



# **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](mailto: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](mailto:vip-tc@lists.accellera.org)

The current Working Group's website address is

[www.accellera.org/activities/vip](http://www.accellera.org/activities/vip)

## Contents

Overview .....	1
Scope .....	1
Purpose .....	1
Normative References .....	2
Definitions, Acronyms, and Abbreviations .....	2
Definitions .....	2
Acronyms and Abbreviations .....	3
Classes and Utilities .....	5
Base .....	8
Overview .....	8
uvm_void .....	9
uvm_object .....	10
uvm_transaction .....	24
uvm_root .....	30
uvm_port_base .....	33
Phasing .....	41
Reporting .....	65
Overview .....	65
uvm_report_object .....	66
uvm_report_handler .....	75
uvm_report_server .....	78
uvm_report_catcher .....	82
Factory .....	88
Overview .....	88
uvm_*_registry .....	89
uvm_factory .....	95
Configuration and Resources .....	106
Overview .....	106
uvm_resource .....	107
uvm_resource_db .....	125
uvm_config_db .....	128
Sequencers .....	131
Overview .....	131
uvm_sequencer_base .....	133
uvm_sequencer_param_base .....	139

uvm_sequencer . . . . .	143
uvm_push_sequencer . . . . .	145
Sequences . . . . .	147
Overview . . . . .	147
uvm_sequence_item . . . . .	148
uvm_sequence_base . . . . .	153
uvm_sequence . . . . .	165
Synchronization . . . . .	167
Overview . . . . .	167
uvm_event . . . . .	168
uvm_event_callback . . . . .	172
uvm_barrier . . . . .	174
uvm_objection . . . . .	177
uvm_heartbeat . . . . .	187
Containers . . . . .	190
Overview . . . . .	190
uvm_pool . . . . .	191
uvm_queue . . . . .	196
TLM . . . . .	199
Overview . . . . .	199
TLM1 . . . . .	201
Overview . . . . .	201
Interfaces . . . . .	209
Ports . . . . .	213
Exports . . . . .	216
Imps . . . . .	219
Analysis Ports . . . . .	222
FIFO . . . . .	225
FIFO Base . . . . .	229
Request-Response Channel . . . . .	232
TLM2 . . . . .	237
Overview . . . . .	237
Generic Payload . . . . .	239
Interfaces . . . . .	253
Sockets . . . . .	257
Ports . . . . .	264

Exports . . . . .	266
Imps . . . . .	268
Macros . . . . .	272
Socket Base . . . . .	273
Temporal Decoupling . . . . .	277
Sequencer Ports . . . . .	282
uvm_seq_item_pull_port . . . . .	282
uvm_sqr_if_base . . . . .	284
Components . . . . .	288
Overview . . . . .	288
uvm_component . . . . .	289
uvm_callback . . . . .	317
uvm_test . . . . .	325
uvm_env . . . . .	327
uvm_agent . . . . .	328
uvm_monitor . . . . .	330
uvm_scoreboard . . . . .	331
uvm_driver . . . . .	332
uvm_push_driver . . . . .	334
uvm_random_stimulus . . . . .	336
uvm_subscriber . . . . .	338
Comparators . . . . .	340
Overview . . . . .	340
uvm_in_order_comparator . . . . .	341
uvm_algorithmic_comparator . . . . .	344
uvm_pair . . . . .	347
uvm_policies . . . . .	349
Macros . . . . .	352
Report Macros . . . . .	352
Component and Object . . . . .	355
Sequence and Do Action . . . . .	377
Callbacks . . . . .	382
TLM . . . . .	386
Registers . . . . .	391

Policies . . . . .	392
Overview . . . . .	392
uvm_printer . . . . .	393
uvm_comparer . . . . .	404
uvm_recorder . . . . .	408
uvm_packer . . . . .	411
Register Layer . . . . .	416
Overview . . . . .	416
Globals . . . . .	417
Register Model . . . . .	424
Blocks . . . . .	424
Address Maps . . . . .	439
Register Files . . . . .	448
Registers . . . . .	452
Fields . . . . .	469
Memories . . . . .	480
Indirect Registers . . . . .	495
FIFO Registers . . . . .	497
Virtual Registers . . . . .	501
Virtual Fields . . . . .	513
Callbacks . . . . .	521
Memory Allocation Mgr . . . . .	529
DUT Integration . . . . .	539
Generic Register Operation Descriptors . . . . .	539
Register Model Adaptor . . . . .	545
Register Sequences . . . . .	549
Backdoors . . . . .	560
HDL Access . . . . .	564
Test Sequences . . . . .	566
Run All Built-In . . . . .	566
Reset . . . . .	568
Register Bit Bash . . . . .	570
Register Access . . . . .	573
Shared Access . . . . .	577
Memory Access . . . . .	582

Memory Walk . . . . .	585
HDL Paths Checking. . . . .	589
Command Line Processor . . . . .	591
Overview . . . . .	591
uvm_cmdline_processor . . . . .	592
Globals . . . . .	599
Types, Enums, Policies . . . . .	599
Globals. . . . .	607
Bibliography. . . . .	612
Index. . . . .	613



## 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<sup>TM</sup>), 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.<sup>1, 2</sup>

## Definitions, Acronyms, and Abbreviations

For the purposes of this document, the following terms and definitions apply. The *IEEE Standards Dictionary: Glossary of Terms & Definitions*<sup>3</sup> 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 **transactions** from another **component**.

**driver:** A component responsible for executing or otherwise processing **transactions**, 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**.

---

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

<sup>2</sup>The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

<sup>3</sup>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 **transactions** to be executed and/or controls the execution of other sequences.

**sequencer:** An advanced stimulus generator which executes **sequences** that define the **transactions** 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 **components**.

**transactor:** See *component*.

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

## 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.

<i>Globals</i>	This category defines a small list of types, variables, functions, and tasks defined in the <i>uvm_pkg</i> scope. These items are accessible from any scope that imports the <i>uvm_pkg</i> . See <a href="#">Types and Enumerations</a> and <a href="#">Globals</a> for details.
<i>Base</i>	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 <i>base</i> classes provide these building blocks. See <a href="#">Core Base Classes</a> for more information.
<i>Reporting</i>	The <i>reporting</i> 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 <a href="#">Reporting Classes</a> for more information.
<i>Factory</i>	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 <a href="#">Factory Classes</a> for details.
<i>Configuration and Resources</i>	The <a href="#">Configuration and Resource Classes</a> 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.
<i>Synchronization</i>	The UVM provides event and barrier synchronization classes for process synchronization. See <a href="#">Synchronization Classes</a> for more information.
<i>Containers</i>	The <a href="#">Container Classes</a> 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

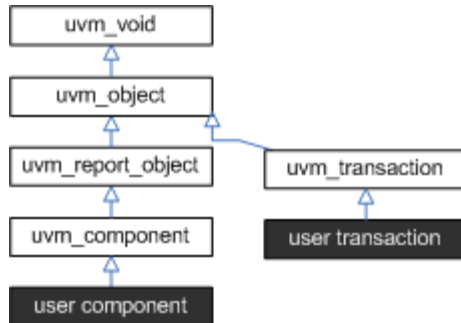
### UVM Class Reference

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

<code>convert2string</code>	provided by the <code>`uvm_field_*</code> macros, <b>Utility and Field Macros for Components and Objects</b> . 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.
<b>RECORDING</b>	
<code>record</code>	The record method deep-records this object's properties according to an optional <i>recorder</i> policy.
<code>do_record</code>	The <code>do_record</code> method is the user-definable hook called by the <b>record</b> method.
<b>COPYING</b>	
<code>copy</code>	The copy method returns a deep copy of this object.
<code>do_copy</code>	The <code>do_copy</code> method is the user-definable hook called by the copy method.
<b>COMPARING</b>	
<code>compare</code>	Deep compares members of this data object with those of the object provided in the <i>rhs</i> (right-hand side) argument, returning 1 on a match, 0 otherwise.
<code>do_compare</code>	The <code>do_compare</code> method is the user-definable hook called by the <b>compare</b> method.
<b>PACKING</b>	
<code>pack</code> <code>pack_bytes</code> <code>pack_ints</code>	The pack methods bitwise-concatenate this object's properties into an array of bits, bytes, or ints.
<code>do_pack</code>	The <code>do_pack</code> method is the user-definable hook called by the <b>pack</b> methods.
<b>UNPACKING</b>	
<code>unpack</code> <code>unpack_bytes</code> <code>unpack_ints</code>	The unpack methods extract property values from an array of bits, bytes, or ints.
<code>do_unpack</code>	The <code>do_unpack</code> method is the user-definable hook called by the <b>unpack</b> method.
<b>CONFIGURATION</b>	
<code>set_int_local</code> <code>set_string_local</code> <code>set_object_local</code>	These methods provide write access to integral, string, and <code>uvm_object</code> -based properties indexed by a <i>field_name</i> 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
```

```
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(),
      $sprintf(" 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 otherwise.

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
    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

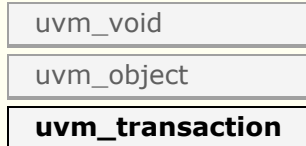
The `uvm_transaction` class is the root base class for UVM transactions. Inheriting all the methods of `uvm_object`, `uvm_transaction` adds a timing and recording interface.

## Summary

### uvm\_transaction

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

#### CLASS HIERARCHY



#### CLASS DECLARATION

```
virtual class uvm_transaction extends uvm_object
```

#### METHODS

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



## VARIABLES

<code>events</code>	The event pool instance for this transaction.
<code>begin_event</code>	The event that is triggered when transaction recording for this transaction begins.
<code>end_event</code>	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

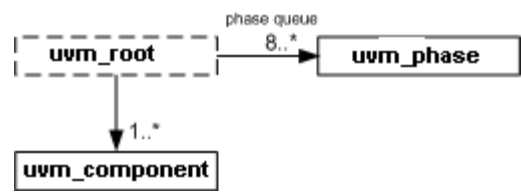
---

```
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.

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

### Port Base Classes

<a href="#">uvm_port_component_base</a>	This class defines an interface for obtaining a port's connectivity lists after or during the end_of_elaboration phase.
<a href="#">uvm_port_component #(PORT)</a>	See description of <a href="#">uvm_port_component_base</a> for information about this class
<a href="#">uvm_port_base #(IF)</a>	Transaction-level communication between components is handled via its ports, exports, and imps, all of which derive from this class.

## 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.

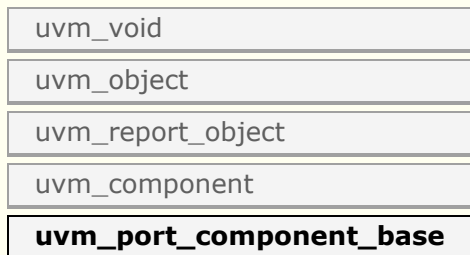
The connectivity lists are returned in the form of handles to objects of this type. This allowing traversal of any port's fan-out and fan-in network through recursive calls to [get\\_connected\\_to](#) and [get\\_provided\\_to](#). Each port's full name and type name can be retrieved using [get\\_full\\_name](#) 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



#### CLASS DECLARATION

