addr_t offset)
```

Set the base address of this map.

reset

```
virtual function void reset(string kind = "SOFT")
```

Reset the mirror for all registers in this address map.

Sets the mirror value of all registers in this address map and all of its submaps to the reset value corresponding to the specified reset event. See uvm_reg_field::reset() for more details. Does not actually set the value of the registers in the design, only the values mirrored in their corresponding mirror.

Note that, unlike the other reset() method, the default reset event for this method is "SOFT".

INTROSPECTION

get_name

Get the simple name

Return the simple object name of this address map.

get_full_name

```
virtual function string get_full_name()
```

Get the hierarchical name

Return the hierarchal name of this address map. The base of the hierarchical name is the root block.

get_root_map

```
virtual function uvm_reg_map get_root_map()
```

Get the externally-visible address map

Get the top-most address map where this address map is instantiated. It corresponds to the externally-visible address map that can be accessed by the verification environment.

get_parent

```
virtual function uvm_reg_block get_parent()
```

Get the parent block

Return the block that is the parent of this address map.

get_parent_map

```
virtual function uvm_reg_map get_parent_map()
```

Get the higher-level address map

Return the address map in which this address map is mapped. returns *null* if this is a top-level address map.

get_base_addr

```
virtual function uvm_reg_addr_t get_base_addr (uvm_hier_e hier = UVM_HIER)
```

Get the base offset address for this map. If this map is the root map, the base address is that set with the *base_addr* argument to uvm_reg_block::create_map(). If this map is a submap of a higher-level map, the base address is offset given this submap by the parent map. See set_submap_offset.

get_n_bytes

```
virtual function int unsigned get_n_bytes (uvm_hier_e hier = UVM_HIER)
```

Get the width in bytes of the bus associated with this map. If *hier* is *UVM_HIER*, then gets the effective bus width relative to the system level. The effective bus width is the narrowest bus width from this map to the top-level root map. Each bus access will be limited to this bus width.

get_base_addr

Gets the endianness of the bus associated with this map. If *hier* is set to *UVM_HIER*, gets the system-level endianness.

get_sequencer

```
virtual function uvm_sequencer_base get_sequencer (uvm_hier_e hier = UVM_HIER
```

Gets the sequencer for the bus associated with this map. If *hier* is set to *UVM_HIER*, gets the sequencer for the bus at the system-level. See set_sequencer.

get_adapter

```
virtual function uvm_reg_adapter get_adapter (uvm_hier_e hier = UVM_HIER)
```

Gets the bus adapter for the bus associated with this map. If *hier* is set to *UVM_HIER*, gets the adapter for the bus used at the system-level. See set_sequencer.

get_submaps

```
virtual function void get_submaps (  ref uvm_reg_map maps[$],
                                   input uvm_hier_e  hier     = UVM_HIER)
```

Get the address sub-maps

Get the address maps instantiated in this address map. If *hier* is *UVM_HIER*, recursively includes the address maps, in the sub-maps.

get_registers

```
virtual function void get_registers (  ref uvm_reg    regs[$],
                                     input uvm_hier_e hier     = UVM_HIER)
```

Get the registers

Get the registers instantiated in this address map. If *hier* is *UVM_HIER*, recursively includes the registers in the sub-maps.

get_fields

```
virtual function void get_fields (  ref uvm_reg_field fields[$],
                                  input uvm_hier_e    hier       = UVM_HIER)
```

Get the fields

Get the fields in the registers instantiated in this address map. If *hier* is *UVM_HIER*, recursively includes the fields of the registers in the sub-maps.

get_virtual_registers

```
virtual function void get_virtual_registers (  ref uvm_vreg   regs[$],
                                             input uvm_hier_e hier     = UVM_
```

Get the virtual registers

Get the virtual registers instantiated in this address map. If *hier* is *UVM_HIER*, recursively includes the virtual registers in the sub-maps.

get_virtual_fields

```
virtual function void get_virtual_fields (  ref uvm_vreg_field fields[$],
                                          input uvm_hier_e     hier       = U
```

Get the virtual fields

Get the virtual fields from the virtual registers instantiated in this address map. If *hier* is *UVM_HIER*, recursively includes the virtual fields in the virtual registers in the sub-maps.

get_physical_addresses

```
virtual function int get_physical_addresses(    uvm_reg_addr_t base_addr,
                                                uvm_reg_addr_t mem_offset,
                                            int unsigned       n_bytes,
                                            ref uvm_reg_addr_t addr[]     )
```

Translate a local address into external addresses

Identify the sequence of addresses that must be accessed physically to access the specified number of bytes at the specified address within this address map. Returns the number of bytes of valid data in each access.

Returns in *addr* a list of address in little endian order, with the granularity of the top-level address map.

A register is specified using a base address with *mem_offset* as 0. A location within a memory is specified using the base address of the memory and the index of the location within that memory.

get_reg_by_offset

```
virtual function uvm_reg get_reg_by_offset(uvm_reg_addr_t offset,
                                           bit            read    = 1)
```

Get register mapped at offset

Identify the register located at the specified offset within this address map for the specified type of access. Returns *null* if no such register is found.

The model must be locked using uvm_reg_block::lock_model() to enable this functionality.

get_mem_by_offset

```
virtual function uvm_mem get_mem_by_offset(uvm_reg_addr_t offset)
```

Get memory mapped at offset

Identify the memory located at the specified offset within this address map. The offset may refer to any memory location in that memory. Returns *null* if no such memory is found.

The model must be locked using uvm_reg_block::lock_model() to enable this functionality.

Bus Access

set_auto_predict

```
function void set_auto_predict(bit on = 1)
```

Sets the auto-predict mode for his map.

When *on* is *TRUE*, the register model will automatically update its mirror (what it thinks should be in the DUT) immediately after any bus read or write operation via this map. Before a uvm_reg::write or uvm_reg::read operation returns, the register's uvm_reg::predict method is called to update the mirrored value in the register.

When *on* is *FALSE*, bus reads and writes via this map do not automatically update the mirror. For real-time updates to the mirror in this mode, you connect a uvm_reg_predictor instance to the bus monitor. The predictor takes observed bus transactions from the bus monitor, looks up the associated uvm_reg register given the address, then calls that register's uvm_reg::predict method. While more complex, this mode will capture all register read/write activity, including that not directly descendant from calls to uvm_reg::write and uvm_reg::read.

By default, auto-prediction is turned off.

get_auto_predict

```
function bit get_auto_predict()
```

Gets the auto-predict mode setting for this map.

do_bus_write

```
virtual task do_bus_write (uvm_reg_item       rw,
                           uvm_sequencer_base sequencer,
                           uvm_reg_adapter    adapter   )
```

Perform a bus write operation.

do_bus_read

```
virtual task do_bus_read (uvm_reg_item       rw,
                          uvm_sequencer_base sequencer,
                          uvm_reg_adapter    adapter   )
```

Perform a bus read operation.

do_write

```
virtual task do_write(uvm_reg_item rw)
```

Perform a write operation.

do_read

```
virtual task do_read(uvm_reg_item rw)
```

Perform a read operation.

uvm_reg_file

Register file abstraction base class

A register file is a collection of register files and registers used to create regular repeated structures.

Register files are usually instantiated as arrays.

Summary

uvm_reg_file

Register file abstraction base class

Class Hierarchy

uvm_void

uvm_object

uvm_reg_file

Class Declaration

```
virtual class uvm_reg_file extends uvm_object
```

Initialization

new	Create a new instance
configure	Configure a register file instance

Introspection

get_name	Get the simple name
get_full_name	Get the hierarchical name
get_parent	Get the parent block
get_regfile	Get the parent register file

Backdoor

clear_hdl_path	Delete HDL paths
add_hdl_path	Add an HDL path
has_hdl_path	Check if a HDL path is specified
get_hdl_path	Get the incremental HDL path(s)
get_full_hdl_path	Get the full hierarchical HDL path(s)
set_default_hdl_path	Set the default design abstraction
get_default_hdl_path	Get the default design abstraction

Initialization

new

```
function new (string name = "")
```

Create a new instance

Creates an instance of a register file abstraction class with the specified name.

configure

```
function void configure (uvm_reg_block blk_parent,
                         uvm_reg_file  regfile_parent,
                         string        hdl_path        = "")
```

Configure a register file instance

Specify the parent block and register file of the register file instance. If the register file is instantiated in a block, *regfile_parent* is specified as *null*. If the register file is instantiated in a register file, *blk_parent* must be the block parent of that register file and *regfile_parent* is specified as that register file.

If the register file corresponds to a hierarchical RTL structure, it's contribution to the HDL path is specified as the *hdl_path*. Otherwise, the register file does not correspond to a hierarchical RTL structure (e.g. it is physically flattened) and does not contribute to the hierarchical HDL path of any contained registers.

Introspection

get_name

Get the simple name

Return the simple object name of this register file.

get_full_name

```
virtual function string get_full_name()
```

Get the hierarchical name

Return the hierarchal name of this register file. The base of the hierarchical name is the root block.

get_parent

```
virtual function uvm_reg_block get_parent ()
```

Get the parent block

get_regfile

```
virtual function uvm_reg_file get_regfile ()
```

Get the parent register file

Returns *null* if this register file is instantiated in a block.

Backdoor

clear_hdl_path

```
function void clear_hdl_path (string kind = "RTL")
```

Delete HDL paths

Remove any previously specified HDL path to the register file instance for the specified design abstraction.

add_hdl_path

```
function void add_hdl_path (string path,
                            string kind  = "RTL")
```

Add an HDL path

Add the specified HDL path to the register file instance for the specified design abstraction. This method may be called more than once for the same design abstraction if the register file is physically duplicated in the design abstraction

has_hdl_path

```
function bit has_hdl_path (string kind = "")
```

Check if a HDL path is specified

Returns TRUE if the register file instance has a HDL path defined for the specified design abstraction. If no design abstraction is specified, uses the default design abstraction specified for the nearest enclosing register file or block

If no design asbtraction is specified, the default design abstraction for this register file is used.

get_hdl_path

```
function void get_hdl_path (  ref string paths[$],
                            input string kind      = "")
```

Get the incremental HDL path(s)

Returns the HDL path(s) defined for the specified design abstraction in the register file instance. If no design abstraction is specified, uses the default design abstraction specified for the nearest enclosing register file or block. Returns only the component of the HDL paths that corresponds to the register file, not a full hierarchical path

If no design asbtraction is specified, the default design abstraction for this register file is used.

get_full_hdl_path

```
function void get_full_hdl_path (  ref string paths[$],
                                 input string kind      = "",
                                 input string separator = ".")
```

Get the full hierarchical HDL path(s)

Returns the full hierarchical HDL path(s) defined for the specified design abstraction in the register file instance. If no design abstraction is specified, uses the default design abstraction specified for the nearest enclosing register file or block. There may be more than one path returned even if only one path was defined for the register file instance, if any of the parent components have more than one path defined for the same design abstraction

If no design asbtraction is specified, the default design abstraction for each ancestor register file or block is used to get each incremental path.

set_default_hdl_path

```
function void set_default_hdl_path (string kind)
```

Set the default design abstraction

Set the default design abstraction for this register file instance.

get_default_hdl_path

```
function string get_default_hdl_path ()
```

Get the default design abstraction

Returns the default design abstraction for this register file instance. If a default design abstraction has not been explicitly set for this register file instance, returns the default design absraction for the nearest register file or block ancestor. Returns “” if no default design abstraction has been specified.

uvm_reg

Register abstraction base class

A register represents a set of fields that are accessible as a single entity.

A register may be mapped to one or more address maps, each with different access rights and policy.

Summary

uvm_reg

Register abstraction base class

Class Hierarchy

uvm_void

uvm_object

uvm_reg

Class Declaration

```
virtual class uvm_reg extends uvm_object
```

Initialization

new	Create a new instance and type-specific configuration
configure	Instance-specific configuration
set_offset	Modify the offset of the register

Introspection

get_name	Get the simple name
get_full_name	Get the hierarchical name
get_parent	Get the parent block
get_regfile	Get the parent register file
get_n_maps	Returns the number of address maps this register is mapped in
is_in_map	Returns 1 if this register is in the specified address *map*
get_maps	Returns all of the address *maps* where this register is mapped
get_rights	Returns the access rights of this register.
get_n_bits	Returns the width, in bits, of this register.
get_n_bytes	Returns the width, in bytes, of this register.
get_max_size	Returns the maximum width, in bits, of all registers.
get_fields	Return the fields in this register
get_field_by_name	Return the named field in this register
get_offset	Returns the offset of this register
get_address	Returns the base external physical address of this register
get_addresses	Identifies the external physical address(es) of this register

Access

set	Set the desired value for this register
get	Return the desired value of the fields in the register.
needs_update	Returns 1 if any of the fields need updating
reset	Reset the desired/mirrored value for this register.
get_reset	Get the specified reset value for this register
has_reset	Check if any field in the register has a reset value specified for the specified reset *kind*.
set_reset	Specify or modify the reset value for this register
write	Write the specified value in this register
read	Read the current value from this register

poke	Deposit the specified value in this register
peek	Read the current value from this register
update	Updates the content of the register in the design to match the desired value
mirror	Read the register and update/check its mirror value
predict	Update the mirrored value for this register.
is_busy	Returns 1 if register is currently being read or written.
FRONTDOOR	
set_frontdoor	Set a user-defined frontdoor for this register
get_frontdoor	Returns the user-defined frontdoor for this register
BACKDOOR	
set_backdoor	Set a user-defined backdoor for this register
get_backdoor	Returns the user-defined backdoor for this register
clear_hdl_path	Delete HDL paths
add_hdl_path	Add an HDL path
add_hdl_path_slice	Append the specified HDL slice to the HDL path of the register instance for the specified design abstraction.
has_hdl_path	Check if a HDL path is specified
get_hdl_path	Get the incremental HDL path(s)
get_hdl_path_kinds	Get design abstractions for which HDL paths have been defined
get_full_hdl_path	Get the full hierarchical HDL path(s)
backdoor_read	User-define backdoor read access
backdoor_write	User-defined backdoor read access
backdoor_read_func	User-defined backdoor read access
backdoor_watch	User-defined DUT register change monitor
COVERAGE	
include_coverage	Specify which coverage model that must be included in various block, register or memory abstraction class instances.
build_coverage	Check if all of the specified coverage models must be built.
add_coverage	Specify that additional coverage models are available.
has_coverage	Check if register has coverage model(s)
set_coverage	Turns on coverage measurement.
get_coverage	Check if coverage measurement is on.
sample	Functional coverage measurement method
sample_values	Functional coverage measurement method for field values
CALLBACKS	
pre_write	Called before register write.
post_write	Called after register write.
pre_read	Called before register read.
post_read	Called after register read.

INITIALIZATION

new

```
function new (     string   name         = "",
               int unsigned n_bits,
                    int      has_coverage      )
```

Create a new instance and type-specific configuration

Creates an instance of a register abstraction class with the specified name.

n_bits specifies the total number of bits in the register. Not all bits need to be implemented. This value is usually a multiple of 8.

has_coverage specifies which functional coverage models are present in the extension of the register abstraction class. Multiple functional coverage models may be specified by adding their symbolic names, as defined by the uvm_coverage_model_e type.

configure

```
function void configure (uvm_reg_block blk_parent,
                         uvm_reg_file  regfile_parent = null,
                         string        hdl_path       = ""   )
```

Instance-specific configuration

Specify the parent block of this register. May also set a parent register file for this register,

If the register is implemented in a single HDL variable, it's name is specified as the *hdl_path*. Otherwise, if the register is implemented as a concatenation of variables (usually one per field), then the HDL path must be specified using the add_hdl_path() or add_hdl_path_slice method.

set_offset

```
virtual function void set_offset (uvm_reg_map    map,
                                  uvm_reg_addr_t offset,
                                  bit            unmapped = 0)
```

Modify the offset of the register

The offset of a register within an address map is set using the uvm_reg_map::add_reg() method. This method is used to modify that offset dynamically.

Modifying the offset of a register will make the register model diverge from the specification that was used to create it.

Introspection

get_name

Get the simple name

Return the simple object name of this register.

get_full_name

```
virtual function string get_full_name()
```

Get the hierarchical name

Return the hierarchal name of this register. The base of the hierarchical name is the root block.

get_parent

```
virtual function uvm_reg_block get_parent ()
```

Get the parent block

get_regfile

```
virtual function uvm_reg_file get_regfile ()
```

Get the parent register file

Returns *null* if this register is instantiated in a block.

get_n_maps

```
virtual function int get_n_maps ()
```

Returns the number of address maps this register is mapped in

is_in_map

```
function bit is_in_map (uvm_reg_map map)
```

Returns 1 if this register is in the specified address *map*

get_maps

```
virtual function void get_maps (ref uvm_reg_map maps[$])
```

Returns all of the address *maps* where this register is mapped

get_rights

```
virtual function string get_rights (uvm_reg_map map = null)
```

Returns the access rights of this register.

Returns "RW", "RO" or "WO". The access rights of a register is always "RW", unless it is a shared register with access restriction in a particular address map.

If no address map is specified and the register is mapped in only one address map, that address map is used. If the register is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the register is not mapped in the specified address map, an error message is issued and "RW" is returned.

get_n_bits

```
virtual function int unsigned get_n_bits ()
```

Returns the width, in bits, of this register.

get_n_bytes

```
virtual function int unsigned get_n_bytes()
```

Returns the width, in bytes, of this register. Rounds up to next whole byte if register is not a multiple of 8.

get_max_size

```
static function int unsigned get_max_size()
```

Returns the maximum width, in bits, of all registers.

get_fields

```
virtual function void get_fields (ref uvm_reg_field fields[$])
```

Return the fields in this register

Fills the specified array with the abstraction class for all of the fields contained in this register. Fields are ordered from least-significant position to most-significant position within the register.

get_field_by_name

```
virtual function uvm_reg_field get_field_by_name(string name)
```

Return the named field in this register

Finds a field with the specified name in this register and returns its abstraction class. If no fields are found, returns null.

get_offset

```
virtual function uvm_reg_addr_t get_offset (uvm_reg_map map = null)
```

Returns the offset of this register

Returns the offset of this register in an address *map*.

If no address map is specified and the register is mapped in only one address map, that address map is used. If the register is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the register is not mapped in the specified address map, an error message is issued.

get_address

```
virtual function uvm_reg_addr_t get_address (uvm_reg_map map = null)
```

Returns the base external physical address of this register

Returns the base external physical address of this register if accessed through the specified address *map*.

If no address map is specified and the register is mapped in only one address map, that address map is used. If the register is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the register is not mapped in the specified address map, an error message is issued.

get_addresses

```
virtual function int get_addresses (     uvm_reg_map    map     = null,
                                    ref uvm_reg_addr_t addr[]          )
```

Identifies the external physical address(es) of this register

Computes all of the external physical addresses that must be accessed to completely read or write this register. The addressed are specified in little endian order. Returns the number of bytes transfered on each access.

If no address map is specified and the register is mapped in only one address map, that address map is used. If the register is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the register is not mapped in the specified address map, an error message is issued.

Access

set

```
virtual function void set (uvm_reg_data_t value,
                           string         fname  = "",
                           int            lineno = 0  )
```

Set the desired value for this register

Sets the desired 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 desired value in its corresponding abstraction class in the RegModel model. Use the uvm_reg::update() method to update the actual register with the mirrored value or the uvm_reg::write() method to set the actual register and its mirrored value.

Unless this method is used, the desired value is equal to the mirrored value.

Refer uvm_reg_field::set() for more details on the effect of setting mirror values on fields with different access policies.

To modify the mirrored field values to a specific value, and thus use the mirrored as a scoreboard for the register values in the DUT, use the uvm_reg::predict() method.

get

```
virtual function uvm_reg_data_t get(string fname  = "",
                                    int    lineno = 0  )
```

Return the desired value of the fields in the register.

Does not actually read the value of the register in the design, only the desired value in the abstraction class. Unless set to a different value using the uvm_reg::set(), the desired value and the mirrored value are identical.

Use the uvm_reg::read() or uvm_reg::peek() method to get the actual register value.

If the register contains write-only fields, the desired/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 something different for these fields, the returned value is the actual content.

needs_update

```
virtual function bit needs_update()
```

Returns 1 if any of the fields need updating

See uvm_reg_field::needs_update() for details. Use the uvm_reg::update() to actually update the DUT register.

reset

```
virtual function void reset(string kind = "HARD")
```

Reset the desired/mirrored value for this register.

Sets the desired and mirror value of the fields in this register to the reset value for the specified reset *kind*. See uvm_reg_field.reset() for more details.

Also resets the semaphore that prevents concurrent access to the register. This semaphore must be explicitly reset if a thread accessing this register array was killed in before the access was completed

get_reset

```
virtual function uvm_reg_data_t get_reset(string kind = "HARD")
```

Get the specified reset value for this register

Return the reset value for this register for the specified reset *kind*.

has_reset

```
virtual function bit has_reset(string kind   = "HARD",
                               bit    delete = 0      )
```

Check if any field in the register has a reset value specified for the specified reset *kind*. If *delete* is TRUE, removes the reset value, if any.

set_reset

```
virtual function void set_reset(uvm_reg_data_t value,
```

```
                                  string         kind   = "HARD")
```

Specify or modify the reset value for this register

Specify or modify the reset value for all the fields in the register corresponding to the cause specified by *kind*.

write

```
virtual task write(output uvm_status_e      status,
                   input uvm_reg_data_t     value,
                   input uvm_path_e         path      = UVM_DEFAULT_PATH,
                   input uvm_reg_map        map       = null,
                   input uvm_sequence_base  parent    = null,
                   input int                prior     = -1,
                   input uvm_object         extension = null,
                   input string             fname     = "",
                   input int                lineno    = 0                )
```

Write the specified value in this register

Write *value* in the DUT register that corresponds to this abstraction class instance using the specified access *path*. If the register is mapped in more than one address map, an address *map* 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 mirrored value will be updated using the uvm_reg::predict() method.

read

```
virtual task read(output uvm_status_e      status,
                  output uvm_reg_data_t    value,
                   input uvm_path_e         path      = UVM_DEFAULT_PATH,
                   input uvm_reg_map        map       = null,
                   input uvm_sequence_base  parent    = null,
                   input int                prior     = -1,
                   input uvm_object         extension = null,
                   input string             fname     = "",
                   input int                lineno    = 0                )
```

Read the current value from this register

Read and return *value* from the DUT register that corresponds to this abstraction class instance using the specified access *path*. If the register is mapped in more than one address map, an address *map* 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 set to zero.

The mirrored value will be updated using the uvm_reg::predict() method.

poke

```
virtual task poke(output uvm_status_e      status,
                   input uvm_reg_data_t     value,
                   input string             kind      = "",
                   input uvm_sequence_base  parent    = null,
                   input uvm_object         extension = null,
                   input string             fname     = "",
                   input int                lineno    = 0    )
```

Deposit the specified value in this register

Deposit the value in the DUT register corresponding to this abstraction class instance, as-is, using a back-door access.

Uses the HDL path for the design abstraction specified by *kind*.

The mirrored value will be updated using the uvm_reg::predict() method.

peek

```
virtual task peek(output uvm_status_e      status,
                  output uvm_reg_data_t    value,
                   input string            kind      = "",
                   input uvm_sequence_base parent    = null,
                   input uvm_object        extension = null,
                   input string            fname     = "",
                   input int               lineno    = 0    )
```

Read the current value from this register

Sample the value in the DUT register corresponding to this absraction class instance using a back-door access. The register value is sampled, not modified.

Uses the HDL path for the design abstraction specified by *kind*.

The mirrored value will be updated using the uvm_reg::predict() method.

update

```
virtual task update(output uvm_status_e      status,
                    input uvm_path_e         path      = UVM_DEFAULT_PATH,
                    input uvm_reg_map        map       = null,
                    input uvm_sequence_base  parent    = null,
                    input int                prior     = -1,
                    input uvm_object         extension = null,
                    input string             fname     = "",
                    input int                lineno    = 0                )
```

Updates the content of the register in the design to match the desired value

This method performs the reverse operation of uvm_reg::mirror(). Write this register if the DUT register is out-of-date with the desired/mirrored value in the abstraction class, as determined by the uvm_reg::needs_update() method.

The update can be performed using the using the physical interfaces (frontdoor) or uvm_reg::poke() (backdoor) access. If the register is mapped in multiple address maps and physical access is used (front-door), an address *map* must be specified.

mirror

```
virtual task mirror(output uvm_status_e      status,
                    input uvm_check_e        check     = UVM_NO_CHECK,
                    input uvm_path_e         path      = UVM_DEFAULT_PATH,
                    input uvm_reg_map        map       = null,
                    input uvm_sequence_base  parent    = null,
                    input int                prior     = -1,
                    input uvm_object         extension = null,
                    input string             fname     = "",
                    input int                lineno    = 0                )
```

Read the register and update/check its mirror value

Read the register and optionally compared the readback value with the current mirrored value if *check* is UVM_CHECK. The mirrored value will be updated using the uvm_reg::predict() method based on the readback value.

The mirroring can be performed using the physical interfaces (frontdoor) or uvm_reg::peek() (backdoor).

If *check* is specified as UVM_CHECK, an error message is issued if the current mirrored value does not match the readback value. Any field whose check has been disabled with uvm_reg_field::set_compare() will not be considered in the comparison.

If the register is mapped in multiple address maps and physical access is used (front-door access), an address *map* must be specified. If the register contains write-only fields, their content is mirrored and optionally checked only if a UVM_BACKDOOR access path is used to read the register.

predict

```
virtual function bit predict (uvm_reg_data_t    value,
                              uvm_reg_byte_en_t be     = -1,
                              uvm_predict_e     kind   = UVM_PREDICT_DIRECT,
                              uvm_path_e        path   = UVM_FRONTDOOR,
                              uvm_reg_map       map    = null,
                              string            fname  = "",
                              int               lineno = 0                 )
```

Update the mirrored value for this register.

Predict the mirror value of the fields in the register based on the specified observed *value* on a specified adress *map*, or based on a calculated value. See uvm_reg_field::predict() for more details.

Returns TRUE if the prediction was succesful for each field in the register.

is_busy

```
function bit is_busy()
```

Returns 1 if register is currently being read or written.

FRONTDOOR

set_frontdoor

```
function void set_frontdoor(uvm_reg_frontdoor ftdr,
                            uvm_reg_map       map    = null,
                            string            fname  = "",
                            int               lineno = 0    )
```

Set a user-defined frontdoor for this register

By default, registers are mapped linearly into the address space of the address maps that instantiate 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

If the register is mapped in multiple address maps, an address *map* must be specified.

get_frontdoor

```
function uvm_reg_frontdoor get_frontdoor(uvm_reg_map map = null)
```

Returns the user-defined frontdoor for this register

If null, no user-defined frontdoor has been defined. A user-defined frontdoor is defined by using the uvm_reg::set_frontdoor() method.

If the register is mapped in multiple address maps, an address *map* must be specified.

Backdoor

set_backdoor

```
function void set_backdoor(uvm_reg_backdoor bkdr,
                           string           fname  = "",
                           int              lineno = 0  )
```

Set a user-defined backdoor for this register

By default, registers are accessed via the built-in string-based DPI routines if an HDL path has been specified using the uvm_reg::configure() or uvm_reg::add_hdl_path() method.

If this default mechanism is not suitable (e.g. because the register is not implemented in pure SystemVerilog) a user-defined access mechanism must be defined and associated with the corresponding register abstraction class

A user-defined backdoor is required if active update of the mirror of this register abstraction class, based on observed changes of the corresponding DUT register, is used.

get_backdoor

```
function uvm_reg_backdoor get_backdoor(bit inherited = 1)
```

Returns the user-defined backdoor for this register

If null, no user-defined backdoor has been defined. A user-defined backdoor is defined by using the uvm_reg::set_backdoor() method.

If *inherited* is TRUE, returns the backdoor of the parent block if none have been specified for this register.

clear_hdl_path

```
function void clear_hdl_path (string kind = "RTL")
```

Delete HDL paths

Remove any previously specified HDL path to the register instance for the specified design abstraction.

add_hdl_path

```
function void add_hdl_path (uvm_hdl_path_slice slices[],
                            string             kind      = "RTL")
```

Add an HDL path

Add the specified HDL path to the register instance for the specified design abstraction. This method may be called more than once for the same design abstraction if the register is physically duplicated in the design abstraction

For example, the following register

```
       1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
Bits:  5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
      +-+---+-------------+---+-------+
      |A|xxx|      B      |xxx|   C   |
      +-+---+-------------+---+-------+
```

would be specified using the following literal value

```
add_hdl_path('{ '{"A_reg", 15, 1},
                '{"B_reg",  6, 7},
                '{'C_reg",  0, 4} } );
```

If the register is implementd using a single HDL variable, The array should specify a single slice with its *offset* and *size* specified as -1. For example:

```
r1.add_hdl_path('{ '{"r1", -1, -1} });
```

add_hdl_path_slice

```
function void add_hdl_path_slice(string name,
                                 int    offset,
                                 int    size,
                                 bit    first   = 0,
                                 string kind    = "RTL")
```

Append the specified HDL slice to the HDL path of the register instance for the specified design abstraction. If *first* is TRUE, starts the specification of a duplicate HDL implementation of the register.

has_hdl_path

```
function bit has_hdl_path (string kind = "")
```

Check if a HDL path is specified

Returns TRUE if the register instance has a HDL path defined for the specified design abstraction. If no design abstraction is specified, uses the default design abstraction specified for the parent block.

get_hdl_path

```
function void get_hdl_path (  ref uvm_hdl_path_concat paths[$],
                            input string              kind      = "")
```

Get the incremental HDL path(s)

Returns the HDL path(s) defined for the specified design abstraction in the register instance. Returns only the component of the HDL paths that corresponds to the register, not a full hierarchical path

If no design asbtraction is specified, the default design abstraction for the parent block is used.

get_hdl_path_kinds

```
function void get_hdl_path_kinds (ref string kinds[$])
```

Get design abstractions for which HDL paths have been defined

get_full_hdl_path

```
function void get_full_hdl_path (  ref uvm_hdl_path_concat paths[$],
                                 input string              kind      = "",
                                 input string              separator = ".")
```

Get the full hierarchical HDL path(s)

Returns the full hierarchical HDL path(s) defined for the specified design abstraction in the register instance. There may be more than one path returned even if only one path was defined for the register instance, if any of the parent components have more than one path defined for the same design abstraction

If no design asbtraction is specified, the default design abstraction for each ancestor block is used to get each incremental path.

backdoor_read

```
virtual task backdoor_read(uvm_reg_item rw)
```

User-define backdoor read access

Override the default string-based DPI backdoor access read for this register type. By default calls uvm_reg::backdoor_read_func().

backdoor_write

```
virtual task backdoor_write(uvm_reg_item rw)
```

User-defined backdoor read access

Override the default string-based DPI backdoor access write for this register type.

backdoor_read_func

```
virtual function uvm_status_e backdoor_read_func(uvm_reg_item rw)
```

User-defined backdoor read access

Override the default string-based DPI backdoor access read for this register type.

backdoor_watch

```
virtual task backdoor_watch()
```

User-defined DUT register change monitor

Watch the DUT register corresponding to this abstraction class instance for any change in value and return when a value-change occurs. This may be implemented a string-based DPI access if the simulation tool provide a value-change callback facility. Such a facility does not exists in the standard SystemVerilog DPI and thus no default implementation for this method can be provided.

COVERAGE

include_coverage

```
static function void include_coverage(string        scope,
                                      uvm_reg_cvr_t models,
                                      uvm_object    accessor = null)
```

Specify which coverage model that must be included in various block, register or memory abstraction class instances.

The coverage models are specified by or'ing or adding the uvm_coverage_model_e coverage model identifiers corresponding to the coverage model to be included.

The scope specifies a hierarchical name or pattern identifying a block, memory or register abstraction class instances. Any block, memory or register whose full hierarchical name matches the specified scope will have the specified functional coverage models included in them.

The scope can be specified as a POSIX regular expression or simple pattern. See uvm_resource_base::Scope Interface for more details.