```
virtual class uvm_port_component_base extends  
uvm_component
```

#### METHODS

<a href="#">get_connected_to</a>	For a port or export type, this function fills <i>list</i> with all of the ports, exports and implementations that this port is connected to.
<a href="#">get_provided_to</a>	For an implementation or export type, this function fills <i>list</i> with all of the ports, exports and implementations that this port 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



#### CLASS DECLARATION

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

#### METHODS

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

<b>Phasing</b>	UVM implements an automated mechanism for phasing the execution of the various components in a testbench.
<a href="#">Pre-Defined Phases</a>	This section describes the set of pre-defined phases provided as a standard part of the UVM library.
<a href="#">User-Defined Phases</a>	To defined your own custom phase, use the following pattern
<a href="#">Phasing Implementation</a>	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.
<a href="#">uvm_phase</a>	This base class defines everything about a phase: behavior, state, and context
<a href="#">uvm_domain</a>	Phasing schedule node representing an independent branch of the schedule.
<a href="#">uvm_bottomup_phase</a>	Virtual base class for function phases that operate bottom-up.
<a href="#">uvm_topdown_phase</a>	Virtual base class for function phases that operate top-down.
<a href="#">uvm_task_phase</a>	Base class for all task phases.

## Pre-Defined Phases

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

## Summary

### Pre-Defined Phases

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

#### COMMON PHASES GLOBAL VARIABLES

The common phases are the set of function and task phases that all **uvm\_components** execute together.

<a href="#">build_ph</a>	Create and configure of testbench structure
<a href="#">connect_ph</a>	Establish cross-component connections.
<a href="#">end_of_elaboration_ph</a>	Fine-tune the testbench.
<a href="#">start_of_simulation_ph</a>	Get ready for DUT to be simulated.
<a href="#">run_ph</a>	Stimulate the DUT.
<a href="#">extract_ph</a>	Extract data from different points of the verification environment.
<a href="#">check_ph</a>	Check for any unexpected conditions in the verification environment.
<a href="#">report_ph</a>	Report results of the test.
<a href="#">final_ph</a>	Tie up loose ends.

#### RUN-TIME SCHEDULE

The run-time schedule is the pre-defined phase

**GLOBAL VARIABLES**

schedule which runs concurrently to the `run_ph` global run phase.

<code>pre_reset_ph</code>	Before reset is asserted.
<code>reset_ph</code>	Reset is asserted.
<code>post_reset_ph</code>	After reset is de-asserted.
<code>pre_configure_ph</code>	Before the DUT is configured by the SW.
<code>configure_ph</code>	The SW configures the DUT.
<code>post_configure_ph</code>	After the SW has configured the DUT.
<code>pre_main_ph</code>	Before the primary test stimulus starts.
<code>main_ph</code>	Primary test stimulus.
<code>post_main_ph</code>	After enough of the primary test stimulus.
<code>pre_shutdown_ph</code>	Before things settle down.
<code>shutdown_ph</code>	Letting things settle down.
<code>post_shutdown_ph</code>	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 regardless 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 verification 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

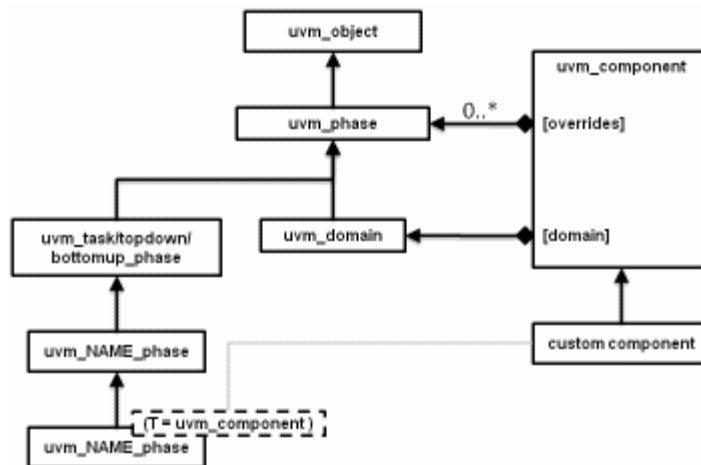
## Phasing Implementation

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 implementation for bottom up function phases.

`uvm_topdown_phase` : A phase implementation for topdown function phases.

`uvm_task_phase` : A phase implementation 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 more 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

<code>phase_started</code>	functionality upon phase - the phase schedule that originated this phase call Generic notification function called prior to <code>exec_func()/exec_task()</code> phase - the phase schedule that originated this phase call
<code>phase_ended</code>	Generic notification function called after <code>exec_func()/exec_task()</code> phase - the phase schedule that originated this phase call
<b>SCHEDULE</b>	
<code>add_phase</code>	Build up a schedule structure inserting phase by phase, specifying linkage
<code>add_schedule</code>	Build up schedule structure by adding another schedule flattened within it.
<code>get_parent</code>	Returns the parent schedule node, if any, for hierarchical graph traversal
<code>get_schedule</code>	Returns the topmost parent schedule node, if any, for hierarchical graph traversal
<code>get_schedule_name</code>	Accessor to return the schedule name associated with this schedule
<b>SYNCHRONIZATION</b>	
<code>get_objection</code>	Return the <code>uvm_objection</code> that gates the termination of the phase.
<code>raise_objection</code>	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.
<code>drop_objection</code>	Drop an objection to ending this phase
<code>sync</code>	Synchronize two domains, fully or partially
<code>unsync</code>	Remove synchronization between two domains, fully or partially
<code>wait_for_state</code>	Wait until this phase compares with the given <i>state</i> and <i>op</i> operand.
<b>JUMPING</b>	
<code>jump</code>	Jump to a specified <i>phase</i> .
<code>jump_all</code>	Make all schedules jump to a specified <i>phase</i> , even if the jump target is local.
<code>get_jump_target</code>	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`

### 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

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

<i>schedule</i>	handle of new schedule to insert within this one
<i>with_phase</i>	specify to add the schedule in parallel with this phase node
<i>after_phase</i>	specify to add the schedule as successor to this phase node
<i>before_phase</i>	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 )

```

Synchronize 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 synchronization 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*.

### 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](#)

`execute`

for each component.

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.

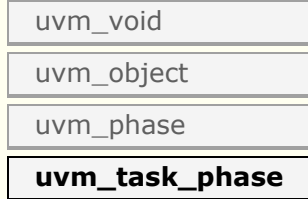
```
class my_comp extends uvm_component;  
  task main_phase(uvm_phase phase);  
    phase.raise_objection(this, "Applying stimulus")  
    ...  
    phase.drop_objection(this, "Applied enough stimulus")  
  endtask  
endclass
```

## Summary

## uvm\_task\_phase

Base class for all task phases.

### CLASS HIERARCHY



### 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)
```

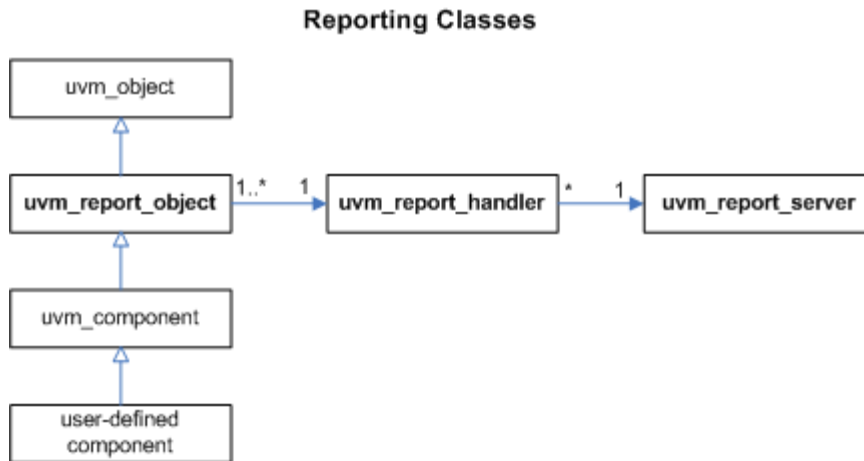
Fork the task-based phase *phase* for the component *comp*.



# 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.

<i>Actions</i>	can be set for (in increasing priority) severity, id, and (severity,id) pair. They include output to the screen <code>UVM_DISPLAY</code> , whether the message counters should be incremented <code>UVM_COUNT</code> , and whether a \$finish should occur <code>UVM_EXIT</code> .
<i>Default Actions</i>	The following provides the default actions assigned to each severity. These can be overridden by any of the <code>set_*_action</code> methods.

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

<i>File descriptors</i>	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.
<i>Default file handle</i>	The default file handle is 0, which means that reports are not sent to a file even if an <code>UVM_LOG</code> attribute is set in the action associated with the report. This can be overridden by any of the <code>set_*_file</code> 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  
report\_summarize

Prints version and copyright information. 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*. 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.

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

`get_report_handler`

default instance.

`reset_report_handler`

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

`get_report_server`

Resets the underlying report handler to its default settings.

`dump_report_state`

Returns the `uvm_report_server` instance associated with this report object.

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:

<i>id</i>	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.
<i>message</i>	the message body, preformatted if necessary to a single string.
<i>verbosity</i>	the verbosity of the message, indicating its relative importance. If this number is less than or equal to the effective verbosity level, see <a href="#">set_report_verbosity_level</a> , 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 <a href="#">uvm_report_catcher</a> , then the verbosity is taken into account.
<i>filename/line</i>	(Optional) The location from which the report was issued. Use the predefined macros, <code>__FILE__</code> and <code>__LINE__</code> . 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 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 takes precedence over 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

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

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

## 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)
```

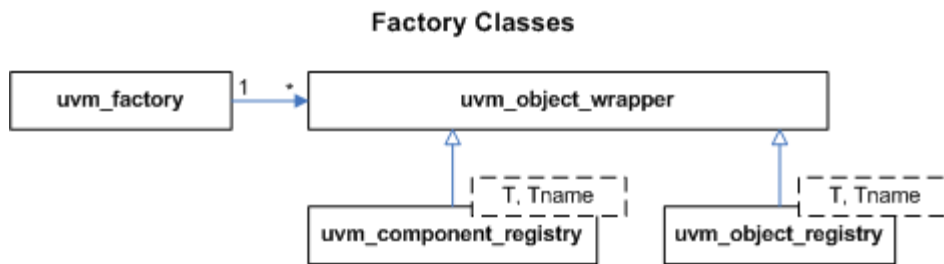
This function is called automatically by `uvm_report_server::summarize()`. It prints the statistics for the active catchers.

# 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

### Factory Component and Object Wrappers

This section defines the proxy component and object classes used by the factory.

[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.

[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.

## 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

`set_inst_override`

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. 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      ctxt = "")
```

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 *ctxt* 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 = null)
```

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**

uvm\_object\_wrapper

uvm\_object\_registry#(T,Tname)

**CLASS DECLARATION**

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

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

### set\_inst\_override

#### USAGE

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.

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.

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      ctxtxt   = "")
```

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 *ctxtxt* 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 = null)
```

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 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

<b>UVM Factory</b>	This page covers the classes that define the UVM factory facility.
<a href="#">uvm_factory</a>	As the name implies, <code>uvm_factory</code> is used to manufacture (create) UVM objects and components.
<a href="#">uvm_object_wrapper</a>	The <code>uvm_object_wrapper</code> provides an abstract interface for creating object and component proxies.

## 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.

<i>type-based</i>	The type-based interface is far less prone to errors in usage. When errors do occur, they are caught at compile-time.
<i>name-based</i>	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_path,
                                           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_path
                                                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.

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 registering 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                         driverD #(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

<code>create_object</code>	Creates a new object with the optional <i>name</i> .
<code>create_component</code>	Creates a new component, passing to its constructor the given <i>name</i> and <i>parent</i> .
<code>get_type_name</code>	Derived classes implement this method to return the type name of the object created by <code>create_component</code> or <code>create_object</code> .

## 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 database 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 returned 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.

Multiple 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 safe.

**uvm\_resource\_pool**: the resource database. This is a singleton class object.

## Contents

## Resources

A resource is a parameterized container that holds arbitrary data.

<a href="#">uvm_resource_types</a>	Provides typedefs and enums used throughout the resources facility.
<a href="#">uvm_resource_options</a>	Provides a namespace for managing options for the resources facility.
<a href="#">uvm_resource_base</a>	Non-parameterized base class for resources.
<a href="#">uvm_resource_pool</a>	The global (singleton) resource database.
<a href="#">uvm_resource #(T)</a>	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

<a href="#">turn_on_auditing</a>	Turn auditing on for the resource database.
<a href="#">turn_off_auditing</a>	Turn auditing off for the resource database.
<a href="#">is_auditing</a>	Returns 1 if the auditing facility is on and 0 if it is off.

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

default\_precedence

new

get\_type\_handle

This variable is used to associate a precedence that a resource has with respect to other resources which match the same scope and name.

The default precedence for an resource that has been created.

constructor for uvm\_resource\_base.

Pure virtual function that returns the type handle of the

	resource container.
<b>LOCKING INTERFACE</b>	The task <code>lock</code> and the functions <code>try_lock</code> and <code>unlock</code> form a locking interface for resources.
<code>lock</code>	Retrieves a lock for this resource.
<code>try_lock</code>	Retrives the lock for this resource.
<code>unlock</code>	Releases the lock held by this semaphore.
<b>READ-ONLY INTERFACE</b>	
<code>set_read_only</code>	Establishes this resource as a read-only resource.
<code>is_read_only</code>	Retruns one if this resource has been set to read-only, zero otherwise
<b>NOTIFICATION</b>	
<code>wait_modified</code>	This task blocks until the resource has been modified -- that is, a <code>uvm_resource#(T)::write</code> operation has been performed.
<b>SCOPE INTERFACE</b>	Each resource has a name, a value and a set of scopes over which it is visible.
<code>set_scope</code>	Set the value of the regular expression that identifies the set of scopes over which this resource is visible.
<code>get_scope</code>	Retrieve the regular expression string that identifies the set of scopes over which this resource is visible.
<code>match_scope</code>	Using the regular expression facility, determine if this resource is visible in a scope.
<b>PRIORITY</b>	Functions for manipulating the search priority of resources.
<code>set priority</code>	Change the search priority of the resource based on the value of the priority enum argument.
<b>UTILITY FUNCTIONS</b>	
<code>do_print</code>	Implementation of <code>do_print</code> which is called by <code>print()</code> .
<b>AUDIT TRAIL</b>	To find out what is happening as the simulation proceeds, an audit trail of each read and write is kept.
<code>print_accessors</code>	Dump the access records for this resource
<code>init_access_record</code>	Inititalize 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()
```

Retrieves the lock for this resource. The function is nonblocking, so it will return immediately. If it was successful 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:

<code>top\..*</code>	all of the scopes whose top-level component is top
<code>top\.env\..*\monitor</code>	all of the scopes in env that end in monitor; i.e. all the monitors two levels down from env
<code>.*\monitor</code>	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")
<code>top\.u[1-5]\.*</code>	all of the scopes rooted and named u1, u2, u3, u4, or u5, and any of their subsopes.

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 are 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 definitions 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 be 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

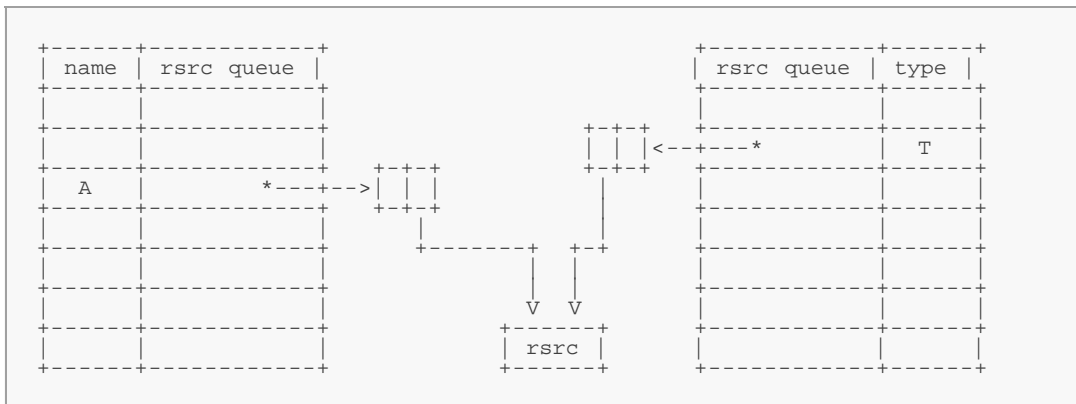
```
function void init_access_record (  
    inout uvm_resource_types::access_t access_record  
)
```

Initialize 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 which 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 itself; 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 incorrectly 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 be 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 normal 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 resource 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 parameter 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](#) traverse 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 resource 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 resources, 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, leaving 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](#)

[get\\_type\\_handle](#)

Static function that returns the static 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](#)

[set\\_override](#)

Simply put this resource into the global resource pool

Put a resource into the global resource pool as an override.

[get\\_by\\_name](#)

[get\\_by\\_type](#)

looks up a resource by *name* in the name map.

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](#)

[write](#)

Return the object stored in the resource container.

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\\_lock](#)

[try\\_read\\_with\\_lock](#)

[write\\_with\\_lock](#)

[try\\_write\\_with\\_lock](#)

[get\\_highest\\_precedence](#)

Locking version of read().

Nonblocking form of read\_with\_lock().

Locking form of write().

Nonblocking form of write\_with\_lock().

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 function 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\_lock;

---

Locking version of read(). Like read(), this returns the contents of the resource container. In addition 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 available 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

<a href="#">get_by_type</a>	Get a resource by type.
<a href="#">get_by_name</a>	Imports a resource by <i>name</i> .
<a href="#">set_default</a>	add a new item into the resources database.
<a href="#">set</a>	Create a new resource, write a <i>val</i> to it, and set it into the database using <i>name</i> and <i>scope</i> as the lookup parameters.
<a href="#">set_anonymous</a>	Create a new resource, write a <i>val</i> to it, and set it into the database.
<a href="#">read_by_name</a>	locate a resource by <i>name</i> and <i>scope</i> and read its value.
<a href="#">read_by_type</a>	Read a value by type.
<a href="#">write_by_name</a>	write a <i>val</i> into the resources database.
<a href="#">write_by_type</a>	write a <i>val</i> into the resources database.
<a href="#">dump</a>	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 auditing.

## 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 it does have a *scope* for lookup purposes. The *accessor* is used for auditing.

## 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 auditing.

## 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 auditing.

## write\_by\_name

---

```
static function bit write_by_name(input string    scope,
                                input string    name,
                                input 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

<code>get</code>	Get the value <i>field_name</i> in <i>inst_name</i> , using component <i>cntxt</i> as the starting search point.
<code>set</code>	Create a new or update an existing configuration setting for <i>field_name</i> in <i>inst_name</i> from <i>cntxt</i> .
<code>exists</code>	Check if a value for <i>field_name</i> is available in <i>inst_name</i> , using component <i>cntxt</i> as the starting search point.
<code>wait_modified</code>	Wait for a configuration setting to be set for <i>field_name</i> in <i>cntxt</i> and <i>inst_name</i> .

## 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 hierarchical 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 = 0
                          )
```

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

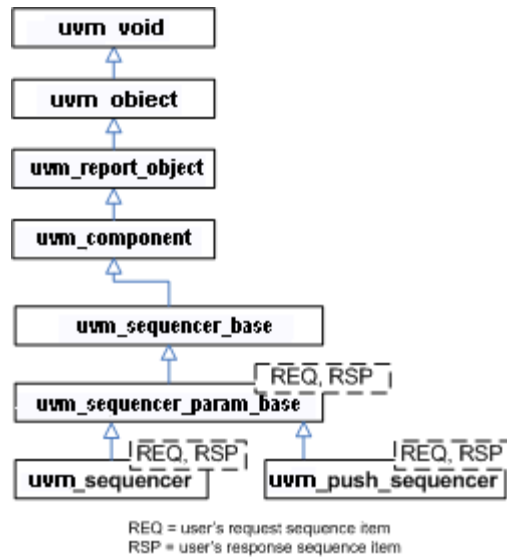
---

```
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.

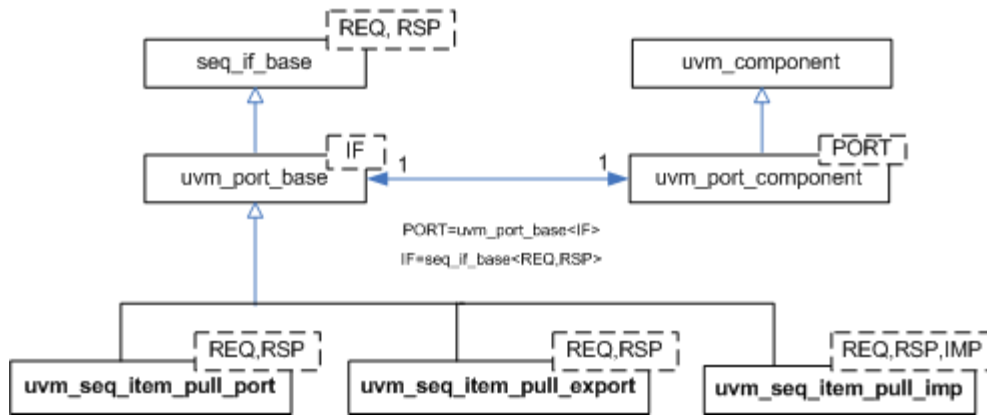
See [Sequence Classes](#) for an overview on sequences and sequence items.

## 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.

## Sequence item port, export, and imp



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

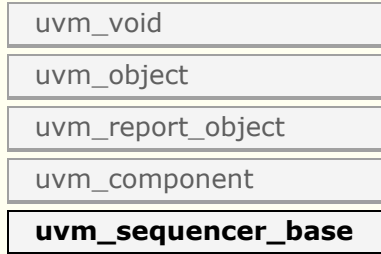
Controls the flow of sequences, which generate the stimulus (sequence item transactions) that is passed on to drivers for execution.