```
uvm_reg::include_coverage("*", UVM_CVR_ALL);
```

The specification of which coverage model to include in which abstraction class is stored in a uvm_reg_cvr_t resource in the uvm_resource_db resource database, in the "uvm_reg::" scope namespace.

build_coverage

```
protected function uvm_reg_cvr_t build_coverage(uvm_reg_cvr_t models)
```

Check if all of the specified coverage models must be built.

Check which of the specified coverage model must be built in this instance of the register abstraction class, as specified by calls to uvm_reg::include_coverage().

Models are specified by adding the symbolic value of individual coverage model as defined in uvm_coverage_model_e. Returns the sum of all coverage models to be built in the register model.

add_coverage

```
virtual protected function void add_coverage(uvm_reg_cvr_t models)
```

Specify that additional coverage models are available.

Add the specified coverage model to the coverage models available in this class. Models are specified by adding the symbolic value of individual coverage model as defined in uvm_coverage_model_e.

This method shall be called only in the constructor of subsequently derived classes.

has_coverage

```
virtual function bit has_coverage(uvm_reg_cvr_t models)
```

Check if register has coverage model(s)

Returns TRUE if the register abstraction class contains a coverage model for all of the models specified. Models are specified by adding the symbolic value of individual coverage model as defined in uvm_coverage_model_e.

set_coverage

```
virtual function uvm_reg_cvr_t set_coverage(uvm_reg_cvr_t is_on)
```

Turns on coverage measurement.

Turns the collection of functional coverage measurements on or off for this register. The functional coverage measurement is turned on for every coverage model specified using uvm_coverage_model_e symbolic identifers. 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 are present in the register abstraction classes, then enabled during construction. See the uvm_reg::has_coverage() method to identify the available functional coverage models.

get_coverage

```
virtual function bit get_coverage(uvm_reg_cvr_t is_on)
```

Check if coverage measurement is on.

Returns TRUE if measurement for all of the specified functional coverage models are currently on. Multiple functional coverage models can be specified by adding the functional coverage model identifiers.

See uvm_reg::set_coverage() for more details.

sample

```
protected virtual function void sample(uvm_reg_data_t data,
                                       uvm_reg_data_t byte_en,
                                       bit            is_read,
                                       uvm_reg_map    map     )
```

Functional coverage measurement method

This method is invoked by the register abstraction class whenever it is read or written with the specified *data* via the specified address *map*. It is invoked after the read or write operation has completed but before the mirror has been updated.

Empty by default, this method may be extended by the abstraction class generator to perform the required sampling in any provided functional coverage model.

sample_values

```
virtual function void sample_values()
```

Functional coverage measurement method for field values

This method is invoked by the user or by the uvm_reg_block::sample_values() method of the parent block to trigger the sampling of the current field values in the register-level functional coverage model.

This method may be extended by the abstraction class generator to perform the required sampling in any provided field-value functional coverage model.

CALLBACKS

pre_write

```
virtual task pre_write(uvm_reg_item rw)
```

Called before register write.

If the specified data value, access *path* or address *map* are modified, the updated data value, access path or address map will be used to perform the register operation. If the *status* is modified to anything other than UVM_IS_OK, the operation is aborted.

The registered callback methods are invoked after the invocation of this method. All register callbacks are executed before the corresponding field callbacks

post_write

```
virtual task post_write(uvm_reg_item rw)
```

Called after register write.

If the specified *status* is modified, the updated status will be returned by the register operation.

The registered callback methods are invoked before the invocation of this method. All register callbacks are executed before the corresponding field callbacks

pre_read

```
virtual task pre_read(uvm_reg_item rw)
```

Called before register read.

If the specified access *path* or address *map* are modified, the updated access path or address map will be used to perform the register operation. If the *status* is modified to anything other than UVM_IS_OK, the operation is aborted.

The registered callback methods are invoked after the invocation of this method. All register callbacks are executed before the corresponding field callbacks

post_read

```
virtual task post_read(uvm_reg_item rw)
```

Called after register read.

If the specified readback data or *status* is modified, the updated readback data or status will be returned by the register operation.

The registered callback methods are invoked before the invocation of this method. All register callbacks are executed before the corresponding field callbacks

uvm_reg_field

Field abstraction class

A field represents a set of bits that behave consistently as a single entity.

A field is contained within a single register, but may have different access policies depending on the adddress map use the access the register (thus the field).

Summary

uvm_reg_field

Field abstraction class

Class Hierarchy

uvm_void

uvm_object

uvm_reg_field

Class Declaration

```
class uvm_reg_field extends uvm_object
```

value	Mirrored field value.

Initialization

new	Create a new field instance
configure	Instance-specific configuration

Introspection

get_name	Get the simple name
get_full_name	Get the hierarchical name
get_parent	Get the parent register
get_lsb_pos	Return the position of the field
get_n_bits	Returns the width, in number of bits, of the field.
get_max_size	Returns the width, in number of bits, of the largest field.
set_access	Modify the access policy of the field
define_access	Define a new access policy value
get_access	Get the access policy of the field
is_known_access	Check if access policy is a built-in one.
set_volatility	Modify the volatility of the field to the specified one.
is_volatile	Indicates if the field value is volatile

Access

set	Set the desired value for this field
get	Return the desired value of the field
reset	Reset the desired/mirrored value for this field.
get_reset	Get the specified reset value for this field
has_reset	Check if the field has a reset value specified
set_reset	Specify or modify the reset value for this field
needs_update	Check if the abstract model contains different desired and mirrored values.
write	Write the specified value in this field
read	Read the current value from this field
poke	Deposit the specified value in this field
peek	Read the current value from this field
mirror	Read the field and update/check its mirror value
set_compare	Sets the compare policy during a mirror update.
get_compare	Returns the compare policy for this field.
is_indv_accessible	Check if this field can be written individually, i.e.
predict	Update the mirrored value for this field.

CALLBACKS

pre_write	Called before field write.
post_write	Called after field write.
pre_read	Called before field read.
post_read	Called after field read.

value

```
rand uvm_reg_data_t value
```

Mirrored field value. This value can be sampled in a functional coverage model or constrained when randomized.

INITIALIZATION

new

```
function new(string name = "uvm_reg_field")
```

Create a new field instance

This method should not be used directly. The uvm_reg_field::type_id::create() factory method should be used instead.

configure

```
function void configure(    uvm_reg        parent,
                        int unsigned       size,
                        int unsigned       lsb_pos,
                            string         access,
                            bit            volatile,
                            uvm_reg_data_t reset,
                            bit            has_reset,
                            bit            is_rand,
                            bit            individually_accessible)
```

Instance-specific configuration

Specify the *parent* register of this field, its *size* in bits, the position of its least-significant bit within the register relative to the least-significant bit of the register, its *access* policy, volatility, "HARD" *reset* value, whether the field value is actually reset (the *reset* value is ignored if *FALSE*), whether the field value may be randomized and whether the field is the only one to occupy a byte lane in the register.

See set_access for a specification of the pre-defined field access policies.

If the field access policy is a pre-defined policy and NOT one of "RW", "WRC", "WRS", "WO", "W1", or "WO1", the value of *is_rand* is ignored and the rand_mode() for the field instance is turned off since it cannot be written.

INTROSPECTION

get_name

Get the simple name

Return the simple object name of this field

get_full_name

```
virtual function string get_full_name()
```

Get the hierarchical name

Return the hierarchal name of this field The base of the hierarchical name is the root block.

get_parent

```
virtual function uvm_reg get_parent()
```

Get the parent register

get_lsb_pos

```
virtual function int unsigned get_lsb_pos()
```

Return the position of the field

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.

get_n_bits

```
virtual function int unsigned get_n_bits()
```

Returns the width, in number of bits, of the field.

get_max_size

```
static function int unsigned get_max_size()
```

Returns the width, in number of bits, of the largest field.

set_access

```
virtual function string set_access(string mode)
```

Modify the access policy of the field

Modify the access policy of the field to the specified one and return the previous access policy.

The pre-defined access policies are as follows. The effect of a read operation are applied after the current value of the field is sampled. The read operation will return the current value, not the value affected by the read operation (if any).

"RO"	W: no effect, R: no effect
"RW"	W: as-is, R: no effect
"RC"	W: no effect, R: clears all bits
"RS"	W: no effect, R: sets all bits
"WRC"	W: as-is, R: clears all bits
"WRS"	W: as-is, R: sets all bits
"WC"	W: clears all bits, R: no effect
"WS"	W: sets all bits, R: no effect
"WSRC"	W: sets all bits, R: clears all bits
"WCRS"	W: clears all bits, R: sets all bits
"W1C"	W: 1/0 clears/no effect on matching bit, R: no effect
"W1S"	W: 1/0 sets/no effect on matching bit, R: no effect
"W1T"	W: 1/0 toggles/no effect on matching bit, R: no effect
"W0C"	W: 1/0 no effect on/clears matching bit, R: no effect
"W0S"	W: 1/0 no effect on/sets matching bit, R: no effect
"W0T"	W: 1/0 no effect on/toggles matching bit, R: no effect
"W1SRC"	W: 1/0 sets/no effect on matching bit, R: clears all bits
"W1CRS"	W: 1/0 clears/no effect on matching bit, R: sets all bits
"W0SRC"	W: 1/0 no effect on/sets matching bit, R: clears all bits
"W0CRS"	W: 1/0 no effect on/clears matching bit, R: sets all bits
"WO"	W: as-is, R: error
"WOC"	W: clears all bits, R: error
"WOS"	W: sets all bits, R: error
"W1"	W: first one after *HARD* reset is as-is, other W have no effects, R: no effect
"WO1"	W: first one after *HARD* reset is as-is, other W have no effects, R: error

It is important to remember that modifying the access of a field will make the register model diverge from the specification that was used to create it.

define_access

```
static function bit define_access(string name)
```

Define a new access policy value

Because field access policies are specified using string values, there is no way for SystemVerilog to verify if a spceific access value is valid or not. To help catch typing errors, user-defined access values must be defined using this method to avoid beign reported as an invalid access policy.

The name of field access policies are always converted to all uppercase.

Returns TRUE if the new access policy was not previously defined. Returns FALSE otherwise but does not issue an error message.

get_access

```
virtual function string get_access(uvm_reg_map map = null)
```

Get the access policy of the field

Returns the current access policy of the field when written and read through the specified address *map*. If the register containing the field is mapped in multiple address map, an address map must be specified. The access policy of a field from a specific address map may be restricted by the register's access policy in that address map. For example, a RW field may only be writable through one of the address maps and read-only through all of the other maps.

is_known_access

```
virtual function bit is_known_access(uvm_reg_map map = null)
```

Check if access policy is a built-in one.

Returns TRUE if the current access policy of the field, when written and read through the specified address *map*, is a built-in access policy.

set_volatility

```
virtual function void set_volatility(bit volatile)
```

Modify the volatility of the field to the specified one.

It is important to remember that modifying the volatility of a field will make the register model diverge from the specification that was used to create it.

is_volatile

```
virtual function bit is_volatile()
```

Indicates if the field value is volatile

UVM uses the IEEE 1685-2009 IP-XACT definition of "volatility". If TRUE, the value of the register is not predictable because it may change between consecutive accesses. This typically indicates a field whose value is updated by the DUT. The nature or cause of the change is not specified. If FALSE, the value of the register is not modified between consecutive accesses.

Access

set

```
virtual function void set(uvm_reg_data_t value,
                          string         fname  = "",
                          int            lineno = 0  )
```

Set the desired value for this field

Sets the desired value of the field to the specified value. Does not actually set the value of the field in the design, only the desired value in the abstrcation class. Use the uvm_reg::update() method to update the actual register with the desired value or the uvm_reg_field::write() method to actually write the field and update its mirrored value.

The final desired value in the mirror is a function of the field access policy 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 uvm_reg::update()) is a zero-time functional replacement for the uvm_reg_field::write() method. For example, the desired value of a read-only field is not modified by this method and the desired 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.

Use the uvm_reg_field::predict() to modify the mirrored value of the field.

get

```
virtual function uvm_reg_data_t get(string fname  = "",
                                    int    lineno = 0  )
```

Return the desired value of the field

Does not actually read the value of the field in the design, only the desired value in the abstraction class. Unless set to a different value using the uvm_reg_field::set(), the desired value and the mirrored value are identical.

Use the uvm_reg_field::read() or uvm_reg_field::peek() method to get the actual field value.

If the field is write-only, the desired/mirrored value is the value last written and assumed to reside in the bits implementing it. Although a physical read operation would something different, the returned value is the actual content.

reset

```
virtual function void reset(string kind = "HARD")
```

Reset the desired/mirrored value for this field.

Sets the desired and mirror value of the field to the reset event specified by *kind*. If the field does not have a reset value specified for the specified reset *kind* the field is unchanged.

Does not actually reset the value of the field in the design, only the value mirrored in the field abstraction class.

Write-once fields can be modified after a “HARD” reset operation.

get_reset

```
virtual function uvm_reg_data_t get_reset(string kind = "HARD")
```

Get the specified reset value for this field

Return the reset value for this field for the specified reset *kind*. Returns the current field value is no reset value has been specified for the specified reset event.

has_reset

```
virtual function bit has_reset(string kind   = "HARD",
                               bit    delete = 0      )
```

Check if the field has a reset value specified

Return TRUE if this field has a reset value specified for the specified reset *kind*. If *delete* is TRUE, removes the reset value, if any.

set_reset

```
virtual function void set_reset(uvm_reg_data_t value,
                                string         kind   = "HARD")
```

Specify or modify the reset value for this field

Specify or modify the reset value for this field corresponding to the cause specified by *kind*.

needs_update

```
virtual function bit needs_update()
```

Check if the abstract model contains different desired and mirrored values.

If a desired field value has been modified in the abstraction class without actually updating the field in the DUT, the state of the DUT (more specifically what the abstraction class *thinks* the state of the DUT is) is outdated. This method returns TRUE if the state of the field in the DUT needs to be updated to match the desired value. The mirror values or actual content of DUT field are not modified. Use the uvm_reg::update() to actually update the DUT field.

write

```
virtual task write (output uvm_status_e      status,
                    input uvm_reg_data_t     value,
                    input uvm_path_e         path      = UVM_DEFAULT_PATH,
                    input uvm_reg_map        map       = null,
                    input uvm_sequence_base  parent    = null,
                    input int                prior     = -1,
                    input uvm_object         extension = null,
                    input string             fname     = "",
                    input int                lineno    = 0               )
```

Write the specified value in this field

Write *value* in the DUT field that corresponds to this abstraction class instance using the specified access *path*. If the register containing this field is mapped in more than one address map, an address *map* 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 field through a physical access is mimicked. For example, read-only bits in the field will not be written.

The mirrored value will be updated using the uvm_reg_field::predict() method.

If a front-door access is used, and if the field is the only field in a byte lane and if the physical interface corresponding to the address map used to access the field support

byte-enabling, then only the field is written. Otherwise, the entire register containing the field is written, and the mirrored values of the other fields in the same register are used in a best-effort not to modify their value.

If a backdoor access is used, a peek-modify-poke process is used. in a best-effort not to modify the value of the other fields in the register.

read

```
virtual task read (output uvm_status_e      status,
                   output uvm_reg_data_t    value,
                    input uvm_path_e        path      = UVM_DEFAULT_PATH,
                    input uvm_reg_map       map       = null,
                    input uvm_sequence_base parent    = null,
                    input int               prior     = -1,
                    input uvm_object        extension = null,
                    input string            fname     = "",
                    input int               lineno    = 0               )
```

Read the current value from this field

Read and return *value* from the DUT field that corresponds to this abstraction class instance using the specified access *path*. If the register containing this field is mapped in more than one address map, an address *map* 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 field through a physical access is mimicked. For example, clear-on-read bits in the filed will be set to zero.

The mirrored value will be updated using the uvm_reg_field::predict() method.

If a front-door access is used, and if the field is the only field in a byte lane and if the physical interface corresponding to the address map used to access the field support byte-enabling, then only the field is read. Otherwise, the entire register containing the field is read, and the mirrored values of the other fields in the same register are updated.

If a backdoor access is used, the entire containing register is peeked and the mirrored value of the other fields in the register is updated.

poke

```
virtual task poke (output uvm_status_e      status,
                    input uvm_reg_data_t    value,
                    input string            kind      = "",
                    input uvm_sequence_base parent    = null,
                    input uvm_object        extension = null,
                    input string            fname     = "",
                    input int               lineno    = 0    )
```

Deposit the specified value in this field

Deposit the value in the DUT field corresponding to this abstraction class instance, as-is, using a back-door access. A peek-modify-poke process is used in a best-effort not to modify the value of the other fields in the register.

The mirrored value will be updated using the uvm_reg_field::predict() method.

peek

```
virtual task peek (output uvm_status_e      status,
                   output uvm_reg_data_t    value,
```

```
                    input string            kind      = "",
                    input uvm_sequence_base parent    = null,
                    input uvm_object        extension = null,
                    input string            fname     = "",
                    input int               lineno    = 0    )
```

Read the current value from this field

Sample the value in the DUT field corresponding to this absraction class instance using a back-door access. The field value is sampled, not modified.

Uses the HDL path for the design abstraction specified by *kind*.

The entire containing register is peeked and the mirrored value of the other fields in the register are updated using the uvm_reg_field::predict() method.

mirror

```
virtual task mirror(output uvm_status_e      status,
                    input uvm_check_e        check     = UVM_NO_CHECK,
                    input uvm_path_e         path      = UVM_DEFAULT_PATH,
                    input uvm_reg_map        map       = null,
                    input uvm_sequence_base  parent    = null,
                    input int                prior     = -1,
                    input uvm_object         extension = null,
                    input string             fname     = "",
                    input int                lineno    = 0                )
```

Read the field and update/check its mirror value

Read the field and optionally compared the readback value with the current mirrored value if *check* is UVM_CHECK. The mirrored value will be updated using the predict() method based on the readback value.

The *path* argument specifies whether to mirror using the UVM_FRONTDOOR (read) or or UVM_BACKDOOR (peek()).

If *check* is specified as UVM_CHECK, an error message is issued if the current mirrored value does not match the readback value, unless set_compare was used disable the check.

If the containing register is mapped in multiple address maps and physical access is used (front-door access), an address *map* must be specified. For write-only fields, their content is mirrored and optionally checked only if a UVM_BACKDOOR access path is used to read the field.

set_compare

```
function void set_compare(uvm_check_e check = UVM_CHECK)
```

Sets the compare policy during a mirror update. The field value is checked against its mirror only when both the *check* argument in uvm_reg_block::mirror, uvm_reg::mirror, or uvm_reg_field::mirror and the compare policy for the field is UVM_CHECK.

get_compare

```
function uvm_check_e get_compare()
```

Returns the compare policy for this field.

is_indv_accessible

```
function bit is_indv_accessible (uvm_path_e  path,
                                 uvm_reg_map local_map)
```

Check if this field can be written individually, i.e. without affecting other fields in the containing register.

predict

```
function bit predict (uvm_reg_data_t    value,
                      uvm_reg_byte_en_t be     = -1,
                      uvm_predict_e     kind   = UVM_PREDICT_DIRECT,
                      uvm_path_e        path   = UVM_FRONTDOOR,
                      uvm_reg_map       map    = null,
                      string            fname  = "",
                      int               lineno = 0                 )
```

Update the mirrored value for this field.

Predict the mirror value of the field based on the specified observed *value* on a bus using the specified address *map*.

If *kind* is specified as UVM_PREDICT_READ, the value was observed in a read transaction on the specified address *map* or backdoor (if *path* is UVM_BACKDOOR). If *kind* is specified as UVM_PREDICT_WRITE, the value was observed in a write transaction on the specified address *map* or backdoor (if *path* is UVM_BACKDOOR). If *kind* is specified as UVM_PREDICT_DIRECT, the value was computed and is updated as-is, without regard to any access policy. For example, the mirrored value of a read-only field is modified by this method if *kind* is specified as UVM_PREDICT_DIRECT.

This method does not allow an update of the mirror when the register containing this field is busy executing a transaction because the results are unpredictable and indicative of a race condition in the testbench.

Returns TRUE if the prediction was succesful.

CALLBACKS

pre_write

```
virtual task pre_write (uvm_reg_item rw)
```

Called before field write.

If the specified data value, access *path* or address *map* are modified, the updated data value, access path or address map will be used to perform the register operation. If the *status* is modified to anything other than UVM_IS_OK, the operation is aborted.

The field callback methods are invoked after the callback methods on the containing register. The registered callback methods are invoked after the invocation of this method.

post_write

```
virtual task post_write (uvm_reg_item rw)
```

Called after field write.

If the specified *status* is modified, the updated status will be returned by the register operation.

The field callback methods are invoked after the callback methods on the containing register. The registered callback methods are invoked before the invocation of this method.

pre_read

```
virtual task pre_read (uvm_reg_item rw)
```

Called before field read.

If the access *path* or address *map* in the *rw* argument are modified, the updated access path or address map will be used to perform the register operation. If the *status* is modified to anything other than UVM_IS_OK, the operation is aborted.

The field callback methods are invoked after the callback methods on the containing register. The registered callback methods are invoked after the invocation of this method.

post_read

```
virtual task post_read (uvm_reg_item rw)
```

Called after field read.

If the specified readback data or~status~ in the *rw* argument is modified, the updated readback data or status will be returned by the register operation.

The field callback methods are invoked after the callback methods on the containing register. The registered callback methods are invoked before the invocation of this method.

uvm_mem

Memory abstraction base class

A memory is a collection of contiguous locations. A memory may be accessible via more than one address map.

Unlike registers, memories are not mirrored because of the potentially large data space: tests that walk the entire memory space would negate any benefit from sparse memory modelling techniques. Rather than relying on a mirror, it is recommended that backdoor access be used instead.

Summary

uvm_mem

Memory abstraction base class

Class Hierarchy

uvm_void

uvm_object

uvm_mem

Class Declaration

```
class uvm_mem extends uvm_object
```

Initialization

new	Create a new instance and type-specific configuration
configure	Instance-specific configuration
set_offset	Modify the offset of the memory
Modifying the offset of a memory will make the abstract model	diverge from the specification that was used to create it.
mam	Memory allocation manager

Introspection

get_name	Get the simple name
get_full_name	Get the hierarchical name
get_parent	Get the parent block
get_n_maps	Returns the number of address maps this memory is mapped in
is_in_map	Return TRUE if this memory is in the specified address *map*
get_maps	Returns all of the address *maps* where this memory is mapped
get_rights	Returns the access rights of this memory.
get_access	Returns the access policy of the memory when written and read via an address map.
get_size	Returns the number of unique memory locations in this memory.
get_n_bytes	Return the width, in number of bytes, of each memory location
get_n_bits	Returns the width, in number of bits, of each memory location
get_max_size	Returns the maximum width, in number of bits, of all memories
get_virtual_registers	Return the virtual registers in this memory
get_virtual_fields	Return the virtual fields in the memory
get_vreg_by_name	Find the named virtual register

get_vfield_by_name	Find the named virtual field
get_vreg_by_offset	Find the virtual register implemented at the specified offset
get_offset	Returns the base offset of a memory location
get_address	Returns the base external physical address of a memory location
get_addresses	Identifies the external physical address(es) of a memory location
HDL Access	
write	Write the specified value in a memory location
read	Read the current value from a memory location
burst_write	Write the specified values in memory locations
burst_read	Read values from memory locations
poke	Deposit the specified value in a memory location
peek	Read the current value from a memory location
Frontdoor	
set_frontdoor	Set a user-defined frontdoor for this memory
get_frontdoor	Returns the user-defined frontdoor for this memory
Backdoor	
set_backdoor	Set a user-defined backdoor for this memory
get_backdoor	Returns the user-defined backdoor for this memory
clear_hdl_path	Delete HDL paths
add_hdl_path	Add an HDL path
add_hdl_path_slice	Add the specified HDL slice to the HDL path for the specified design abstraction.
has_hdl_path	Check if a HDL path is specified
get_hdl_path	Get the incremental HDL path(s)
get_full_hdl_path	Get the full hierarchical HDL path(s)
get_hdl_path_kinds	Get design abstractions for which HDL paths have been defined
backdoor_read	User-define backdoor read access
backdoor_write	User-defined backdoor read access
backdoor_read_func	User-defined backdoor read access
Callbacks	
pre_write	Called before memory write.
post_write	Called after memory write.
pre_read	Called before memory read.
post_read	Called after memory read.
Coverage	
build_coverage	Check if all of the specified coverage model must be built.
add_coverage	Specify that additional coverage models are available.
has_coverage	Check if memory has coverage model(s)
set_coverage	Turns on coverage measurement.
get_coverage	Check if coverage measurement is on.
sample	Functional coverage measurement method

Initialization

new

```
function new (        string   name,
              longint unsigned size,
                  int unsigned n_bits,
                      string   access       = "RW",
                      int      has_coverage = UVM_NO_COVERAGE)
```

Create a new instance and type-specific configuration

Creates an instance of a memory abstraction class with the specified name.

size specifies the total number of memory locations. *n_bits* specifies the total number of bits in each memory location. *access* specifies the access policy of this memory and may be one of "RW for RAMs and "RO" for ROMs.

has_coverage specifies which functional coverage models are present in the extension of the register abstraction class. Multiple functional coverage models may be specified by adding their symbolic names, as defined by the uvm_coverage_model_e type.

configure

```
function void configure (uvm_reg_block parent,
                         string        hdl_path = "")
```

Instance-specific configuration

Specify the parent block of this memory.

If this memory is implemented in a single HDL variable, it's name is specified as the *hdl_path*. Otherwise, if the memory is implemented as a concatenation of variables (usually one per bank), then the HDL path must be specified using the add_hdl_path() or add_hdl_path_slice() method.

set_offset

Modify the offset of the memory

The offset of a memory within an address map is set using the uvm_reg_map::add_mem() method. This method is used to modify that offset dynamically.

Modifying the offset of a memory will make the abstract model

diverge from the specification that was used to create it.

mam

```
uvm_mem_mam mam
```

Memory allocation manager

Memory allocation manager for the memory corresponding to this abstraction class instance. Can be used to allocate regions of consecutive addresses of specific sizes, such as DMA buffers, or to locate virtual register array.

INTROSPECTION

get_name

Get the simple name

Return the simple object name of this memory.

get_full_name

```
virtual function string get_full_name()
```

Get the hierarchical name

Return the hierarchal name of this memory. The base of the hierarchical name is the root block.

get_parent

```
virtual function uvm_reg_block get_parent ()
```

Get the parent block

get_n_maps

```
virtual function int get_n_maps ()
```

Returns the number of address maps this memory is mapped in

is_in_map

```
function bit is_in_map (uvm_reg_map map)
```

Return TRUE if this memory is in the specified address *map*

get_maps

```
virtual function void get_maps (ref uvm_reg_map maps[$])
```

Returns all of the address *maps* where this memory is mapped

get_rights

```
virtual function string get_rights (uvm_reg_map map = null)
```

Returns the access rights of this memory.

Returns "RW", "RO" or "WO". The access rights of a memory is always "RW", unless it is a shared memory with access restriction in a particular address map.

If no address map is specified and the memory is mapped in only one address map, that address map is used. If the memory is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the memory is not mapped in the specified address map, an error message is issued and "RW" is returned.

get_access

```
virtual function string get_access(uvm_reg_map map = null)
```

Returns the access policy of the memory when written and read via an address map.

If the memory is mapped in more than one address map, an address *map* must be specified. If access restrictions are present when accessing a memory through the specified address map, 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 "RO".

get_size

```
function longint unsigned get_size()
```

Returns the number of unique memory locations in this memory.

get_n_bytes

```
function int unsigned get_n_bytes()
```

Return the width, in number of bytes, of each memory location

get_n_bits

```
function int unsigned get_n_bits()
```

Returns the width, in number of bits, of each memory location

get_max_size

```
static function int unsigned get_max_size()
```

Returns the maximum width, in number of bits, of all memories

get_virtual_registers

```
virtual function void get_virtual_registers(ref uvm_vreg regs[$])
```

Return the virtual registers in this memory

Fills the specified array with the abstraction class for all of the virtual registers implemented in this memory. The order in which the virtual registers are located in the

array is not specified.

get_virtual_fields

```
virtual function void get_virtual_fields(ref uvm_vreg_field fields[$])
```

Return the virtual fields in the memory

Fills the specified dynamic array with the abstraction class for all of the virtual fields implemented in this memory. The order in which the virtual fields are located in the array is not specified.

get_vreg_by_name

```
virtual function uvm_vreg get_vreg_by_name(string name)
```

Find the named virtual register

Finds a virtual register with the specified name implemented in this memory and returns its abstraction class instance. If no virtual register with the specified name is found, returns *null*.

get_vfield_by_name

```
virtual function uvm_vreg_field get_vfield_by_name(string name)
```

Find the named virtual field

Finds a virtual field with the specified name implemented in this memory and returns its abstraction class instance. If no virtual field with the specified name is found, returns *null*.

get_vreg_by_offset

```
virtual function uvm_vreg get_vreg_by_offset(uvm_reg_addr_t offset,
                                             uvm_reg_map    map     = null)
```

Find the virtual register implemented at the specified offset

Finds the virtual register implemented in this memory at the specified *offset* in the specified address *map* and returns its abstraction class instance. If no virtual register at the offset is found, returns *null*.

get_offset

```
virtual function uvm_reg_addr_t get_offset (uvm_reg_addr_t offset = 0,
                                            uvm_reg_map    map    = null)
```

Returns the base offset of a memory location

Returns the base offset of the specified location in this memory in an address *map*.

If no address map is specified and the memory is mapped in only one address map, that

address map is used. If the memory is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the memory is not mapped in the specified address map, an error message is issued.

get_address

```
virtual function uvm_reg_addr_t get_address(uvm_reg_addr_t offset = 0,
                                            uvm_reg_map    map    = null)
```

Returns the base external physical address of a memory location

Returns the base external physical address of the specified location in this memory if accessed through the specified address *map*.

If no address map is specified and the memory is mapped in only one address map, that address map is used. If the memory is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the memory is not mapped in the specified address map, an error message is issued.

get_addresses

```
virtual function int get_addresses(    uvm_reg_addr_t offset = 0,
                                       uvm_reg_map    map    = null,
                                   ref uvm_reg_addr_t addr[]        )
```

Identifies the external physical address(es) of a memory location

Computes all of the external physical addresses that must be accessed to completely read or write the specified location in this memory. The addressed are specified in little endian order. Returns the number of bytes transfered on each access.

If no address map is specified and the memory is mapped in only one address map, that address map is used. If the memory is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the memory is not mapped in the specified address map, an error message is issued.

HDL Access

write

```
virtual task write(output uvm_status_e      status,
                   input  uvm_reg_addr_t    offset,
                   input  uvm_reg_data_t    value,
                   input  uvm_path_e        path      = UVM_DEFAULT_PATH,
                   input  uvm_reg_map       map       = null,
                   input  uvm_sequence_base parent    = null,
                   input  int               prior     = -1,
                   input  uvm_object        extension = null,
                   input  string            fname     = "",
                   input  int               lineno    = 0               )
```

Write the specified value in a memory location

Write *value* in the memory location that corresponds to this abstraction class instance at the specified *offset* using the specified access *path*. If the memory is mapped in more than one address map, an address *map* 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, a read-only memory will not be written.

read

```
virtual task read(output uvm_status_e      status,
                   input uvm_reg_addr_t    offset,
                  output uvm_reg_data_t    value,
                   input uvm_path_e        path      = UVM_DEFAULT_PATH,
                   input uvm_reg_map       map       = null,
                   input uvm_sequence_base parent    = null,
                   input int               prior     = -1,
                   input uvm_object        extension = null,
                   input string            fname     = "",
                   input int               lineno    = 0               )
```

Read the current value from a memory location

Read and return *value* from the memory location that corresponds to this abstraction class instance at the specified *offset* using the specified access *path*. If the register is mapped in more than one address map, an address *map* must be specified if a physical access is used (front-door access).

burst_write

```
virtual task burst_write(output uvm_status_e      status,
                         input uvm_reg_addr_t    offset,
                         input uvm_reg_data_t    value[],
                         input uvm_path_e        path      = UVM_DEFAULT_PAT
                         input uvm_reg_map       map       = null,
                         input uvm_sequence_base parent    = null,
                         input int               prior     = -1,
                         input uvm_object        extension = null,
                         input string            fname     = "",
                         input int               lineno    = 0
```

Write the specified values in memory locations

Burst-write the specified *values* in the memory locations beginning at the specified *offset*. If the memory is mapped in more than one address map, an address *map* must be specified if not using the backdoor. If a back-door access path is used, the effect of writing the register through a physical access is mimicked. For example, a read-only memory will not be written.

burst_read

```
virtual task burst_read(output uvm_status_e      status,
                         input uvm_reg_addr_t    offset,
                        output uvm_reg_data_t    value[],
                         input uvm_path_e        path      = UVM_DEFAULT_PATH
                         input uvm_reg_map       map       = null,
                         input uvm_sequence_base parent    = null,
                         input int               prior     = -1,
                         input uvm_object        extension = null,
                         input string            fname     = "",
                         input int               lineno    = 0
```

Read values from memory locations

Burst-read into *values* the data the memory locations beginning at the specified *offset*. If the memory is mapped in more than one address map, an address *map* must be specified if not using the backdoor. If a back-door access path is used, the effect of writing the register through a physical access is mimicked. For example, a read-only memory will not be written.

poke

```
virtual task poke(output uvm_status_e      status,
                  input uvm_reg_addr_t     offset,
                  input uvm_reg_data_t     value,
                  input string             kind      = "",
                  input uvm_sequence_base  parent    = null,
                  input uvm_object         extension = null,
                  input string             fname     = "",
                  input int                lineno    = 0    )
```

Deposit the specified value in a memory location

Deposit the value in the DUT memory location corresponding to this abstraction class instance at the secified *offset*, as-is, using a back-door access.

Uses the HDL path for the design abstraction specified by *kind*.

peek

```
virtual task peek(output uvm_status_e      status,
                   input uvm_reg_addr_t    offset,
                  output uvm_reg_data_t    value,
                   input string            kind      = "",
                   input uvm_sequence_base parent    = null,
                   input uvm_object        extension = null,
                   input string            fname     = "",
                   input int               lineno    = 0    )
```

Read the current value from a memory location

Sample the value in the DUT memory location corresponding to this absraction class instance at the specified *offset* using a back-door access. The memory location value is sampled, not modified.

Uses the HDL path for the design abstraction specified by *kind*.

FRONTDOOR

set_frontdoor

```
function void set_frontdoor(uvm_reg_frontdoor ftdr,
                            uvm_reg_map       map    = null,
                            string            fname  = "",
                            int               lineno = 0    )
```

Set a user-defined frontdoor for this memory

By default, memorys are mapped linearly into the address space of the address maps that instantiate them. If memorys are accessed using a different mechanism, a user-defined access mechanism must be defined and associated with the corresponding

memory abstraction class

If the memory is mapped in multiple address maps, an address *map* must be specified.

get_frontdoor

```
function uvm_reg_frontdoor get_frontdoor(uvm_reg_map map = null)
```

Returns the user-defined frontdoor for this memory

If null, no user-defined frontdoor has been defined. A user-defined frontdoor is defined by using the uvm_mem::set_frontdoor() method.

If the memory is mapped in multiple address maps, an address *map* must be specified.

Backdoor

set_backdoor

```
function void set_backdoor (uvm_reg_backdoor bkdr,
                            string           fname  = "",
                            int              lineno = 0  )
```

Set a user-defined backdoor for this memory

By default, memories are accessed via the built-in string-based DPI routines if an HDL path has been specified using the uvm_mem::configure() or uvm_mem::add_hdl_path() method. If this default mechanism is not suitable (e.g. because the memory is not implemented in pure SystemVerilog) a user-defined access mechanism must be defined and associated with the corresponding memory abstraction class

get_backdoor

```
function uvm_reg_backdoor get_backdoor(bit inherited = 1)
```

Returns the user-defined backdoor for this memory

If null, no user-defined backdoor has been defined. A user-defined backdoor is defined by using the uvm_reg::set_backdoor() method.

If *inherit* is TRUE, returns the backdoor of the parent block if none have been specified for this memory.

clear_hdl_path

```
function void clear_hdl_path (string kind = "RTL")
```

Delete HDL paths

Remove any previously specified HDL path to the memory instance for the specified design abstraction.

add_hdl_path

```
function void add_hdl_path (uvm_hdl_path_slice slices[],
                            string             kind      = "RTL")
```

Add an HDL path

Add the specified HDL path to the memory instance for the specified design abstraction. This method may be called more than once for the same design abstraction if the memory is physically duplicated in the design abstraction

add_hdl_path_slice

```
function void add_hdl_path_slice(string name,
                                 int    offset,
                                 int    size,
                                 bit    first   = 0,
                                 string kind    = "RTL")
```

Add the specified HDL slice to the HDL path for the specified design abstraction. If *first* is TRUE, starts the specification of a duplicate HDL implementation of the memory.

has_hdl_path

```
function bit has_hdl_path (string kind = "")
```

Check if a HDL path is specified

Returns TRUE if the memory instance has a HDL path defined for the specified design abstraction. If no design abstraction is specified, uses the default design abstraction specified for the parent block.

get_hdl_path

```
function void get_hdl_path (  ref uvm_hdl_path_concat paths[$],
                            input string              kind      = "")
```

Get the incremental HDL path(s)

Returns the HDL path(s) defined for the specified design abstraction in the memory instance. Returns only the component of the HDL paths that corresponds to the memory, not a full hierarchical path

If no design asbtraction is specified, the default design abstraction for the parent block is used.

get_full_hdl_path

```
function void get_full_hdl_path (  ref uvm_hdl_path_concat paths[$],
                                 input string              kind      = "",
                                 input string              separator = ".")
```

Get the full hierarchical HDL path(s)

Returns the full hierarchical HDL path(s) defined for the specified design abstraction in the memory instance. There may be more than one path returned even if only one path

was defined for the memory instance, if any of the parent components have more than one path defined for the same design abstraction

If no design asbtraction is specified, the default design abstraction for each ancestor block is used to get each incremental path.

get_hdl_path_kinds

```
function void get_hdl_path_kinds (ref string kinds[$])
```

Get design abstractions for which HDL paths have been defined

backdoor_read

```
virtual protected task backdoor_read(uvm_reg_item rw)
```

User-define backdoor read access

Override the default string-based DPI backdoor access read for this memory type. By default calls uvm_mem::backdoor_read_func().

backdoor_write

```
virtual task backdoor_write(uvm_reg_item rw)
```

User-defined backdoor read access

Override the default string-based DPI backdoor access write for this memory type.

backdoor_read_func

```
virtual function uvm_status_e backdoor_read_func(uvm_reg_item rw)
```

User-defined backdoor read access

Override the default string-based DPI backdoor access read for this memory type.

Callbacks

pre_write

```
virtual task pre_write(uvm_reg_item rw)
```

Called before memory write.

If the *offset*, *value*, access *path*, or address *map* are modified, the updated offset, data value, access path or address map will be used to perform the memory operation. If the *status* is modified to anything other than UVM_IS_OK, the operation is aborted.

The registered callback methods are invoked after the invocation of this method.

post_write

```
virtual task post_write(uvm_reg_item rw)
```

Called after memory write.

If the *status* is modified, the updated status will be returned by the memory operation.

The registered callback methods are invoked before the invocation of this method.

pre_read

```
virtual task pre_read(uvm_reg_item rw)
```

Called before memory read.

If the *offset*, access *path* or address *map* are modified, the updated offset, access path or address map will be used to perform the memory operation. If the *status* is modified to anything other than UVM_IS_OK, the operation is aborted.

The registered callback methods are invoked after the invocation of this method.

post_read

```
virtual task post_read(uvm_reg_item rw)
```

Called after memory read.

If the readback data or *status* is modified, the updated readback //data or status will be returned by the memory operation.

The registered callback methods are invoked before the invocation of this method.

COVERAGE

build_coverage

```
protected function uvm_reg_cvr_t build_coverage(uvm_reg_cvr_t models)
```

Check if all of the specified coverage model must be built.

Check which of the specified coverage model must be built in this instance of the memory abstraction class, as specified by calls to uvm_reg::include_coverage().

Models are specified by adding the symbolic value of individual coverage model as defined in uvm_coverage_model_e. Returns the sum of all coverage models to be built in the memory model.

add_coverage