## 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



#### CLASS DECLARATION

```
class uvm_sequencer_base extends uvm_component
```

#### METHODS

<code>new</code>	Creates and initializes an instance of this class using the normal constructor arguments for <code>uvm_component</code> : <code>name</code> is the name of the instance, and <code>parent</code> is the handle to the hierarchical parent.
<code>is_child</code>	Returns 1 if the child sequence is a child of the parent sequence, 0 otherwise.
<code>user_priority_arbitration</code>	When the sequencer arbitration mode is set to <code>SEQ_ARB_USER</code> (via the <code>set_arbitration</code> method), the sequencer will call this function each time that it needs to arbitrate among sequences.
<code>execute_item</code>	This task allows the user to supply an item or sequence to the sequencer and have it be executed procedurally.
<code>start_phase_sequence</code>	Start the default sequence for this phase, if any.
<code>wait_for_grant</code>	This task issues a request for the specified sequence.
<code>wait_for_item_done</code>	A sequence may optionally call <code>wait_for_item_done</code> .
<code>is_blocked</code>	Returns 1 if the sequence referred to by <code>sequence_ptr</code> is currently locked out of the sequencer.
<code>has_lock</code>	Returns 1 if the sequence referred to in the parameter currently has a lock on this sequencer, 0 otherwise.
<code>lock</code>	Requests a lock for the sequence specified by <code>sequence_ptr</code> .
<code>grab</code>	Requests a lock for the sequence specified by <code>sequence_ptr</code> .
<code>unlock</code>	Removes any locks and grabs obtained by the specified <code>sequence_ptr</code> .
<code>ungrab</code>	Removes any locks and grabs obtained by the specified <code>sequence_ptr</code> .
<code>stop_sequences</code>	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.

<code>is_grabbed</code>	Returns 1 if any sequence currently has a lock or grab on this sequencer, 0 otherwise.
<code>current_grabber</code>	Returns a reference to the sequence that currently has a lock or grab on the sequence.
<code>has_do_available</code>	Returns 1 if any sequence running on this sequencer is ready to supply a transaction, 0 otherwise.
<code>set_arbitration</code>	Specifies the arbitration mode for the sequencer.
<code>get_arbitration</code>	Return the current arbitration mode set for this sequencer.
<code>wait_for_sequences</code>	Waits for a sequence to have a new item available.
<code>send_request</code>	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 referred 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 issuing 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

<i>SEQ_ARB_FIFO</i>	Requests are granted in FIFO order (default)
<i>SEQ_ARB_WEIGHTED</i>	Requests are granted randomly by weight
<i>SEQ_ARB_RANDOM</i>	Requests are granted randomly
<i>SEQ_ARB_STRICT_FIFO</i>	Requests at highest priority granted in fifo order
<i>SEQ_ARB_STRICT_RANDOM</i>	Requests at highest priority granted in randomly
<i>SEQ_ARB_USER</i>	Arbitration is delegated to the user-defined function, <i>user_priority_arbitration</i> . 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)

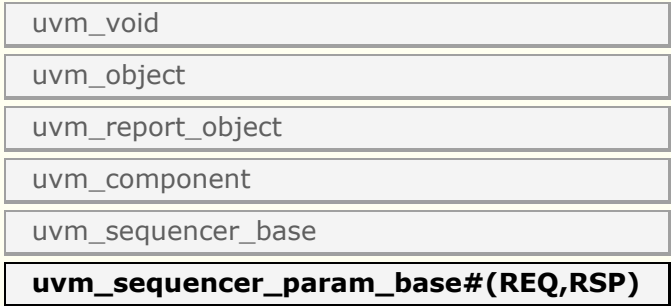
Extends [uvm\\_sequencer\\_base](#) with an API depending on specific request (REQ) and response (RSP) types.

## 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



#### CLASS DECLARATION

```
class uvm_sequencer_param_base #(
    type REQ = uvm_sequence_item,
    type RSP = REQ
) extends uvm_sequencer_base
```

<a href="#">new</a>	Creates and initializes an instance of this class using the normal constructor arguments for <a href="#">uvm_component</a> : name is the name of the instance, and parent is the handle to the hierarchical parent, if any.
<a href="#">send_request</a>	The <a href="#">send_request</a> function may only be called after a <a href="#">wait_for_grant</a> call.
<a href="#">get_current_item</a>	Returns the <a href="#">request_item</a> currently being executed by the sequencer.

#### REQUESTS

<a href="#">get_num_reqs_sent</a>	Returns the number of requests that have been sent by this sequencer.
<a href="#">set_num_last_reqs</a> <a href="#">get_num_last_reqs</a>	Sets the size of the <a href="#">last_requests</a> buffer. Returns the size of the last requests buffer, as set by <a href="#">set_num_last_reqs</a> .
<a href="#">last_req</a>	Returns the last request item by default.

#### RESPONSES

<a href="#">rsp_export</a>	Drivers or monitors can connect to this port to send responses to the sequencer.
<a href="#">get_num_rsps_received</a>	Returns the number of responses received thus far by this sequencer.
<a href="#">set_num_last_rsps</a> <a href="#">get_num_last_rsps</a>	Sets the size of the <a href="#">last_responses</a> buffer. Returns the max size of the last responses buffer, as set by <a href="#">set_num_last_rsps</a> .
<a href="#">last_rsp</a>	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 requested.

# 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 *n*th-before-

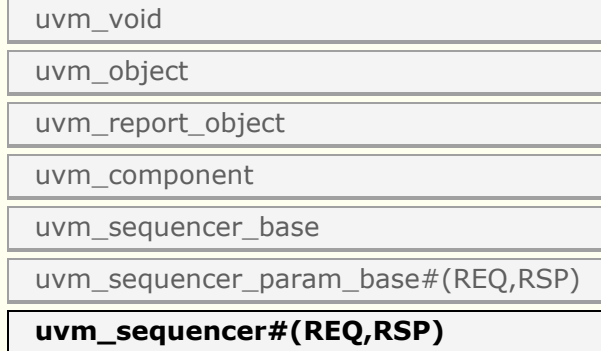
last response item. If n is greater than the last response buffer size, the function will return null.

# uvm\_sequencer #(REQ,RSP)

## Summary

### uvm\_sequencer #(REQ,RSP)

#### CLASS HIERARCHY



#### 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.

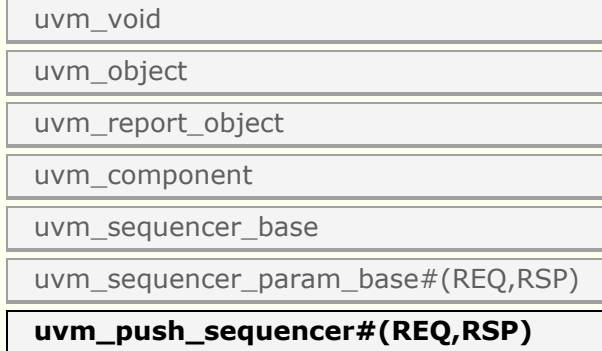


# uvm\_push\_sequencer #(REQ,RSP)

## Summary

### uvm\_push\_sequencer #(REQ,RSP)

#### CLASS HIERARCHY



#### 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)
```

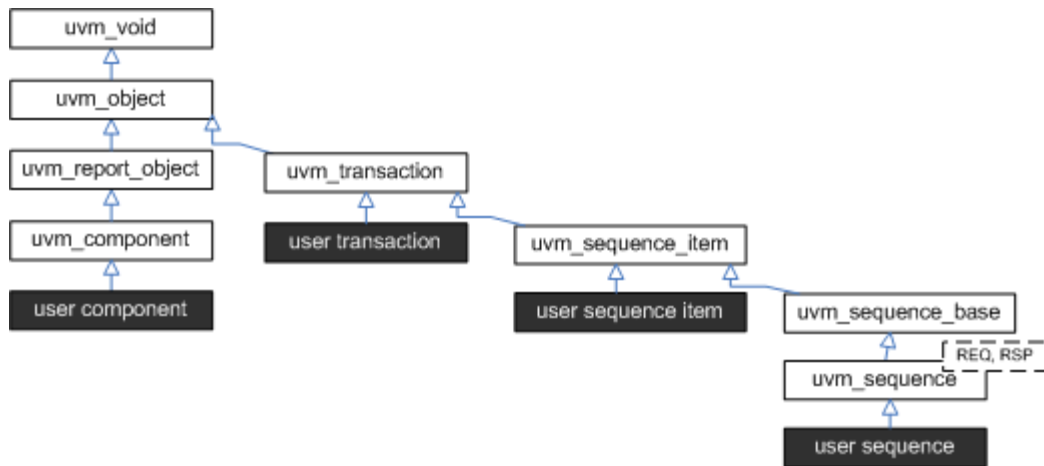
The push sequencer continuously selects from its list of available sequences and sends the next item from the selected sequence out its [req\\_port](#) using `req_port.put(item)`. Typically, the `req_port` would be connected to the `req_export` on an instance of an [uvm\\_push\\_driver #\(REQ,RSP\)](#), 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 initialization 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 executing other *uvm\_sequences*.

## Summary

### Sequence Classes

Sequences encapsulate user-defined procedures that generate multiple [uvm\\_sequence\\_item](#)-based transactions.

# uvm\_sequence\_item

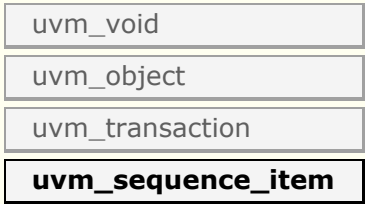
The base class for user-defined sequence items and also the base class for the uvm\_sequence class. The uvm\_sequence\_item class provides the basic functionality for objects, both sequence items and sequences, to operate in the sequence mechanism.

## Summary

### uvm\_sequence\_item

The base class for user-defined sequence items and also the base class for the uvm\_sequence class.

#### CLASS HIERARCHY



#### 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 its parent. A parent sequence will have a depth of 1, its child will have a depth of 2, and its 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_seq.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

<code>is_item</code>	<p><code>uvm_sequence_base</code>. Returns 1 on items and 0 on sequences.</p>
<code>get_sequence_state</code>	Returns the sequence state as an enumerated value.
<code>wait_for_sequence_state</code>	Waits until the sequence reaches the given <i>state</i> .
<b>SEQUENCE EXECUTION</b>	
<code>start</code>	Executes this sequence, returning when the sequence has completed.
<code>pre_body</code>	This task is a user-definable callback that is called before the execution of <b>body</b> <i>only</i> when the sequence is started with <b>start</b> .
<code>pre_do</code>	This task is a user-definable callback task that is called <i>on the parent sequence</i> , if any, the sequence has issued a <code>wait_for_grant()</code> call and after the sequencer has selected this sequence, and before the item is randomized.
<code>mid_do</code>	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.
<code>body</code>	This is the user-defined task where the main sequence code resides.
<code>post_do</code>	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 <code>item_done</code> or <code>put</code> methods.
<code>post_body</code>	This task is a user-definable callback task that is called after the execution of <b>body</b> <i>only</i> when the sequence is started with <b>start</b> .
<code>starting_phase</code>	If non-null, specifies the phase in which this sequence was started.
<b>SEQUENCE CONTROL</b>	
<code>set_priority</code>	The priority of a sequence may be changed at any point in time.
<code>get_priority</code>	This function returns the current priority of the sequence.
<code>is_relevant</code>	The default <code>is_relevant</code> implementation returns 1, indicating that the sequence is always relevant.
<code>wait_for_relevant</code>	This method is called by the sequencer when all available sequences are not relevant.
<code>lock</code>	Requests a lock on the specified sequencer.
<code>grab</code>	Requests a lock on the specified sequencer.
<code>unlock</code>	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\_response\_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 method 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 *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 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

---

```
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_response_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

<a href="#">new</a>	Creates and initializes a new sequence object.
<a href="#">send_request</a>	This method will send the request item to the sequencer, which will forward it to the driver.
<a href="#">get_current_item</a>	Returns the request item currently being executed by the sequencer.
<a href="#">get_response</a>	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 received.

If a `transaction_id` 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.

- [uvme\\_event](#) - UVM's event class augments the SystemVerilog event datatype with such services as setting callbacks and data delivery.
- [uvme\\_barrier](#) - A barrier is used to prevent a pre-configured number of processes from continuing until all have reached a certain point in simulation.
- `uvme_event_pool` and `uvme_barrier_pool` - The event and barrier pool classes are specializations of [uvme\\_object\\_string\\_pool #\(T\)](#) used to store collections of [uvme\\_events](#) and [uvme\\_barriers](#), respectively, indexed by string name. Each pool class contains a static, "global" pool instance for sharing across all processes.
- [uvme\\_event\\_callback](#) - The event callback is used to create callback objects that may be attached to [uvme\\_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



#### CLASS DECLARATION

```
class uvm_event extends uvm_object
```

#### METHODS

<code>new</code>	Creates a new event object.
<code>wait_on</code>	Waits for the event to be activated for the first time.
<code>wait_off</code>	If the event has already triggered and is "on", this task waits for the event to be turned "off" via a call to <code>reset</code> .
<code>wait_trigger</code>	Waits for the event to be triggered.
<code>wait_pttrigger</code>	Waits for a persistent trigger of the event.
<code>wait_trigger_data</code>	This method calls <code>wait_trigger</code> followed by <code>get_trigger_data</code> .
<code>wait_pttrigger_data</code>	This method calls <code>wait_pttrigger</code> followed by <code>get_trigger_data</code> .
<code>trigger</code>	Triggers the event, resuming all waiting processes.
<code>get_trigger_data</code>	Gets the data, if any, provided by the last call to <code>trigger</code> .
<code>get_trigger_time</code>	Gets the time that this event was last triggered.
<code>is_on</code>	Indicates whether the event has been triggered since it was last reset.
<code>is_off</code>	Indicates whether the event has been triggered or been reset.
<code>reset</code>	Resets the event to its off state.
<code>add_callback</code>	Registers a callback object, <code>cb</code> , with this event.
<code>delete_callback</code>	Unregisters the given callback, <code>cb</code> , from this event.
<code>cancel</code>	Decrements the number of waiters on the event.
<code>get_num_waiters</code>	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 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\\_pttrigger](#) 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



#### CLASS DECLARATION

```
virtual class uvm_event_callback extends uvm_object
```

#### METHODS

<a href="#">new</a>	Creates a new callback object.
<a href="#">pre_trigger</a>	This callback is called just before triggering the associated event.
<a href="#">post_trigger</a>	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

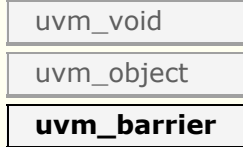
The uvm\_barrier class provides a multiprocess synchronization mechanism. It enables a set of processes to block until the desired number of processes get to the synchronization point, at which time all of the processes are released.

## Summary

### uvm\_barrier

The uvm\_barrier class provides a multiprocess synchronization mechanism.

#### CLASS HIERARCHY



#### CLASS DECLARATION

```
class uvm_barrier extends uvm_object
```

#### METHODS

<a href="#">new</a>	Creates a new barrier object.
<a href="#">wait_for</a>	Waits for enough processes to reach the barrier before continuing.
<a href="#">reset</a>	Resets the barrier.
<a href="#">set_auto_reset</a>	Determines if the barrier should reset itself after the threshold is reached.
<a href="#">set_threshold</a>	Sets the process threshold.
<a href="#">get_threshold</a>	Gets the current threshold setting for the barrier.
<a href="#">get_num_waiters</a>	Returns the number of processes currently waiting at the barrier.
<a href="#">cancel</a>	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

<b>Objection Mechanism</b>	The following classes define the objection mechanism and end-of-test functionality, which is based on <code>uvm_objection</code> .
<code>uvm_objection</code>	Objections provide a facility for coordinating status information between two or more participating components, objects, and even module-based IP.
<code>uvm_test_done_objection</code>	Provides built-in end-of-test coordination
<code>uvm_callbacks_objection</code>	The <code>uvm_callbacks_objection</code> is a specialized <code>uvm_objection</code> which contains callbacks for the raised and dropped events.
<code>uvm_objection_callback</code>	The <code>uvm_objection</code> is the callback type that defines the callback implementations for an objection callback.

# 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

```
graph TD; uvm_void --> uvm_object; uvm_void --> uvm_report_object; uvm_report_object --> uvm_objection;
```

uvm\_void

uvm\_object

uvm\_report\_object

**uvm\_objection**

## CLASS DECLARATION

```
class uvm_objection extends uvm_report_object
```

<code>new</code>	Creates a new objection instance.
<code>trace_mode</code>	Set or get the trace mode for the objection object.

### OBJECTION CONTROL

<code>m_set_hier_mode</code>	Hierarchical mode only needs to be set for intermediate components, not for <code>uvm_root</code> or a leaf component.
<code>raise_objection</code>	Raises the number of objections for the source <i>object</i> by <i>count</i> , which defaults to 1.
<code>drop_objection</code>	Drops the number of objections for the source <i>object</i> by <i>count</i> , which defaults to 1.
<code>set_drain_time</code>	Sets the drain time on the given <i>object</i> to <i>drain</i> .

### CALLBACK HOOKS

<code>raised</code>	Objection callback that is called when a <code>raise_objection</code> has reached <i>obj</i> .
<code>dropped</code>	Objection callback that is called when a <code>drop_objection</code> has reached <i>obj</i> .
<code>all_dropped</code>	Objection callback that is called when a <code>drop_objection</code> has reached <i>obj</i> , and the total count for <i>obj</i> goes to zero.

### OBJECTION STATUS

<code>get_objectors</code>	Returns the current list of objecting objects (objects that raised an objection but have not dropped it).
<code>wait_for</code>	Waits for the raised, dropped, or all_dropped <i>event</i> to occur in the given <i>obj</i> .
<code>get_objection_count</code>	Returns the current number of objections raised by the given <i>object</i> .
<code>get_objection_total</code>	Returns the current number of objections raised by the given <i>object</i> and all descendants.
<code>get_drain_time</code>	Returns the current drain time set for the given <i>object</i> (default: 0 ns).
<code>display_objections</code>	Displays objection information about the given <i>object</i> .

## 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.

Raising 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 descendants 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 propagated 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

<code>new</code>	Creates the singleton test_done objection.
<code>qualify</code>	Checks that the given <i>object</i> is derived from either <code>uvm_component</code> or <code>uvm_sequence_base</code> .
<code>stop_request</code>	Calling this function triggers the process of shutting down the currently running task-based phase.

#### VARIABLES

<code>stop_timeout</code>	These set watchdog timers for task-based phases and stop tasks.
---------------------------	---

#### METHODS

<code>all_dropped</code>	This callback is called when the given <i>object's</i> objection count reaches zero; if the <i>object</i> is the implicit top-level, <code>uvm_root</code> then it means there are no more objections raised for the <code>uvm_test_done</code> objection.
<code>raise_objection</code>	Calls <code>uvm_objection::raise_objection</code> after calling <code>qualify</code> .
<code>drop_objection</code>	Calls <code>uvm_objection::drop_objection</code> after calling <code>qualify</code> .
<code>force_stop</code>	Forces the propagation of the <code>all_dropped()</code> 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 happen 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

```

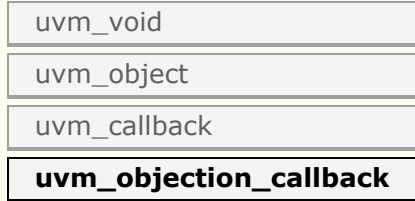
## Summary

### uvm\_objection\_callback

The `uvm_objection` is the callback type that defines the callback implementations

for an objection callback.

#### CLASS HIERARCHY



#### CLASS DECLARATION

```
class uvm_objection_callback extends uvm_callback
```

#### METHODS

<code>raised</code>	Objection raised callback function.
<code>dropped</code>	Objection dropped callback function.
<code>all_dropped</code>	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

<code>new</code>	Creates a new heartbeat instance associated with <i>cntxt</i> .
<code>set_mode</code>	Sets or retrieves the heartbeat mode.
<code>set_heartbeat</code>	Sets up the heartbeat event and assigns a list of objects to watch.
<code>add</code>	Add a single component to the set of components to be monitored.
<code>remove</code>	Remove a single component to the set of components being monitored.
<code>start</code>	Starts the heartbeat monitor.
<code>stop</code>	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

<b>Pool Classes</b>	This section defines the <code>&lt;uvm_pool #(T)&gt;</code> class and derivative.
<code>uvm_pool #(KEY,T)</code> <code>uvm_object_string_pool #(T)</code>	Implements a class-based dynamic associative array. This provides a specialization of the generic <code>uvm_pool #(KEY,T)</code> class for an associative array of <code>uvm_object</code> -based objects indexed by string.

# 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

<b>uvm_pool #(KEY,T)</b>
Implements a class-based dynamic associative array.
<b>CLASS HIERARCHY</b>
<div><div>uvm_void</div><div>uvm_object</div><div><b>uvm_pool#(KEY,T)</b></div></div>
<b>CLASS DECLARATION</b>
<pre>class uvm_pool #(type KEY = int,                   T = uvm_void) extends uvm_object</pre>
<b>METHODS</b>
<div><div><code>new</code></div><div><code>get_global_pool</code></div><div><code>get_global</code></div><div><code>get</code></div><div><code>add</code></div><div><code>num</code></div><div><code>delete</code></div><div><code>exists</code></div><div><code>first</code></div><div><code>last</code></div><div><code>next</code></div><div><code>prev</code></div></div> <div><div>Creates a new pool with the given <i>name</i>.</div><div>Returns the singleton global pool for the item type, T.</div><div>Returns the specified item instance from the global item pool.</div><div>Returns the item with the given <i>key</i>.</div><div>Adds the given (<i>key</i>, <i>item</i>) pair to the pool.</div><div>Returns the number of uniquely keyed items stored in the pool.</div><div>Removes the item with the given <i>key</i> from the pool.</div><div>Returns 1 if a item with the given <i>key</i> exists in the pool, 0 otherwise.</div><div>Returns the key of the first item stored in the pool.</div><div>Returns the key of the last item stored in the pool.</div><div>Returns the key of the next item in the pool.</div><div>Returns the key of the previous item in the pool.</div></div>

### 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\_object\_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

<a href="#">new</a>	Creates a new pool with the given <i>name</i> .
<a href="#">get_type_name</a>	Returns the type name of this object.
<a href="#">get_global_pool</a>	Returns the singleton global pool for the item type, T.
<a href="#">get</a>	Returns the object item at the given string <i>key</i> .
<a href="#">delete</a>	Removes the item with the given string <i>key</i> 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)

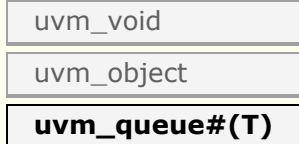
Implements a class-based dynamic queue. Allows queues to be allocated on demand, and passed and stored by reference.

## Summary

### uvm\_queue #(T)

Implements a class-based dynamic queue.

#### CLASS HIERARCHY



#### CLASS DECLARATION