```
virtual protected function void add_coverage(uvm_reg_cvr_t models)
```

Specify that additional coverage models are available.

Add the specified coverage model to the coverage models available in this class. Models are specified by adding the symbolic value of individual coverage model as defined in uvm_coverage_model_e.

This method shall be called only in the constructor of subsequently derived classes.

has_coverage

```
virtual function bit has_coverage(uvm_reg_cvr_t models)
```

Check if memory has coverage model(s)

Returns TRUE if the memory abstraction class contains a coverage model for all of the models specified. Models are specified by adding the symbolic value of individual coverage model as defined in uvm_coverage_model_e.

set_coverage

```
virtual function uvm_reg_cvr_t set_coverage(uvm_reg_cvr_t is_on)
```

Turns on coverage measurement.

Turns the collection of functional coverage measurements on or off for this memory. The functional coverage measurement is turned on for every coverage model specified using uvm_coverage_model_e symbolic identifers. 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 are present in the memory abstraction classes, then enabled during construction. See the uvm_mem::has_coverage() method to identify the available functional coverage models.

get_coverage

```
virtual function bit get_coverage(uvm_reg_cvr_t is_on)
```

Check if coverage measurement is on.

Returns TRUE if measurement for all of the specified functional coverage models are currently on. Multiple functional coverage models can be specified by adding the functional coverage model identifiers.

See uvm_mem::set_coverage() for more details.

sample

```
protected virtual function void sample(uvm_reg_addr_t offset,
                                       bit            is_read,
                                       uvm_reg_map    map     )
```

Functional coverage measurement method

This method is invoked by the memory abstraction class whenever an address within one of its address map is succesfully read or written. The specified offset is the offset within

the memory, not an absolute address.

Empty by default, this method may be extended by the abstraction class generator to perform the required sampling in any provided functional coverage model.

uvm_reg_indirect_data

Indirect data access abstraction class

Models the behavior of a register used to indirectly access a register array, indexed by a second *address* register.

This class should not be instantiated directly. A type-specific class extension should be used to provide a factory-enabled constructor and specify the *n_bits* and coverage models.

Summary

uvm_reg_indirect_data

Indirect data access abstraction class

Class Hierarchy

- uvm_void
- uvm_object
- uvm_reg
- **uvm_reg_indirect_data**

Class Declaration

```
class uvm_reg_indirect_data extends uvm_reg
```

Methods

new	Create an instance of this class
configure	Configure the indirect data register.

Methods

new

```
function new(     string   name      = "uvm_reg_indirect",
             int unsigned n_bits,
                  int      has_cover                      )
```

Create an instance of this class

Should not be called directly, other than via super.new(). The value of *n_bits* must match the number of bits in the indirect register array.

configure

```
function void configure (uvm_reg       idx,
                         uvm_reg       reg_a[],
                         uvm_reg_block blk_parent,
                         uvm_reg_file  regfile_parent = null)
```

Configure the indirect data register.

The *idx* register specifies the index, in the *reg_a* register array, of the register to access. The *idx* must be written to first. A read or write operation to this register will subsequently read or write the indexed register in the register array.

The number of bits in each register in the register array must be equal to *n_bits* of this register.

See uvm_reg::configure() for the remaining arguments.

uvm_reg_fifo

This special register models a DUT FIFO accessed via write/read, where writes push to the FIFO and reads pop from it.

Backdoor access is not enabled, as it is not yet possible to force complete FIFO state, i.e. the write and read indexes used to access the FIFO data.

Summary

uvm_reg_fifo

This special register models a DUT FIFO accessed via write/read, where writes push to the FIFO and reads pop from it.

Class Hierarchy

uvm_void
uvm_object
uvm_reg
uvm_reg_fifo

Class Declaration

```
class uvm_reg_fifo extends uvm_reg
```

fifo	The abstract representation of the FIFO.
Initialization	
new	Creates an instance of a FIFO register having *size* elements of *n_bits* each.
set_compare	Sets the compare policy during a mirror (read) of the DUT FIFO.
Introspection	
size	The number of entries currently in the FIFO.
capacity	The maximum number of entries, or depth, of the FIFO.
Access	
write	Pushes the given value to the DUT FIFO.
read	Reads the next value out of the DUT FIFO.
set	Pushes the given value to the abstract FIFO.
update	Pushes (writes) all values preloaded using <set(()> to the DUT>.
mirror	Reads the next value out of the DUT FIFO.
get	Returns the next value from the abstract FIFO, but does not pop it.
do_predict	Updates the abstract (mirror) FIFO based on write() and read() operations.
Special Overrides	
pre_write	Special pre-processing for a write() or update().
pre_read	Special post-processing for a write() or update().

fifo

```
rand uvm_reg_data_t fifo[$]
```

The abstract representation of the FIFO. Constrained to be no larger than the size

parameter. It is public to enable subtypes to add constraints on it and randomize.

Initialization

new

```
function new(     string   name      = "reg_fifo",
             int unsigned size,
             int unsigned n_bits,
                 int      has_cover              )
```

Creates an instance of a FIFO register having *size* elements of *n_bits* each.

set_compare

```
function void set_compare(uvm_check_e check = UVM_CHECK)
```

Sets the compare policy during a mirror (read) of the DUT FIFO. The DUT read value is checked against its mirror only when both the *check* argument in the mirror() call and the compare policy for the field is UVM_CHECK.

Introspection

size

```
function int unsigned size()
```

The number of entries currently in the FIFO.

capacity

```
function int unsigned capacity()
```

The maximum number of entries, or depth, of the FIFO.

Access

write

Pushes the given value to the DUT FIFO. If auto-predition is enabled, the written value is also pushed to the abstract FIFO before the call returns. If auto-prediction is not enabled (see <uvm_map::set_auto_predict>), the value is pushed to abstract FIFO only when the write operation is observed on the target bus. This mode requires using the <uvm_reg_predictor #(BUSTYPE)> class. If the write is via an update() operation, the abstract FIFO already contains the written value and is thus not affected by either prediction mode.

read

Reads the next value out of the DUT FIFO. If auto-prediction is enabled, the frontmost value in abstract FIFO is popped.

set

```
virtual function void set(uvm_reg_data_t value,
                          string         fname  = "",
                          int            lineno = 0  )
```

Pushes the given value to the abstract FIFO. You may call this method several times before an update() as a means of preloading the DUT FIFO. Calls to *set()* to a full FIFO are ignored. You must call update() to update the DUT FIFO with your set values.

update

```
virtual task update(output uvm_status_e      status,
                    input  uvm_path_e        path      = UVM_DEFAULT_PATH,
                    input  uvm_reg_map       map       = null,
                    input  uvm_sequence_base parent    = null,
                    input  int               prior     = -1,
                    input  uvm_object        extension = null,
                    input  string            fname     = "",
                    input  int               lineno    = 0               )
```

Pushes (writes) all values preloaded using <set(()> to the DUT>. You must *update* after *set* before any blocking statements, else other reads/writes to the DUT FIFO may cause the mirror to become out of sync with the DUT.

mirror

Reads the next value out of the DUT FIFO. If auto-prediction is enabled, the frontmost value in abstract FIFO is popped. If the *check* argument is set and comparison is enabled with set_compare().

get

```
virtual function uvm_reg_data_t get(string fname  = "",
                                    int    lineno = 0  )
```

Returns the next value from the abstract FIFO, but does not pop it. Used to get the expected value in a mirror() operation.

do_predict

```
virtual function void do_predict(uvm_reg_item      rw,
                                 uvm_predict_e     kind = UVM_PREDICT_DIRECT,
                                 uvm_reg_byte_en_t be   = -1
```

Updates the abstract (mirror) FIFO based on write() and read() operations. When auto-prediction is on, this method is called before each read, write, peek, or poke operation

returns. When auto-prediction is off, this method is called by a uvm_reg_predictor upon receipt and conversion of an observed bus operation to this register.

If a write prediction, the observed write value is pushed to the abstract FIFO as long as it is not full and the operation did not originate from an update(). If a read prediction, the observed read value is compared with the frontmost value in the abstract FIFO if set_compare() enabled comparison and the FIFO is not empty.

Special Overrides

pre_write

```
virtual task pre_write(uvm_reg_item rw)
```

Special pre-processing for a write() or update(). Called as a result of a write() or update(). It is an error to attempt a write to a full FIFO or a write while an update is still pending. An update is pending after one or more calls to set(). If in your application the DUT allows writes to a full FIFO, you must override *pre_write* as appropriate.

pre_read

```
virtual task pre_read(uvm_reg_item rw)
```

Special post-processing for a write() or update(). Aborts the operation if the internal FIFO is empty. If in your application the DUT does not behave this way, you must override *pre_write* as appropriate.

uvm_vreg

A virtual register is a collection of fields, overlaid on top of a memory, usually in an array. The semantics and layout of virtual registers comes from an agreement between the software and the hardware, not any physical structures in the DUT.

Contents

uvm_vreg

Virtual register abstraction base class

A virtual register represents a set of fields that are logically implemented in consecutive memory locations.

All virtual register accesses eventually turn into memory accesses.

A virtual register array may be implemented on top of any memory abstraction class and possibly dynamically resized and/or relocated.

Summary

uvm_vreg

Virtual register abstraction base class

Class Hierarchy

uvm_void
uvm_object
uvm_vreg

Class Declaration

```
class uvm_vreg extends uvm_object
```

Initialization

new	Create a new instance and type-specific configuration
configure	Instance-specific configuration
implement	Dynamically implement, resize or relocate a virtual register array
allocate	Randomly implement, resize or relocate a virtual register array
get_region	Get the region where the virtual register array is implemented
release_region	Dynamically un-implement a virtual register array

Introspection

get_name	Get the simple name

get_full_name	Get the hierarchical name
get_parent	Get the parent block
get_memory	Get the memory where the virtual regoster array is implemented
get_n_maps	Returns the number of address maps this virtual register array is mapped in
is_in_map	Return TRUE if this virtual register array is in the specified address *map*
get_maps	Returns all of the address *maps* where this virtual register array is mapped
get_rights	Returns the access rights of this virtual reigster array
get_access	Returns the access policy of the virtual register array when written and read via an address map.
get_size	Returns the size of the virtual register array.
get_n_bytes	Returns the width, in bytes, of a virtual register.
get_n_memlocs	Returns the number of memory locations used by a single virtual register.
get_incr	Returns the number of memory locations between two individual virtual registers in the same array.
get_fields	Return the virtual fields in this virtual register
get_field_by_name	Return the named virtual field in this virtual register
get_offset_in_memory	Returns the offset of a virtual register
get_address	Returns the base external physical address of a virtual register

HDL Access

write	Write the specified value in a virtual register
read	Read the current value from a virtual register
poke	Deposit the specified value in a virtual register
peek	Sample the current value in a virtual register
reset	Reset the access semaphore

Callbacks

pre_write	Called before virtual register write.
post_write	Called after virtual register write.
pre_read	Called before virtual register read.
post_read	Called after virtual register read.

Initialization

new

```
function new(     string   name,
             int unsigned n_bits)
```

Create a new instance and type-specific configuration

Creates an instance of a virtual register abstraction class with the specified name.

n_bits specifies the total number of bits in a virtual register. Not all bits need to be mapped to a virtual field. This value is usually a multiple of 8.

configure

```
function void configure(         uvm_reg_block  parent,
                                 uvm_mem        mem     = null,
                        longint  unsigned       size    = 0,
                                 uvm_reg_addr_t offset  = 0,
                            int  unsigned       incr    = 0    )
```

Instance-specific configuration

Specify the *parent* block of this virtual register array. If one of the other parameters are specified, the virtual register is assumed to be dynamic and can be later (re-)implemented using the uvm_vreg::implement() method.

If *mem* is specified, then the virtual register array is assumed to be statically implemented in the memory corresponding to the specified memory abstraction class and *size*, *offset* and *incr* must also be specified. Static virtual register arrays cannot be re-implemented.

implement

```
virtual function bit implement(longint unsigned       n,
                               uvm_mem        mem    = null,
                               uvm_reg_addr_t offset = 0,
                           int unsigned       incr   = 0    )
```

Dynamically implement, resize or relocate a virtual register array

Implement an array 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 increment 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 array is in the same memory, at the same base offset using the same offset increment as originally implemented. Only the number of virtual registers in the virtual register array is modified.

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 base offset and offset increment. Returns FALSE otherwise.

The memory region used to implement a virtual register array is reserved in the memory allocation manager associated with the memory to prevent it from being allocated for another purpose.

allocate

```
virtual function uvm_mem_region allocate(longint unsigned    n,
                                         uvm_mem_mam mam)
```

Randomly implement, resize or relocate a virtual register array

Implement a virtual register array of the specified size in 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 uvm_mem_region memory region descriptor if the memory allocation manager was able to allocate a region that can implement the virtual register array. Returns *null* otherwise.

A region implementing a virtual register array must not be released using the uvm_mem_mam::release_region() method. It must be released using the uvm_vreg::release_region() method.

get_region

```
virtual function uvm_mem_region get_region()
```

Get the region where the virtual register array is implemented

Returns a reference to the uvm_mem_region memory region descriptor that implements the virtual register array.

Returns *null* if the virtual registers array is not currently implemented. A region implementing a virtual register array must not be released using the uvm_mem_mam::release_region() method. It must be released using the uvm_vreg::release_region() method.

release_region

```
virtual function void release_region()
```

Dynamically un-implement a virtual register array

Release the memory region used to implement a virtual register array and return it to the pool of available memory that can be allocated by the memory's default allocation manager. The virtual register array is subsequently considered as unimplemented and can no longer be accessed.

Statically-implemented virtual registers cannot be released.

INTROSPECTION

get_name

Get the simple name

Return the simple object name of this register.

get_full_name

```
virtual function string get_full_name()
```

Get the hierarchical name

Return the hierarchal name of this register. The base of the hierarchical name is the root block.

get_parent

```
virtual function uvm_reg_block get_parent()
```

Get the parent block

get_memory

```
virtual function uvm_mem get_memory()
```

Get the memory where the virtual regoster array is implemented

get_n_maps

```
virtual function int get_n_maps ()
```

Returns the number of address maps this virtual register array is mapped in

is_in_map

```
function bit is_in_map (uvm_reg_map map)
```

Return TRUE if this virtual register array is in the specified address *map*

get_maps

```
virtual function void get_maps (ref uvm_reg_map maps[$])
```

Returns all of the address *maps* where this virtual register array is mapped

get_rights

```
virtual function string get_rights(uvm_reg_map map = null)
```

Returns the access rights of this virtual reigster array

Returns "RW", "RO" or "WO". The access rights of a virtual register array is always "RW", unless it is implemented in a shared memory with access restriction in a particular address map.

If no address map is specified and the memory is mapped in only one address map, that address map is used. If the memory is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the memory is not mapped in the specified address map, an error message is issued and "RW" is returned.

get_access

```
virtual function string get_access(uvm_reg_map map = null)
```

Returns the access policy of the virtual register array when written and read via an address map.

If the memory implementing the virtual register array is mapped in more than one address map, an address *map* must be specified. If access restrictions are present when accessing a memory through the specified address map, the access mode returned takes the access restrictions into account. For example, a read-write memory accessed through an address map with read-only restrictions would return "RO".

get_size

```
virtual function int unsigned get_size()
```

Returns the size of the virtual register array.

get_n_bytes

```
virtual function int unsigned get_n_bytes()
```

Returns the width, in bytes, of a 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.

get_n_memlocs

```
virtual function int unsigned get_n_memlocs()
```

Returns the number of memory locations used by a single virtual register.

get_incr

```
virtual function int unsigned get_incr()
```

Returns the number of memory locations between two individual virtual registers in the same array.

get_fields

```
virtual function void get_fields(ref uvm_vreg_field fields[$])
```

Return the virtual fields in this virtual register

Fills the specified array with the abstraction class for all of the virtual fields contained in this virtual register. Fields are ordered from least-significant position to most-significant position within the register.

get_field_by_name

```
virtual function uvm_vreg_field get_field_by_name(string name)
```

Return the named virtual field in this virtual register

Finds a virtual field with the specified name in this virtual register and returns its abstraction class. If no fields are found, returns null.

get_offset_in_memory

```
virtual function uvm_reg_addr_t get_offset_in_memory(longint unsigned idx)
```

Returns the offset of a virtual register

Returns the base offset of the specified virtual register, in the overall address space of the memory that implements the virtual register array.

get_address

```
virtual function uvm_reg_addr_t get_address(longint unsigned    idx,
                                            uvm_reg_map map  = null)
```

Returns the base external physical address of a virtual register

Returns the base external physical address of the specified virtual reigster if accessed through the specified address *map*.

If no address map is specified and the memory implementing the virtual register array is mapped in only one address map, that address map is used. If the memory is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the memory is not mapped in the specified address map, an error message is issued.

HDL Access

write

```
virtual task write(input  longint unsigned          idx,
                   output uvm_status_e      status,
                    input uvm_reg_data_t    value,
                    input uvm_path_e        path      = UVM_DEFAULT_PA
                    input uvm_reg_map       map       = null,
                    input uvm_sequence_base parent    = null,
                    input uvm_object        extension = null,
                    input string            fname     = "",
                    input int               lineno    = 0
```

Write the specified value in a virtual register

Write *value* in the DUT memory location(s) that implements the virtual register array that corresponds to this abstraction class instance using the specified access *path*.

If the memory implementing the virtual register array is mapped in more than one address map, an address *map* must be specified if a physical access is used (front-door access).

The operation is eventually mapped into set of memory-write operations at the location where the virtual register specified by *idx* in the virtual register array is implemented.

read

```
virtual task read(input  longint unsigned          idx,
                  output uvm_status_e      status,
                  output uvm_reg_data_t    value,
                   input uvm_path_e        path      = UVM_DEFAULT_PAT
```

```
                          input uvm_reg_map       map       = null,
                          input uvm_sequence_base parent    = null,
                          input uvm_object        extension = null,
                          input string            fname     = "",
                          input int               lineno    = 0
```

Read the current value from a virtual register

Read from the DUT memory location(s) that implements the virtual register array that corresponds to this abstraction class instance using the specified access *path* and return the readback *value*.

If the memory implementing the virtual register array is mapped in more than one address map, an address *map* must be specified if a physical access is used (front-door access).

The operation is eventually mapped into set of memory-read operations at the location where the virtual register specified by *idx* in the virtual register array is implemented.

poke

```
virtual task poke(input  longint unsigned          idx,
                  output uvm_status_e      status,
                  input  uvm_reg_data_t    value,
                  input  uvm_sequence_base parent    = null,
                  input  uvm_object        extension = null,
                  input  string            fname     = "",
                  input  int               lineno    = 0    )
```

Deposit the specified value in a virtual register

Deposit *value* in the DUT memory location(s) that implements the virtual register array that corresponds to this abstraction class instance using the memory backdoor access.

The operation is eventually mapped into set of memory-poke operations at the location where the virtual register specified by *idx* in the virtual register array is implemented.

peek

```
virtual task peek(input  longint unsigned          idx,
                  output uvm_status_e      status,
                  output uvm_reg_data_t    value,
                  input  uvm_sequence_base parent    = null,
                  input  uvm_object        extension = null,
                  input  string            fname     = "",
                  input  int               lineno    = 0    )
```

Sample the current value in a virtual register

Sample the DUT memory location(s) that implements the virtual register array that corresponds to this abstraction class instance using the memory backdoor access, and return the sampled *value*.

The operation is eventually mapped into set of memory-peek operations at the location where the virtual register specified by *idx* in the virtual register array is implemented.

reset

```
function void reset(string kind = "HARD")
```

Reset the access semaphore

Reset the semaphore that prevents concurrent access to the virtual register. This semaphore must be explicitly reset if a thread accessing this virtual register array was killed in before the access was completed

CALLBACKS

pre_write

```
virtual task pre_write(longint unsigned       idx,
                       ref uvm_reg_data_t wdat,
                       ref uvm_path_e     path,
                       ref uvm_reg_map    map  )
```

Called before virtual register write.

If the specified data value, access *path* or address *map* are modified, the updated data value, access path or address map will be used to perform the virtual register operation.

The registered callback methods are invoked after the invocation of this method. All register callbacks are executed after the corresponding field callbacks The pre-write virtual register and field callbacks are executed before the corresponding pre-write memory callbacks

post_write

```
virtual task post_write(longint unsigned       idx,
                        uvm_reg_data_t wdat,
                        uvm_path_e     path,
                        uvm_reg_map    map,
                    ref uvm_status_e   status)
```

Called after virtual register write.

If the specified *status* is modified, the updated status will be returned by the virtual register operation.

The registered callback methods are invoked before the invocation of this method. All register callbacks are executed before the corresponding field callbacks The post-write virtual register and field callbacks are executed after the corresponding post-write memory callbacks

pre_read

```
virtual task pre_read(longint unsigned    idx,
                      ref uvm_path_e  path,
                      ref uvm_reg_map map  )
```

Called before virtual register read.

If the specified access *path* or address *map* are modified, the updated access path or address map will be used to perform the register operation.

The registered callback methods are invoked after the invocation of this method. All register callbacks are executed after the corresponding field callbacks The pre-read virtual register and field callbacks are executed before the corresponding pre-read

memory callbacks

post_read

```
virtual task post_read(longint unsigned        idx,
                         ref uvm_reg_data_t rdat,
                       input uvm_path_e      path,
                       input uvm_reg_map     map,
                         ref uvm_status_e    status)
```

Called after virtual register read.

If the specified readback data or *status* is modified, the updated readback data or status will be returned by the register operation.

The registered callback methods are invoked before the invocation of this method. All register callbacks are executed before the corresponding field callbacks The post-read virtual register and field callbacks are executed after the corresponding post-read memory callbacks

uvm_vreg_cbs

Pre/post read/write callback facade class

Summary

uvm_vreg_cbs

Pre/post read/write callback facade class

Class Hierarchy

- uvm_void
- uvm_object
- uvm_callback
- **uvm_vreg_cbs**

Class Declaration

```
class uvm_vreg_cbs extends uvm_callback
```

Methods

pre_write	Callback called before a write operation.
post_write	Called after register write.
pre_read	Called before register read.
post_read	Called after register read.

Types

uvm_vreg_cb	Convenience callback type declaration
uvm_vreg_cb_iter	Convenience callback iterator type declaration

Methods

pre_write

```
virtual task pre_write(        uvm_vreg       rg,
                       longint unsigned       idx,
                           ref uvm_reg_data_t wdat,
                           ref uvm_path_e     path,
                           ref uvm_reg_map    map  )
```

Callback called before a write operation.

The registered callback methods are invoked after the invocation of the uvm_vreg::pre_write() method. All virtual register callbacks are executed after the corresponding virtual field callbacks The pre-write virtual register and field callbacks are executed before the corresponding pre-write memory callbacks

The written value *wdat*, access *path* and address *map*, if modified, modifies the actual value, access path or address map used in the virtual register operation.

post_write

```
virtual task post_write(        uvm_vreg       rg,
                        longint unsigned       idx,
                                uvm_reg_data_t wdat,
                                uvm_path_e     path,
                                uvm_reg_map    map,
                            ref uvm_status_e   status)
```

Called after register write.

The registered callback methods are invoked before the invocation of the uvm_reg::post_write() method. All register callbacks are executed before the corresponding virtual field callbacks The post-write virtual register and field callbacks are executed after the corresponding post-write memory callbacks

The *status* of the operation, if modified, modifies the actual returned status.

pre_read

```
virtual task pre_read(        uvm_vreg    rg,
                      longint unsigned    idx,
                          ref uvm_path_e  path,
                          ref uvm_reg_map map  )
```

Called before register read.

The registered callback methods are invoked after the invocation of the uvm_reg::pre_read() method. All register callbacks are executed after the corresponding virtual field callbacks The pre-read virtual register and field callbacks are executed before the corresponding pre-read memory callbacks

The access *path* and address *map*, if modified, modifies the actual access path or address map used in the register operation.

post_read

```
virtual task post_read(        uvm_vreg       rg,
                       longint unsigned       idx,
                           ref uvm_reg_data_t rdat,
                         input uvm_path_e     path,
                         input uvm_reg_map    map,
                           ref uvm_status_e   status)
```

Called after register read.

The registered callback methods are invoked before the invocation of the uvm_reg::post_read() method. All register callbacks are executed before the corresponding virtual field callbacks The post-read virtual register and field callbacks are executed after the corresponding post-read memory callbacks

The readback value *rdat* and the *status* of the operation, if modified, modifies the actual returned readback value and status.

TYPES

uvm_vreg_cb

Convenience callback type declaration

Use this declaration to register virtual register callbacks rather than the more verbose parameterized class

uvm_vreg_cb_iter

Convenience callback iterator type declaration

Use this declaration to iterate over registered virtual register callbacks rather than the more verbose parameterized class

Virtual Register Field Classes

This section defines the virtual field and callback classes.

A virtual field is set of contiguous bits in one or more memory locations. The semantics and layout of virtual fields comes from an agreement between the software and the hardware, not any physical structures in the DUT.

Contents

uvm_vreg_field

Virtual field abstraction class

A virtual field represents a set of adjacent bits that are logically implemented in consecutive memory locations.

Summary

uvm_vreg_field

Virtual field abstraction class

Class Hierarchy

uvm_void

uvm_object

uvm_vreg_field

Class Declaration

```
class uvm_vreg_field extends uvm_object
```

INITIALIZATION

new	Create a new virtual field instance
configure	Instance-specific configuration

Introspection

get_name	Get the simple name
get_full_name	Get the hierarchical name
get_parent	Get the parent virtual register
get_lsb_pos_in_register	Return the position of the virtual field / Returns the index of the least significant bit of the virtual field in the virtual register that instantiates it.
get_n_bits	Returns the width, in bits, of the virtual field.
get_access	Returns the access policy of the virtual field register when written and read via an address map.

HDL Access

write	Write the specified value in a virtual field

read	Read the current value from a virtual field
poke	Deposit the specified value in a virtual field
peek	Sample the current value from a virtual field

CALLBACKS

pre_write	Called before virtual field write.
post_write	Called after virtual field write
pre_read	Called before virtual field read.
post_read	Called after virtual field read.

INITIALIZATION

new

```
function new(string name = "uvm_vreg_field")
```

Create a new virtual field instance

This method should not be used directly. The uvm_vreg_field::type_id::create() method should be used instead.

configure

```
function void configure(    uvm_vreg parent,
                         int unsigned size,
                         int unsigned lsb_pos)
```

Instance-specific configuration

Specify the *parent* virtual register of this virtual field, its *size* in bits, and the position of its least-significant bit within the virtual register relative to the least-significant bit of the virtual register.

INTROSPECTION

get_name

Get the simple name

Return the simple object name of this virtual field

get_full_name

```
virtual function string get_full_name()
```

Get the hierarchical name

Return the hierarchal name of this virtual field The base of the hierarchical name is the root block.

get_parent

```
virtual function uvm_vreg get_parent()
```

Get the parent virtual register

get_lsb_pos_in_register

```
virtual function int unsigned get_lsb_pos_in_register()
```

Return the position of the virtual field / Returns the index of the least significant bit of the virtual 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 register.

get_n_bits

```
virtual function int unsigned get_n_bits()
```

Returns the width, in bits, of the virtual field.

get_access

```
virtual function string get_access(uvm_reg_map map = null)
```

Returns the access policy of the virtual field register when written and read via an address map.

If the memory implementing the virtual field is mapped in more than one address map, an address *map* must be specified. If access restrictions are present when accessing a memory through the specified address map, the access mode returned takes the access restrictions into account. For example, a read-write memory accessed through an address map with read-only restrictions would return "RO".

HDL Access

write

```
virtual task write(input  longint unsigned          idx,
                   output uvm_status_e      status,
                   input  uvm_reg_data_t    value,
                   input  uvm_path_e        path      = UVM_DEFAULT_PA
                   input  uvm_reg_map       map       = null,
                   input  uvm_sequence_base parent    = null,
                   input  uvm_object        extension = null,
                   input  string            fname     = "",
                   input  int               lineno    = 0
```

Write the specified value in a virtual field

Write *value* in the DUT memory location(s) that implements the virtual field that corresponds to this abstraction class instance using the specified access *path*.

If the memory implementing the virtual register array containing this virtual field is mapped in more than one address map, an address *map* must be specified if a physical

access is used (front-door access).

The operation is eventually mapped into memory read-modify-write operations at the location where the virtual register specified by *idx* in the virtual register array is implemented. If a backdoor is available for the memory implemeting the virtual field, it will be used for the memory-read operation.

read

```
virtual task read(input  longint unsigned          idx,
                  output uvm_status_e      status,
                  output uvm_reg_data_t    value,
                   input uvm_path_e        path      = UVM_DEFAULT_PAT
                   input uvm_reg_map       map       = null,
                   input uvm_sequence_base parent    = null,
                   input uvm_object        extension = null,
                   input string            fname     = "",
                   input int               lineno    = 0
```

Read the current value from a virtual field

Read from the DUT memory location(s) that implements the virtual field that corresponds to this abstraction class instance using the specified access *path*, and return the readback *value*.

If the memory implementing the virtual register array containing this virtual field is mapped in more than one address map, an address *map* must be specified if a physical access is used (front-door access).

The operation is eventually mapped into memory read operations at the location(s) where the virtual register specified by *idx* in the virtual register array is implemented.

poke

```
virtual task poke(input  longint unsigned          idx,
                  output uvm_status_e      status,
                   input uvm_reg_data_t    value,
                   input uvm_sequence_base parent    = null,
                   input uvm_object        extension = null,
                   input string            fname     = "",
                   input int               lineno    = 0    )
```

Deposit the specified value in a virtual field

Deposit *value* in the DUT memory location(s) that implements the virtual field that corresponds to this abstraction class instance using the specified access *path*.

The operation is eventually mapped into memory peek-modify-poke operations at the location where the virtual register specified by *idx* in the virtual register array is implemented.

peek

```
virtual task peek(input  longint unsigned          idx,
                  output uvm_status_e      status,
                  output uvm_reg_data_t    value,
                   input uvm_sequence_base parent    = null,
                   input uvm_object        extension = null,
                   input string            fname     = "",
                   input int               lineno    = 0    )
```

Sample the current value from a virtual field

Sample from the DUT memory location(s) that implements the virtual field that corresponds to this abstraction class instance using the specified access *path*, and return the readback *value*.

If the memory implementing the virtual register array containing this virtual field is mapped in more than one address map, an address *map* must be specified if a physical access is used (front-door access).

The operation is eventually mapped into memory peek operations at the location(s) where the virtual register specified by *idx* in the virtual register array is implemented.

CALLBACKS

pre_write

```
virtual task pre_write(longint unsigned       idx,
                       ref uvm_reg_data_t wdat,
                       ref uvm_path_e     path,
                       ref uvm_reg_map    map  )
```

Called before virtual field write.

If the specified data value, access *path* or address *map* are modified, the updated data value, access path or address map will be used to perform the virtual register operation.

The virtual field callback methods are invoked before the callback methods on the containing virtual register. The registered callback methods are invoked after the invocation of this method. The pre-write virtual register and field callbacks are executed before the corresponding pre-write memory callbacks

post_write

```
virtual task post_write(longint unsigned       idx,
                        uvm_reg_data_t wdat,
                        uvm_path_e     path,
                        uvm_reg_map    map,
                    ref uvm_status_e   status)
```

Called after virtual field write

If the specified *status* is modified, the updated status will be returned by the virtual register operation.

The virtual field callback methods are invoked after the callback methods on the containing virtual register. The registered callback methods are invoked before the invocation of this method. The post-write virtual register and field callbacks are executed after the corresponding post-write memory callbacks

pre_read

```
virtual task pre_read(longint unsigned    idx,
                      ref uvm_path_e  path,
                      ref uvm_reg_map map  )
```

Called before virtual field read.

If the specified access *path* or address *map* are modified, the updated access path or address map will be used to perform the virtual register operation.

The virtual field callback methods are invoked after the callback methods on the containing virtual register. The registered callback methods are invoked after the invocation of this method. The pre-read virtual register and field callbacks are executed before the corresponding pre-read memory callbacks

post_read

```
virtual task post_read(longint unsigned        idx,
                       ref uvm_reg_data_t rdat,
                           uvm_path_e      path,
                           uvm_reg_map     map,
                       ref uvm_status_e   status)
```

Called after virtual field read.

If the specified readback data *rdat* or *status* is modified, the updated readback data or status will be returned by the virtual register operation.

The virtual field callback methods are invoked after the callback methods on the containing virtual register. The registered callback methods are invoked before the invocation of this method. The post-read virtual register and field callbacks are executed after the corresponding post-read memory callbacks

uvm_vreg_field_cbs

Pre/post read/write callback facade class

Summary

uvm_vreg_field_cbs

Pre/post read/write callback facade class

Class Hierarchy

- uvm_void
- uvm_object
- uvm_callback
- **uvm_vreg_field_cbs**

Class Declaration

```
class uvm_vreg_field_cbs extends uvm_callback
```

Methods

pre_write	Callback called before a write operation.
post_write	Called after a write operation
pre_read	Called before a virtual field read.
post_read	Called after a virtual field read.

Types

uvm_vreg_field_cb	Convenience callback type declaration
uvm_vreg_field_cb_iter	Convenience callback iterator type declaration

METHODS

pre_write

```
virtual task pre_write(        uvm_vreg_field field,
                       longint unsigned       idx,
                           ref uvm_reg_data_t wdat,
                           ref uvm_path_e     path,
                           ref uvm_reg_map    map   )
```

Callback called before a write operation.

The registered callback methods are invoked before the invocation of the virtual register pre-write callbacks and after the invocation of the uvm_vreg_field::pre_write() method.

The written value *wdat*, access *path* and address *map*, if modified, modifies the actual value, access path or address map used in the register operation.

post_write

```
virtual task post_write(        uvm_vreg_field field,
                        longint unsigned       idx,
                                uvm_reg_data_t wdat,
                                uvm_path_e     path,
                                uvm_reg_map    map,
                            ref uvm_status_e   status)
```

Called after a write operation

The registered callback methods are invoked after the invocation of the virtual register post-write callbacks and before the invocation of the uvm_vreg_field::post_write() method.

The *status* of the operation, if modified, modifies the actual returned status.

pre_read

```
virtual task pre_read(        uvm_vreg_field field,
                      longint unsigned       idx,
                          ref uvm_path_e     path,
                          ref uvm_reg_map    map   )
```

Called before a virtual field read.

The registered callback methods are invoked after the invocation of the virtual register pre-read callbacks and after the invocation of the uvm_vreg_field::pre_read() method.

The access *path* and address *map*, if modified, modifies the actual access path or address map used in the register operation.

post_read

```
virtual task post_read(        uvm_vreg_field field,
                       longint unsigned       idx,
                           ref uvm_reg_data_t rdat,
                               uvm_path_e     path,
                               uvm_reg_map    map,
```

```
                                ref uvm_status_e   status)
```

Called after a virtual field read.

The registered callback methods are invoked after the invocation of the virtual register post-read callbacks and before the invocation of the uvm_vreg_field::post_read() method.

The readback value *rdat* and the *status* of the operation, if modified, modifies the actual returned readback value and status.

TYPES

uvm_vreg_field_cb

Convenience callback type declaration

Use this declaration to register virtual field callbacks rather than the more verbose parameterized class

uvm_vreg_field_cb_iter

Convenience callback iterator type declaration

Use this declaration to iterate over registered virtual field callbacks rather than the more verbose parameterized class

Register Callbacks

This section defines the base class used for all register callback extensions. It also includes pre-defined callback extensions for use on read-only and write-only registers.

Contents

uvm_reg_cbs

Facade class for field, register, memory and backdoor access callback methods.

Summary

uvm_reg_cbs

Facade class for field, register, memory and backdoor access callback methods.

Class Hierarchy

uvm_void
uvm_object
uvm_callback
uvm_reg_cbs

Class Declaration