```
class uvm_queue #(type T = int) extends uvm_object
```

#### METHODS

<code>new</code>	Creates a new queue with the given <i>name</i> .
<code>get_global_queue</code>	Returns the singleton global queue for the item type, T.
<code>get_global</code>	Returns the specified item instance from the global item queue.
<code>get</code>	Returns the item at the given <i>index</i> .
<code>size</code>	Returns the number of items stored in the queue.
<code>insert</code>	Inserts the item at the given <i>index</i> in the queue.
<code>delete</code>	Removes the item at the given <i>index</i> from the queue; if <i>index</i> is not provided, the entire contents of the queue are deleted.
<code>pop_front</code>	Returns the first element in the queue ( <i>index</i> =0), or <i>null</i> if the queue is empty.
<code>pop_back</code>	Returns the last element in the queue ( <i>index</i> =size()-1), or <i>null</i> if the queue is empty.
<code>push_front</code>	Inserts the given <i>item</i> at the front of the queue.
<code>push_back</code>	Inserts the given <i>item</i> 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 <i>analysis</i> 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

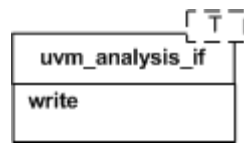
The TLM1 ports provide blocking and nonblocking pass-by-value transaction-level interfaces. The semantics of these interfaces are limited to message passing.

## 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.





# TLM1 Interfaces, Ports, Exports and Transport Interfaces

Each TLM1 interface is either blocking, non-blocking, or a combination of these two.

<i>blocking</i>	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.
<i>non-blocking</i>	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.
<i>combination</i>	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 an 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 Ports,

The primitive, unidirectional *put*, *get*, and *peek* interfaces are combined to form bidirectional master and slave interfaces. 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 interface.

## UNIDIRECTIONAL INTERFACES & PORTS

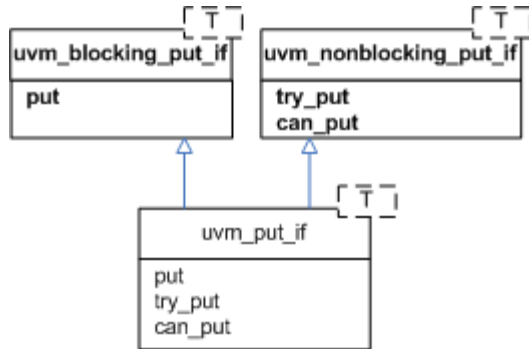
---

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

---

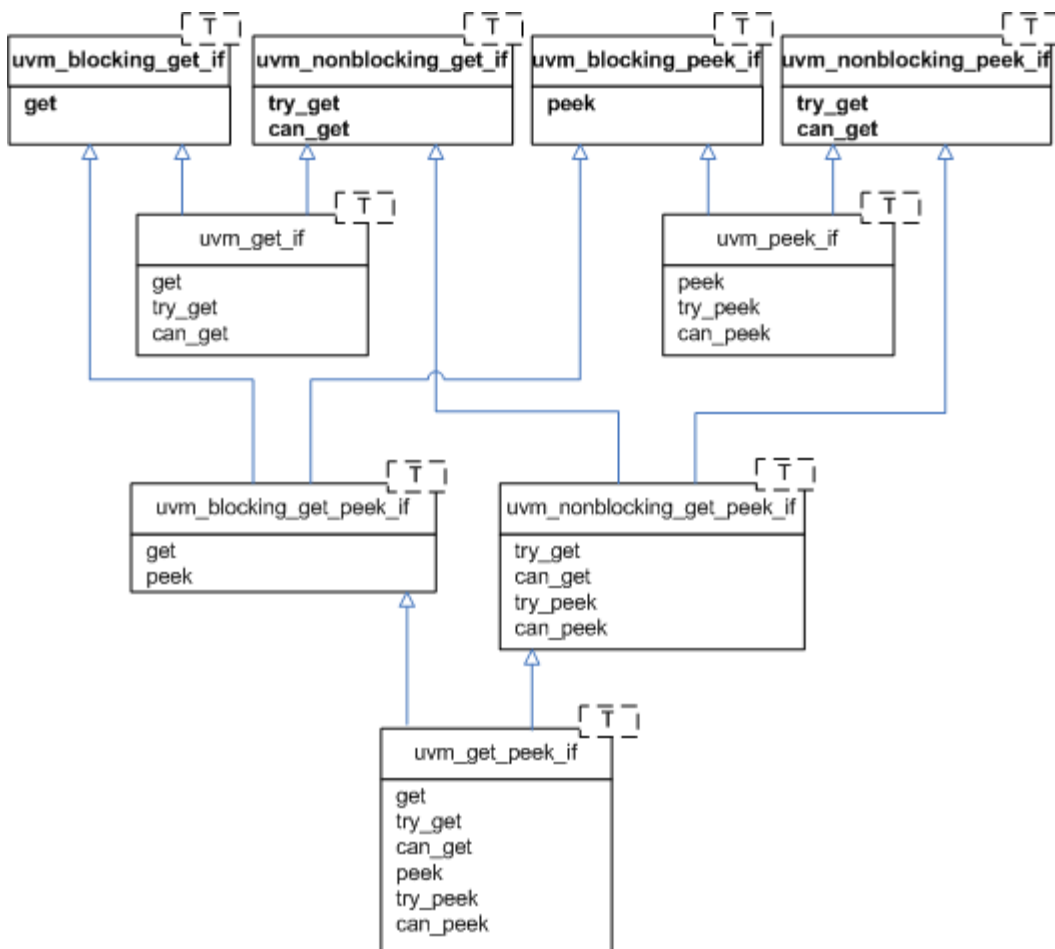
The *put* interfaces are used to send, or *put*, transactions to other components. Successful completion of a put guarantees its delivery, not execution.



### Get and Peek

---

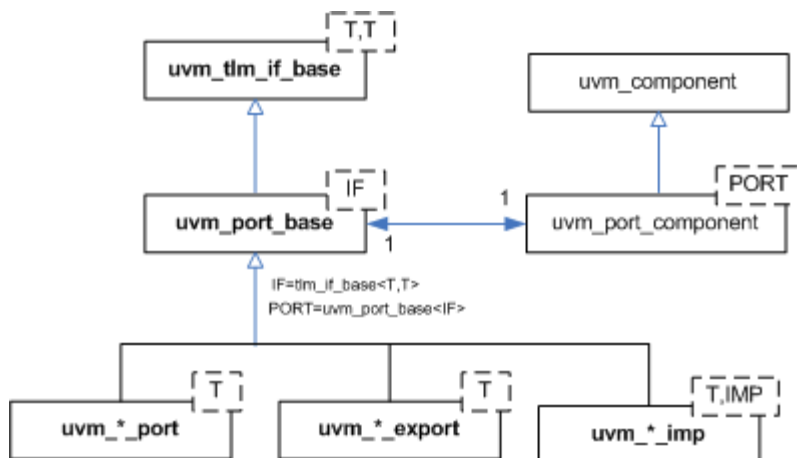
The *get* interfaces are used to retrieve transactions from other components. The *peek* interfaces are used for the same purpose, except the retrieved transaction is not consumed; successive calls to *peek* will return the same object. Combined *get\_peek* interfaces are also defined.



## 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

---

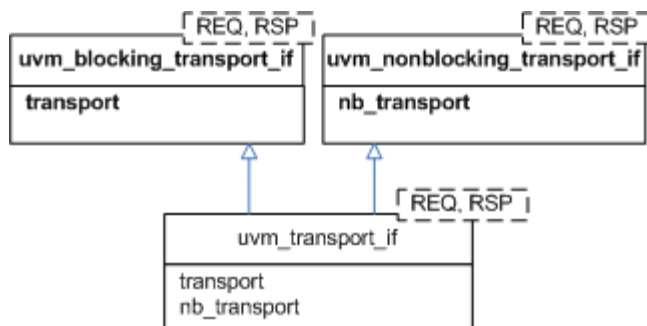
The bidirectional interfaces consist of blocking, non-blocking, and combined blocking and non-blocking variants of the *transport*, *master*, and *slave* interfaces.

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.

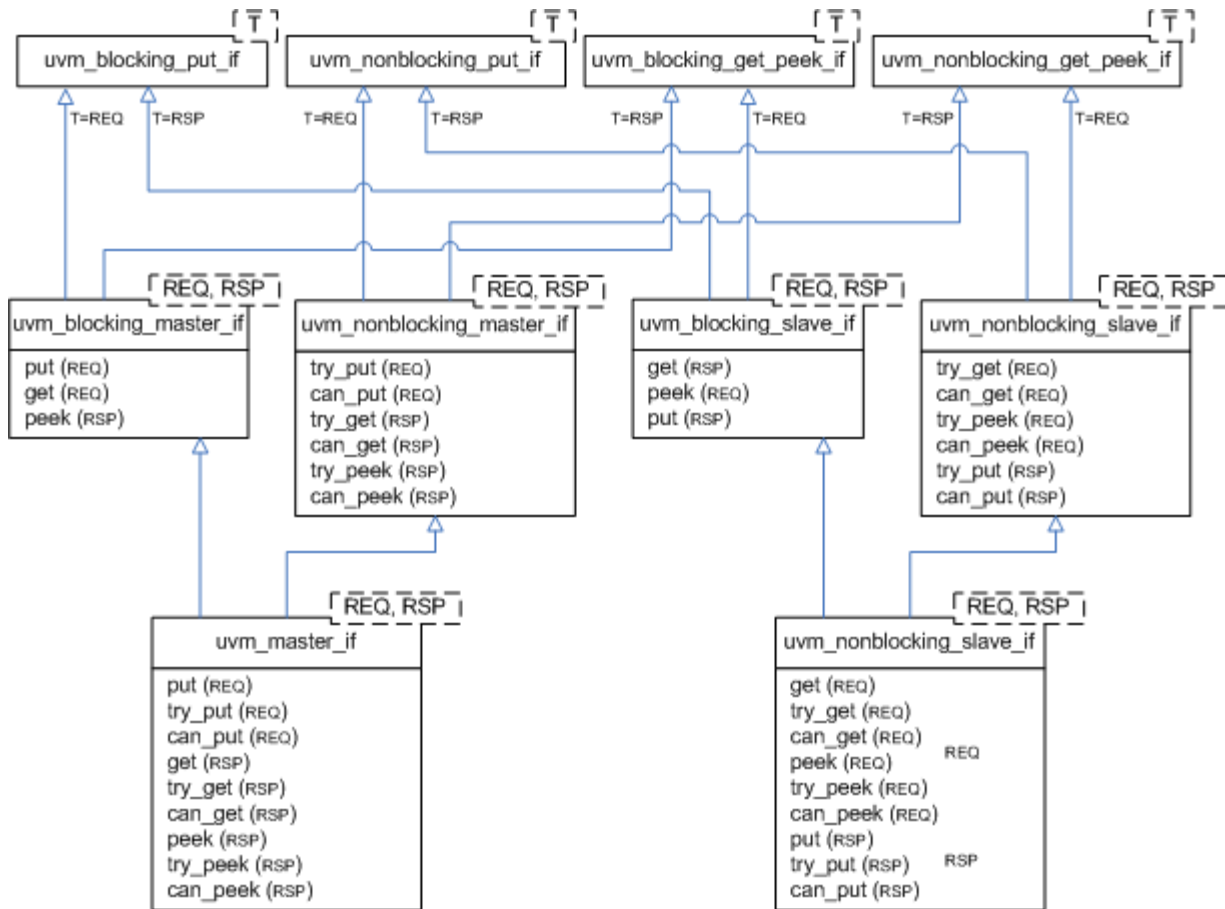


### Master and Slave

---

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

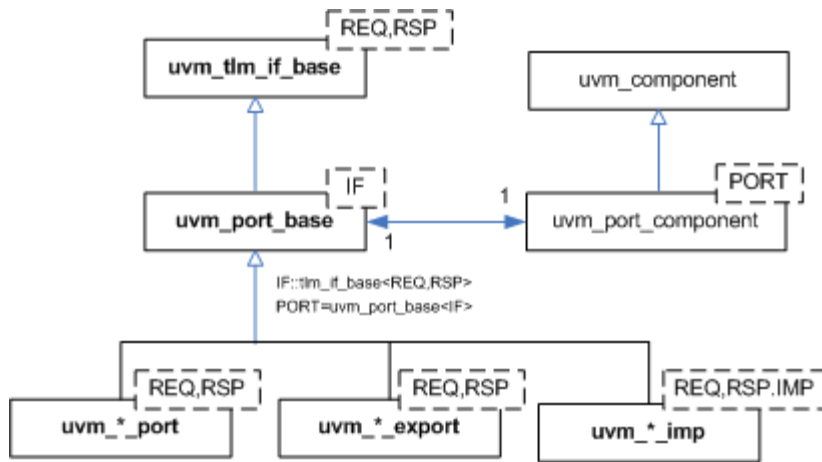
responses. The slave gets or peeks requests and puts responses. Because the put and the get come from different function interface methods, the requests and responses are not coupled as they are with the *transport* interface.



## 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 uvvm_*_port #(type REQ=int, RSP=int)
  extends uvvm_port_base #(tlm_if_base #(REQ, RSP));

class uvvm_*_export #(type REQ=int, RSP=int)
  extends uvvm_port_base #(tlm_if_base #(REQ, RSP));

class uvvm_*_imp #(type REQ=int, RSP=int)
  extends uvvm_port_base #(tlm_if_base #(REQ, RSP));
```

where the asterisk can be any of

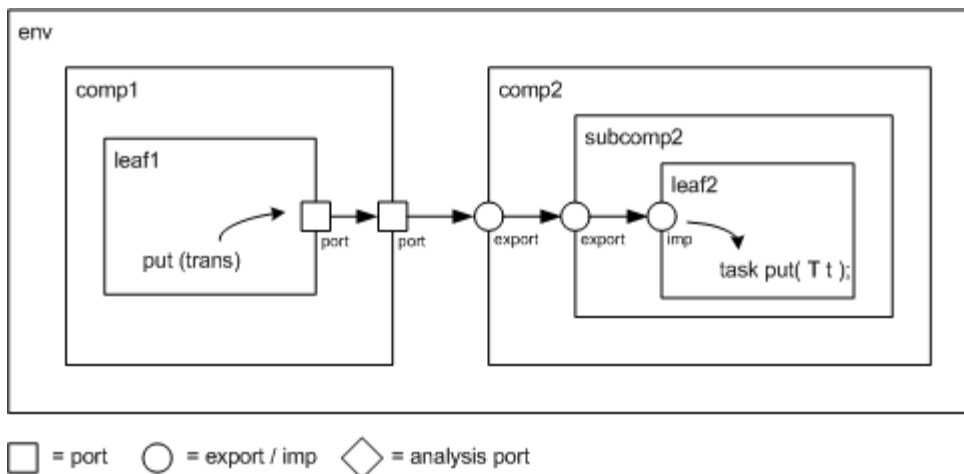
```
transport
blocking_transport
nonblocking_transport

blocking_master
nonblocking_master
master

blocking_slave
nonblocking_slave
slave
```

## Usage

This example illustrates basic TLM connectivity using the blocking put interface.



<i>port-to-port</i>	leaf1's <i>out</i> port is connected to its parent's (comp1) <i>out</i> port
<i>port-to-export</i>	comp1's <i>out</i> port is connected to comp2's <i>in</i> export
<i>export-to-export</i>	comp2's <i>in</i> export is connected to its child's (subcomp2) <i>in</i> export
<i>export-to-imp</i>	subcomp2's <i>in</i> export is connected leaf2's <i>in</i> imp port.
<i>imp-to-implementation</i>	leaf2's <i>in</i> 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 primitive 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

### TLM Port Classes

The following classes define the TLM port classes.

<code>uvm_*_port #(T)</code>	These unidirectional ports are instantiated by components that <i>require</i> , or <i>use</i> , the associated interface to convey transactions.
<code>uvm_*_port #(REQ,RSP)</code>	These bidirectional ports are instantiated by components that <i>require</i> , or <i>use</i> , the associated interface to convey transactions.

## 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 associated 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 associated 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

### TLM Export Classes

The following classes define the TLM export classes.

**uvm\_\*\_export #(T)** The unidirectional uvm\_\*\_export is a port that *forwards* or *promotes* an interface implementation from a child component to its parent.

**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.

## 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 ports**

The following defines the TLM implementation (imp) classes.

#### **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*.

#### **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*.

## 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

#### **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*.

#### 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

<i>REQ</i>	Request transaction type
<i>RSP</i>	Response transaction type
<i>IMP</i>	Component type that implements the interface methods, typically the the parent of this imp port.
<i>REQ_IMP</i>	Component type that implements the request side of the interface. Defaults to <i>IMP</i> . For master and slave imps only.
<i>RSP_IMP</i>	Component type that implements the response side of the interface. Defaults to <i>IMP</i> . 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

<b>Analysis Ports</b>	This section defines the port, export, and imp classes used for transaction analysis.
<a href="#">uvm_analysis_port</a>	Broadcasts a value to all subscribers implementing a <a href="#">uvm_analysis_imp</a> .
<a href="#">uvm_analysis_imp</a>	Receives all transactions broadcasted by a <a href="#">uvm_analysis_port</a> .
<a href="#">uvm_analysis_export</a>	Exports a lower-level <a href="#">uvm_analysis_imp</a> to its parent.

## 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

<b>TLM FIFO Classes</b>	This section defines TLM-based FIFO classes.
<code>uvm_tlm_fifo</code>	This class provides storage of transactions between two independently running processes.
<code>uvm_tlm_analysis_fifo</code>	An <code>analysis_fifo</code> is a <code>uvm_tlm_fifo</code> with an unbounded size and a write interface.

# 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**

<code>new</code>	The <i>name</i> and <i>parent</i> are the normal <code>uvm_component</code> constructor arguments.
<code>size</code>	Returns the capacity of the FIFO-- that is, the number of entries the FIFO is capable of holding.
<code>used</code>	Returns the number of entries put into the FIFO.
<code>is_empty</code>	Returns 1 when there are no entries in the FIFO, 0 otherwise.
<code>is_full</code>	Returns 1 when the number of entries in the FIFO is equal to its <code>size</code> , 0 otherwise.
<code>flush</code>	Removes all entries from the FIFO, after which <code>used</code> 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 interface 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

<b>TLM Channel Classes</b>	This section defines built-in TLM channel classes.
<code>uvm_tlm_req_rsp_channel #(REQ,RSP)</code>	The <code>uvm_tlm_req_rsp_channel</code> contains a request FIFO of type <i>REQ</i> and a response FIFO of type <i>RSP</i> .
<code>uvm_tlm_transport_channel #(REQ,RSP)</code>	A <code>uvm_tlm_transport_channel</code> is a <code>uvm_tlm_req_rsp_channel #(REQ,RSP)</code> that implements the transport interface.

## 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 response 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**

<code>put_request_export</code>	The <code>put_export</code> provides both the blocking and non-blocking put interface methods to the request FIFO:
<code>get_peek_response_export</code>	The <code>get_peek_response_export</code> 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

<i>Blocking (b_transport)</i>	completes the entire transaction within a single method call
<i>Non-blocking (nb_transport)</i>	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-of-response- 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 initiator and target. Some socket types must be bound to impls 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

<i>Call to <code>b_transport()</code></i>	start-of-life of transaction
<i>Return from <code>b_transport()</code></i>	end-of-life of transaction
<i>Phase argument to <code>nb_transport()</code></i>	timing point within lifetime of transaction
<i>Return value of <code>nb_transport()</code></i>	whether return path is being used (also shortcut to final phase)
<i>Response status within transaction object</i>	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

### TLM Generic Payload & Extensions

The Generic Payload transaction represents a generic bus read/write access.

#### GLOBALS

Defines, Constants, enums.

[uvm\\_tlm\\_command\\_e](#)

Command attribute type definition

[uvm\\_tlm\\_response\\_status\\_e](#)

Response status attribute type definition

#### GENERIC PAYLOAD

[uvm\\_tlm\\_generic\\_payload](#)

This class provides a transaction definition commonly used in memory-mapped bus-based systems.

[uvm\\_tlm\\_gp](#)

This typedef provides a short, more convenient name for the [uvm\\_tlm\\_generic\\_payload](#) type.

[uvm\\_tlm\\_extension\\_base](#)

The class [uvm\\_tlm\\_extension\\_base](#) is the non-parameterized base class for all generic payload extensions.

[uvm\\_tlm\\_extension](#)

TLM extension class.

## GLOBALS

Defines, Constants, enums.

### [uvm\\_tlm\\_command\\_e](#)

Command attribute 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](#)

Response status attribute type definition

*UVM\_TLM\_OK\_RESPONSE*

Bus operation completed successfully

*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

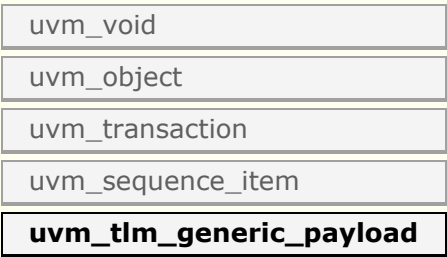
This class provides a transaction definition commonly used in memory-mapped bus-based systems. It's intended to be a general purpose transaction class that lends itself to many applications. The class is derived from `uvm_sequence_item` which enables it to be generated in sequences and transported to drivers through sequencers.