```
virtual class uvm_reg_cbs extends uvm_callback
```

Methods	
pre_write	Called before a write operation.
post_write	Called after user-defined backdoor register write.
pre_read	Callback called before a read operation.
post_read	Callback called after a read operation.
post_predict	Called by the uvm_reg_field::predict() method after a successful UVM_PREDICT_READ or UVM_PREDICT_WRITE prediction.
encode	Data encoder
decode	Data decode

Methods

pre_write

```
virtual task pre_write(uvm_reg_item rw)
```

Called before a write operation.

All registered *pre_write* callback methods are invoked after the invocation of the *pre_write* method of associated object (uvm_reg, uvm_reg_field, uvm_mem, or uvm_reg_backdoor). If the element being written is a uvm_reg, all *pre_write* callback methods are invoked before the contained uvm_reg_fields.

Backdoor	uvm_reg_backdoor::pre_write, uvm_reg_cbs::pre_write cbs for backdoor.
Register	uvm_reg::pre_write, uvm_reg_cbs::pre_write cbs for reg, then foreach field: uvm_reg_field::pre_write, uvm_reg_cbs::pre_write cbs for field
RegField	uvm_reg_field::pre_write, uvm_reg_cbs::pre_write cbs for field
Memory	uvm_mem::pre_write, uvm_reg_cbs::pre_write cbs for mem

The *rw* argument holds information about the operation.

- Modifying the *value* modifies the actual value written.
- For memories, modifying the *offset* modifies the offset used in the operation.
- For non-backdoor operations, modifying the access *path* or address *map* modifies the actual path or map used in the operation.

If the *rw.status* is modified to anything other than UVM_IS_OK, the operation is aborted.

See uvm_reg_item for details on *rw* information.

post_write

```
virtual task post_write(uvm_reg_item rw)
```

Called after user-defined backdoor register write.

All registered *post_write* callback methods are invoked before the invocation of the *post_write* method of the associated object (uvm_reg, uvm_reg_field, uvm_mem, or uvm_reg_backdoor). If the element being written is a uvm_reg, all *post_write* callback methods are invoked before the contained uvm_reg_fields.

Summary of callback order

Backdoor	uvm_reg_cbs::post_write cbs for backdoor, uvm_reg_backdoor::post_write
Register	uvm_reg_cbs::post_write cbs for reg, uvm_reg::post_write, then foreach field: uvm_reg_cbs::post_write cbs for field, uvm_reg_field::post_read
RegField	uvm_reg_cbs::post_write cbs for field, uvm_reg_field::post_write
Memory	uvm_reg_cbs::post_write cbs for mem, uvm_mem::post_write

The *rw* argument holds information about the operation.

- Modifying the *status* member modifies the returned status.
- Modiying the *value* or *offset* members has no effect, as the operation has already completed.

See uvm_reg_item for details on *rw* information.

pre_read

```
virtual task pre_read(uvm_reg_item rw)
```

Callback called before a read operation.

All registered *pre_read* callback methods are invoked after the invocation of the *pre_read* method of associated object (uvm_reg, uvm_reg_field, uvm_mem, or uvm_reg_backdoor). If the element being read is a uvm_reg, all *pre_read* callback methods are invoked before the contained uvm_reg_fields.

Backdoor	uvm_reg_backdoor::pre_read, uvm_reg_cbs::pre_read cbs for backdoor
Register	uvm_reg::pre_read, uvm_reg_cbs::pre_read cbs for reg, then foreach field: uvm_reg_field::pre_read, uvm_reg_cbs::pre_read cbs for field
RegField	uvm_reg_field::pre_read, uvm_reg_cbs::pre_read cbs for field
Memory	uvm_mem::pre_read, uvm_reg_cbs::pre_read cbs for mem

The *rw* argument holds information about the operation.

- The *value* member of *rw* is not used has no effect if modified.
- For memories, modifying the *offset* modifies the offset used in the operation.
- For non-backdoor operations, modifying the access *path* or address *map* modifies the actual path or map used in the operation.

If the *rw.status* is modified to anything other than UVM_IS_OK, the operation is aborted.

See uvm_reg_item for details on *rw* information.

post_read

```
virtual task post_read(uvm_reg_item rw)
```

Callback called after a read operation.

All registered *post_read* callback methods are invoked before the invocation of the *post_read* method of the associated object (uvm_reg, uvm_reg_field, uvm_mem, or uvm_reg_backdoor). If the element being read is a uvm_reg, all *post_read* callback methods are invoked before the contained uvm_reg_fields.

Backdoor	uvm_reg_cbs::post_read cbs for backdoor, uvm_reg_backdoor::post_read
Register	uvm_reg_cbs::post_read cbs for reg, uvm_reg::post_read, then foreach field: uvm_reg_cbs::post_read cbs for field, uvm_reg_field::post_read
RegField	uvm_reg_cbs::post_read cbs for field, uvm_reg_field::post_read
Memory	uvm_reg_cbs::post_read cbs for mem, uvm_mem::post_read

The *rw* argument holds information about the operation.

- Modifying the readback *value* or *status* modifies the actual returned value and status.
- Modiying the *value* or *offset* members has no effect, as the operation has already completed.

See uvm_reg_item for details on *rw* information.

post_predict

```
virtual function void post_predict(input uvm_reg_field  fld,
                                   input uvm_reg_data_t previous,
                                   inout uvm_reg_data_t value,
                                   input uvm_predict_e  kind,
                                   input uvm_path_e     path,
                                   input uvm_reg_map    map      )
```

Called by the uvm_reg_field::predict() method after a successful UVM_PREDICT_READ or UVM_PREDICT_WRITE prediction.

previous is the previous value in the mirror and *value* is the latest predicted value. Any change to *value* will modify the predicted mirror value.

encode

```
virtual function void encode(ref uvm_reg_data_t data[])
```

Data encoder

The registered callback methods are invoked in order of registration after all the *pre_write* methods have been called. The encoded data is passed through each invocation in sequence. This allows the *pre_write* methods to deal with clear-text data.

By default, the data is not modified.

decode

```
virtual function void decode(ref uvm_reg_data_t data[])
```

Data decode

The registered callback methods are invoked in *reverse order* of registration before all the *post_read* methods are called. The decoded data is passed through each invocation in sequence. This allows the *post_read* methods to deal with clear-text data.

The reversal of the invocation order is to allow the decoding of the data to be performed in the opposite order of the encoding with both operations specified in the same callback extension.

By default, the data is not modified.

Typedefs

Summary

Typedefs

uvm_reg_cb	Convenience callback type declaration for registers
uvm_reg_cb_iter	Convenience callback iterator type declaration for registers
uvm_reg_bd_cb	Convenience callback type declaration for backdoor
uvm_reg_bd_cb_iter	Convenience callback iterator type declaration for backdoor
uvm_mem_cb	Convenience callback type declaration for memories
uvm_mem_cb_iter	Convenience callback iterator type declaration for memories
uvm_reg_field_cb	Convenience callback type declaration for fields
uvm_reg_field_cb_iter	Convenience callback iterator type declaration for fields

Predefined Extensions

uvm_reg_cb

Convenience callback type declaration for registers

Use this declaration to register register callbacks rather than the more verbose parameterized class

uvm_reg_cb_iter

Convenience callback iterator type declaration for registers

Use this declaration to iterate over registered register callbacks rather than the more verbose parameterized class

uvm_reg_bd_cb

Convenience callback type declaration for backdoor

Use this declaration to register register backdoor callbacks rather than the more verbose parameterized class

uvm_reg_bd_cb_iter

Convenience callback iterator type declaration for backdoor

Use this declaration to iterate over registered register backdoor callbacks rather than the more verbose parameterized class

uvm_mem_cb

Convenience callback type declaration for memories

Use this declaration to register memory callbacks rather than the more verbose parameterized class

uvm_mem_cb_iter

Convenience callback iterator type declaration for memories

Use this declaration to iterate over registered memory callbacks rather than the more verbose parameterized class

uvm_reg_field_cb

Convenience callback type declaration for fields

Use this declaration to register field callbacks rather than the more verbose parameterized class

uvm_reg_field_cb_iter

Convenience callback iterator type declaration for fields

Use this declaration to iterate over registered field callbacks rather than the more verbose parameterized class

Predefined Extensions

uvm_reg_read_only_cbs

Pre-defined register callback method for read-only registers that will issue an error if a write() operation is attempted.

Summary

uvm_reg_read_only_cbs

Pre-defined register callback method for read-only registers that will issue an error if a write() operation is attempted.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_callback
- uvm_reg_cbs

uvm_reg_read_only_cbs

CLASS DECLARATION

```
class uvm_reg_read_only_cbs extends uvm_reg_cbs
```

METHODS

pre_write	Produces an error message and sets status to UVM_NOT_OK.
add	Add this callback to the specified register and its contained fields.
remove	Remove this callback from the specified register and its contained fields.

METHODS

pre_write

```
virtual task pre_write(uvm_reg_item rw)
```

Produces an error message and sets status to UVM_NOT_OK.

add

```
static function void add(uvm_reg rg)
```

Add this callback to the specified register and its contained fields.

remove

```
static function void remove(uvm_reg rg)
```

Remove this callback from the specified register and its contained fields.

uvm_reg_write_only_cbs

Pre-defined register callback method for write-only registers that will issue an error if a read() operation is attempted.

Summary

uvm_reg_write_only_cbs

Pre-defined register callback method for write-only registers that will issue an error if a read() operation is attempted.

CLASS HIERARCHY

uvm_void

uvm_object

uvm_callback

uvm_reg_cbs

uvm_reg_write_only_cbs

Class Declaration

```
class uvm_reg_write_only_cbs extends uvm_reg_cbs
```

Methods

pre_read	Produces an error message and sets status to UVM_NOT_OK.
add	Add this callback to the specified register and its contained fields.
remove	Remove this callback from the specified register and its contained fields.

Methods

pre_read

```
virtual task pre_read(uvm_reg_item rw)
```

Produces an error message and sets status to UVM_NOT_OK.

add

```
static function void add(uvm_reg rg)
```

Add this callback to the specified register and its contained fields.

remove

```
static function void remove(uvm_reg rg)
```

Remove this callback from the specified register and its contained fields.

Memory Allocation Manager

Manages the exclusive allocation of consecutive memory locations called *regions*. The regions can subsequently be accessed like little memories of their own, without knowing in which memory or offset they are actually located.

The memory allocation manager should be used by any application-level process that requires reserved space in the memory, such as DMA buffers.

A region will remain reserved until it is explicitly released.

Contents

uvm_mem_mam

Memory allocation manager

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.

Summary

uvm_mem_mam

Memory allocation manager

Class Declaration

```
class uvm_mem_mam
```

Initialization

alloc_mode_e	Memory allocation mode
locality_e	Location of memory regions
default_alloc	Region allocation policy
new	Create a new manager instance
reconfigure	Reconfigure the manager

Memory Management

reserve_region	Reserve a specific memory region
request_region	Request and reserve a memory region
release_region	Release the specified region
release_all_regions	Forcibly release all allocated memory regions.

Introspection

convert2string	Image of the state of the manager
for_each	Iterate over all currently allocated regions
get_memory	Get the managed memory implementation

INITIALIZATION

alloc_mode_e

Memory allocation mode

Specifies how to allocate a memory region

GREEDY	Consume new, previously unallocated memory
THRIFTY	Reused previously released memory as much as possible (not yet implemented)

locality_e

Location of memory regions

Specifies where to locate new memory regions

BROAD	Locate new regions randomly throughout the address space
NEARBY	Locate new regions adjacent to existing regions

default_alloc

```
uvm_mem_mam_policy default_alloc
```

Region allocation policy

This object is repeatedly randomized when allocating new regions.

new

```
function new(string          name,
             uvm_mem_mam_cfg cfg,
             uvm_mem         mem   = null)
```

Create a new manager instance

Create an instance of a memory allocation manager with the specified name and configuration. This instance manages all memory region allocation within the address range specified in the configuration descriptor.

If a reference to a memory abstraction class is provided, the memory locations within the regions can be accessed through the region descriptor, using the uvm_mem_region::read() and uvm_mem_region::write() methods.

reconfigure

```
function uvm_mem_mam_cfg reconfigure(uvm_mem_mam_cfg cfg = null)
```

Reconfigure the manager

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. All currently allocated regions must fall within the new address space.

Returns the previous configuration.

if no new configuration is specified, simply returns the current configuration.

Memory Management

reserve_region

```
function uvm_mem_region reserve_region(bit [63:0]   start_offset,
                                       int unsigned n_bytes,
                                           string   fname        = "",
                                           int      lineno       = 0 )
```

Reserve a specific memory region

Reserve a memory region of the specified number of bytes starting at the specified offset. 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.

Regions can be reserved to create “holes” in the managed address space.

request_region

```
function uvm_mem_region request_region(int unsigned           n_bytes,
                                       uvm_mem_mam_policy alloc    = null
                                       string             fname    = "",
                                       int                lineno   = 0
```

Request and reserve a memory region

Request and reserve a memory region of the specified number of bytes starting at a random location. If an policy is specified, it is randomized to determine the start offset of the region. If no policy is specified, the policy found in the uvm_mem_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 policy.

If the memory allocation is configured to *THRIFTY* or *NEARBY*, a suitable region is first sought procedurally.

release_region

```
function void release_region(uvm_mem_region region)
```

Release the specified region

Release a previously allocated memory region. An error is issued if the specified region has not been previously allocated or is no longer allocated.

release_all_regions

```
function void release_all_regions()
```

Forcibly release all allocated memory regions.

Introspection

convert2string

```
function string convert2string()
```

Image of the state of the manager

Create a human-readable description of the state of the memory manager and the currently allocated regions.

for_each

```
function uvm_mem_region for_each(bit reset = 0)
```

Iterate over all currently allocated regions

If reset is *TRUE*, reset the iterator and return the first allocated region. Returns *null* when there are no additional allocated regions to iterate on.

get_memory

```
function uvm_mem get_memory()
```

Get the managed memory implementation

Return the reference to the 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.

uvm_mem_region

Allocated memory region descriptor

Each instance of this class describes an allocated memory region. Instances of this class are created only by the memory manager, and returned by the uvm_mem_mam::reserve_region() and uvm_mem_mam::request_region() methods.

Summary

uvm_mem_region

Allocated memory region descriptor

Class Declaration

```
class uvm_mem_region
```

Methods

get_start_offset	Get the start offset of the region
get_end_offset	Get the end offset of the region
get_len	Size of the memory region
get_n_bytes	Number of bytes in the region
release_region	Release this region
get_memory	Get the memory where the region resides
get_virtual_registers	Get the virtual register array in this region
write	Write to a memory location in the region.
read	Read from a memory location in the region.
burst_write	Write to a set of memory location in the region.
burst_read	Read from a set of memory location in the region.
poke	Deposit in a memory location in the region.
peek	Sample a memory location in the region.

Methods

get_start_offset

```
function bit [63:0] get_start_offset()
```

Get the start offset of the region

Return the address offset, within the memory, where this memory region starts.

get_end_offset

```
function bit [63:0] get_end_offset()
```

Get the end offset of the region

Return the address offset, within the memory, where this memory region ends.

get_len

```
function int unsigned get_len()
```

Size of the memory region

Return the number of consecutive memory locations (not necessarily bytes) in the allocated region.

get_n_bytes

```
function int unsigned get_n_bytes()
```

Number of bytes in the region

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.

release_region

```
function void release_region()
```

Release this region

get_memory

```
function uvm_mem get_memory()
```

Get the memory where the region resides

Return a reference to the 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.

get_virtual_registers

```
function uvm_vreg get_virtual_registers()
```

Get the virtual register array in this region

Return a reference to the virtual register array abstraction class implemented in this region. Returns *null* if the memory region is not known to implement virtual registers.

write

```
task write(output uvm_status_e      status,
           input uvm_reg_addr_t     offset,
           input uvm_reg_data_t     value,
           input uvm_path_e         path      = UVM_DEFAULT_PATH,
           input uvm_reg_map        map       = null,
           input uvm_sequence_base  parent    = null,
           input int                prior     = -1,
           input uvm_object         extension = null,
           input string             fname     = "",
           input int                lineno    = 0               )
```

Write to a memory location in the region.

Write to the memory location that corresponds to the specified *offset* within this region. Requires that the memory abstraction class be associated with the memory allocation manager that allocated this region.

See uvm_mem::write() for more details.

read

```
task read(output uvm_status_e      status,
          input uvm_reg_addr_t     offset,
         output uvm_reg_data_t     value,
          input uvm_path_e         path      = UVM_DEFAULT_PATH,
          input uvm_reg_map        map       = null,
          input uvm_sequence_base  parent    = null,
          input int                prior     = -1,
          input uvm_object         extension = null,
          input string             fname     = "",
          input int                lineno    = 0               )
```

Read from a memory location in the region.

Read from the memory location that corresponds to the specified *offset* within this region. Requires that the memory abstraction class be associated with the memory allocation manager that allocated this region.

See uvm_mem::read() for more details.

burst_write

```
task burst_write(output uvm_status_e      status,
                 input uvm_reg_addr_t     offset,
                 input uvm_reg_data_t     value[],
                 input uvm_path_e         path      = UVM_DEFAULT_PATH,
                 input uvm_reg_map        map       = null,
                 input uvm_sequence_base  parent    = null,
                 input int                prior     = -1,
                 input uvm_object         extension = null,
                 input string             fname     = "",
                 input int                lineno    = 0               )
```

Write to a set of memory location in the region.

Write to the memory locations that corresponds to the specified *burst* within this region. Requires that the memory abstraction class be associated with the memory allocation manager that allocated this region.

See uvm_mem::burst_write() for more details.

burst_read

```
task burst_read(output uvm_status_e      status,
                input uvm_reg_addr_t     offset,
               output uvm_reg_data_t     value[],
                input uvm_path_e         path      = UVM_DEFAULT_PATH,
                input uvm_reg_map        map       = null,
                input uvm_sequence_base  parent    = null,
                input int                prior     = -1,
                input uvm_object         extension = null,
                input string             fname     = "",
                input int                lineno    = 0               )
```

Read from a set of memory location in the region.

Read from the memory locations that corresponds to the specified *burst* within this region. Requires that the memory abstraction class be associated with the memory allocation manager that allocated this region.

See uvm_mem::burst_read() for more details.

poke

```
task poke(output uvm_status_e      status,
```

```
            input uvm_reg_addr_t    offset,
            input uvm_reg_data_t    value,
            input uvm_sequence_base parent    = null,
            input uvm_object        extension = null,
            input string            fname     = "",
            input int               lineno    = 0    )
```

Deposit in a memory location in the region.

Deposit the specified value in the memory location that corresponds to the specified *offset* within this region. Requires that the memory abstraction class be associated with the memory allocation manager that allocated this region.

See uvm_mem::poke() for more details.

peek

```
task peek(output uvm_status_e      status,
           input uvm_reg_addr_t    offset,
          output uvm_reg_data_t    value,
           input uvm_sequence_base parent    = null,
           input uvm_object        extension = null,
           input string            fname     = "",
           input int               lineno    = 0    )
```

Sample a memory location in the region.

Sample the memory location that corresponds to the specified *offset* within this region. Requires that the memory abstraction class be associated with the memory allocation manager that allocated this region.

See uvm_mem::peek() for more details.

uvm_mem_mam_policy

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. If a procedural region allocation policy is required, it can be implemented in the pre/post_randomize() method.

Summary

uvm_mem_mam_policy

An instance of this class is randomized to determine the starting offset of a randomly allocated memory region.

Class Declaration

```
class uvm_mem_mam_policy
```

Variables

len	Number of addresses required
start_offset	The starting offset of the region
min_offset	Minimum address offset in the managed address space
max_offset	Maximum address offset in the managed address space
in_use	Regions already allocated in the managed address space

VARIABLES

len

```
int unsigned len
```

Number of addresses required

start_offset

```
rand bit [63:0] start_offset
```

The starting offset of the region

min_offset

```
bit [63:0] min_offset
```

Minimum address offset in the managed address space

max_offset

```
bit [63:0] max_offset
```

Maximum address offset in the managed address space

in_use

```
uvm_mem_region in_use[$]
```

Regions already allocated in the managed address space

uvm_mem_mam_cfg

Specifies the memory managed by an instance of a uvm_mem_mam memory allocation manager class.

Summary

uvm_mem_mam_cfg

Specifies the memory managed by an instance of a uvm_mem_mam memory allocation manager class.

CLASS DECLARATION

```
class uvm_mem_mam_cfg
```

Variables

n_bytes	Number of bytes in each memory location
end_offset	Last address of managed space
mode	Region allocation mode
locality	Region location mode

Variables

n_bytes

```
rand int unsigned n_bytes
```

Number of bytes in each memory location

end_offset

```
rand bit [63:0] end_offset
```

Last address of managed space

mode

```
rand uvm_mem_mam::alloc_mode_e mode
```

Region allocation mode

locality

```
rand uvm_mem_mam::locality_e locality
```

Region location mode

Generic Register Operation Descriptors

This section defines the abtract register transaction item. It also defines a descriptor for a physical bus operation that is used by uvm_reg_adapter subtypes to convert from a protocol-specific address/data/rw operation to a bus-independent, canonical r/w operation.

Contents

uvm_reg_item

Defines an abstract register transaction item. No bus-specific information is present, although a handle a uvm_reg_map is provided in case the user wishes to implement a custom address translation algorithm.

Summary

uvm_reg_item

Defines an abstract register transaction item.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_transaction
- uvm_sequence_item
- **uvm_reg_item**

Class Declaration

```
class uvm_reg_item extends uvm_sequence_item
```

Variables

element_kind	Kind of element being accessed: REG, MEM, or FIELD.
element	A handle to the RegModel model element associated with this transaction.
kind	Kind of access: READ or WRITE.
value	The value to write to, or after completion, the value read from the DUT.
offset	For memory accesses, the offset address.
status	The result of the transaction: IS_OK, HAS_X, or ERROR.
local_map	The local map used to obtain addresses.
map	The original map specified for the operation.

path	The path being used: UVM_FRONTDOOR or UVM_BACKDOOR.
parent	The sequence from which the operation originated.
prior	The priority requested of this transfer, as defined by uvm_sequence_base::start_item.
extension	Handle to optional user data, as conveyed in the call to write, read, mirror, or update call.
bd_kind	If path is UVM_BACKDOOR, this member specifies the abstraction kind for the backdoor access, e.g.
fname	The file name from where this transaction originated, if provided at the call site.
lineno	The file name from where this transaction originated, if provided at the call site.

METHODS

new	Create a new instance of this type, giving it the optional *name*.
convert2string	Returns a string showing the contents of this transaction.
do_copy	Copy the *rhs* object into this object.

VARIABLES

element_kind

```
uvm_elem_kind_e element_kind
```

Kind of element being accessed: REG, MEM, or FIELD. See uvm_elem_kind_e.

element

```
uvm_object element
```

A handle to the RegModel model element associated with this transaction. Use element_kind to determine the type to cast to: uvm_reg, uvm_mem, or uvm_reg_field.

kind

```
rand uvm_access_e kind
```

Kind of access: READ or WRITE.

value

```
rand uvm_reg_data_t value[]
```

The value to write to, or after completion, the value read from the DUT. Burst operations use the values property.

offset

```
rand uvm_reg_addr_t offset
```

For memory accesses, the offset address. For bursts, the *starting* offset address.

status

```
uvm_status_e status
```

The result of the transaction: IS_OK, HAS_X, or ERROR. See uvm_status_e.

local_map

```
uvm_reg_map local_map
```

The local map used to obtain addresses. Users may customize address-translation using this map. Access to the sequencer and bus adapter can be obtained by getting this map's root map, then calling uvm_reg_map::get_sequencer and uvm_reg_map::get_adapter.

map

```
uvm_reg_map map
```

The original map specified for the operation. The actual map used may differ when a test or sequence written at the block level is reused at the system level.

path

```
uvm_path_e path
```

The path being used: UVM_FRONTDOOR or UVM_BACKDOOR.

parent

```
rand uvm_sequence_base parent
```

The sequence from which the operation originated.

prior

```
int prior = -1
```

The priority requested of this transfer, as defined by uvm_sequence_base::start_item.

extension

```
rand uvm_object extension
```

Handle to optional user data, as conveyed in the call to write, read, mirror, or update call. Must derive from uvm_object.

bd_kind

```
string bd_kind = ""
```

If path is UVM_BACKDOOR, this member specifies the abstraction kind for the backdoor access, e.g. "RTL" or "GATES".

fname

```
string fname = ""
```

The file name from where this transaction originated, if provided at the call site.

lineno

```
int lineno = 0
```

The file name from where this transaction originated, if provided at the call site.

Methods

new

```
function new(string name = "")
```

Create a new instance of this type, giving it the optional *name*.

convert2string

```
virtual function string convert2string()
```

Returns a string showing the contents of this transaction.

do_copy

```
virtual function void do_copy(uvm_object rhs)
```

Copy the *rhs* object into this object. The *rhs* object must derive from uvm_reg_item.

uvm_reg_bus_op

Struct that defines a generic bus transaction for register and memory accesses, having *kind* (read or write), *address*, *data*, and *byte enable* information. If the bus is narrower than the register or memory location being accessed, there will be multiple of these bus operations for every abstract uvm_reg_item transaction. In this case, *data* represents the portion of uvm_reg_item::value being transferred during this bus cycle. If the bus is

wide enough to perform the register or memory operation in a single cycle, *data* will be the same as uvm_reg_item::value.

Summary

uvm_reg_bus_op

Struct that defines a generic bus transaction for register and memory accesses, having *kind* (read or write), *address*, *data*, and *byte enable* information.

VARIABLES

info	The bus-independent read/write information.
kind	Kind of access: READ or WRITE.
addr	The bus address.
data	The data to write.
n_bits	The number of bits of uvm_reg_item::value being transferred by this transaction.
byte_en	Enables for the byte lanes on the bus.
status	The result of the transaction: UVM_IS_OK, UVM_HAS_X, UVM_NOT_OK.

VARIABLES

info

The bus-independent read/write information. See uvm_reg_item.

kind

```
uvm_access_e kind
```

Kind of access: READ or WRITE.

addr

```
uvm_reg_addr_t addr
```

The bus address.

data

```
uvm_reg_data_t data
```

The data to write. If the bus width is smaller than the register or memory width, *data* represents only the portion of *value* that is being transferred this bus cycle.

n_bits

```
int n_bits
```

The number of bits of uvm_reg_item::value being transferred by this transaction.

byte_en

```
uvm_reg_byte_en_t byte_en
```

Enables for the byte lanes on the bus. Meaningful only when the bus supports byte enables and the operation originates from a field write/read.

status

```
uvm_status_e status
```

The result of the transaction: UVM_IS_OK, UVM_HAS_X, UVM_NOT_OK. See uvm_status_e.

Classes for Adapting Between Register and Bus Operations

This section defines classes used to convert transaction streams between generic register address/data reads and writes and physical bus accesses.

Contents

uvm_reg_adapter

This class defines an interface for converting between uvm_reg_bus_op and a specific bus transaction.

Summary

uvm_reg_adapter

This class defines an interface for converting between uvm_reg_bus_op and a specific bus transaction.

Class Hierarchy

uvm_void

uvm_object

uvm_reg_adapter

Class Declaration

```
virtual class uvm_reg_adapter extends uvm_object
```

new	Create a new instance of this type, giving it the optional *name*.
supports_byte_enable	Set this bit in extensions of this class if the bus protocol supports byte enables.
provides_responses	Set this bit in extensions of this class if the bus driver provides separate response items.
reg2bus	Extensions of this class *must* implement this method to convert a uvm_reg_item to the uvm_sequence_item subtype that defines the bus transaction.
bus2reg	Extensions of this class *must* implement this method to copy members of the given *bus_item* to corresponding members of the provided *bus_rw* instance.
Example	The following example illustrates how to implement a RegModel-BUS adapter class for the APB bus protocol.

new

```
function new(string name = "")
```

Create a new instance of this type, giving it the optional *name*.

supports_byte_enable

```
bit supports_byte_enable
```

Set this bit in extensions of this class if the bus protocol supports byte enables.

provides_responses

```
bit provides_responses
```

Set this bit in extensions of this class if the bus driver provides separate response items.

reg2bus

```
pure virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw)
```

Extensions of this class *must* implement this method to convert a uvm_reg_item to the uvm_sequence_item subtype that defines the bus transaction.

The method must allocate a new bus item, assign its members from the corresponding members from the given *bus_rw* item, then return it. The bus item gets returned in a uvm_sequence_item base handle.

bus2reg

```
pure virtual function void bus2reg(     uvm_sequence_item bus_item,
                                   ref uvm_reg_bus_op    rw       )
```

Extensions of this class *must* implement this method to copy members of the given *bus_item* to corresponding members of the provided *bus_rw* instance. Unlike reg2bus, the resulting transaction is not allocated from scratch. This is to accommodate applications where the bus response must be returned in the original request.

EXAMPLE

The following example illustrates how to implement a RegModel-BUS adapter class for the APB bus protocol.

```
class rreg2apb_adapter extends uvm_reg_adapter;
  `uvm_object_utils(reg2apb_adapter)

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

  endfunction

  virtual function uvm_sequence_item reg2bus(uvm_reg_bus_op rw);
```

```
    apb_item apb = apb_item::type_id::create("apb_item");
    apb.op   = (rw.kind == UVM_READ) ? apb::READ : apb::WRITE;
    apb.addr = rw.addr;
    apb.data = rw.data;
    return apb;
  endfunction

  virtual function void bus2reg(uvm_sequencer_item bus_item,
                                uvm_reg_bus_op rw);
    apb_item apb;
    if (!$cast(apb,bus_item)) begin
      `uvm_fatal("CONVERT_APB2REG","Bus item is not of type apb_item")
    end
    rw.kind  = apb.op==apb::READ ? UVM_READ : UVM_WRITE;
    rw.addr = apb.addr;
    rw.data = apb.data;
    rw.status = UVM_IS_OK;
  endfunction

endclass
```

uvm_reg_tlm_adapter

For converting between uvm_reg_bus_op and uvm_tlm_gp items.

Summary

uvm_reg_tlm_adapter

For converting between uvm_reg_bus_op and uvm_tlm_gp items.

Class Hierarchy

- uvm_void
- uvm_object
- uvm_reg_adapter
- **uvm_reg_tlm_adapter**

Class Declaration

```
class uvm_reg_tlm_adapter extends uvm_reg_adapter
```

Methods

reg2bus	Converts a uvm_reg_bus_op struct to a uvm_tlm_gp item.
bus2reg	Converts a uvm_tlm_gp item to a uvm_reg_bus_op.

Methods

reg2bus

```
virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw)
```

Converts a uvm_reg_bus_op struct to a uvm_tlm_gp item.

bus2reg

```
virtual function void bus2reg(     uvm_sequence_item bus_item,
                              ref uvm_reg_bus_op    rw       )
```

Converts a uvm_tlm_gp item to a uvm_reg_bus_op. into the provided *rw* transaction.

Register Sequence and Predictor Classes

This section defines the base classes used for register stimulus generation. It also defines a predictor component, which is used to update the register model's mirror values based on transactions observed on a physical bus.

Contents

uvm_reg_sequence

This class provides base functionality for both user-defined RegModel test sequences and "register translation sequences".

- When used as a base for user-defined RegModel test sequences, this class provides convenience methods for reading and writing registers and memories. Users implement the body() method to interact directly with the RegModel model (held in the model property) or indirectly via the delegation methods in this class.
- When used as a translation sequence, objects of this class are executed directly on a bus sequencerwhich are used in support of a layered sequencer use model, a pre-defined convert-and-execute algorithm is provided.

Register operations do not require extending this class if none of the above services are needed. Register test sequences can be extend from the base uvm_sequence #(REQ,RSP) base class or even from outside a sequence.

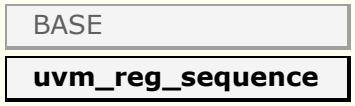
Note- The convenience API not yet implemented.

Summary

uvm_reg_sequence

This class provides base functionality for both user-defined RegModel test sequences and "register translation sequences".

Class Hierarchy



Class Declaration

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

BASE	Specifies the sequence type to extend from.

model	Block abstraction this sequence executes on, defined only when this sequence is a user-defined test sequence.
adapter	Adapter to use for translating between abstract register transactions and physical bus transactions, defined only when this sequence is a translation sequence.
reg_seqr	Layered upstream "register" sequencer.
new	Create a new instance, giving it the optional *name*.
body	Continually gets a register transaction from the configured upstream sequencer, reg_seqr, and executes the corresponding bus transaction via <do_rw_access>.
do_reg_item	Executes the given register transaction, *rw*, via the sequencer on which this sequence was started (i.e.
Convenience Write/Read API	The following methods delegate to the corresponding method in the register or memory element.
write_reg	Writes the given register *rg* using uvm_reg::write, supplying 'this' as the *parent* argument.
read_reg	Reads the given register *rg* using uvm_reg::read, supplying 'this' as the *parent* argument.
poke_reg	Pokes the given register *rg* using uvm_reg::poke, supplying 'this' as the *parent* argument.
peek_reg	Peeks the given register *rg* using uvm_reg::peek, supplying 'this' as the *parent* argument.
update_reg	Updates the given register *rg* using uvm_reg::update, supplying 'this' as the *parent* argument.
mirror_reg	Mirrors the given register *rg* using uvm_reg::mirror, supplying 'this' as the *parent* argument.
write_mem	Writes the given memory *mem* using uvm_mem::write, supplying 'this' as the *parent* argument.
read_mem	Reads the given memory *mem* using uvm_mem::read, supplying 'this' as the *parent* argument.
poke_mem	Pokes the given memory *mem* using uvm_mem::poke, supplying 'this' as the *parent* argument.
peek_mem	Peeks the given memory *mem* using uvm_mem::peek, supplying 'this' as the *parent* argument.

BASE

Specifies the sequence type to extend from.

When used as a translation sequence running on a bus sequencer, *BASE* must be compatible with the sequence type expected by the bus sequencer.

When used as a test sequence running on a particular sequencer, *BASE* must be compatible with the sequence type expected by that sequencer.

When used as a virtual test sequence without a sequencer, *BASE* does not need to be specified, i.e. the default specialization is adequate.

To maximize opportunities for reuse, user-defined RegModel sequences should "promote" the BASE parameter.

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

This way, the RegModel sequence can be extended from user-defined base sequences.

model

```
uvm_reg_block model
```

Block abstraction this sequence executes on, defined only when this sequence is a user-defined test sequence.

adapter

```
uvm_reg_adapter adapter
```

Adapter to use for translating between abstract register transactions and physical bus transactions, defined only when this sequence is a translation sequence.

reg_seqr

```
uvm_sequencer #(uvm_reg_item) reg_seqr
```

Layered upstream "register" sequencer.

Specifies the upstream sequencer between abstract register transactions and physical bus transactions. Defined only when this sequence is a translation sequence, and we want to "pull" from an upstream sequencer.

new

```
function new (string name = "uvm_reg_sequence_inst")
```

Create a new instance, giving it the optional *name*.

body

```
virtual task body()
```

Continually gets a register transaction from the configured upstream sequencer, reg_seqr, and executes the corresponding bus transaction via <do_rw_access>.

User-defined RegModel test sequences must override body() and not call super.body(), else a warning will be issued and the calling process not return.

do_reg_item

```
virtual task do_reg_item(uvm_reg_item rw)
```

Executes the given register transaction, *rw*, via the sequencer on which this sequence was started (i.e. m_sequencer). Uses the configured adapter to convert the register transaction into the type expected by this sequencer.

Convenience Write/Read API

The following methods delegate to the corresponding method in the register or memory element. They allow a sequence *body()* to do reads and writes without having to explicitly supply itself to *parent* sequence argument. Thus, a register write

```
model.regA.write(status, value, .parent(this));
```

can be written instead as

```
write_reg(model.regA, status, value);
```

write_reg

```
virtual task write_reg( input uvm_reg        rg,
                       output uvm_status_e   status,
                        input uvm_reg_data_t value,
                        input uvm_path_e     path      = UVM_DEFAULT_PATH,
                        input uvm_reg_map    map       = null,
                        input int            prior     = -1,
                        input uvm_object     extension = null,
                        input string         fname     = "",
                        input int            lineno    = 0               )
```

Writes the given register *rg* using uvm_reg::write, supplying 'this' as the *parent* argument. Thus,

```
write_reg(model.regA, status, value);
```

is equivalent to

```
model.regA.write(status, value, .parent(this));
```

read_reg

```
virtual task read_reg( input uvm_reg        rg,
                      output uvm_status_e   status,
                      output uvm_reg_data_t value,
                       input uvm_path_e     path      = UVM_DEFAULT_PATH,
                       input uvm_reg_map    map       = null,
                       input int            prior     = -1,
                       input uvm_object     extension = null,
                       input string         fname     = "",
                       input int            lineno    = 0               )
```

Reads the given register *rg* using uvm_reg::read, supplying 'this' as the *parent* argument. Thus,

```
read_reg(model.regA, status, value);
```

is equivalent to

```
```

```
model.regA.read(status, value, .parent(this));
```

poke_reg

```
virtual task poke_reg( input uvm_reg        rg,
                      output uvm_status_e   status,
                       input uvm_reg_data_t value,
                       input string         kind      = "",
                       input uvm_object     extension = null,
                       input string         fname     = "",
                       input int            lineno    = 0    )
```

Pokes the given register *rg* using uvm_reg::poke, supplying 'this' as the *parent* argument. Thus,

```
poke_reg(model.regA, status, value);
```

is equivalent to

```
model.regA.poke(status, value, .parent(this));
```

peek_reg

```
virtual task peek_reg( input uvm_reg        rg,
                      output uvm_status_e   status,
                      output uvm_reg_data_t value,
                       input string         kind      = "",
                       input uvm_object     extension = null,
                       input string         fname     = "",
                       input int            lineno    = 0    )
```

Peeks the given register *rg* using uvm_reg::peek, supplying 'this' as the *parent* argument. Thus,

```
peek_reg(model.regA, status, value);
```

is equivalent to

```
model.regA.peek(status, value, .parent(this));
```

update_reg

```
virtual task update_reg( input uvm_reg      rg,
                        output uvm_status_e status,
                         input uvm_path_e   path      = UVM_DEFAULT_PATH,
                         input uvm_reg_map  map       = null,
                         input int          prior     = -1,
                         input uvm_object   extension = null,
                         input string       fname     = "",
```

```
                          input int          lineno    = 0                 )
```

Updates the given register *rg* using uvm_reg::update, supplying 'this' as the *parent* argument. Thus,

```
update_reg(model.regA, status, value);
```

is equivalent to

```
model.regA.update(status, value, .parent(this));
```

mirror_reg

```
virtual task mirror_reg( input uvm_reg      rg,
                        output uvm_status_e status,
                         input uvm_check_e  check     = UVM_NO_CHECK,
                         input uvm_path_e   path      = UVM_DEFAULT_PATH,
                         input uvm_reg_map  map       = null,
                         input int          prior     = -1,
                         input uvm_object   extension = null,
                         input string       fname     = "",
                         input int          lineno    = 0                 )
```

Mirrors the given register *rg* using uvm_reg::mirror, supplying 'this' as the *parent* argument. Thus,

```
mirror_reg(model.regA, status, UVM_CHECK);
```

is equivalent to

```
model.regA.mirror(status, UVM_CHECK, .parent(this));
```

write_mem

```
virtual task write_mem( input uvm_mem        mem,
                       output uvm_status_e   status,
                         input uvm_reg_addr_t offset,
                         input uvm_reg_data_t value,
                         input uvm_path_e    path      = UVM_DEFAULT_PATH,
                         input uvm_reg_map   map       = null,
                         input int           prior     = -1,
                         input uvm_object    extension = null,
                         input string        fname     = "",
                         input int           lineno    = 0                 )
```

Writes the given memory *mem* using uvm_mem::write, supplying 'this' as the *parent* argument. Thus,

```
write_mem(model.regA, status, offset, value);
```

is equivalent to

```
model.regA.write(status, offset, value, .parent(this));
```

read_mem

```
virtual task read_mem( input uvm_mem         mem,
                      output uvm_status_e    status,
                       input uvm_reg_addr_t  offset,
                      output uvm_reg_data_t  value,
                       input uvm_path_e      path      = UVM_DEFAULT_PATH,
                       input uvm_reg_map     map       = null,
                       input int             prior     = -1,
                       input uvm_object      extension = null,
                       input string          fname     = "",
                       input int             lineno    = 0               )
```

Reads the given memory *mem* using uvm_mem::read, supplying 'this' as the *parent* argument. Thus,

```
read_mem(model.regA, status, offset, value);
```

is equivalent to

```
model.regA.read(status, offset, value, .parent(this));
```

poke_mem

```
virtual task poke_mem( input uvm_mem         mem,
                      output uvm_status_e    status,
                       input uvm_reg_addr_t  offset,
                       input uvm_reg_data_t  value,
                       input string          kind      = "",
                       input uvm_object      extension = null,
                       input string          fname     = "",
                       input int             lineno    = 0    )
```

Pokes the given memory *mem* using uvm_mem::poke, supplying 'this' as the *parent* argument. Thus,

```
poke_mem(model.regA, status, offset, value);
```

is equivalent to

```
model.regA.poke(status, offset, value, .parent(this));
```

peek_mem

```
virtual task peek_mem( input uvm_mem         mem,
                      output uvm_status_e    status,
                       input uvm_reg_addr_t  offset,
                      output uvm_reg_data_t  value,
                       input string          kind      = "",
                       input uvm_object      extension = null,
                       input string          fname     = "",
                       input int             lineno    = 0    )
```

Peeks the given memory *mem* using uvm_mem::peek, supplying 'this' as the *parent* argument. Thus,

```
peek_mem(model.regA, status, offset, value);
```

is equivalent to

```
model.regA.peek(status, offset, value, .parent(this));
```

uvm_reg_frontdoor

Facade class for register and memory frontdoor access.

User-defined frontdoor access sequence

Base class for user-defined access to register and memory reads and writes through a physical interface.

By default, different registers and memories are mapped to different addresses in the address space and are accessed via those exclusively through physical addresses.

The frontdoor allows access using a non-linear and/or non-mapped mechanism. Users can extend this class to provide the physical access to these registers.

Summary

uvm_reg_frontdoor

Facade class for register and memory frontdoor access.

Class Hierarchy

uvm_reg_sequence#(uvm_sequence#(uvm_sequence_item))

uvm_reg_frontdoor

Class Declaration

```
virtual class uvm_reg_frontdoor extends uvm_reg_sequence
#(
   uvm_sequence #(uvm_sequence_item)
)
```

Variables

rw_info	Holds information about the register being read or written
sequencer	Sequencer executing the operation

METHODS

new	Constructor, new object givne optional *name*.

VARIABLES

rw_info

```
uvm_reg_item rw_info
```

Holds information about the register being read or written

sequencer

```
uvm_sequencer_base sequencer
```

Sequencer executing the operation

METHODS

new

```
function new(string name = "")
```

Constructor, new object givne optional *name*.

uvm_reg_predictor

Updates the register model mirror based on observed bus transactions

This class converts observed bus transactions of type *BUSTYPE* to generic registers transactions, determines the register being accessed based on the bus address, then updates the register's mirror value with the observed bus data, subject to the register's access mode. See uvm_reg::predict for details.

Memories can be large, so their accesses are not predicted. Users can periodically use backdoor peek/poke to update the memory mirror.

Summary

uvm_reg_predictor

Updates the register model mirror based on observed bus transactions

CLASS HIERARCHY

uvm_void

uvm_object

uvm_report_object

uvm_component

uvm_reg_predictor

Class Declaration

```
class uvm_reg_predictor #(
   type BUSTYPE = int
) extends uvm_component
```

Variables

bus_in	Observed bus transactions of type *BUSTYPE* are received from this port and processed.
reg_ap	Analysis output port that publishes uvm_reg_item transactions converted from bus transactions received on *bus_in*.
map	The map used to convert a bus address to the corresponding register or memory handle.
adapter	The adapter used to convey the parameters of a bus operation in terms of a canonical uvm_reg_bus_op datum.

Methods

new	Create a new instance of this type, giving it the optional *name* and *parent*.
pre_predict	Override this method to change the value or re-direct the target register
check_phase	Checks that no pending register transactions are still enqueued.

Variables

bus_in

```
uvm_analysis_imp #(                   BUSTYPE,
                   uvm_reg_predictor #(BUSTYPE)) bus_in
```

Observed bus transactions of type *BUSTYPE* are received from this port and processed.

For each incoming transaction, the predictor will attempt to get the register or memory handle corresponding to the observed bus address.

If there is a match, the predictor calls the register or memory's predict method, passing in the observed bus data. The register or memory mirror will be updated with this data, subject to its configured access behavior--RW, RO, WO, etc. The predictor will also convert the bus transaction to a generic uvm_reg_item and send it out the *reg_ap* analysis port.

If the register is wider than the bus, the predictor will collect the multiple bus transactions needed to determine the value being read or written.

reg_ap

```
uvm_analysis_port #(uvm_reg_item) reg_ap
```

Analysis output port that publishes uvm_reg_item transactions converted from bus transactions received on *bus_in*.

map

```
uvm_reg_map map
```

The map used to convert a bus address to the corresponding register or memory handle. Must be configured before the run phase.

adapter

```
uvm_reg_adapter adapter
```

The adapter used to convey the parameters of a bus operation in terms of a canonical uvm_reg_bus_op datum. The *adapter* must be configured before the run phase.

METHODS

new

```
function new (string        name,
              uvm_component parent)
```

Create a new instance of this type, giving it the optional *name* and *parent*.

pre_predict

```
virtual function void pre_predict(uvm_reg_item rw)
```

Override this method to change the value or re-direct the target register

check_phase

```
virtual function void check_phase(uvm_phase phase)
```

Checks that no pending register transactions are still enqueued.

uvm_reg_backdoor

Base class for user-defined back-door register and memory access.

This class can be extended by users to provide user-specific back-door access to registers and memories that are not implemented in pure SystemVerilog or that are not accessible using the default DPI backdoor mechanism.

Summary

uvm_reg_backdoor

Base class for user-defined back-door register and memory access.

Class Hierarchy

uvm_void

uvm_object

uvm_reg_backdoor

Class Declaration

```
class uvm_reg_backdoor extends uvm_object
```

Methods

new	Create an instance of this class
do_pre_read	Execute the pre-read callbacks
do_post_read	Execute the post-read callbacks
do_pre_write	Execute the pre-write callbacks
do_post_write	Execute the post-write callbacks
write	User-defined backdoor write operation.
read	User-defined backdoor read operation.
read_func	User-defined backdoor read operation.
is_auto_updated	Indicates if wait_for_change() method is implemented
wait_for_change	Wait for a change in the value of the register or memory element in the DUT.
pre_read	Called before user-defined backdoor register read.
post_read	Called after user-defined backdoor register read.
pre_write	Called before user-defined backdoor register write.
post_write	Called after user-defined backdoor register write.

Methods

new

```
function new(string name = "")
```

Create an instance of this class

Create an instance of the user-defined backdoor class for the specified register or memory

do_pre_read

```
protected task do_pre_read(uvm_reg_item rw)
```

Execute the pre-read callbacks

This method *must* be called as the first statement in a user extension of the read() method.

do_post_read

```
protected task do_post_read(uvm_reg_item rw)
```

Execute the post-read callbacks

This method *must* be called as the last statement in a user extension of the read() method.

do_pre_write

```
protected task do_pre_write(uvm_reg_item rw)
```

Execute the pre-write callbacks

This method *must* be called as the first statement in a user extension of the write() method.

do_post_write

```
protected task do_post_write(uvm_reg_item rw)
```

Execute the post-write callbacks

This method *must* be called as the last statement in a user extension of the write() method.

write

```
virtual task write(uvm_reg_item rw)
```

User-defined backdoor write operation.

Call do_pre_write(). Deposit the specified value in the specified register HDL implementation. Call do_post_write(). Returns an indication of the success of the operation.

read

```
virtual task read(uvm_reg_item rw)
```

User-defined backdoor read operation.

Overload this method only if the backdoor requires the use of task.

Call do_pre_read(). Peek the current value of the specified HDL implementation. Call do_post_read(). Returns the current value and an indication of the success of the

operation.

By default, calls read_func().

read_func

```
virtual function void read_func(uvm_reg_item rw)
```

User-defined backdoor read operation.

Peek the current value in the HDL implementation. Returns the current value and an indication of the success of the operation.

is_auto_updated

```
virtual function bit is_auto_updated(uvm_reg_field field)
```

Indicates if wait_for_change() method is implemented

Implement to return TRUE if and only if wait_for_change() is implemented to watch for changes in the HDL implementation of the specified field

wait_for_change

```
virtual local task wait_for_change(uvm_object element)
```

Wait for a change in the value of the register or memory element in the DUT.

When this method returns, the mirror value for the register corresponding to this instance of the backdoor class will be updated via a backdoor read operation.

pre_read

```
virtual task pre_read(uvm_reg_item rw)
```

Called before user-defined backdoor register read.

The registered callback methods are invoked after the invocation of this method.

post_read

```
virtual task post_read(uvm_reg_item rw)
```

Called after user-defined backdoor register read.

The registered callback methods are invoked before the invocation of this method.

pre_write

```
virtual task pre_write(uvm_reg_item rw)
```

Called before user-defined backdoor register write.

The registered callback methods are invoked after the invocation of this method.

The written value, if modified, modifies the actual value that will be written.

post_write

```
virtual task post_write(uvm_reg_item rw)
```

Called after user-defined backdoor register write.

The registered callback methods are invoked before the invocation of this method.

UVM HDL Backdoor Access support routines.

These routines provide an interface to the DPI/PLI implementation of backdoor access used by registers.

If you DON'T want to use the DPI HDL API, then compile your SystemVerilog code with the vlog switch

```
vlog ... +define+UVM_HDL_NO_DPI ...
```

Summary

UVM HDL Backdoor Access support routines.

These routines provide an interface to the DPI/PLI implementation of backdoor access used by registers.

VARIABLES

UVM_HDL_MAX_WIDTH	Sets the maximum size bit vector for backdoor access.

METHODS

uvm_hdl_check_path	Checks that the given HDL *path* exists.
uvm_hdl_deposit	Sets the given HDL *path* to the specified *value*.
uvm_hdl_force	Forces the *value* on the given *path*.
uvm_hdl_force_time	Forces the *value* on the given *path* for the specified amount of *force_time*.
uvm_hdl_release_and_read	Releases a value previously set with uvm_hdl_force.
uvm_hdl_release	Releases a value previously set with uvm_hdl_force.
uvm_hdl_read()	Gets the value at the given *path*.

VARIABLES

UVM_HDL_MAX_WIDTH

```
parameter int UVM_HDL_MAX_WIDTH = `UVM_HDL_MAX_WIDTH
```

Sets the maximum size bit vector for backdoor access. This parameter will be looked up by the DPI-C code using: vpi_handle_by_name("uvm_pkg::UVM_HDL_MAX_WIDTH", 0);

METHODS

uvm_hdl_check_path

```
import "DPI-C" function int uvm_hdl_check_path(string path)
```

Checks that the given HDL *path* exists. Returns 0 if NOT found, 1 otherwise.

uvm_hdl_deposit

```
import "DPI-C" function int uvm_hdl_deposit(string         path,
                                            uvm_hdl_data_t value)
```

Sets the given HDL *path* to the specified *value*. Returns 1 if the call succeeded, 0 otherwise.

uvm_hdl_force

```
import "DPI-C" function int uvm_hdl_force(string         path,
                                          uvm_hdl_data_t value)
```

Forces the *value* on the given *path*. Returns 1 if the call succeeded, 0 otherwise.

uvm_hdl_force_time

```
task uvm_hdl_force_time(string         path,
                        uvm_hdl_data_t value,
                        time           force_time = )
```

Forces the *value* on the given *path* for the specified amount of *force_time*. If *force_time* is 0, uvm_hdl_deposit is called. Returns 1 if the call succeeded, 0 otherwise.

uvm_hdl_release_and_read

```
import "DPI-C" function int uvm_hdl_release_and_read(
          string         path,
   inout  uvm_hdl_data_t value
)
```

Releases a value previously set with uvm_hdl_force. Returns 1 if the call succeeded, 0 otherwise. *value* is set to the HDL value after the release. For 'reg', the value will still be the forced value until it has bee procedurally reassigned. For 'wire', the value will change immediately to the resolved value of its continuous drivers, if any. If none, its value remains as forced until the next direct assignment.

uvm_hdl_release

```
import "DPI-C" function int uvm_hdl_release(string path)
```

Releases a value previously set with uvm_hdl_force. Returns 1 if the call succeeded, 0 otherwise.

uvm_hdl_read()

```
import "DPI-C" function int uvm_hdl_read(       string         path,
                                         output uvm_hdl_data_t value)
```

Gets the value at the given *path*. Returns 1 if the call succeeded, 0 otherwise.

uvm_reg_mem_built_in_seq

Sequence that executes a user-defined selection of pre-defined register and memory test sequences.

Summary

uvm_reg_mem_built_in_seq

Sequence that executes a user-defined selection of pre-defined register and memory test sequences.

Class Hierarchy

uvm_reg_sequence#(uvm_sequence#(uvm_reg_item))

uvm_reg_mem_built_in_seq

Class Declaration

```
class uvm_reg_mem_built_in_seq extends uvm_reg_sequence
#(
   uvm_sequence #(uvm_reg_item)
)
```

Variables

model	The block to be tested.
tests	The pre-defined test sequences to be executed.

Methods

body	Executes any or all the built-in register and memory sequences.

Variables

model

The block to be tested. Declared in the base class.

```
uvm_reg_block model;
```

tests

```
bit [63:0] tests = UVM_DO_ALL_REG_MEM_TESTS
```

The pre-defined test sequences to be executed.

Methods

body

```
virtual task body()
```

Executes any or all the built-in register and memory sequences. Do not call directly.
Use seq.start() instead.

uvm_reg_hw_reset_seq

Test the hard reset values of registers

The test sequence performs the following steps

1. resets the DUT and the block abstraction class associated with this sequence.

2. reads all of the registers in the block, via all of the available address maps, comparing the value read with the expected reset value.

If bit-type resource named "NO_REG_TESTS" or "NO_REG_HW_RESET_TEST" in the "REG::" namespace matches the full name of the block or register, the block or register is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.get_full_name(),".*"},
                           "NO_REG_TESTS", 1, this);
```

This is usually the first test executed on any DUT.

Summary

uvm_reg_hw_reset_seq

Test the hard reset values of registers

Class Hierarchy

uvm_reg_sequence#(uvm_sequence#(uvm_reg_item))

uvm_reg_hw_reset_seq

Class Declaration

```
class uvm_reg_hw_reset_seq extends uvm_reg_sequence #(
   uvm_sequence #(uvm_reg_item)
)
```

Variables

model	The block to be tested.
body	Executes the Hardware Reset sequence.

Methods

reset_blk	Reset the DUT that corresponds to the specified block abstraction class.

Variables

model

The block to be tested. Declared in the base class.

```
uvm_reg_block model;
```

body

```
virtual task body()
```

Executes the Hardware Reset sequence. Do not call directly. Use seq.start() instead.

METHODS

reset_blk

```
virtual task reset_blk(uvm_reg_block blk)
```

Reset the DUT that corresponds to the specified block abstraction class.

Currently empty. Will rollback the environment's phase to the *reset* phase once the new phasing is available.

In the meantime, the DUT should be reset before executing this test sequence or this method should be implemented in an extension to reset the DUT.

Bit Bashing Test Sequences

This section defines classes that test individual bits of the registers defined in a register model.

Contents

uvm_reg_single_bit_bash_seq

Verify the implementation of a single register by attempting to write 1's and 0's to every bit in it, via every address map in which the register is mapped, making sure that the resulting value matches the mirrored value.

If bit-type resource named "NO_REG_TESTS" or "NO_REG_BIT_BASH_TEST" in the "REG::" namespace matches the full name of the register, the register is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.r0.get_full_name()},
                           "NO_REG_TESTS", 1, this);
```

Registers that contain fields with unknown access policies cannot be tested.

The DUT should be idle and not modify any register durign this test.

Summary

uvm_reg_single_bit_bash_seq

Verify the implementation of a single register by attempting to write 1's and 0's to every bit in it, via every address map in which the register is mapped, making sure that the resulting value matches the mirrored value.

Class Hierarchy

```
uvm_reg_sequence#(uvm_sequence#(uvm_reg_item))
  uvm_reg_single_bit_bash_seq
```

Class Declaration

```
class uvm_reg_single_bit_bash_seq extends
uvm_reg_sequence #(
   uvm_sequence #(uvm_reg_item)
)
```

VARIABLES

rg	The register to be tested

VARIABLES

rg

```
uvm_reg rg
```

The register to be tested

uvm_reg_bit_bash_seq

Verify the implementation of all registers in a block by executing the uvm_reg_single_bit_bash_seq sequence on it.

If bit-type resource named "NO_REG_TESTS" or "NO_REG_BIT_BASH_TEST" in the "REG::" namespace matches the full name of the block, the block is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.get_full_name(),".*"},
                           "NO_REG_TESTS", 1, this);
```

Summary

uvm_reg_bit_bash_seq

Verify the implementation of all registers in a block by executing the uvm_reg_single_bit_bash_seq sequence on it.

CLASS HIERARCHY

```
uvm_reg_sequence#(uvm_sequence#(uvm_reg_item))
  uvm_reg_bit_bash_seq
```

CLASS DECLARATION

```
class uvm_reg_bit_bash_seq extends uvm_reg_sequence #(
   uvm_sequence #(uvm_reg_item)
)
```

VARIABLES

model	The block to be tested.
reg_seq	The sequence used to test one register

METHODS

body	Executes the Register Bit Bash sequence.
do_block	Test all of the registers in a a given *block*
reset_blk	Reset the DUT that corresponds to the specified block abstraction class.

VARIABLES

model

The block to be tested. Declared in the base class.

```
uvm_reg_block model;
```

reg_seq

```
protected uvm_reg_single_bit_bash_seq reg_seq
```

The sequence used to test one register

METHODS

body

```
virtual task body()
```

Executes the Register Bit Bash sequence. Do not call directly. Use seq.start() instead.

do_block

```
protected virtual task do_block(uvm_reg_block blk)
```

Test all of the registers in a a given *block*

reset_blk

```
virtual task reset_blk(uvm_reg_block blk)
```

Reset the DUT that corresponds to the specified block abstraction class.

Currently empty. Will rollback the environment's phase to the *reset* phase once the new phasing is available.

In the meantime, the DUT should be reset before executing this test sequence or this method should be implemented in an extension to reset the DUT.

Register Access Test Sequences

This section defines sequences that test DUT register access via the available frontdoor and backdoor paths defined in the provided register model.

Contents

uvm_reg_single_access_seq

Verify the accessibility of a register by writing through its default address map then reading it via the backdoor, then reversing the process, making sure that the resulting value matches the mirrored value.

If bit-type resource named “NO_REG_TESTS” or “NO_REG_ACCESS_TEST” in the “REG::” namespace matches the full name of the register, the register is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.r0.get_full_name()},
                           "NO_REG_TESTS", 1, this);
```

Registers without an available backdoor or that contain read-only fields only, or fields with unknown access policies cannot be tested.

The DUT should be idle and not modify any register during this test.

Summary

uvm_reg_single_access_seq

Verify the accessibility of a register by writing through its default address map then reading it via the backdoor, then reversing the process, making sure that the resulting value matches the mirrored value.

Class Hierarchy

uvm_reg_sequence#(uvm_sequence#(uvm_reg_item))

uvm_reg_single_access_seq

CLASS DECLARATION

```
class uvm_reg_single_access_seq extends uvm_reg_sequence
#(
   uvm_sequence #(uvm_reg_item)
)
```

VARIABLES

rg	The register to be tested

VARIABLES

rg

```
uvm_reg rg
```

The register to be tested

uvm_reg_access_seq

Verify the accessibility of all registers in a block by executing the uvm_reg_single_access_seq sequence on every register within it.

If bit-type resource named "NO_REG_TESTS" or "NO_REG_ACCESS_TEST" in the "REG::" namespace matches the full name of the block, the block is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.get_full_name(),".*"},
                           "NO_REG_TESTS", 1, this);
```

Summary

uvm_reg_access_seq

Verify the accessibility of all registers in a block by executing the uvm_reg_single_access_seq sequence on every register within it.

CLASS HIERARCHY

uvm_reg_sequence#(uvm_sequence#(uvm_reg_item))

uvm_reg_access_seq

CLASS DECLARATION

```
class uvm_reg_access_seq extends uvm_reg_sequence #(
   uvm_sequence #(uvm_reg_item)
)
```

VARIABLES

model	The block to be tested.
reg_seq	The sequence used to test one register

METHODS

body	Executes the Register Access sequence.
do_block	Test all of the registers in a block
reset_blk	Reset the DUT that corresponds to the specified block abstraction class.

VARIABLES

model

The block to be tested. Declared in the base class.

```
uvm_reg_block model;
```

reg_seq

```
protected uvm_reg_single_access_seq reg_seq
```

The sequence used to test one register

METHODS

body

```
virtual task body()
```

Executes the Register Access sequence. Do not call directly. Use seq.start() instead.

do_block

```
protected virtual task do_block(uvm_reg_block blk)
```

Test all of the registers in a block

reset_blk

```
virtual task reset_blk(uvm_reg_block blk)
```

Reset the DUT that corresponds to the specified block abstraction class.

Currently empty. Will rollback the environment's phase to the *reset* phase once the new phasing is available.

In the meantime, the DUT should be reset before executing this test sequence or this method should be implemented in an extension to reset the DUT.

uvm_reg_mem_access_seq

Verify the accessibility of all registers and memories in a block by executing the uvm_reg_access_seq and uvm_mem_access_seq sequence respectively on every register and memory within it.

Blocks and registers with the NO_REG_TESTS or the NO_REG_ACCESS_TEST attribute are not verified.

Summary

uvm_reg_mem_access_seq

Verify the accessibility of all registers and memories in a block by executing the uvm_reg_access_seq and uvm_mem_access_seq sequence respectively on every register and memory within it.

Class Hierarchy

uvm_reg_sequence#(uvm_sequence#(uvm_reg_item))

uvm_reg_mem_access_seq

Class Declaration

```
class uvm_reg_mem_access_seq extends uvm_reg_sequence #(
   uvm_sequence #(uvm_reg_item)
)
```

Shared Register and Memory Access Test Sequences

This section defines sequences for testing registers and memories that are shared between two or more physical interfaces, i.e. are associated with more than one uvm_reg_map instance.

Contents

uvm_reg_shared_access_seq

Verify the accessibility of a shared register by writing through each address map then reading it via every other address maps in which the register is readable and the backdoor, making sure that the resulting value matches the mirrored value.

If bit-type resource named "NO_REG_TESTS" or "NO_REG_SHARED_ACCESS_TEST" in the "REG::" namespace matches the full name of the register, the register is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.r0.get_full_name()},
                           "NO_REG_TESTS", 1, this);
```

Registers that contain fields with unknown access policies cannot be tested.

The DUT should be idle and not modify any register during this test.

Summary

uvm_reg_shared_access_seq

Verify the accessibility of a shared register by writing through each address map

then reading it via every other address maps in which the register is readable and the backdoor, making sure that the resulting value matches the mirrored value.

Class Hierarchy

uvm_reg_sequence#(uvm_sequence#(uvm_reg_item))

uvm_reg_shared_access_seq

Class Declaration

```
class uvm_reg_shared_access_seq extends uvm_reg_sequence
#(
   uvm_sequence #(uvm_reg_item)
)
```

Variables

rg	The register to be tested

Variables

rg

```
uvm_reg rg
```

The register to be tested

uvm_mem_shared_access_seq

Verify the accessibility of a shared memory by writing through each address map then reading it via every other address maps in which the memory is readable and the backdoor, making sure that the resulting value matches the written value.

If bit-type resource named “NO_REG_TESTS”, “NO_MEM_TESTS”, “NO_REG_SHARED_ACCESS_TEST” or “NO_MEM_SHARED_ACCESS_TEST” in the “REG::” namespace matches the full name of the memory, the memory is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.mem0.get_full_name()},
                           "NO_MEM_TESTS", 1, this);
```

The DUT should be idle and not modify the memory during this test.

Summary

uvm_mem_shared_access_seq

Verify the accessibility of a shared memory by writing through each address map then reading it via every other address maps in which the memory is readable and the backdoor, making sure that the resulting value matches the written value.

Class Hierarchy

uvm_reg_sequence#(uvm_sequence#(uvm_reg_item))

uvm_mem_shared_access_seq

Class Declaration

```
class uvm_mem_shared_access_seq extends uvm_reg_sequence
#(
   uvm_sequence #(uvm_reg_item)
)
```

Variables

mem	The memory to be tested

Variables

mem

```
uvm_mem mem
```

The memory to be tested

uvm_reg_mem_shared_access_seq

Verify the accessibility of all shared registers and memories in a block by executing the uvm_reg_shared_access_seq and uvm_mem_shared_access_seq sequence respectively on every register and memory within it.

If bit-type resource named “NO_REG_TESTS”, “NO_MEM_TESTS”, “NO_REG_SHARED_ACCESS_TEST” or “NO_MEM_SHARED_ACCESS_TEST” in the “REG::” namespace matches the full name of the block, the block is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.get_full_name(),".*"},
                           "NO_REG_TESTS", 1, this);
```

Summary

uvm_reg_mem_shared_access_seq

Verify the accessibility of all shared registers and memories in a block by executing the uvm_reg_shared_access_seq and uvm_mem_shared_access_seq sequence respectively on every register and memory within it.

Class Hierarchy

```
uvm_reg_sequence#(uvm_sequence#(uvm_reg_item))
```

uvm_reg_mem_shared_access_seq

Class Declaration

```
class uvm_reg_mem_shared_access_seq extends
uvm_reg_sequence #(
   uvm_sequence #(uvm_reg_item)
)
```

VARIABLES

model

The block to be tested

```
uvm_reg_block model;
```

reg_seq

```
protected uvm_reg_shared_access_seq reg_seq
```

The sequence used to test one register

mem_seq

```
protected uvm_mem_shared_access_seq mem_seq
```

The sequence used to test one memory

METHODS

body

```
virtual task body()
```

Executes the Shared Register and Memory sequence

do_block

```
protected virtual task do_block(uvm_reg_block blk)
```

Test all of the registers and memories in a block

reset_blk

```
virtual task reset_blk(uvm_reg_block blk)
```

Reset the DUT that corresponds to the specified block abstraction class.

Currently empty. Will rollback the environment's phase to the *reset* phase once the new phasing is available.

In the meantime, the DUT should be reset before executing this test sequence or this method should be implemented in an extension to reset the DUT.

Memory Access Test Sequence

Contents

uvm_mem_single_access_seq

Verify the accessibility of a memory by writing through its default address map then reading it via the backdoor, then reversing the process, making sure that the resulting value matches the written value.

If bit-type resource named "NO_REG_TESTS", "NO_MEM_TESTS", or "NO_MEM_ACCESS_TEST" in the "REG::" namespace matches the full name of the memory, the memory is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.mem0.get_full_name()},
                           "NO_MEM_TESTS", 1, this);
```

Memories without an available backdoor cannot be tested.

The DUT should be idle and not modify the memory during this test.

Summary

uvm_mem_single_access_seq

Verify the accessibility of a memory by writing through its default address map then reading it via the backdoor, then reversing the process, making sure that the resulting value matches the written value.

Class Hierarchy

```
uvm_reg_sequence#(uvm_sequence#(uvm_reg_item))
  uvm_mem_single_access_seq
```

Class Declaration

```
class uvm_mem_single_access_seq extends uvm_reg_sequence
#(
   uvm_sequence #(uvm_reg_item)
)
```

Variables

mem	The memory to be tested

VARIABLES

mem

```
uvm_mem mem
```

The memory to be tested

uvm_mem_access_seq

Verify the accessibility of all memories in a block by executing the uvm_mem_single_access_seq sequence on every memory within it.

If bit-type resource named "NO_REG_TESTS", "NO_MEM_TESTS", or "NO_MEM_ACCESS_TEST" in the "REG::" namespace matches the full name of the block, the block is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.get_full_name(),".*"},
                           "NO_MEM_TESTS", 1, this);
```

Summary

uvm_mem_access_seq

Verify the accessibility of all memories in a block by executing the uvm_mem_single_access_seq sequence on every memory within it.

CLASS HIERARCHY

```
uvm_reg_sequence#(uvm_sequence#(uvm_reg_item))
```

uvm_mem_access_seq

CLASS DECLARATION

```
class uvm_mem_access_seq extends uvm_reg_sequence #(
   uvm_sequence #(uvm_reg_item)
)
```

VARIABLES

model	The block to be tested.
mem_seq	The sequence used to test one memory

METHODS

body	Execute the Memory Access sequence.
do_block	Test all of the memories in a given *block*
reset_blk	Reset the DUT that corresponds to the specified block abstraction class.

VARIABLES

model

The block to be tested. Declared in the base class.

```
uvm_reg_block model;
```

mem_seq

```
protected uvm_mem_single_access_seq mem_seq
```

The sequence used to test one memory

METHODS

body

```
virtual task body()
```

Execute the Memory Access sequence. Do not call directly. Use seq.start() instead.

do_block

```
protected virtual task do_block(uvm_reg_block blk)
```

Test all of the memories in a given *block*

reset_blk

```
virtual task reset_blk(uvm_reg_block blk)
```

Reset the DUT that corresponds to the specified block abstraction class.

Currently empty. Will rollback the environment's phase to the *reset* phase once the new phasing is available.

In the meantime, the DUT should be reset before executing this test sequence or this method should be implemented in an extension to reset the DUT.

Memory Walking-Ones Test Sequences

This section defines sequences for applying a "walking-ones" algorithm on one or more memories.

Contents

uvm_mem_single_walk_seq

Runs the walking-ones algorithm on the memory given by the mem property, which must be assigned prior to starting this sequence.

If bit-type resource named "NO_REG_TESTS", "NO_MEM_TESTS", or "NO_MEM_WALK_TEST" in the "REG::" namespace matches the full name of the memory, the memory is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.mem0.get_full_name()},
                           "NO_MEM_TESTS", 1, this);
```

The walking ones algorithm is performed for each map in which the memory is defined.

```
for (k = 0 thru memsize-1)
  write addr=k data=~k
  if (k > 0) {
    read addr=k-1, expect data=~(k-1)
    write addr=k-1 data=k-1
  if (k == last addr)
    read addr=k, expect data=~k
```

Summary

uvm_mem_single_walk_seq

Runs the walking-ones algorithm on the memory given by the mem property, which must be assigned prior to starting this sequence.

Class Hierarchy

uvm_reg_sequence#(uvm_sequence#(uvm_reg_item))

uvm_mem_single_walk_seq

Class Declaration

```
class uvm_mem_single_walk_seq extends uvm_reg_sequence #(
   uvm_sequence #(uvm_reg_item)
)
```

Variables

mem	The memory to test; must be assigned prior to starting sequence.

Methods

new	Creates a new instance of the class with the given name.
body	Performs the walking-ones algorithm on each map of the memory specifed in mem.

Variables

mem

```
uvm_mem mem
```

The memory to test; must be assigned prior to starting sequence.

Methods

new

```
function new(string name = "uvm_mem_walk_seq")
```

Creates a new instance of the class with the given name.

body

```
virtual task body()
```

Performs the walking-ones algorithm on each map of the memory specifed in mem.

uvm_mem_walk_seq

Verifies the all memories in a block by executing the uvm_mem_single_walk_seq sequence on every memory within it.

If bit-type resource named "NO_REG_TESTS", "NO_MEM_TESTS", or "NO_MEM_WALK_TEST" in the "REG::" namespace matches the full name of the block, the block is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.get_full_name(),".*"},
                           "NO_MEM_TESTS", 1, this);
```

Summary

uvm_mem_walk_seq

Verifies the all memories in a block by executing the uvm_mem_single_walk_seq sequence on every memory within it.

Class Hierarchy

uvm_reg_sequence#(uvm_sequence#(uvm_reg_item))

uvm_mem_walk_seq

Class Declaration

```
class uvm_mem_walk_seq extends uvm_reg_sequence #(
   uvm_sequence #(uvm_reg_item)
)
```

Variables

model	The block to be tested.
mem_seq	The sequence used to test one memory

Methods

body	Executes the mem walk sequence, one block at a time.
do_block	Test all of the memories in a given *block*
reset_blk	Reset the DUT that corresponds to the specified block abstraction class.

Variables

model

The block to be tested. Declared in the base class.

```
uvm_reg_block model;
```

mem_seq

```
protected uvm_mem_single_walk_seq mem_seq
```

The sequence used to test one memory

Methods

body

```
virtual task body()
```

Executes the mem walk sequence, one block at a time. Do not call directly. Use seq.start() instead.

do_block

```
protected virtual task do_block(uvm_reg_block blk)
```

Test all of the memories in a given *block*

reset_blk

```
virtual task reset_blk(uvm_reg_block blk)
```

Reset the DUT that corresponds to the specified block abstraction class.

Currently empty. Will rollback the environment's phase to the *reset* phase once the new phasing is available.

In the meantime, the DUT should be reset before executing this test sequence or this method should be implemented in an extension to reset the DUT.

HDL Paths Checking Test Sequence

Summary

uvm_reg_mem_hdl_paths_seq

Verify the correctness of HDL paths specified for registers and memories.

This sequence is be used to check that the specified backdoor paths are indeed accessible by the simulator. By default, the check is performed for the default design abstraction. If the simulation contains multiple models of the DUT, HDL paths for multiple design abstractions can be checked.

If a path is not accessible by the simulator, it cannot be used for read/write backdoor accesses. In that case a warning is produced. A simulator may have finer-grained access permissions such as separate read or write permissions. These extra access permissions are NOT checked.

The test is performed in zero time and does not require any reads/writes to/from the DUT.

Summary

uvm_reg_mem_hdl_paths_seq

Verify the correctness of HDL paths specified for registers and memories.

Class Hierarchy

uvm_reg_sequence#(uvm_sequence#(uvm_reg_item))

uvm_reg_mem_hdl_paths_seq

Class Declaration

```
class uvm_reg_mem_hdl_paths_seq extends uvm_reg_sequence
#(
   uvm_sequence #(uvm_reg_item)
)
```

Variables

abstractions	If set, check the HDL paths for the specified design abstractions.

Variables

abstractions

```
string abstractions[$]
```

If set, check the HDL paths for the specified design abstractions. If empty, check the HDL path for the default design abstraction, as specified with uvm_reg_block::set_default_hdl_path()

Command Line Processor Class

This class provides a general interface to the command line arguments that were provided for the given simulation. Users can retrieve the complete arguments using methods such as *get_args()* and *get_arg_matches()* but also retrieve the suffixes of arguments using *get_arg_values()*.

The uvm_cmdline_processor class also provides support for setting various UVM variables from the command line such as components' verbosities and configuration settings for integral types and strings. Command line arguments that are in uppercase should only have one setting to invocation. Command line arguments that in lowercase can have multiple settings per invocation.

All of these capablities are described in the uvm_cmdline_processor section.

Summary

Command Line Processor Class

This class provides a general interface to the command line arguments that were provided for the given simulation.

uvm_cmdline_processor

This class provides an interface to the command line arguments that were provided for the given simulation. The class is intended to be used as a singleton, but that isn't required. The generation of the data structures which hold the command line argument information happens during construction of the class object. A global variable called *uvm_cmdline_proc* is created at initialization time and may be used to access command line information.

The uvm_cmdline_processor class also provides support for setting various UVM variables from the command line such as components' verbosities and configuration settings for integral types and strings. Each of these capablities is described in the Built-in UVM Aware Command Line Arguments section.

Summary

uvm_cmdline_processor

This class provides an interface to the command line arguments that were provided for the given simulation.

Class Hierarchy

uvm_void
uvm_object
uvm_report_object
uvm_cmdline_processor

Class Declaration

```
class uvm_cmdline_processor extends uvm_report_object
```

Singleton

get_inst	Returns the singleton instance of the UVM command line processor.

Basic Arguments

get_args	This function returns a queue with all of the command line arguments that were used to start the simulation.
get_plusargs	This function returns a queue with all of the plus arguments that were used to start the simulation.
get_uvmargs	This function returns a queue with all of the uvm arguments that were used to start the simulation.
get_arg_matches	This function loads a queue with all of the arguments that match the input expression and returns the number of items that matched.

Argument Values

get_arg_value	This function finds the first argument which matches the *match* arg and returns the suffix of the argument.
get_arg_values	This function finds all the arguments which matches the *match* arg and returns the suffix of the arguments in a list of values.

Tool information

get_tool_name	Returns the simulation tool that is executing the simlation.
get_tool_version	Returns the version of the simulation tool that is executing the simlation.

Command Line Debug

+UVM_DUMP_CMDLINE_ARGS	*+UVM_DUMP_CMDLINE_ARGS* allows the user to dump all command line arguments to the reporting mechanism.

Built-in UVM Aware Command Line Arguments

+UVM_TESTNAME	*+UVM_TESTNAME=<class name>* allows the user to specify which uvm_test (or uvm_component) should be created via the factory and cycled through the UVM phases.

+UVM_VERBOSITY	*+UVM_VERBOSITY=<verbosity>* allows the user to specify the initial verbosity for all components.
+uvm_set_verbosity	*+uvm_set_verbosity=<comp>,<id>,<verbosity>,<phase>* and *+uvm_set_verbosity=<comp>,<id>,<verbosity>,time,<time>* allow the users to manipulate the verbosity of specific components at specific phases (and times during the "run" phases) of the simulation.
+uvm_set_action	*+uvm_set_action=<comp>,<id>,<severity>,<action>* provides the equivalent of various uvm_report_object's set_report_*_action APIs.
+uvm_set_severity	*+uvm_set_severity=<comp>,<id>,<current severity>,<new severity>* provides the equivalent of the various uvm_report_object's set_report_*_severity_override APIs.
+UVM_TIMEOUT	*+UVM_TIMEOUT=<timeout>,<overridable>* allows users to change the global timeout of the UVM framework.
+UVM_MAX_QUIT_COUNT	*+UVM_MAX_QUIT_COUNT=<count>,<overridable>* allows users to change max quit count for the report server.
+UVM_PHASE_TRACE	*+UVM_PHASE_TRACE* turns on tracing of phase executions.
+UVM_OBJECTION_TRACE	*+UVM_OBJECTION_TRACE* turns on tracing of objection activity.
+uvm_set_inst_override, +uvm_set_type_override	*+uvm_set_inst_override=<req_type>,<override_type>,<full_inst_path>* and *+uvm_set_type_override=<req_type>,<override_type>[,<replace>]* work like the name based overrides in the factory-- factory.set_inst_override_by_name() and factory.set_type_override_by_name().
+uvm_set_config_int, +uvm_set_config_string	*+uvm_set_config_int=<comp>,<field>,<value>* and *+uvm_set_config_string=<comp>,<field>,<value>* work like their procedural counterparts: set_config_int() and set_config_string().

Singleton

get_inst

```
static function uvm_cmdline_processor get_inst()
```

Returns the singleton instance of the UVM command line processor.

Basic Arguments

get_args

```
function void get_args (output string args[$])
```

This function returns a queue with all of the command line arguments that were used to start the simulation. Note that element 0 of the array will always be the name of the executable which started the simulation.

get_plusargs

```
function void get_plusargs (output string args[$])
```

This function returns a queue with all of the plus arguments that were used to start the simulation. Plusarguments may be used by the simulator vendor, or may be specific to a company or individiual user. Plusargs never have extra arguments (i.e. if there is a plusarg as the second argument on the command line, the third argument is unrelated); this is not necessarily the case with vendor specific dash arguments.

get_uvmargs

This function returns a queue with all of the uvm arguments that were used to start the simulation. An UVM argument is taken to be any argument that starts with a - or + and uses the keyword UVM (case insensitive) as the first three letters of the argument.

get_arg_matches

```
function int get_arg_matches (     string match,
                               ref string args[$])
```

This function loads a queue with all of the arguments that match the input expression and returns the number of items that matched. If the input expression is bracketed with //, then it is taken as an extended regular expression otherwise, it is taken as the beginning of an argument to match. For example:

```
string myargs[$]
initial begin
   void'(uvm_cmdline_proc.get_arg_matches("+foo",myargs)); //matches +foo,
+foobar
                                                           //doesn't
match +barfoo
   void'(uvm_cmdline_proc.get_arg_matches("/foo/",myargs)); //matches +foo,
+foobar,
                                                            //foo.sv,
barfoo, etc.
   void'(uvm_cmdline_proc.get_arg_matches("/^foo.*\.sv",myargs)); //matches
foo.sv
                                                                 //and
foo123.sv,
                                                                 //not
barfoo.sv.
```

Argument Values

get_arg_value

```
function int get_arg_value (       string match,
                            output string value )
```

This function finds the first argument which matches the *match* arg and returns the suffix of the argument. This is similar to the $value$plusargs system task, but does not take a formating string. The return value is the number of command line arguments that match the *match* string, and *value* is the value of the first match.

get_arg_values

```
function int get_arg_values (     string match,
                              ref string values[$])
```

This function finds all the arguments which matches the *match* arg and returns the suffix of the arguments in a list of values. The return value is the number of matches that were found (it is the same as values.size()). For example if '+foo=1,yes,on +foo=5,no,off' was provided on the command line and the following code was executed:

```
string foo_values[$]
initial begin
   void'(uvm_cmdline_proc.get_arg_values("+foo=",foo_values));
```

The foo_values queue would contain two entries. These entries are shown here:

0 "1,yes,on"

1 "5,no,off"

Splitting the resultant string is left to user but using the uvm_split_string() function is recommended.

Tool information

get_tool_name

```
function string get_tool_name ()
```

Returns the simulation tool that is executing the simlation. This is a vendor specific string.

get_tool_version

```
function string get_tool_version ()
```

Returns the version of the simulation tool that is executing the simlation. This is a vendor specific string.

Command Line Debug

+UVM_DUMP_CMDLINE_ARGS

+UVM_DUMP_CMDLINE_ARGS allows the user to dump all command line arguments to the reporting mechanism. The output in is tree format.

Built-in UVM Aware Command Line Arguments

+UVM_TESTNAME

+UVM_TESTNAME=<class name> allows the user to specify which uvm_test (or uvm_component) should be created via the factory and cycled through the UVM phases. If multiple of these settings are provided, the first occurrence is used and a warning is issued for subsequent settings. For example:

```
<sim command> +UVM_TESTNAME=read_modify_write_test
```

+UVM_VERBOSITY

+UVM_VERBOSITY=<verbosity> allows the user to specify the initial verbosity for all components. If multiple of these settings are provided, the first occurrence is used and a warning is issued for subsequent settings. For example:

```
<sim command> +UVM_VERBOSITY=UVM_HIGH
```

+uvm_set_verbosity

+uvm_set_verbosity=<comp>,<id>,<verbosity>,<phase> and *+uvm_set_verbosity=<comp>,<id>,<verbosity>,time,<time>* allow the users to manipulate the verbosity of specific components at specific phases (and times during the "run" phases) of the simulation. The *id* argument can be either *ALL* for all IDs or a specific message id. Wildcarding is not supported for *id* due to performance concerns. Settings for non-"run" phases are executed in order of occurrence on the command line. Settings for "run" phases (times) are sorted by time and then executed in order of occurrence for settings of the same time. For example:

```
<sim command>
+uvm_set_verbosity=uvm_test_top.env0.agent1.*,_ALL_,UVM_FULL,time,800
```

+uvm_set_action

+uvm_set_action=<comp>,<id>,<severity>,<action> provides the equivalent of various uvm_report_object's set_report_*_action APIs. The special keyword, *ALL*, can be provided for both/either the *id* and/or *severity* arguments. The action can be UVM_NO_ACTION or a | separated list of the other UVM message actions. For example:

```
<sim command>
+uvm_set_action=uvm_test_top.env0.*,_ALL_,UVM_ERROR,UVM_NO_ACTION
```

+uvm_set_severity

+uvm_set_severity=<comp>,<id>,<current severity>,<new severity> provides the equivalent of the various uvm_report_object's set_report_*_severity_override APIs. The special keyword, *ALL*, can be provided for both/either the *id* and/or *current severity* arguments. For example:

```
<sim command>
+uvm_set_severity=uvm_test_top.env0.*,BAD_CRC,UVM_ERROR,UVM_WARNING
```

+UVM_TIMEOUT

+UVM_TIMEOUT=<timeout>,<overridable> allows users to change the global timeout of the UVM framework. The <overridable> argument ('YES' or 'NO') specifies whether user code can subsequently change this value. If set to 'NO' and the user code tries to change the global timeout value, an warning message will be generated.

```
<sim command> +uvm_timeout=200000,NO
```

+UVM_MAX_QUIT_COUNT

+UVM_MAX_QUIT_COUNT=<count>,<overridable> allows users to change max quit count for the report server. The <overridable> argument ('0' or '1') specifies whether user code can subsequently change this value. If set to '0' and the user code tries to change the max quit count value, an warning message will be generated.

```
<sim command> +UVM_MAX_QUIT_COUNT=5,0
```

+UVM_PHASE_TRACE

+UVM_PHASE_TRACE turns on tracing of phase executions. Users simply need to put the argument on the command line.

+UVM_OBJECTION_TRACE

+UVM_OBJECTION_TRACE turns on tracing of objection activity. Users simply need to put the argument on the command line.

+uvm_set_inst_override, +uvm_set_type_override

+uvm_set_inst_override=<req_type>,<override_type>,<full_inst_path> and *+uvm_set_type_override=<req_type>,<override_type>[,<replace>]* work like the name based overrides in the factory--factory.set_inst_override_by_name() and factory.set_type_override_by_name(). For uvm_set_type_override, the third argument is 0 or 1 (the default is 1 if this argument is left off); this argument specifies whether previous type overrides for the type should be replaced. For example:

```
<sim command> +uvm_set_type_override=eth_packet,short_eth_packet
```

+uvm_set_config_int, +uvm_set_config_string

+uvm_set_config_int=<comp>,<field>,<value> and *+uvm_set_config_string=<comp>,<field>,<value>* work like their procedural counterparts: set_config_int() and set_config_string(). For the value of int config

settings, `b (0b), `o, `d, `h (`x or 0x) as the first two characters of the value are treated as base specifiers for interpreting the base of the number. Size specifiers are not used since SystemVerilog does not allow size specifiers in string to value conversions. For example:

```
<sim command> +uvm_set_config_int=uvm_test_top.soc_env,mode,5
```

No equivalent of set_config_object() exists since no way exists to pass an uvm_object into the simulation via the command line.

Types and Enumerations

Summary

Types and Enumerations

Field automation	
`UVM_MAX_STREAMBITS	Defines the maximum bit vector size for integral types.
uvm_bitstream_t	The bitstream type is used as a argument type for passing integral values in such methods as set_int_local, get_int_local, get_config_int, report, pack and unpack.
uvm_radix_enum	Specifies the radix to print or record in.
uvm_recursion_policy_enum	Specifies the policy for copying objects.
uvm_active_passive_enum	Convenience value to define whether a component, usually an agent, is in "active" mode or "passive" mode.
`uvm_field_* macro flags	Defines what operations a given field should be involved in.
Reporting	
uvm_severity	Defines all possible values for report severity.
uvm_action	Defines all possible values for report actions.
uvm_verbosity	Defines standard verbosity levels for reports.
Port Type	
uvm_port_type_e	Specifies the type of port
Sequences	
uvm_sequencer_arb_mode	Specifies a sequencer's arbitration mode
uvm_sequence_state_enum	Defines current sequence state
uvm_sequence_lib_mode	Specifies the random selection mode of a sequence library
Phasing	
uvm_phase_type	This is an attribute of a uvm_phase object which defines the phase execution type.
uvm_phase_state	The set of possible states of a phase.
uvm_phase_transition	These are the phase state transition for callbacks which provide additional information that may be useful during callbacks
uvm_wait_op	Specifies the operand when using methods like uvm_phase::wait_for_state.
Objections	
uvm_objection_event	Enumerated the possible objection events one could wait on.
Default Policy Classes	Policy classes copying, comparing, packing, unpacking, and recording uvm_object-based objects.
uvm_default_table_printer	The table printer is a global object that can be used with uvm_object::do_print to get tabular style printing.
uvm_default_tree_printer	The tree printer is a global object that can be used with uvm_object::do_print to get multi-line tree style printing.
uvm_default_line_printer	The line printer is a global object that can be used with uvm_object::do_print to get single-line style printing.
uvm_default_printer	The default printer policy.
uvm_default_packer	The default packer policy.
uvm_default_comparer	The default compare policy.
uvm_default_recorder	The default recording policy.

FIELD AUTOMATION

`UVM_MAX_STREAMBITS

Defines the maximum bit vector size for integral types.

uvm_bitstream_t

The bitstream type is used as a argument type for passing integral values in such methods as set_int_local, get_int_local, get_config_int, report, pack and unpack.

uvm_radix_enum

Specifies the radix to print or record in.

UVM_BIN	Selects binary (%b) format
UVM_DEC	Selects decimal (%d) format
UVM_UNSIGNED	Selects unsigned decimal (%u) format
UVM_OCT	Selects octal (%o) format
UVM_HEX	Selects hexidecimal (%h) format
UVM_STRING	Selects string (%s) format
UVM_TIME	Selects time (%t) format
UVM_ENUM	Selects enumeration value (name) format

uvm_recursion_policy_enum

Specifies the policy for copying objects.

UVM_DEEP	Objects are deep copied (object must implement copy method)
UVM_SHALLOW	Objects are shallow copied using default SV copy.
UVM_REFERENCE	Only object handles are copied.

uvm_active_passive_enum

Convenience value to define whether a component, usually an agent, is in "active" mode or "passive" mode.

`uvm_field_* macro flags

Defines what operations a given field should be involved in. Bitwise OR all that apply.

UVM_DEFAULT	All field operations turned on
UVM_COPY	Field will participate in uvm_object::copy
UVM_COMPARE	Field will participate in uvm_object::compare

UVM_PRINT	Field will participate in uvm_object::print
UVM_RECORD	Field will participate in uvm_object::record
UVM_PACK	Field will participate in uvm_object::pack
UVM_NOCOPY	Field will not participate in uvm_object::copy
UVM_NOCOMPARE	Field will not participate in uvm_object::compare
UVM_NOPRINT	Field will not participate in uvm_object::print
UVM_NORECORD	Field will not participate in uvm_object::record
UVM_NOPACK	Field will not participate in uvm_object::pack
UVM_DEEP	Object field will be deep copied
UVM_SHALLOW	Object field will be shallow copied
UVM_REFERENCE	Object field will copied by reference
UVM_READONLY	Object field will NOT be automatically configured.

REPORTING

uvm_severity

Defines all possible values for report severity.

UVM_INFO	Informative messsage.
UVM_WARNING	Indicates a potential problem.
UVM_ERROR	Indicates a real problem. Simulation continues subject to the configured message action.
UVM_FATAL	Indicates a problem from which simulation can not recover. Simulation exits via $finish after a #0 delay.

uvm_action

Defines all possible values for report actions. Each report is configured to execute one or more actions, determined by the bitwise OR of any or all of the following enumeration constants.

UVM_NO_ACTION	No action is taken
UVM_DISPLAY	Sends the report to the standard output
UVM_LOG	Sends the report to the file(s) for this (severity,id) pair
UVM_COUNT	Counts the number of reports with the COUNT attribute. When this value reaches max_quit_count, the simulation terminates
UVM_EXIT	Terminates the simulation immediately.
UVM_CALL_HOOK	Callback the report hook methods
UVM_STOP	Causes *$stop* to be executed, putting the simulation into interactive mode.

uvm_verbosity

Defines standard verbosity levels for reports.

UVM_NONE	Report is always printed. Verbosity level setting can not disable it.
UVM_LOW	Report is issued if configured verbosity is set to UVM_LOW or above.
UVM_MEDIUM	Report is issued if configured verbosity is set to UVM_MEDIUM or above.
UVM_HIGH	Report is issued if configured verbosity is set to UVM_HIGH or above.
UVM_FULL	Report is issued if configured verbosity is set to UVM_FULL or above.

Port Type

uvm_port_type_e

Specifies the type of port

UVM_PORT	The port requires the interface that is its type parameter.
UVM_EXPORT	The port provides the interface that is its type parameter via a connection to some other export or implementation.
UVM_IMPLEMENTATION	The port provides the interface that is its type parameter, and it is bound to the component that implements the interface.

Sequences

uvm_sequencer_arb_mode

Specifies a sequencer's arbitration mode

SEQ_ARB_FIFO	Requests are granted in FIFO order (default)
SEQ_ARB_WEIGHTED	Requests are granted randomly by weight
SEQ_ARB_RANDOM	Requests are granted randomly
SEQ_ARB_STRICT_FIFO	Requests at highest priority granted in fifo order
SEQ_ARB_STRICT_RANDOM	Requests at highest priority granted in randomly
SEQ_ARB_USER	Arbitration is delegated to the user-defined function, user_priority_arbitration. That function will specify the next sequence to grant.

uvm_sequence_state_enum

Defines current sequence state

CREATED	The sequence has been allocated.
PRE_BODY	The sequence is started and the pre_body task is being

	executed.
BODY	The sequence is started and the body task is being executed.
POST_BODY	The sequence is started and the post_body task is being executed.
ENDED	The sequence has ended by the completion of the body task.
STOPPED	The sequence has been forcibly ended by issuing a kill() on the sequence.
FINISHED	The sequence is completely finished executing.

uvm_sequence_lib_mode

Specifies the random selection mode of a sequence library

UVM_SEQ_LIB_RAND	Random sequence selection
UVM_SEQ_LIB_RANDC	Random cyclic sequence selection
UVM_SEQ_LIB_ITEM	Emit only items, no sequence execution
UVM_SEQ_LIB_USER	Apply a user-defined random-selection algorithm

PHASING

uvm_phase_type

This is an attribute of a uvm_phase object which defines the phase execution type. Every phase we define has a type. It is used only for information, as the type behavior is captured in three derived classes uvm_task/topdown/bottomup_phase.

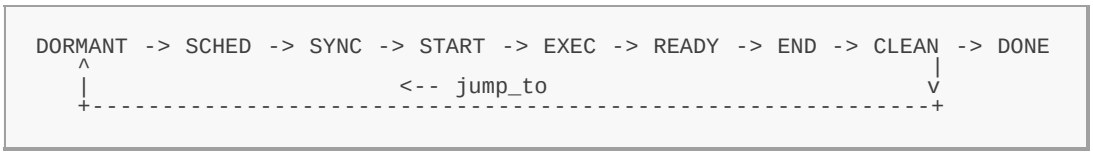
UVM_PHASE_TASK	The phase is a task-based phase, a fork is done for each participating component and so the traversal order is arbitrary
UVM_PHASE_TOPDOWN	The phase is a function phase, components are traversed from top-down, allowing them to add to the component tree as they go.
UVM_PHASE_BOTTOMUP	The phase is a function phase, components are traversed from the bottom up, allowing roll-up / consolidation functionality.
UVM_PHASE_SCHEDULE_NODE	The phase is not an imp, but a dummy phase graph node representing the beginning of a VIP schedule of phases.
UVM_PHASE_ENDSCHEDULE_NODE	The phase is not an imp, but a dummy phase graph node representing the end of a VIP schedule of phases
UVM_PHASE_DOMAIN_NODE	The phase is not an imp, but a dummy phase graph node representing an entire domain branch with schedules beneath

uvm_phase_state

The set of possible states of a phase. This is an attribute of a schedule node in the graph, not of a phase, to maintain independent per-domain state

UVM_PHASE_DORMANT	Nothing has happened with the phase in this domain.
UVM_PHASE_SCHEDULED	At least one immediate predecessor has completed. Scheduled phases block until all predecessors complete or until a jump is executed.
UVM_PHASE_SYNCING	All predecessors complete, checking that all synced phases (e.g. across domains) are at or beyond this point
UVM_PHASE_STARTED	phase ready to execute, running phase_started() callback
UVM_PHASE_EXECUTING	An executing phase is one where the phase callbacks are being executed. It's process is tracked by the phaser.
UVM_PHASE_READY_TO_END	no objections remain, awaiting completion of predecessors of its successors. For example, when phase 'run' is ready to end, its successor will be 'extract', whose predecessors are 'run' and 'post_shutdown'. Therefore, 'run' will be waiting for 'post_shutdown' to be ready to end.
UVM_PHASE_ENDED	phase completed execution, now running phase_ended() callback
UVM_PHASE_CLEANUP	all processes related to phase are being killed
UVM_PHASE_DONE	A phase is done after it terminated execution. Becoming done may enable a waiting successor phase to execute.

The state transitions occur as follows



uvm_phase_transition

These are the phase state transition for callbacks which provide additional information that may be useful during callbacks

UVM_COMPLETED	the phase completed normally
UVM_FORCED_STOP	the phase was forced to terminate prematurely
UVM_SKIPPED	the phase was in the path of a forward jump
UVM_RERUN	the phase was in the path of a backwards jump

uvm_wait_op

Specifies the operand when using methods like uvm_phase::wait_for_state.

UVM_EQ	equal
UVM_NE	not equal
UVM_LT	less than
UVM_LTE	less than or equal to
UVM_GT	greater than
UVM_GTE	greater than or equal to

Objections

uvm_objection_event

Enumerated the possible objection events one could wait on. See uvm_objection::wait_for.

UVM_RAISED	an objection was raised
UVM_DROPPED	an objection was raised
UVM_ALL_DROPPED	all objections have been dropped

Default Policy Classes

Policy classes copying, comparing, packing, unpacking, and recording uvm_object-based objects.

uvm_default_table_printer

```
uvm_table_printer uvm_default_table_printer = new()
```

The table printer is a global object that can be used with uvm_object::do_print to get tabular style printing.

uvm_default_tree_printer

```
uvm_tree_printer uvm_default_tree_printer = new()
```

The tree printer is a global object that can be used with uvm_object::do_print to get multi-line tree style printing.

uvm_default_line_printer

```
uvm_line_printer uvm_default_line_printer = new()
```

The line printer is a global object that can be used with uvm_object::do_print to get single-line style printing.

uvm_default_printer

```
uvm_printer uvm_default_printer = uvm_default_table_printer
```

The default printer policy. Used when calls to uvm_object::print or uvm_object::sprint do not specify a printer policy.

The default printer may be set to any legal uvm_printer derived type, including the global line, tree, and table printers described above.

uvm_default_packer

```
uvm_packer uvm_default_packer = new()
```

The default packer policy. Used when calls to uvm_object::pack and uvm_object::unpack do not specify a packer policy.

uvm_default_comparer

```
uvm_comparer uvm_default_comparer = new()
```

The default compare policy. Used when calls to uvm_object::compare do not specify a comparer policy.

uvm_default_recorder

```
uvm_recorder uvm_default_recorder = new()
```

The default recording policy. Used when calls to uvm_object::record do not specify a recorder policy.

Globals

Summary

Globals

Simulation Control

run_test	Convenience function for uvm_top.run_test().
uvm_test_done	An instance of the uvm_test_done_objection class, this object is used by components to coordinate when to end the currently running task-based phase.
global_stop_request	Convenience function for uvm_test_done.stop_request().
set_global_timeout	Convenience function for uvm_top.set_timeout().
set_global_stop_timeout	Convenience function for uvm_test_done.stop_timeout = timeout.

Reporting

uvm_report_enabled	Returns 1 if the configured verbosity in *uvm_top* is greater than *verbosity* and the action associated with the given *severity* and *id* is not UVM_NO_ACTION, else returns 0.
uvm_report_info uvm_report_warning uvm_report_error uvm_report_fatal	These methods, defined in package scope, are convenience functions that delegate to the corresponding component methods in *uvm_top*.

Configuration

set_config_int	This is the global version of set_config_int in uvm_component.
set_config_object	This is the global version of set_config_object in uvm_component.
set_config_string	This is the global version of set_config_string in uvm_component.

Miscellaneous

uvm_is_match	Returns 1 if the two strings match, 0 otherwise.
uvm_string_to_bits	Converts an input string to its bit-vector equivalent.
uvm_bits_to_string	Converts an input bit-vector to its string equivalent.
uvm_wait_for_nba_region	Callers of this task will not return until the NBA region, thus allowing other processes any number of delta cycles (#0) to settle out before continuing.
uvm_split_string	Returns a queue of strings, *values*, that is the result of the *str* split based on the *sep*.

Simulation Control

run_test

```
task run_test (string test_name = "")
```

Convenience function for uvm_top.run_test(). See uvm_root for more information.

uvm_test_done

```
uvm_test_done_objection uvm_test_done = uvm_test_done_objection::get()
```

An instance of the uvm_test_done_objection class, this object is used by components to coordinate when to end the currently running task-based phase. When all participating components have dropped their raised objections, an implicit call to global_stop_request is issued to end the run phase (or any other task-based phase).

global_stop_request

```
function void global_stop_request()
```

Convenience function for uvm_test_done.stop_request(). See uvm_test_done_objection::stop_request for more information.

set_global_timeout

```
function void set_global_timeout(time timeout,
                                 bit  overridable = 1)
```

Convenience function for uvm_top.set_timeout(). See uvm_root::set_timeout for more information. The overridable bit controls whether subsequent settings will be honored.

set_global_stop_timeout

```
function void set_global_stop_timeout(time timeout)
```

Convenience function for uvm_test_done.stop_timeout = timeout. See <uvm_uvm_test_done::stop_timeout> for more information.

Reporting

uvm_report_enabled

```
function bit uvm_report_enabled (int          verbosity,
                                 uvm_severity severity   = UVM_INFO,
                                 string       id         = ""        )
```

Returns 1 if the configured verbosity in *uvm_top* is greater than *verbosity* and the action associated with the given *severity* and *id* is not UVM_NO_ACTION, else returns 0.

See also uvm_report_object::uvm_report_enabled.

Static methods of an extension of uvm_report_object, e.g. uvm_compoent-based objects, can not call *uvm_report_enabled* because the call will resolve to the uvm_report_object::uvm_report_enabled, which is non-static. Static methods can not call non-static methods of the same class.

uvm_report_info

```
function void uvm_report_info(string id,
                              string message,
                              int    verbosity = UVM_MEDIUM,
                              string filename  = "",
                              int    line      = 0          )
```

uvm_report_warning

```
function void uvm_report_warning(string id,
                                 string message,
                                 int    verbosity = UVM_MEDIUM,
                                 string filename  = "",
                                 int    line      = 0          )
```

uvm_report_error

```
function void uvm_report_error(string id,
                               string message,
                               int    verbosity = UVM_LOW,
                               string filename  = "",
                               int    line      = 0        )
```

uvm_report_fatal

```
function void uvm_report_fatal(string id,
                               string message,
                               int    verbosity = UVM_NONE,
                               string filename  = "",
                               int    line      = 0        )
```

These methods, defined in package scope, are convenience functions that delegate to the corresponding component methods in *uvm_top*. They can be used in module-based code to use the same reporting mechanism as class-based components. See uvm_report_object for details on the reporting mechanism.

Note: Verbosity is ignored for warnings, errors, and fatals to ensure users do not inadvertently filter them out. It remains in the methods for backward compatibility.

CONFIGURATION

set_config_int

```
function void set_config_int (string          inst_name,
                              string          field_name,
                              uvm_bitstream_t value      )
```

This is the global version of set_config_int in uvm_component. This function places the configuration setting for an integral field in a global override table, which has highest precedence over any component-level setting. See uvm_component::set_config_int for details on setting configuration.

set_config_object

```
function void set_config_object (string     inst_name,
                                  string     field_name,
                                  uvm_object value,
                                  bit        clone      = 1)
```

This is the global version of set_config_object in uvm_component. This function places the configuration setting for an object field in a global override table, which has highest precedence over any component-level setting. See uvm_component::set_config_object for details on setting configuration.

set_config_string

```
function void set_config_string (string inst_name,
                                  string field_name,
                                  string value      )
```

This is the global version of set_config_string in uvm_component. This function places the configuration setting for an string field in a global override table, which has highest precedence over any component-level setting. See uvm_component::set_config_string for details on setting configuration.

MISCELLANEOUS

uvm_is_match

```
function bit uvm_is_match (string expr,
                           string str  )
```

Returns 1 if the two strings match, 0 otherwise.

The first string, *expr*, is a string that may contain '*' and '?' characters. A * matches zero or more characters, and ? matches any single character. The 2nd argument, *str*, is the string begin matched against. It must not contain any wildcards.

uvm_string_to_bits

```
function logic[UVM_LARGE_STRING:0] uvm_string_to_bits(string str)
```

Converts an input string to its bit-vector equivalent. Max bit-vector length is approximately 14000 characters.

uvm_bits_to_string

```
function string uvm_bits_to_string(logic [UVM_LARGE_STRING:0] str)
```

Converts an input bit-vector to its string equivalent. Max bit-vector length is approximately 14000 characters.

uvm_wait_for_nba_region

```
task uvm_wait_for_nba_region
```

Callers of this task will not return until the NBA region, thus allowing other processes any number of delta cycles (#0) to settle out before continuing. See uvm_sequencer_base::wait_for_sequences for example usage.

uvm_split_string

```
function automatic void uvm_split_string (    string str,
                                               byte   sep,
                                           ref string values[$])
```

Returns a queue of strings, *values*, that is the result of the *str* split based on the *sep*. For example:

```
uvm_split_string("1,on,false", ",", splits);
```

Results in the 'splits' queue containing the three elements: 1, on and false.

Bibliography

[B1] Open SystemC Initiative (OSCI), Transaction Level Modeling (TLM) Library, Release 1.0.

[B2] Open SystemC Initiative (OSCI), Transaction Level Modeling (TLM-2.0) Library, Release 2.0.

[B3] IEEE Std 1685™, IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows.

[B4] For practical UVM examples, see the following Internet location:
http://www.accellera.org/activities/vip.

Index

$#!

- `uvm_do_on_pri
- `uvm_do_on_pri_with
- `uvm_do_on_with
- `uvm_do_pri
- `uvm_do_pri_with
- `uvm_do_with
- `uvm_error
- `uvm_error_context
- `uvm_fatal
- `uvm_fatal_context
- `uvm_field_*macro flags
- `uvm_field_*macros
- `uvm_field_aa_*_int macros
- `uvm_field_aa_*_string macros
- `uvm_field_aa_int_byte
- `uvm_field_aa_int_byte_unsigned
- `uvm_field_aa_int_enumkey
- `uvm_field_aa_int_int
- `uvm_field_aa_int_int_unsigned
- `uvm_field_aa_int_integer
- `uvm_field_aa_int_integer_unsigned
- `uvm_field_aa_int_key
- `uvm_field_aa_int_longint
- `uvm_field_aa_int_longint_unsigned
- `uvm_field_aa_int_shortint
- `uvm_field_aa_int_shortint_unsigned
- `uvm_field_aa_int_string
- `uvm_field_aa_object_int
- `uvm_field_aa_object_string
- `uvm_field_aa_string_string
- `uvm_field_array_*macros
- `uvm_field_array_enum
- `uvm_field_array_int
- `uvm_field_array_object
- `uvm_field_array_string
- `uvm_field_enum
- `uvm_field_event
- `uvm_field_int
- `uvm_field_object
- `uvm_field_queue_*macros
- `uvm_field_queue_enum
- `uvm_field_queue_int
- `uvm_field_queue_object
- `uvm_field_queue_string
- `uvm_field_real
- `uvm_field_sarray_*macros
- `uvm_field_sarray_enum
- `uvm_field_sarray_int
- `uvm_field_sarray_object
- `uvm_field_sarray_string
- `uvm_field_string
- `uvm_field_utils_begin
- `uvm_field_utils_end
- `uvm_get_imp_decl
- `uvm_get_peek_imp_decl
- `uvm_info
- `uvm_info_context
- `uvm_master_imp_decl
- `UVM_MAX_STREAMBITS
- `uvm_nonblocking_get_imp_decl

`uvm_nonblocking_get_peek_imp_decl
`uvm_nonblocking_master_imp_decl
`uvm_nonblocking_peek_imp_decl
`uvm_nonblocking_put_imp_decl
`uvm_nonblocking_slave_imp_decl
`uvm_nonblocking_transport_imp_decl
`uvm_object_param_utils
`uvm_object_param_utils_begin
`uvm_object_registry
`uvm_object_utils
`uvm_object_utils_begin
`uvm_object_utils_end
`uvm_pack_array
`uvm_pack_arrayN
`uvm_pack_enum
`uvm_pack_enumN
`uvm_pack_int
`uvm_pack_intN
`uvm_pack_queue
`uvm_pack_queueN
`uvm_pack_real
`uvm_pack_sarray
`uvm_pack_sarrayN
`uvm_pack_string
`uvm_peek_imp_decl
`uvm_put_imp_decl
`uvm_rand_send
`uvm_rand_send_pri
`uvm_rand_send_pri_with
`uvm_rand_send_with
`uvm_record_attribute
`uvm_record_field
`UVM_REG_ADDR_WIDTH
`UVM_REG_BYTENABLE_WIDTH
`UVM_REG_CVR_WIDTH
`UVM_REG_DATA_WIDTH
`uvm_register_cb
`uvm_send
`uvm_send_pri
`uvm_sequence_library_utils
`uvm_set_super_type
`uvm_slave_imp_decl
`UVM_TLM_B_MASK
`UVM_TLM_B_TRANSPORT_IMP
`UVM_TLM_FUNCTION_ERROR
`UVM_TLM_NB_BW_MASK
`UVM_TLM_NB_FW_MASK
`UVM_TLM_NB_TRANSPORT_BW_IMP
`UVM_TLM_NB_TRANSPORT_FW_IMP
`UVM_TLM_TASK_ERROR
`uvm_transport_imp_decl
`uvm_unpack_array
`uvm_unpack_arrayN
`uvm_unpack_enum
`uvm_unpack_enumN
`uvm_unpack_int
`uvm_unpack_intN
`uvm_unpack_queue
`uvm_unpack_queueN
`uvm_unpack_real

A

B

C

Callbacks Classes
can_get
uvm_tlm_if_base#(T1,T2)
can_peek
uvm_tlm_if_base#(T1,T2)
can_put
uvm_tlm_if_base#(T1,T2)
cancel
uvm_barrier
uvm_event
capacity
uvm_reg_fifo
catch
uvm_report_catcher
CB
uvm_callbacks#(T,CB)
Change Message State
uvm_report_catcher
check_config_usage
uvm_component
check_data_width
uvm_reg_block
check_ph
Pre-Defined Phases
check_phase
uvm_component
uvm_reg_predictor
check_type
uvm_comparer
Classes for Adapting Between Register and Bus Operations
clear_extension
uvm_tlm_generic_payload
clear_extensions
uvm_tlm_generic_payload
clear_hdl_path
uvm_mem
uvm_reg
uvm_reg_block
uvm_reg_file
clear_response_queue
uvm_sequence_base
clone
uvm_object
Command Line Debug
uvm_cmdline_processor
Command Line Processor Class
Common Phases Global Variables
Pre-Defined Phases
Comparators
comps/uvm_algorithmic_comparator.svh
comps/uvm_in_order_comparator.svh
comparators.txt

convert2string
uvm_mem_mam
uvm_object
uvm_reg_item
uvm_tlm_generic_payload

copy
uvm_object

Copying
uvm_object

Core Base Classes
Coverage
uvm_mem
uvm_reg
uvm_reg_block

create
uvm_component_registry#(T,Tname)
uvm_object
uvm_object_registry#(T,Tname)
uvm_tlm_extension_base

create_component
uvm_component
uvm_component_registry#(T,Tname)
uvm_object_wrapper

create_component_by_name
uvm_factory

create_component_by_type
uvm_factory

create_item
uvm_sequence_base

create_map
uvm_reg_block

create_object
uvm_component
uvm_object_registry#(T,Tname)
uvm_object_wrapper

create_object_by_name
uvm_factory

create_object_by_type
uvm_factory

CREATED
Creation
uvm_factory
uvm_object

Current Message State
uvm_report_catcher

current_grabber
uvm_sequencer_base

D

E

F

G

get_config_string
uvm_component

get_connected_to
uvm_port_component_base

get_coverage
uvm_mem
uvm_reg
uvm_reg_block

get_current_item
uvm_sequence#(REQ,RSP)
uvm_sequencer_param_base#(REQ,RSP)

get_data
uvm_tlm_generic_payload

get_data_length
uvm_tlm_generic_payload

get_default_hdl_path
uvm_reg_block
uvm_reg_file

get_default_path
uvm_reg_block

get_depth
uvm_component
uvm_sequence_item

get_domain
uvm_component

get_drain_time
uvm_objection

get_end_offset
uvm_mem_region

get_end_time
uvm_transaction

get_event_pool
uvm_transaction

get_extension
uvm_tlm_generic_payload

get_field_by_name
uvm_reg
uvm_reg_block
uvm_vreg

get_fields
uvm_reg
uvm_reg_block
uvm_reg_map
uvm_vreg

get_file_handle
uvm_report_handler

get_first
uvm_callbacks#(T,CB)

get_first_child
uvm_component

get_fname
uvm_report_catcher

get_frontdoor
- uvm_mem
- uvm_reg

get_full_hdl_path
- uvm_mem
- uvm_reg
- uvm_reg_block
- uvm_reg_file

get_full_name
- uvm_component
- uvm_mem
- uvm_object
- uvm_port_base#(IF)
- uvm_reg
- uvm_reg_block
- uvm_reg_field
- uvm_reg_file
- uvm_reg_map
- uvm_vreg
- uvm_vreg_field

get_global
- uvm_pool#(KEY,T)
- uvm_queue#(T)

get_global_pool
- uvm_object_string_pool#(T)
- uvm_pool#(KEY,T)

get_global_queue
- uvm_queue#(T)

get_hdl_path
- uvm_mem
- uvm_reg
- uvm_reg_block
- uvm_reg_file

get_hdl_path_kinds
- uvm_mem
- uvm_reg

get_highest_precedence
- uvm_resource#(T)
- uvm_resource_pool

get_id
- uvm_report_catcher

get_id_count
- uvm_report_server

get_if
- uvm_port_base#(IF)

get_incr
- uvm_vreg

get_initiator
- uvm_transaction

get_inst
- uvm_cmdline_processor

get_inst_count
- uvm_object

get_inst_id

uvm_object

get_is_active
uvm_agent

get_jump_target
uvm_phase

get_last
uvm_callbacks#(T,CB)

get_len
uvm_mem_region

get_line
uvm_report_catcher

get_lsb_pos
uvm_reg_field

get_lsb_pos_in_register
uvm_vreg_field

get_map_by_name
uvm_reg_block

get_maps
uvm_mem
uvm_reg
uvm_reg_block
uvm_vreg

get_max_quit_count
uvm_report_server

get_max_size
uvm_mem
uvm_reg
uvm_reg_field

get_mem_by_name
uvm_reg_block

get_mem_by_offset
uvm_reg_map

get_memory
uvm_mem_mam
uvm_mem_region
uvm_vreg

get_message
uvm_report_catcher

get_n_bits
uvm_mem
uvm_reg
uvm_reg_field
uvm_vreg_field

get_n_bytes
uvm_mem
uvm_mem_region
uvm_reg
uvm_reg_map
uvm_vreg

get_n_maps
uvm_mem
uvm_reg
uvm_vreg

get_n_memlocs
uvm_vreg

get_name
uvm_mem
uvm_object
uvm_port_base#(IF)
uvm_reg
uvm_reg_block
uvm_reg_field
uvm_reg_file
uvm_reg_map
uvm_tlm_time
uvm_vreg
uvm_vreg_field

get_next
uvm_callbacks#(T,CB)

get_next_child
uvm_component

get_next_item
uvm_sqr_if_base#(REQ,RSP)

get_num_children
uvm_component

get_num_extensions
uvm_tlm_generic_payload

get_num_last_reqs
uvm_sequencer_param_base#(REQ,RSP)

get_num_last_rsps
uvm_sequencer_param_base#(REQ,RSP)

get_num_reqs_sent
uvm_sequencer_param_base#(REQ,RSP)

get_num_rsps_received
uvm_sequencer_param_base#(REQ,RSP)

get_num_waiters
uvm_barrier
uvm_event

get_object_type
uvm_object

get_objection
uvm_phase

get_objection_count
uvm_objection

get_objection_total
uvm_objection

get_objectors
uvm_objection

get_offset
uvm_mem
uvm_reg

get_offset_in_memory
uvm_vreg

get_packed_size
uvm_packer

get_parent

uvm_component
uvm_mem
uvm_phase
uvm_port_base#(IF)
uvm_reg
uvm_reg_block
uvm_reg_field
uvm_reg_file
uvm_reg_map
uvm_vreg
uvm_vreg_field

get_parent_map
uvm_reg_map

get_parent_sequence
uvm_sequence_item

get_peek_export
uvm_tlm_fifo_base#(T)

get_peek_request_export
uvm_tlm_req_rsp_channel#(REQ,RSP)

get_peek_response_export
uvm_tlm_req_rsp_channel#(REQ,RSP)

get_phase_type
uvm_phase

get_physical_addresses
uvm_reg_map

get_plusargs
uvm_cmdline_processor

get_port
uvm_port_component#(PORT)

get_prev
uvm_callbacks#(T,CB)

get_priority
uvm_sequence_base

get_provided_to
uvm_port_component_base

get_quit_count
uvm_report_server

get_radix_str
uvm_printer_knobs

get_realtime
uvm_tlm_time

get_reg_by_name
uvm_reg_block

get_reg_by_offset
uvm_reg_map

get_regfile
uvm_reg
uvm_reg_file

get_region
uvm_vreg

get_registers
uvm_reg_block
uvm_reg_map

get_report_action
uvm_report_object

get_report_catcher
uvm_report_catcher

get_report_file_handle
uvm_report_object

get_report_handler
uvm_report_object

get_report_server
uvm_report_object

get_report_verbosity_level
uvm_report_object

get_reset
uvm_reg
uvm_reg_field

get_response
uvm_sequence#(REQ,RSP)

get_response_queue_depth
uvm_sequence_base

get_response_queue_error_report_disabled
uvm_sequence_base

get_response_status
uvm_tlm_generic_payload

get_response_string
uvm_tlm_generic_payload

get_rights
uvm_mem
uvm_reg
uvm_vreg

get_root_blocks
uvm_reg_block

get_root_map
uvm_reg_map

get_root_sequence
uvm_sequence_item

get_root_sequence_name
uvm_sequence_item

get_run_count
uvm_phase

get_schedule
uvm_component
uvm_phase

get_schedule_name
uvm_phase

get_scope
uvm_resource_base

get_sequence_id
uvm_sequence_item

get_sequence_path
uvm_sequence_item

get_sequence_state
uvm_sequence_base

get_sequencer
uvm_reg_map
uvm_sequence_item

get_server
uvm_report_server

get_severity
uvm_report_catcher

get_severity_count
uvm_report_server

get_size
uvm_mem
uvm_vreg

get_start_offset
uvm_mem_region

get_state
uvm_phase

get_streaming_width
uvm_tlm_generic_payload

get_submap_offset
uvm_reg_map

get_submaps
uvm_reg_map

get_threshold
uvm_barrier

get_tool_name
uvm_cmdline_processor

get_tool_version
uvm_cmdline_processor

get_tr_handle
uvm_transaction

get_transaction_id
uvm_transaction

get_trigger_data
uvm_event

get_trigger_time
uvm_event

get_type
uvm_object
uvm_resource#(T)

get_type_handle
uvm_resource#(T)
uvm_resource_base
uvm_tlm_extension_base

get_type_handle_name
uvm_tlm_extension_base

get_type_name
uvm_callback
uvm_component_registry#(T,Tname)
uvm_object
uvm_object_registry#(T,Tname)
uvm_object_string_pool#(T)
uvm_object_wrapper
uvm_port_base#(IF)

H

uvm_reg_file

has_lock
uvm_sequence_base
uvm_sequencer_base

has_reset
uvm_reg
uvm_reg_field

HDL Access
uvm_mem
uvm_vreg
uvm_vreg_field

HDL Paths Checking Test Sequence header
uvm_printer_knobs

hex_radix
uvm_printer_knobs

Hierarchical Reporting Interface
uvm_component

Hierarchy Interface
uvm_component

I

uvm_queue#(T)

Interface Masks

Introspection

uvm_mem
uvm_mem_mam
uvm_reg
uvm_reg_block
uvm_reg_field
uvm_reg_fifo
uvm_reg_file
uvm_reg_map
uvm_vreg
uvm_vreg_field

is

uvm_phase

is_active

uvm_transaction

is_after

uvm_phase

is_auditing

uvm_resource_options

is_auto_updated

uvm_reg_backdoor

is_before

uvm_phase

is_blocked

uvm_sequence_base
uvm_sequencer_base

is_busy

uvm_reg

is_child

uvm_sequencer_base

is_dmi_allowed

uvm_tlm_generic_payload

is_empty

uvm_tlm_fifo

is_enabled

uvm_callback

is_export

uvm_port_base#(IF)
uvm_port_component_base

is_full

uvm_tlm_fifo

is_grabbed

uvm_sequencer_base

is_hdl_path_root

uvm_reg_block

is_imp

uvm_port_base#(IF)
uvm_port_component_base

is_in_map

uvm_mem
uvm_reg

J

K

L

M

uvm_resource_base

max_offset
uvm_mem_mam_policy

max_size
uvm_port_base#(IF)

mcd
uvm_printer_knobs

mem
uvm_mem_shared_access_seq
uvm_mem_single_access_seq
uvm_mem_single_walk_seq

mem_seq
uvm_mem_access_seq
uvm_mem_walk_seq
uvm_reg_mem_shared_access_seq

Memory Access Test Sequence
Memory Allocation Manager
Memory Management
uvm_mem_mam

Memory Walking-Ones Test Sequences
Methods
Global
uvm_*_export#(REQ,RSP)
uvm_*_export#(T)
uvm_*_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_*_imp#(T,IMP)
uvm_*_port#(REQ,RSP)
uvm_*_port#(T)
uvm_agent
uvm_algorithmic_comparator#(BEFORE,AFTER,TRANSFORMER)
uvm_analysis_export
uvm_analysis_port
uvm_barrier
uvm_bottomup_phase
uvm_built_in_pair#(T1,T2)
uvm_callback
uvm_callback_iter
uvm_callbacks_objection
uvm_comparer
uvm_component_registry#(T,Tname)
uvm_config_db#(T)
uvm_domain
uvm_driver#(REQ,RSP)
uvm_env
uvm_event
uvm_event_callback
uvm_hdl_path_concat
uvm_heartbeat
uvm_in_order_comparator#(T,comp_type,convert,pair_type)
uvm_mem_access_seq
uvm_mem_region
uvm_mem_single_walk_seq
uvm_mem_walk_seq
uvm_monitor
uvm_object_string_pool#(T)
uvm_object_wrapper
uvm_objection_callback
uvm_pair#(T1,T2)

uvm_pool#(KEY,T)
uvm_port_base#(IF)
uvm_port_component#(PORT)
uvm_port_component_base
uvm_printer_knobs
uvm_push_driver#(REQ,RSP)
uvm_push_sequencer#(REQ,RSP)
uvm_queue#(T)
uvm_random_stimulus#(T)
uvm_recorder
uvm_reg_access_seq
uvm_reg_backdoor
uvm_reg_bit_bash_seq
uvm_reg_cbs
uvm_reg_frontdoor
uvm_reg_hw_reset_seq
uvm_reg_indirect_data
uvm_reg_item
uvm_reg_mem_built_in_seq
uvm_reg_mem_shared_access_seq
uvm_reg_predictor
uvm_reg_read_only_cbs
uvm_reg_tlm_adapter
uvm_reg_write_only_cbs
uvm_report_handler
uvm_report_server
uvm_resource_db
uvm_resource_options
uvm_root
uvm_scoreboard
uvm_seq_item_pull_imp#(REQ,RSP,IMP)
uvm_sequence#(REQ,RSP)
uvm_sequencer#(REQ,RSP)
uvm_sequencer_base
uvm_sqr_if_base#(REQ,RSP)
uvm_subscriber
uvm_table_printer
uvm_task_phase
uvm_test
uvm_test_done_objection
uvm_tlm_analysis_fifo
uvm_tlm_b_initiator_socket
uvm_tlm_b_target_socket
uvm_tlm_extension
uvm_tlm_extension_base
uvm_tlm_fifo
uvm_tlm_fifo_base#(T)
uvm_tlm_nb_initiator_socket
uvm_tlm_nb_passthrough_target_socket
uvm_tlm_nb_target_socket
uvm_tlm_nb_transport_bw_export
uvm_tlm_nb_transport_bw_port
uvm_tlm_req_rsp_channel#(REQ,RSP)
uvm_tlm_transport_channel#(REQ,RSP)
uvm_topdown_phase
uvm_transaction
uvm_tree_printer
uvm_utils
uvm_vreg_cbs
uvm_vreg_field_cbs

N

uvm_*_export#(REQ,RSP)
uvm_*_export#(T)
uvm_*_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_*_imp#(T,IMP)
uvm_*_port#(REQ,RSP)
uvm_*_port#(T)
uvm_agent
uvm_algorithmic_comparator#(BEFORE,AFTER,TRANSFORMER)
uvm_analysis_export
uvm_barrier
uvm_bottomup_phase
uvm_built_in_pair#(T1,T2)
uvm_callback
uvm_callback_iter
uvm_component
uvm_driver#(REQ,RSP)
uvm_env
uvm_event
uvm_event_callback
uvm_heartbeat
uvm_line_printer
uvm_mem
uvm_mem_mam
uvm_mem_single_walk_seq
uvm_monitor
uvm_object
uvm_object_string_pool#(T)
uvm_objection
uvm_pair#(T1,T2)
uvm_phase
uvm_pool#(KEY,T)
uvm_port_base#(IF)
uvm_push_driver#(REQ,RSP)
uvm_push_sequencer#(REQ,RSP)
uvm_queue#(T)
uvm_random_stimulus#(T)
uvm_reg
uvm_reg_adapter
uvm_reg_backdoor
uvm_reg_block
uvm_reg_field
uvm_reg_fifo
uvm_reg_file
uvm_reg_frontdoor
uvm_reg_indirect_data
uvm_reg_item
uvm_reg_map
uvm_reg_predictor
uvm_reg_sequence
uvm_report_catcher
uvm_report_handler
uvm_report_object
uvm_report_server
uvm_resource_base
uvm_scoreboard
uvm_seq_item_pull_imp#(REQ,RSP,IMP)
uvm_sequence#(REQ,RSP)
uvm_sequence_base
uvm_sequence_item
uvm_sequencer#(REQ,RSP)

O

uvm_printer_knobs

offset

uvm_reg_item

P

uvm_sqr_if_base#(REQ,RSP)
uvm_tlm_if_base#(T1,T2)

Put
put_ap
uvm_tlm_fifo_base#(T)

put_export
uvm_tlm_fifo_base#(T)

put_request_export
uvm_tlm_req_rsp_channel#(REQ,RSP)

put_response_export
uvm_tlm_req_rsp_channel#(REQ,RSP)

Q

qualify
uvm_test_done_objection

R

```
    uvm_root
rw_info
    uvm_reg_frontdoor
```

S

uvm_component

set_config_object
Global
uvm_component

set_config_string
Global
uvm_component

set_coverage
uvm_mem
uvm_reg
uvm_reg_block

set_data
uvm_tlm_generic_payload

set_data_length
uvm_tlm_generic_payload

set_default
uvm_resource_db

set_default_hdl_path
uvm_reg_block
uvm_reg_file

set_default_index
uvm_port_base#(IF)

set_default_map
uvm_reg_block

set_depth
uvm_sequence_item

set_dmi_allowed
uvm_tlm_generic_payload

set_domain
uvm_component

set_drain_time
uvm_objection

set_extension
uvm_tlm_generic_payload

set_frontdoor
uvm_mem
uvm_reg

set_global_stop_timeout
set_global_timeout
set_hdl_path_root
uvm_reg_block

set_heartbeat
uvm_heartbeat

set_id
uvm_report_catcher

set_id_count
uvm_report_server

set_id_info
uvm_sequence_item

set_initiator
uvm_transaction

set_inst_override

uvm_component
uvm_component_registry#(T,Tname)
uvm_object_registry#(T,Tname)

set_inst_override_by_name
uvm_factory

set_inst_override_by_type
uvm_component
uvm_factory

set_int_local
uvm_object

set_max_quit_count
uvm_report_server

set_message
uvm_report_catcher

set_mode
uvm_heartbeat

set_name
uvm_component
uvm_object

set_name_override
uvm_resource_pool

set_num_last_reqs
uvm_sequencer_param_base#(REQ,RSP)

set_num_last_rsps
uvm_sequencer_param_base#(REQ,RSP)

set_object_local
uvm_object

set_offset
uvm_mem
uvm_reg

set_override
uvm_resource#(T)
uvm_resource_pool

set_parent_sequence
uvm_sequence_item

set_phase_imp
uvm_component

set_priority
uvm_resource_pool
uvm_sequence_base

set_priority_name
uvm_resource_pool

set_priority_type
uvm_resource_pool

set_quit_count
uvm_report_server

set_read
uvm_tlm_generic_payload

set_read_only
uvm_resource_base

set_report_default_file
uvm_report_object

set_report_default_file_hier
uvm_component

set_report_handler
uvm_report_object

set_report_id_action
uvm_report_object

set_report_id_action_hier
uvm_component

set_report_id_file
uvm_report_object

set_report_id_file_hier
uvm_component

set_report_id_verbosity
uvm_report_object

set_report_id_verbosity_hier
uvm_component

set_report_max_quit_count
uvm_report_object

set_report_severity_action
uvm_report_object

set_report_severity_action_hier
uvm_component

set_report_severity_file
uvm_report_object

set_report_severity_file_hier
uvm_component

set_report_severity_id_action
uvm_report_object

set_report_severity_id_action_hier
uvm_component

set_report_severity_id_file
uvm_report_object

set_report_severity_id_file_hier
uvm_component

set_report_severity_id_override
uvm_report_object

set_report_severity_id_verbosity
uvm_report_object

set_report_severity_id_verbosity_hier
uvm_component

set_report_severity_override
uvm_report_object

set_report_verbosity_level
uvm_report_object

set_report_verbosity_level_hier
uvm_component

set_reset
uvm_reg
uvm_reg_field

set_response_queue_depth
uvm_sequence_base

set_response_queue_error_report_disabled
uvm_sequence_base

set_response_status
uvm_tlm_generic_payload

set_scope
uvm_resource_base

set_sequencer
uvm_reg_map
uvm_sequence_item

set_server
uvm_report_server

set_severity
uvm_report_catcher

set_severity_count
uvm_report_server

set_streaming_width
uvm_tlm_generic_payload

set_string_local
uvm_object

set_submap_offset
uvm_reg_map

set_threshold
uvm_barrier

set_time_resolution
uvm_tlm_time

set_timeout
uvm_root

set_transaction_id
uvm_transaction

set_type_override
uvm_component
uvm_component_registry#(T,Tname)
uvm_object_registry#(T,Tname)
uvm_resource_pool

set_type_override_by_name
uvm_factory

set_type_override_by_type
uvm_component
uvm_factory

set_use_sequence_info
uvm_sequence_item

set_verbosity
uvm_report_catcher

set_volatility
uvm_reg_field

set_write
uvm_tlm_generic_payload

Setup
uvm_report_object

sev
uvm_comparer

Shared Register and Memory Access Test Sequences

uvm_component
uvm_heartbeat

stop_request
uvm_test_done_objection

stop_sequences
uvm_sequencer#(REQ,RSP)
uvm_sequencer_base

stop_stimulus_generation
uvm_random_stimulus#(T)

stop_timeout
uvm_test_done_objection

STOPPED
summarize
uvm_report_server

summarize_report_catcher
uvm_report_catcher

supports_byte_enable
uvm_reg_adapter

suspend
uvm_component

sync
uvm_phase

Synchronization
uvm_phase

Synchronization Classes

T

T
uvm_callbacks#(T,CB)

T1 first
uvm_pair#(T1,T2)

T2 second
uvm_pair#(T1,T2)

tests
uvm_reg_mem_built_in_seq

TLM Channel Classes
TLM Export Classes
TLM FIFO Classes
TLM Generic Payload&Extensions
TLM IF Class
TLM Implementation Port Declaration Macros
tlm interfaces
TLM Interfaces
TLM Port Classes
TLM Socket Base Classes
TLM Sockets
tlm transport methods
uvm_tlm_if

TLM1
TLM1 Interfaces,Ports,Exports and Transport Interfaces
TLM2

TLM2 Export Classes
TLM2 imps(interface implementations)
TLM2 Interfaces,Ports,Exports and Transport Interfaces Subset
TLM2 ports
Tool information
uvm_cmdline_processor

top_levels
uvm_root

tr_handle
uvm_recorder

trace_mode
uvm_objection

transport
uvm_tlm_if_base#(T1,T2)

Transport
transport_export
uvm_tlm_transport_channel#(REQ,RSP)

traverse
uvm_bottomup_phase
uvm_task_phase
uvm_topdown_phase

trigger
uvm_event

try_get
uvm_tlm_if_base#(T1,T2)

try_lock
uvm_resource_base

try_next_item
uvm_sqr_if_base#(REQ,RSP)

try_peek
uvm_tlm_if_base#(T1,T2)

try_put
uvm_tlm_if_base#(T1,T2)

try_read_with_lock
uvm_resource#(T)

try_write_with_lock
uvm_resource#(T)

turn_off_auditing
uvm_resource_options

turn_on_auditing
uvm_resource_options

Type Interface
uvm_resource#(T)

Type&Instance Overrides
uvm_factory

type_name
uvm_printer_knobs

Typedefs
Types
Global
uvm_vreg_cbs
uvm_vreg_field_cbs

Types and Enumerations

U

uvm_reg_field
uvm_reg_field_cb
uvm_reg_field_cb_iter
uvm_reg_fifo
uvm_reg_file
uvm_reg_frontdoor
uvm_reg_hw_reset_seq
uvm_reg_indirect_data
uvm_reg_item
uvm_reg_map
uvm_reg_mem_access_seq
uvm_reg_mem_built_in_seq
uvm_reg_mem_hdl_paths_seq
uvm_reg_mem_shared_access_seq
uvm_reg_mem_tests_e
uvm_reg_predictor
uvm_reg_read_only_cbs
uvm_reg_sequence
uvm_reg_shared_access_seq
uvm_reg_single_access_seq
uvm_reg_single_bit_bash_seq
uvm_reg_tlm_adapter
uvm_reg_write_only_cbs
uvm_report_catcher
uvm_report_enabled
Global
uvm_report_object

uvm_report_error
Global
uvm_report_catcher
uvm_report_object
uvm_sequence_item

uvm_report_fatal
Global
uvm_report_catcher
uvm_report_object
uvm_sequence_item

uvm_report_handler
uvm_report_info
Global
uvm_report_catcher
uvm_report_object
uvm_sequence_item

uvm_report_object
uvm_report_server
uvm_report_warning
Global
uvm_report_catcher
uvm_report_object
uvm_sequence_item

UVM_RERUN
uvm_resource#(T)
uvm_resource_base
uvm_resource_db
uvm_resource_options
uvm_resource_pool
uvm_resource_types
uvm_root

uvm_scoreboard
uvm_seq_item_pull_export#(REQ,RSP)
uvm_seq_item_pull_imp#(REQ,RSP,IMP)
uvm_seq_item_pull_port#(REQ,RSP)
UVM_SEQ_LIB_ITEM
UVM_SEQ_LIB_RAND
UVM_SEQ_LIB_RANDC
UVM_SEQ_LIB_USER
uvm_sequence#(REQ,RSP)
uvm_sequence_base
uvm_sequence_item
uvm_sequence_lib_mode
uvm_sequence_state_enum
uvm_sequencer#(REQ,RSP)
uvm_sequencer_arb_mode
uvm_sequencer_base
uvm_sequencer_param_base#(REQ,RSP)
uvm_severity
UVM_SHALLOW
UVM_SKIPPED
uvm_split_string
uvm_sqr_if_base#(REQ,RSP)
uvm_status_e
UVM_STOP
UVM_STRING
uvm_string_to_bits
uvm_subscriber
uvm_table_printer
uvm_task_phase
uvm_test
uvm_test_done
uvm_test_done_objection
UVM_TIME
UVM_TLM_ACCEPTED
UVM_TLM_ADDRESS_ERROR_RESPONSE
uvm_tlm_analysis_fifo
uvm_tlm_b_initiator_socket
uvm_tlm_b_initiator_socket_base
uvm_tlm_b_passthrough_initiator_socket
uvm_tlm_b_passthrough_initiator_socket_base
uvm_tlm_b_passthrough_target_socket
uvm_tlm_b_passthrough_target_socket_base
uvm_tlm_b_target_socket
uvm_tlm_b_target_socket_base
uvm_tlm_b_transport_export
uvm_tlm_b_transport_imp
uvm_tlm_b_transport_port
UVM_TLM_BURST_ERROR_RESPONSE
UVM_TLM_BYTE_ENABLE_ERROR_RESPONSE
uvm_tlm_command_e
UVM_TLM_COMMAND_ERROR_RESPONSE
UVM_TLM_COMPLETED
uvm_tlm_extension
uvm_tlm_extension_base
uvm_tlm_fifo
uvm_tlm_fifo_base#(T)
UVM_TLM_GENERIC_ERROR_RESPONSE
uvm_tlm_generic_payload
uvm_tlm_gp
uvm_tlm_if

uvm_tlm_if_base#(T1,T2)
UVM_TLM_IGNORE_COMMAND
UVM_TLM_INCOMPLETE_RESPONSE
uvm_tlm_nb_initiator_socket
uvm_tlm_nb_initiator_socket_base
uvm_tlm_nb_passthrough_initiator_socket
uvm_tlm_nb_passthrough_initiator_socket_base
uvm_tlm_nb_passthrough_target_socket
uvm_tlm_nb_passthrough_target_socket_base
uvm_tlm_nb_target_socket
uvm_tlm_nb_target_socket_base
uvm_tlm_nb_transport_bw_export
uvm_tlm_nb_transport_bw_imp
uvm_tlm_nb_transport_bw_port
uvm_tlm_nb_transport_fw_export
uvm_tlm_nb_transport_fw_imp
uvm_tlm_nb_transport_fw_port
UVM_TLM_OK_RESPONSE
uvm_tlm_phase_e
UVM_TLM_READ_COMMAND
uvm_tlm_req_rsp_channel#(REQ,RSP)
uvm_tlm_response_status_e
uvm_tlm_sync_e
uvm_tlm_time
uvm_tlm_transport_channel#(REQ,RSP)
UVM_TLM_UPDATED
UVM_TLM_WRITE_COMMAND
uvm_top
uvm_root
uvm_topdown_phase
uvm_transaction
uvm_tree_printer
UVM_UNSIGNED
uvm_utils
uvm_verbosity
uvm_void
uvm_vreg
uvm_vreg_cb
uvm_vreg_cbs
uvm_vreg_cb_iter
uvm_vreg_cbs
uvm_vreg_cbs
uvm_vreg_field
uvm_vreg_field_cb
uvm_vreg_field_cbs
uvm_vreg_field_cb_iter
uvm_vreg_field_cbs
uvm_vreg_field_cbs
uvm_wait_for_nba_region
uvm_wait_op
UVM_WARNING
UVM_WRITE

V

W

wait_for_change
uvm_reg_backdoor

wait_for_grant
uvm_sequence_base
uvm_sequencer_base

wait_for_item_done
uvm_sequence_base
uvm_sequencer_base

wait_for_relevant
uvm_sequence_base

wait_for_sequence_state
uvm_sequence_base

wait_for_sequences
uvm_sequencer_base
uvm_sqr_if_base#(REQ,RSP)

wait_for_state
uvm_phase

wait_modified
uvm_config_db#(T)
uvm_resource_base

wait_off
uvm_event

wait_on
uvm_event

wait_ptrigger
uvm_event

wait_ptrigger_data
uvm_event

wait_trigger
uvm_event

wait_trigger_data
uvm_event

Why is this necessary
uvm_tlm_time

write
uvm_analysis_port
uvm_mem
uvm_mem_region
uvm_reg
uvm_reg_backdoor
uvm_reg_field
uvm_reg_fifo
uvm_resource#(T)
uvm_subscriber
uvm_tlm_if_base#(T1,T2)
uvm_vreg
uvm_vreg_field

write_by_name
uvm_resource_db

write_by_type
uvm_resource_db

write_mem
uvm_reg_sequence

write_mem_by_name
uvm_reg_block

write_reg
uvm_reg_sequence

write_reg_by_name
uvm_reg_block

write_with_lock
uvm_resource#(T)

Class Index

P

U

uvm_mem_mam
uvm_mem_mam_cfg
uvm_mem_mam_policy
uvm_mem_region
uvm_mem_shared_access_seq
uvm_mem_single_access_seq
uvm_mem_single_walk_seq
uvm_mem_walk_seq
uvm_monitor
uvm_object
uvm_object_registry#(T,Tname)
uvm_object_string_pool#(T)
uvm_object_wrapper
uvm_objection
uvm_objection_callback
uvm_packer
uvm_pair#(T1,T2)
uvm_phase
uvm_pool#(KEY,T)
uvm_port_base#(IF)
uvm_port_component#(PORT)
uvm_port_component_base
uvm_printer
uvm_printer_knobs
uvm_push_driver#(REQ,RSP)
uvm_push_sequencer#(REQ,RSP)
uvm_queue#(T)
uvm_random_stimulus#(T)
uvm_recorder
uvm_reg
uvm_reg_access_seq
uvm_reg_adapter
uvm_reg_backdoor
uvm_reg_bit_bash_seq
uvm_reg_block
uvm_reg_bus_op
uvm_reg_cbs
uvm_reg_field
uvm_reg_fifo
uvm_reg_file
uvm_reg_frontdoor
uvm_reg_hw_reset_seq
uvm_reg_indirect_data
uvm_reg_item
uvm_reg_map
uvm_reg_mem_access_seq
uvm_reg_mem_built_in_seq
uvm_reg_mem_hdl_paths_seq
uvm_reg_mem_shared_access_seq
uvm_reg_predictor
uvm_reg_read_only_cbs
uvm_reg_sequence
uvm_reg_shared_access_seq
uvm_reg_single_access_seq
uvm_reg_single_bit_bash_seq
uvm_reg_tlm_adapter
uvm_reg_write_only_cbs
uvm_report_catcher
uvm_report_handler
uvm_report_object

uvm_report_server
uvm_resource#(T)
uvm_resource_base
uvm_resource_db
uvm_resource_options
uvm_resource_pool
uvm_resource_types
uvm_root
uvm_scoreboard
uvm_seq_item_pull_export#(REQ,RSP)
uvm_seq_item_pull_imp#(REQ,RSP,IMP)
uvm_seq_item_pull_port#(REQ,RSP)
uvm_sequence#(REQ,RSP)
uvm_sequence_base
uvm_sequence_item
uvm_sequencer#(REQ,RSP)
uvm_sequencer_base
uvm_sequencer_param_base#(REQ,RSP)
uvm_sqr_if_base#(REQ,RSP)
uvm_subscriber
uvm_table_printer
uvm_task_phase
uvm_test
uvm_test_done_objection
uvm_tlm_analysis_fifo
uvm_tlm_b_initiator_socket
uvm_tlm_b_initiator_socket_base
uvm_tlm_b_passthrough_initiator_socket
uvm_tlm_b_passthrough_initiator_socket_base
uvm_tlm_b_passthrough_target_socket
uvm_tlm_b_passthrough_target_socket_base
uvm_tlm_b_target_socket
uvm_tlm_b_target_socket_base
uvm_tlm_b_transport_export
uvm_tlm_b_transport_imp
uvm_tlm_b_transport_port
uvm_tlm_extension
uvm_tlm_extension_base
uvm_tlm_fifo
uvm_tlm_fifo_base#(T)
uvm_tlm_generic_payload
uvm_tlm_gp
uvm_tlm_if
uvm_tlm_if_base#(T1,T2)
uvm_tlm_nb_initiator_socket
uvm_tlm_nb_initiator_socket_base
uvm_tlm_nb_passthrough_initiator_socket
uvm_tlm_nb_passthrough_initiator_socket_base
uvm_tlm_nb_passthrough_target_socket
uvm_tlm_nb_passthrough_target_socket_base
uvm_tlm_nb_target_socket
uvm_tlm_nb_target_socket_base
uvm_tlm_nb_transport_bw_export
uvm_tlm_nb_transport_bw_imp
uvm_tlm_nb_transport_bw_port
uvm_tlm_nb_transport_fw_export
uvm_tlm_nb_transport_fw_imp
uvm_tlm_nb_transport_fw_port
uvm_tlm_req_rsp_channel#(REQ,RSP)
uvm_tlm_time

File Index

U

Macro Index

$#!

`uvm_field_array_object
`uvm_field_array_string
`uvm_field_enum
`uvm_field_event
`uvm_field_int
`uvm_field_object
`uvm_field_queue_enum
`uvm_field_queue_int
`uvm_field_queue_object
`uvm_field_queue_string
`uvm_field_real
`uvm_field_sarray_enum
`uvm_field_sarray_int
`uvm_field_sarray_object
`uvm_field_sarray_string
`uvm_field_string
`uvm_field_utils_begin
`uvm_field_utils_end
`uvm_get_imp_decl
`uvm_get_peek_imp_decl
`uvm_info
`uvm_info_context
`uvm_master_imp_decl
`UVM_MAX_STREAMBITS
`uvm_nonblocking_get_imp_decl
`uvm_nonblocking_get_peek_imp_decl
`uvm_nonblocking_master_imp_decl
`uvm_nonblocking_peek_imp_decl
`uvm_nonblocking_put_imp_decl
`uvm_nonblocking_slave_imp_decl
`uvm_nonblocking_transport_imp_decl
`uvm_object_param_utils
`uvm_object_param_utils_begin
`uvm_object_registry
`uvm_object_utils
`uvm_object_utils_begin
`uvm_object_utils_end
`uvm_pack_array
`uvm_pack_arrayN
`uvm_pack_enum
`uvm_pack_enumN
`uvm_pack_int
`uvm_pack_intN
`uvm_pack_queue
`uvm_pack_queueN
`uvm_pack_real
`uvm_pack_sarray
`uvm_pack_sarrayN
`uvm_pack_string
`uvm_peek_imp_decl
`uvm_put_imp_decl
`uvm_rand_send
`uvm_rand_send_pri
`uvm_rand_send_pri_with
`uvm_rand_send_with
`uvm_record_attribute
`uvm_record_field
`UVM_REG_ADDR_WIDTH
`UVM_REG_BYTENABLE_WIDTH
`UVM_REG_CVR_WIDTH

Method Index

A

B

C

compose_message
- uvm_report_server

configure
- uvm_mem
- uvm_reg
- uvm_reg_block
- uvm_reg_field
- uvm_reg_file
- uvm_reg_indirect_data
- uvm_reg_map
- uvm_vreg
- uvm_vreg_field

configure_phase
- uvm_component

connect
- uvm_port_base#(IF)
- uvm_tlm_nb_passthrough_target_socket
- uvm_tlm_nb_target_socket

Connect
- uvm_tlm_b_initiator_socket
- uvm_tlm_b_target_socket
- uvm_tlm_nb_initiator_socket

connect_phase
- uvm_component

convert2string
- uvm_mem_mam
- uvm_object
- uvm_reg_item
- uvm_tlm_generic_payload

copy
- uvm_object

create
- uvm_component_registry#(T,Tname)
- uvm_object
- uvm_object_registry#(T,Tname)
- uvm_tlm_extension_base

create_component
- uvm_component
- uvm_component_registry#(T,Tname)
- uvm_object_wrapper

create_component_by_name
- uvm_factory

create_component_by_type
- uvm_factory

create_item
- uvm_sequence_base

create_map
- uvm_reg_block

create_object
- uvm_component
- uvm_object_registry#(T,Tname)
- uvm_object_wrapper

create_object_by_name
- uvm_factory

D

uvm_reg_mem_shared_access_seq

do_bus_read
uvm_reg_map

do_bus_write
uvm_reg_map

do_compare
uvm_object

do_copy
uvm_object
uvm_reg_item

do_end_tr
uvm_component
uvm_transaction

do_kill
uvm_sequence_base

do_kill_all
uvm_component

do_pack
uvm_object

do_post_read
uvm_reg_backdoor

do_post_write
uvm_reg_backdoor

do_pre_read
uvm_reg_backdoor

do_pre_write
uvm_reg_backdoor

do_predict
uvm_reg_fifo

do_print
uvm_object
uvm_resource_base

do_read
uvm_reg_map

do_record
uvm_object

do_reg_item
uvm_reg_sequence

do_unpack
uvm_object

do_write
uvm_reg_map

drop_objection
uvm_objection
uvm_phase
uvm_test_done_objection

dropped
uvm_callbacks_objection
uvm_component
uvm_objection
uvm_objection_callback

dump

E

F

uvm_reg_block

find_blocks
uvm_reg_block

find_override_by_name
uvm_factory

find_override_by_type
uvm_factory

find_unused_resources
uvm_resource_pool

finish_item
uvm_sequence_base
uvm_sequence_item

first
uvm_callback_iter
uvm_pool#(KEY,T)

flush
uvm_in_order_comparator#(T,comp_type,convert,pair_type)
uvm_tlm_fifo

for_each
uvm_mem_mam

force_stop
uvm_test_done_objection

format_action
uvm_report_handler

format_header
uvm_printer

format_row
uvm_printer

G

generate_stimulus
- uvm_random_stimulus#(T)

get
- uvm_component_registry#(T,Tname)
- uvm_config_db#(T)
- uvm_object_registry#(T,Tname)
- uvm_object_string_pool#(T)
- uvm_pool#(KEY,T)
- uvm_queue#(T)
- uvm_reg
- uvm_reg_field
- uvm_reg_fifo
- uvm_resource_pool
- uvm_sqr_if_base#(REQ,RSP)
- uvm_tlm_if_base#(T1,T2)

get_abstime
- uvm_tlm_time

get_accept_time
- uvm_transaction

get_access
- uvm_mem
- uvm_reg_field
- uvm_vreg
- uvm_vreg_field

get_action
- uvm_report_catcher
- uvm_report_handler

get_adapter
- uvm_reg_map

get_address
- uvm_mem
- uvm_reg
- uvm_tlm_generic_payload
- uvm_vreg

get_addresses
- uvm_mem
- uvm_reg

get_arbitration
- uvm_sequencer_base

get_arg_matches
- uvm_cmdline_processor

get_arg_value
- uvm_cmdline_processor

get_arg_values
- uvm_cmdline_processor

get_args
- uvm_cmdline_processor

get_auto_predict
uvm_reg_map

get_backdoor
uvm_mem
uvm_reg
uvm_reg_block

get_base_addr
uvm_reg_map

get_begin_time
uvm_transaction

get_block_by_name
uvm_reg_block

get_blocks
uvm_reg_block

get_by_name
uvm_resource#(T)
uvm_resource_db
uvm_resource_pool

get_by_type
uvm_resource#(T)
uvm_resource_db
uvm_resource_pool

get_byte_enable
uvm_tlm_generic_payload

get_byte_enable_length
uvm_tlm_generic_payload

get_cb
uvm_callback_iter

get_child
uvm_component

get_children
uvm_component

get_client
uvm_report_catcher

get_command
uvm_tlm_generic_payload

get_common_domain
uvm_domain

get_comp
uvm_port_base#(IF)

get_compare
uvm_reg_field

get_config
uvm_utils

get_config_int
uvm_component

get_config_object
uvm_component

get_config_string
uvm_component

get_connected_to
uvm_port_component_base

get_coverage
uvm_mem
uvm_reg
uvm_reg_block

get_current_item
uvm_sequence#(REQ,RSP)
uvm_sequencer_param_base#(REQ,RSP)

get_data
uvm_tlm_generic_payload

get_data_length
uvm_tlm_generic_payload

get_default_hdl_path
uvm_reg_block
uvm_reg_file

get_default_path
uvm_reg_block

get_depth
uvm_component
uvm_sequence_item

get_domain
uvm_component

get_drain_time
uvm_objection

get_end_offset
uvm_mem_region

get_end_time
uvm_transaction

get_event_pool
uvm_transaction

get_extension
uvm_tlm_generic_payload

get_field_by_name
uvm_reg
uvm_reg_block
uvm_vreg

get_fields
uvm_reg
uvm_reg_block
uvm_reg_map
uvm_vreg

get_file_handle
uvm_report_handler

get_first
uvm_callbacks#(T,CB)

get_first_child
uvm_component

get_fname
uvm_report_catcher

get_frontdoor
uvm_mem
uvm_reg

get_full_hdl_path
uvm_mem

uvm_reg
uvm_reg_block
uvm_reg_file

get_full_name
uvm_component
uvm_mem
uvm_object
uvm_port_base#(IF)
uvm_reg
uvm_reg_block
uvm_reg_field
uvm_reg_file
uvm_reg_map
uvm_vreg
uvm_vreg_field

get_global
uvm_pool#(KEY,T)
uvm_queue#(T)

get_global_pool
uvm_object_string_pool#(T)
uvm_pool#(KEY,T)

get_global_queue
uvm_queue#(T)

get_hdl_path
uvm_mem
uvm_reg
uvm_reg_block
uvm_reg_file

get_hdl_path_kinds
uvm_mem
uvm_reg

get_highest_precedence
uvm_resource#(T)
uvm_resource_pool

get_id
uvm_report_catcher

get_id_count
uvm_report_server

get_if
uvm_port_base#(IF)

get_incr
uvm_vreg

get_initiator
uvm_transaction

get_inst
uvm_cmdline_processor

get_inst_count
uvm_object

get_inst_id
uvm_object

get_is_active
uvm_agent

get_jump_target
uvm_phase

get_last
uvm_callbacks#(T,CB)

get_len
uvm_mem_region

get_line
uvm_report_catcher

get_lsb_pos
uvm_reg_field

get_lsb_pos_in_register
uvm_vreg_field

get_map_by_name
uvm_reg_block

get_maps
uvm_mem
uvm_reg
uvm_reg_block
uvm_vreg

get_max_quit_count
uvm_report_server

get_max_size
uvm_mem
uvm_reg
uvm_reg_field

get_mem_by_name
uvm_reg_block

get_mem_by_offset
uvm_reg_map

get_memory
uvm_mem_mam
uvm_mem_region
uvm_vreg

get_message
uvm_report_catcher

get_n_bits
uvm_mem
uvm_reg
uvm_reg_field
uvm_vreg_field

get_n_bytes
uvm_mem
uvm_mem_region
uvm_reg
uvm_reg_map
uvm_vreg

get_n_maps
uvm_mem
uvm_reg
uvm_vreg

get_n_memlocs
uvm_vreg

get_name
uvm_mem
uvm_object
uvm_port_base#(IF)

uvm_reg
uvm_reg_block
uvm_reg_field
uvm_reg_file
uvm_reg_map
uvm_tlm_time
uvm_vreg
uvm_vreg_field

get_next
uvm_callbacks#(T,CB)

get_next_child
uvm_component

get_next_item
uvm_sqr_if_base#(REQ,RSP)

get_num_children
uvm_component

get_num_extensions
uvm_tlm_generic_payload

get_num_last_reqs
uvm_sequencer_param_base#(REQ,RSP)

get_num_last_rsps
uvm_sequencer_param_base#(REQ,RSP)

get_num_reqs_sent
uvm_sequencer_param_base#(REQ,RSP)

get_num_rsps_received
uvm_sequencer_param_base#(REQ,RSP)

get_num_waiters
uvm_barrier
uvm_event

get_object_type
uvm_object

get_objection
uvm_phase

get_objection_count
uvm_objection

get_objection_total
uvm_objection

get_objectors
uvm_objection

get_offset
uvm_mem
uvm_reg

get_offset_in_memory
uvm_vreg

get_packed_size
uvm_packer

get_parent
uvm_component
uvm_mem
uvm_phase
uvm_port_base#(IF)
uvm_reg
uvm_reg_block

uvm_reg_field
uvm_reg_file
uvm_reg_map
uvm_vreg
uvm_vreg_field

get_parent_map
uvm_reg_map

get_parent_sequence
uvm_sequence_item

get_phase_type
uvm_phase

get_physical_addresses
uvm_reg_map

get_plusargs
uvm_cmdline_processor

get_port
uvm_port_component#(PORT)

get_prev
uvm_callbacks#(T,CB)

get_priority
uvm_sequence_base

get_provided_to
uvm_port_component_base

get_quit_count
uvm_report_server

get_radix_str
uvm_printer_knobs

get_realtime
uvm_tlm_time

get_reg_by_name
uvm_reg_block

get_reg_by_offset
uvm_reg_map

get_regfile
uvm_reg
uvm_reg_file

get_region
uvm_vreg

get_registers
uvm_reg_block
uvm_reg_map

get_report_action
uvm_report_object

get_report_catcher
uvm_report_catcher

get_report_file_handle
uvm_report_object

get_report_handler
uvm_report_object

get_report_server
uvm_report_object

get_report_verbosity_level

uvm_report_object

get_reset
uvm_reg
uvm_reg_field

get_response
uvm_sequence#(REQ,RSP)

get_response_queue_depth
uvm_sequence_base

get_response_queue_error_report_disabled
uvm_sequence_base

get_response_status
uvm_tlm_generic_payload

get_response_string
uvm_tlm_generic_payload

get_rights
uvm_mem
uvm_reg
uvm_vreg

get_root_blocks
uvm_reg_block

get_root_map
uvm_reg_map

get_root_sequence
uvm_sequence_item

get_root_sequence_name
uvm_sequence_item

get_run_count
uvm_phase

get_schedule
uvm_component
uvm_phase

get_schedule_name
uvm_phase

get_scope
uvm_resource_base

get_sequence_id
uvm_sequence_item

get_sequence_path
uvm_sequence_item

get_sequence_state
uvm_sequence_base

get_sequencer
uvm_reg_map
uvm_sequence_item

get_server
uvm_report_server

get_severity
uvm_report_catcher

get_severity_count
uvm_report_server

get_size
uvm_mem

uvm_vreg

get_start_offset
uvm_mem_region

get_state
uvm_phase

get_streaming_width
uvm_tlm_generic_payload

get_submap_offset
uvm_reg_map

get_submaps
uvm_reg_map

get_threshold
uvm_barrier

get_tool_name
uvm_cmdline_processor

get_tool_version
uvm_cmdline_processor

get_tr_handle
uvm_transaction

get_transaction_id
uvm_transaction

get_trigger_data
uvm_event

get_trigger_time
uvm_event

get_type
uvm_object
uvm_resource#(T)

get_type_handle
uvm_resource#(T)
uvm_resource_base
uvm_tlm_extension_base

get_type_handle_name
uvm_tlm_extension_base

get_type_name
uvm_callback
uvm_component_registry#(T,Tname)
uvm_object
uvm_object_registry#(T,Tname)
uvm_object_string_pool#(T)
uvm_object_wrapper
uvm_port_base#(IF)

get_use_response_handler
uvm_sequence_base

get_use_sequence_info
uvm_sequence_item

get_uvmargs
uvm_cmdline_processor

get_verbosity
uvm_report_catcher

get_verbosity_level
uvm_report_handler

H

I

uvm_port_base#(IF)
uvm_port_component_base

is_full
uvm_tlm_fifo

is_grabbed
uvm_sequencer_base

is_hdl_path_root
uvm_reg_block

is_imp
uvm_port_base#(IF)
uvm_port_component_base

is_in_map
uvm_mem
uvm_reg
uvm_vreg

is_indv_accessible
uvm_reg_field

is_item
uvm_sequence_base
uvm_sequence_item

is_known_access
uvm_reg_field

is_locked
uvm_reg_block

is_null
uvm_packer

is_off
uvm_event

is_on
uvm_event

is_port
uvm_port_base#(IF)
uvm_port_component_base

is_quit_count_reached
uvm_report_server

is_read
uvm_tlm_generic_payload

is_read_only
uvm_resource_base

is_recording_enabled
uvm_transaction

is_relevant
uvm_sequence_base

is_response_error
uvm_tlm_generic_payload

is_response_ok
uvm_tlm_generic_payload

is_unbounded
uvm_port_base#(IF)

is_volatile
uvm_reg_field

is_write

uvm_tlm_generic_payload

issue
uvm_report_catcher

item_done
uvm_sqr_if_base#(REQ,RSP)

J

jump
uvm_phase

jump_all
uvm_phase

K

kill
uvm_component
uvm_sequence_base

L

last
uvm_callback_iter
uvm_pool#(KEY,T)

last_req
uvm_sequencer_param_base#(REQ,RSP)

last_rsp
uvm_sequencer_param_base#(REQ,RSP)

lock
uvm_resource_base
uvm_sequence_base
uvm_sequencer_base

lock_model
uvm_reg_block

lookup
uvm_component

lookup_name
uvm_resource_pool

lookup_regex
uvm_resource_pool

lookup_regex_names
uvm_resource_pool

lookup_scope
uvm_resource_pool

lookup_type
uvm_resource_pool

M

N

uvm_env
uvm_event
uvm_event_callback
uvm_heartbeat
uvm_mem
uvm_mem_mam
uvm_mem_single_walk_seq
uvm_monitor
uvm_object
uvm_object_string_pool#(T)
uvm_objection
uvm_pair#(T1,T2)
uvm_phase
uvm_pool#(KEY,T)
uvm_port_base#(IF)
uvm_push_driver#(REQ,RSP)
uvm_push_sequencer#(REQ,RSP)
uvm_queue#(T)
uvm_random_stimulus#(T)
uvm_reg
uvm_reg_adapter
uvm_reg_backdoor
uvm_reg_block
uvm_reg_field
uvm_reg_fifo
uvm_reg_file
uvm_reg_frontdoor
uvm_reg_indirect_data
uvm_reg_item
uvm_reg_map
uvm_reg_predictor
uvm_reg_sequence
uvm_report_catcher
uvm_report_handler
uvm_report_object
uvm_report_server
uvm_resource_base
uvm_scoreboard
uvm_seq_item_pull_imp#(REQ,RSP,IMP)
uvm_sequence#(REQ,RSP)
uvm_sequence_base
uvm_sequence_item
uvm_sequencer#(REQ,RSP)
uvm_sequencer_base
uvm_sequencer_param_base#(REQ,RSP)
uvm_subscriber
uvm_task_phase
uvm_test
uvm_test_done_objection
uvm_tlm_analysis_fifo
uvm_tlm_b_initiator_socket
uvm_tlm_b_target_socket
uvm_tlm_extension
uvm_tlm_extension_base
uvm_tlm_fifo
uvm_tlm_fifo_base#(T)
uvm_tlm_generic_payload
uvm_tlm_nb_initiator_socket
uvm_tlm_nb_target_socket
uvm_tlm_nb_transport_bw_export

P

uvm_phase

poke
uvm_mem
uvm_mem_region
uvm_reg
uvm_reg_field
uvm_vreg
uvm_vreg_field

poke_mem
uvm_reg_sequence

poke_reg
uvm_reg_sequence

pop_back
uvm_queue#(T)

pop_front
uvm_queue#(T)

post_body
uvm_sequence_base

post_configure_phase
uvm_component

post_do
uvm_sequence_base

post_main_phase
uvm_component

post_predict
uvm_reg_cbs

post_read
uvm_mem
uvm_reg
uvm_reg_backdoor
uvm_reg_cbs
uvm_reg_field
uvm_vreg
uvm_vreg_cbs
uvm_vreg_field
uvm_vreg_field_cbs

post_reset_phase
uvm_component

post_shutdown_phase
uvm_component

post_trigger
uvm_event_callback

post_write
uvm_mem
uvm_reg
uvm_reg_backdoor
uvm_reg_cbs
uvm_reg_field
uvm_vreg
uvm_vreg_cbs
uvm_vreg_field
uvm_vreg_field_cbs

pre_abort
uvm_component

pre_body
uvm_sequence_base

pre_configure_phase
uvm_component

pre_do
uvm_sequence_base

pre_main_phase
uvm_component

pre_predict
uvm_reg_predictor

pre_read
uvm_mem
uvm_reg
uvm_reg_backdoor
uvm_reg_cbs
uvm_reg_field
uvm_reg_fifo
uvm_reg_write_only_cbs
uvm_vreg
uvm_vreg_cbs
uvm_vreg_field
uvm_vreg_field_cbs

pre_reset_phase
uvm_component

pre_shutdown_phase
uvm_component

pre_trigger
uvm_event_callback

pre_write
uvm_mem
uvm_reg
uvm_reg_backdoor
uvm_reg_cbs
uvm_reg_field
uvm_reg_fifo
uvm_reg_read_only_cbs
uvm_vreg
uvm_vreg_cbs
uvm_vreg_field
uvm_vreg_field_cbs

predict
uvm_reg
uvm_reg_field

prev
uvm_callback_iter
uvm_pool#(KEY,T)

print
uvm_factory
uvm_object

print_accessors
uvm_resource_base

print_array_footer
uvm_printer

print_array_header
uvm_printer

print_array_range
uvm_printer

print_catcher
uvm_report_catcher

print_config
uvm_component

print_config_settings
uvm_component

print_config_with_audit
uvm_component

print_generic
uvm_printer

print_int
uvm_printer

print_msg
uvm_comparer

print_object
uvm_printer

print_override_info
uvm_component

print_resources
uvm_resource_pool

print_string
uvm_printer

print_time
uvm_printer

print_topology
uvm_root

process_report
uvm_report_server

push_back
uvm_queue#(T)

push_front
uvm_queue#(T)

put
uvm_sqr_if_base#(REQ,RSP)
uvm_tlm_if_base#(T1,T2)

Q

R

record_error_tr
uvm_component

record_event_tr
uvm_component

record_field
uvm_recorder

record_field_real
uvm_recorder

record_generic
uvm_recorder

record_object
uvm_recorder

record_string
uvm_recorder

record_time
uvm_recorder

reg2bus
uvm_reg_adapter
uvm_reg_tlm_adapter

register
uvm_factory

release_all_regions
uvm_mem_mam

release_region
uvm_mem_mam
uvm_mem_region
uvm_vreg

remove
uvm_heartbeat
uvm_reg_read_only_cbs
uvm_reg_write_only_cbs

report
uvm_report_handler

report_error_hook
uvm_report_object

report_fatal_hook
uvm_report_object

report_header
uvm_report_object

report_hook
uvm_report_object

report_info_hook
uvm_report_object

report_phase
uvm_component

report_summarize
uvm_report_object

report_warning_hook
uvm_report_object

request_region
uvm_mem_mam

reseed

uvm_object

reserve_region
uvm_mem_mam

reset
uvm_barrier
uvm_event
uvm_reg
uvm_reg_block
uvm_reg_field
uvm_reg_map
uvm_tlm_time
uvm_vreg

reset_blk
uvm_mem_access_seq
uvm_mem_walk_seq
uvm_reg_access_seq
uvm_reg_bit_bash_seq
uvm_reg_hw_reset_seq
uvm_reg_mem_shared_access_seq

reset_phase
uvm_component

reset_quit_count
uvm_report_server

reset_report_handler
uvm_report_object

reset_severity_counts
uvm_report_server

resolve_bindings
uvm_component
uvm_port_base#(IF)

response_handler
uvm_sequence_base

resume
uvm_component

run_hooks
uvm_report_handler

run_phase
uvm_component
uvm_push_sequencer#(REQ,RSP)

run_test
Global
uvm_root

S

set_byte_enable
uvm_tlm_generic_payload

set_byte_enable_length
uvm_tlm_generic_payload

set_command
uvm_tlm_generic_payload

set_compare
uvm_reg_field
uvm_reg_fifo

set_config_int
Global
uvm_component

set_config_object
Global
uvm_component

set_config_string
Global
uvm_component

set_coverage
uvm_mem
uvm_reg
uvm_reg_block

set_data
uvm_tlm_generic_payload

set_data_length
uvm_tlm_generic_payload

set_default
uvm_resource_db

set_default_hdl_path
uvm_reg_block
uvm_reg_file

set_default_index
uvm_port_base#(IF)

set_default_map
uvm_reg_block

set_depth
uvm_sequence_item

set_dmi_allowed
uvm_tlm_generic_payload

set_domain
uvm_component

set_drain_time
uvm_objection

set_extension
uvm_tlm_generic_payload

set_frontdoor
uvm_mem
uvm_reg

set_global_stop_timeout
set_global_timeout
set_hdl_path_root
uvm_reg_block

set_heartbeat
uvm_heartbeat

set_id
uvm_report_catcher

set_id_count
uvm_report_server

set_id_info
uvm_sequence_item

set_initiator
uvm_transaction

set_inst_override
uvm_component
uvm_component_registry#(T,Tname)
uvm_object_registry#(T,Tname)

set_inst_override_by_name
uvm_factory

set_inst_override_by_type
uvm_component
uvm_factory

set_int_local
uvm_object

set_max_quit_count
uvm_report_server

set_message
uvm_report_catcher

set_mode
uvm_heartbeat

set_name
uvm_component
uvm_object

set_name_override
uvm_resource_pool

set_num_last_reqs
uvm_sequencer_param_base#(REQ,RSP)

set_num_last_rsps
uvm_sequencer_param_base#(REQ,RSP)

set_object_local
uvm_object

set_offset
uvm_mem
uvm_reg

set_override
uvm_resource#(T)
uvm_resource_pool

set_parent_sequence
uvm_sequence_item

set_phase_imp
uvm_component

set_priority
uvm_resource_pool
uvm_sequence_base

set_priority_name

uvm_resource_pool

set_priority_type
uvm_resource_pool

set_quit_count
uvm_report_server

set_read
uvm_tlm_generic_payload

set_read_only
uvm_resource_base

set_report_default_file
uvm_report_object

set_report_default_file_hier
uvm_component

set_report_handler
uvm_report_object

set_report_id_action
uvm_report_object

set_report_id_action_hier
uvm_component

set_report_id_file
uvm_report_object

set_report_id_file_hier
uvm_component

set_report_id_verbosity
uvm_report_object

set_report_id_verbosity_hier
uvm_component

set_report_max_quit_count
uvm_report_object

set_report_severity_action
uvm_report_object

set_report_severity_action_hier
uvm_component

set_report_severity_file
uvm_report_object

set_report_severity_file_hier
uvm_component

set_report_severity_id_action
uvm_report_object

set_report_severity_id_action_hier
uvm_component

set_report_severity_id_file
uvm_report_object

set_report_severity_id_file_hier
uvm_component

set_report_severity_id_override
uvm_report_object

set_report_severity_id_verbosity
uvm_report_object

set_report_severity_id_verbosity_hier
uvm_component

set_report_severity_override
uvm_report_object

set_report_verbosity_level
uvm_report_object

set_report_verbosity_level_hier
uvm_component

set_reset
uvm_reg
uvm_reg_field

set_response_queue_depth
uvm_sequence_base

set_response_queue_error_report_disabled
uvm_sequence_base

set_response_status
uvm_tlm_generic_payload

set_scope
uvm_resource_base

set_sequencer
uvm_reg_map
uvm_sequence_item

set_server
uvm_report_server

set_severity
uvm_report_catcher

set_severity_count
uvm_report_server

set_streaming_width
uvm_tlm_generic_payload

set_string_local
uvm_object

set_submap_offset
uvm_reg_map

set_threshold
uvm_barrier

set_time_resolution
uvm_tlm_time

set_timeout
uvm_root

set_transaction_id
uvm_transaction

set_type_override
uvm_component
uvm_component_registry#(T,Tname)
uvm_object_registry#(T,Tname)
uvm_resource_pool

set_type_override_by_name
uvm_factory

set_type_override_by_type
uvm_component
uvm_factory

set_use_sequence_info
uvm_sequence_item

T

trace_mode
uvm_objection

transport
uvm_tlm_if_base#(T1,T2)

traverse
uvm_bottomup_phase
uvm_task_phase
uvm_topdown_phase

trigger
uvm_event

try_get
uvm_tlm_if_base#(T1,T2)

try_lock
uvm_resource_base

try_next_item
uvm_sqr_if_base#(REQ,RSP)

try_peek
uvm_tlm_if_base#(T1,T2)

try_put
uvm_tlm_if_base#(T1,T2)

try_read_with_lock
uvm_resource#(T)

try_write_with_lock
uvm_resource#(T)

turn_off_auditing
uvm_resource_options

turn_on_auditing
uvm_resource_options

U

W

uvm_resource_base

wait_off
uvm_event

wait_on
uvm_event

wait_ptrigger
uvm_event

wait_ptrigger_data
uvm_event

wait_trigger
uvm_event

wait_trigger_data
uvm_event

write
uvm_analysis_port
uvm_mem
uvm_mem_region
uvm_reg
uvm_reg_backdoor
uvm_reg_field
uvm_reg_fifo
uvm_resource#(T)
uvm_subscriber
uvm_tlm_if_base#(T1,T2)
uvm_vreg
uvm_vreg_field

write_by_name
uvm_resource_db

write_by_type
uvm_resource_db

write_mem
uvm_reg_sequence

write_mem_by_name
uvm_reg_block

write_reg
uvm_reg_sequence

write_reg_by_name
uvm_reg_block

write_with_lock
uvm_resource#(T)

Type Index

A

alloc_mode_e
uvm_mem_mam

L

locality_e
uvm_mem_mam

U

uvm_access_e
uvm_action
uvm_active_passive_enum
uvm_bitstream_t
uvm_check_e
uvm_coverage_model_e
uvm_elem_kind_e
uvm_endianness_e
uvm_hdl_path_slice
uvm_hier_e
uvm_mem_cb
uvm_mem_cb_iter
uvm_objection_event
uvm_path_e
uvm_phase_state
uvm_phase_transition
uvm_phase_type
uvm_port_type_e
uvm_predict_e
uvm_radix_enum
uvm_recursion_policy_enum
uvm_reg_addr_logic_t
uvm_reg_addr_t
uvm_reg_bd_cb
uvm_reg_bd_cb_iter
uvm_reg_byte_en_t
uvm_reg_cb
uvm_reg_cb_iter
uvm_reg_cvr_t
uvm_reg_data_logic_t
uvm_reg_data_t
uvm_reg_field_cb
uvm_reg_field_cb_iter
uvm_reg_mem_tests_e
uvm_sequence_lib_mode
uvm_sequence_state_enum

uvm_sequencer_arb_mode
uvm_severity
uvm_status_e
uvm_tlm_command_e
uvm_tlm_phase_e
uvm_tlm_response_status_e
uvm_tlm_sync_e
uvm_verbosity
uvm_vreg_cb
uvm_vreg_cbs
uvm_vreg_cb_iter
uvm_vreg_cbs
uvm_vreg_field_cb
uvm_vreg_field_cbs
uvm_vreg_field_cb_iter
uvm_vreg_field_cbs
uvm_wait_op

Variable Index

$#!

A

B

C

D

E

F

H

I

K

L

M

N

O

P

R

S

T

U

V

Constant Index

B

BEGIN_REQ
BEGIN_RESP
BODY

C

CREATED

E

END_REQ
END_RESP
ENDED

F

FINISHED

P

POST_BODY
PRE_BODY

S

SEQ_ARB_FIFO
SEQ_ARB_RANDOM
SEQ_ARB_STRICT_FIFO
SEQ_ARB_STRICT_RANDOM
SEQ_ARB_USER
SEQ_ARB_WEIGHTED
STOPPED

U

UNINITIALIZED_PHASE

UVM_ALL_DROPPED
UVM_BACKDOOR
UVM_BIG_ENDIAN
UVM_BIG_FIFO
UVM_BIN
UVM_CALL_HOOK
UVM_CHECK
UVM_COMPLETED
UVM_COUNT
UVM_CVR_ADDR_MAP
UVM_CVR_ALL
UVM_CVR_FIELD_VALS
UVM_CVR_REG_BITS
UVM_DEC
UVM_DEEP
UVM_DEFAULT_PATH
UVM_DISPLAY
UVM_DO_ALL_REG_MEM_TESTS
UVM_DO_MEM_ACCESS
UVM_DO_MEM_WALK
UVM_DO_REG_ACCESS
UVM_DO_REG_BIT_BASH
UVM_DO_REG_HW_RESET
UVM_DO_SHARED_ACCESS
UVM_DROPPED
UVM_ENUM
UVM_EQ
UVM_ERROR
UVM_EXIT
UVM_EXPORT
UVM_FATAL
UVM_FIELD
UVM_FORCED_STOP
UVM_FRONTDOOR
UVM_FULL
UVM_GT
UVM_GTE
UVM_HAS_X
UVM_HEX
UVM_HIER
UVM_HIGH
UVM_IMPLEMENTATION
UVM_INFO
UVM_IS_OK
UVM_LITTLE_ENDIAN
UVM_LITTLE_FIFO
UVM_LOG
UVM_LOW
UVM_LT
UVM_LTE
UVM_MEDIUM
UVM_MEM
UVM_NE
UVM_NO_ACTION
UVM_NO_CHECK
UVM_NO_COVERAGE
UVM_NO_ENDIAN
UVM_NO_HIER
UVM_NONE
UVM_NOT_OK

UVM_OCT
UVM_PHASE_BOTTOMUP
UVM_PHASE_CLEANUP
UVM_PHASE_DOMAIN_NODE
UVM_PHASE_DONE
UVM_PHASE_DORMANT
UVM_PHASE_ENDED
UVM_PHASE_ENDSCHEDULE_NODE
UVM_PHASE_EXECUTING
UVM_PHASE_READY_TO_END
UVM_PHASE_SCHEDULE_NODE
UVM_PHASE_SCHEDULED
UVM_PHASE_STARTED
UVM_PHASE_SYNCING
UVM_PHASE_TASK
UVM_PHASE_TOPDOWN
UVM_PORT
UVM_PREDICT
UVM_PREDICT_DIRECT
UVM_PREDICT_READ
UVM_PREDICT_WRITE
UVM_RAISED
UVM_READ
UVM_REFERENCE
UVM_REG
UVM_RERUN
UVM_SEQ_LIB_ITEM
UVM_SEQ_LIB_RAND
UVM_SEQ_LIB_RANDC
UVM_SEQ_LIB_USER
UVM_SHALLOW
UVM_SKIPPED
UVM_STOP
UVM_STRING
UVM_TIME
UVM_TLM_ACCEPTED
UVM_TLM_ADDRESS_ERROR_RESPONSE
UVM_TLM_BURST_ERROR_RESPONSE
UVM_TLM_BYTE_ENABLE_ERROR_RESPONSE
UVM_TLM_COMMAND_ERROR_RESPONSE
UVM_TLM_COMPLETED
UVM_TLM_GENERIC_ERROR_RESPONSE
UVM_TLM_IGNORE_COMMAND
UVM_TLM_INCOMPLETE_RESPONSE
UVM_TLM_OK_RESPONSE
UVM_TLM_READ_COMMAND
UVM_TLM_UPDATED
UVM_TLM_WRITE_COMMAND
UVM_UNSIGNED
UVM_WARNING
UVM_WRITE

Port Index

A

B

G

M

P

put_export
uvm_tlm_fifo_base#(T)

put_request_export
uvm_tlm_req_rsp_channel#(REQ,RSP)

put_response_export
uvm_tlm_req_rsp_channel#(REQ,RSP)

R

req_export
uvm_push_driver#(REQ,RSP)

req_port
uvm_push_sequencer#(REQ,RSP)

request_ap
uvm_tlm_req_rsp_channel#(REQ,RSP)

response_ap
uvm_tlm_req_rsp_channel#(REQ,RSP)

rsp_export
uvm_sequencer_param_base#(REQ,RSP)

rsp_port
uvm_driver#(REQ,RSP)
uvm_push_driver#(REQ,RSP)

S

seq_item_port
uvm_driver#(REQ,RSP)

slave_export
uvm_tlm_req_rsp_channel#(REQ,RSP)

T

transport_export
uvm_tlm_transport_channel#(REQ,RSP)