Summary

uvm\_tlm\_generic\_payload

This class provides a transaction definition commonly used in memory-mapped bus-based systems.

CLASS HIERARCHY



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 <i>UVM_TLM_WRITE_COMMAND</i> .
<code>set_address</code>	Set the value of the <code>m_address</code> variable
<code>get_address</code>	Get the value of the <code>m_address</code> variable
<code>get_data</code>	Return the value of the <code>m_data</code> array
<code>set_data</code>	Set the value of the <code>m_data</code> array
<code>get_data_length</code>	Return the current size of the <code>m_data</code> array
<code>set_data_length</code>	Set the value of the <code>m_length</code>
<code>get_streaming_width</code>	Get the value of the <code>m_streaming_width</code> array
<code>set_streaming_width</code>	Set the value of the <code>m_streaming_width</code> array
<code>get_byte_enable</code>	Return the value of the <code>m_byte_enable</code> array
<code>set_byte_enable</code>	Set the value of the <code>m_byte_enable</code> array
<code>get_byte_enable_length</code>	Return the current size of the <code>m_byte_enable</code> array
<code>set_byte_enable_length</code>	Set the size <code>m_byte_enable_length</code> of the <code>m_byte_enable</code> array i.e <code>m_byte_enable.size()</code>
<code>set_dmi_allowed</code>	DMI hint.
<code>is_dmi_allowed</code>	DMI hint.
<code>get_response_status</code>	Return the current value of the <code>m_response_status</code> variable
<code>set_response_status</code>	Set the current value of the <code>m_response_status</code> variable
<code>is_response_ok</code>	Return TRUE if the current value of the <code>m_response_status</code> variable is <i>UVM_TLM_OK_RESPONSE</i>
<code>is_response_error</code>	Return TRUE if the current value of the <code>m_response_status</code> variable is not <i>UVM_TLM_OK_RESPONSE</i>
<code>get_response_string</code>	Return the current value of the <code>m_response_status</code> variable as a string

#### EXTENSIONS MECHANISM

<code>set_extension</code>	Add an instance-specific extension.
<code>get_num_extensions</code>	Return the current number of instance specific extensions.
<code>get_extension</code>	Return the instance specific extension bound under the specified key.
<code>clear_extension</code>	Remove the instance-specific extension bound under the specified key.
<code>clear_extensions</code>	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  $\text{address\_attribute} + \text{streaming\_width} - 1$ . 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 `SsystemC`. 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 its 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



#### CLASS DECLARATION

```
virtual class uvm_tlm_extension_base extends uvm_object
```

#### METHODS

`new`

`get_type_handle`

`get_type_handle_name`

`create`

An interface to polymorphically retrieve a handle that uniquely identifies the type of the sub-class

An interface to polymorphically retrieve the name that uniquely identifies the type of the sub-class

## 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.

## Summary

### tlm interfaces

#### GLOBALS

Global macro's & enums

`uvm_tlm_phase_e`

Nonblocking transport synchronization state values between an initiator and a target.

`uvm_tlm_sync_e`

Pre-defined phase state values for the nonblocking transport Base Protocol between an initiator and a target.

``UVM_TLM_TASK_ERROR`

Defines Not-Yet-Implemented TLM tasks

``UVM_TLM_FUNCTION_ERROR`

Defines Not-Yet-Implemented TLM functions

#### TLM IF CLASS

Base class type to define the transport functions.

## GLOBALS

Global macro's & enums

### uvm\_tlm\_phase\_e

Nonblocking transport synchronization state values between an initiator and a target.

<code>UNINITIALIZED_PHASE</code>	Defaults for constructor
<code>BEGIN_REQ</code>	Beginning of request phase
<code>END_REQ</code>	End of request phase
<code>BEGIN_RESP</code>	Beginning of response phase
<code>END_RESP</code>	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.

<code>UVM_TLM_ACCEPTED</code>	Transaction has been accepted
<code>UVM_TLM_UPDATED</code>	Transaction has been modified
<code>UVM_TLM_COMPLETED</code>	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.

<a href="#">nb_transport_fw</a>	Forward path call.
<a href="#">nb_transport_bw</a>	Implementation of the backward path.
<a href="#">b_transport</a>	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 successful 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

<b>TLM Sockets</b>	Each <code>uvm_tlm_*_socket</code> class is derived from a corresponding <code>uvm_tlm_*_socket_base</code> class.
<code>uvm_tlm_b_initiator_socket</code>	IS-A forward port; has no backward path except via the payload contents
<code>uvm_tlm_b_target_socket</code>	IS-A forward imp; has no backward path except via the payload contents.
<code>uvm_tlm_nb_initiator_socket</code>	IS-A forward port; HAS-A backward imp
<code>uvm_tlm_nb_target_socket</code>	IS-A forward imp; HAS-A backward port
<code>uvm_tlm_b_passthrough_initiator_socket</code>	IS-A forward port;
<code>uvm_tlm_b_passthrough_target_socket</code>	IS-A forward export;
<code>uvm_tlm_nb_passthrough_initiator_socket</code>	IS-A forward port; HAS-A backward export
<code>uvm_tlm_nb_passthrough_target_socket</code>	IS-A forward export; HAS-A backward port

# uvm\_tlm\_b\_initiator\_socket

IS-A forward port; has no backward path except via the payload contents

## Summary

<b>uvm_tlm_b_initiator_socket</b>
IS-A forward port; has no backward path except via the payload contents
<b>CLASS HIERARCHY</b>
<div>uvm_tlm_b_initiator_socket_base#(T)</div>

## 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

<b>new</b>	Construct a new instance of this socket <i>imp</i> is a reference to the class implementing the <code>nb_transport_bw()</code> method.
<b>Connect</b>	Connect this socket to the specified <code>uvm_tlm_nb_target_socket</code>

## 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

<b>new</b>	Construct a new instance of this socket <i>imp</i> is a reference to the class implementing the <code>nb_transport_fw()</code> 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

### TLM2 ports

The following defines TLM2 port classes.

<code>uvm_tlm_b_transport_port</code>	Class providing the blocking transport port, The port can be bound to one export.
<code>uvm_tlm_nb_transport_fw_port</code>	Class providing the non-blocking backward transport port.
<code>uvm_tlm_nb_transport_bw_port</code>	Class providing the non-blocking backward transport port.

## 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

### TLM2 Export Classes

This section defines the export classes for connecting TLM2 interfaces.

[uvm\\_tlm\\_b\\_transport\\_export](#)

Blocking transport export class.

[uvm\\_tlm\\_nb\\_transport\\_fw\\_export](#)

Non-blocking forward transport export class

[uvm\\_tlm\\_nb\\_transport\\_bw\\_export](#)

Non-blocking backward transport export class

## 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

[new](#)

## METHODS

[new](#)

# TLM2 impls (interface implementations)

This section defines the implementation classes for connecting TLM2 interfaces.

TLM impls bind a TLM interface with the object that contains the interface implementation. In addition to the transaction type and the phase type, the impls 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 impl resides. The constructor of the impl takes as an argument an object of type IMP and installs it as the implementation object. Most often the impl constructor argument is "this".

## Contents

### TLM2 impls (interface implementations)

This section defines the implementation classes for connecting TLM2 interfaces.

#### IMP BINDING MACROS

``UVM_TLM_NB_TRANSPORT_FW_IMP`

The macro wraps the forward path call function nb\_transport\_fw()

``UVM_TLM_NB_TRANSPORT_BW_IMP`

Implementation of the backward path.

``UVM_TLM_B_TRANSPORT_IMP`

The macro wraps the function b\_transport() Execute a blocking transaction.

#### IMP BINDING CLASSES

`uvm_tlm_b_transport_imp`

Used like exports, except an additional class parameter specifies the type of the implementation object.

`uvm_tlm_nb_transport_fw_imp`

Used like exports, except an additional class parameter specifies the type of the implementation object.

`uvm_tlm_nb_transport_bw_imp`

Used like exports, except an additional class parameter specifies the type of the implementation object.

## 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 successful 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 additional class parameter specifies 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 additional class parameter specifies 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 additional class parameter specifies 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 additional class parameter specifies 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 additional class parameter specifies 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 additional class parameter specifies 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

<code>`UVM_TLM_NB_FW_MASK</code>	Define Non blocking Forward mask onehot assignment = `b001
<code>`UVM_TLM_NB_BW_MASK</code>	Define Non blocking backward mask onehot assignment = `b010
<code>`UVM_TLM_B_MASK</code>	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.

<i>Termination Sockets</i>	A termination socket must be the terminus of every TLM path. A transaction originates with an initiator socket and ultimately ends up in a target socket. There may be zero or more passthrough sockets between initiator and target.
<i>Passthrough Sockets</i>	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

<b>TLM Socket Base Classes</b>	A collection of base classes, one for each socket type.
<code>uvm_tlm_b_target_socket_base</code>	IS-A forward imp; has no backward path except via the payload contents.
<code>uvm_tlm_b_initiator_socket_base</code>	IS-A forward port; has no backward path except via the payload contents
<code>uvm_tlm_nb_target_socket_base</code>	IS-A forward imp; HAS-A backward port
<code>uvm_tlm_nb_initiator_socket_base</code>	IS-A forward port; HAS-A backward imp
<code>uvm_tlm_nb_passthrough_initiator_socket_base</code>	IS-A forward port; HAS-A backward export
<code>uvm_tlm_nb_passthrough_target_socket_base</code>	IS-A forward export; HAS-A backward port
<code>uvm_tlm_b_passthrough_initiator_socket_base</code>	IS-A forward port
<code>uvm_tlm_b_passthrough_target_socket_base</code>	IS-A forward export

# uvm\_tlm\_b\_target\_socket\_base

IS-A forward imp; has no backward path except via the payload contents.

## Summary

<b>uvm_tlm_b_target_socket_base</b>
IS-A forward imp; has no backward path except via the payload contents.
<b>CLASS HIERARCHY</b>
<div>uvm_port_base#(uvm_tlm_if#(T))</div> <div><b>uvm_tlm_b_target_socket_base</b></div>
<b>CLASS DECLARATION</b>

```
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
```

<a href="#">set_time_resolution</a>	Set the default canonical time resolution.
<a href="#">new</a>	Create a new canonical time value.
<a href="#">get_name</a>	Return the name of this instance
<a href="#">reset</a>	Reset the value to 0
<a href="#">get_realtime</a>	Return the current canonical time value, scaled for the caller's timescale
<a href="#">incr</a>	Increment the time value by the specified number of scaled time unit
<a href="#">decr</a>	Decrement the time value by the specified number of scaled time unit
<a href="#">get_abstime</a>	Return the current canonical time value, in the number of specified time unit, regardless of the current timescale of the caller.
<a href="#">set_abstime</a>	Set the current canonical time value, to the number of specified time unit, regardless 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, regardless 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, regardless 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;  
    end  
endmodule
```

```

    $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

```

yields

```

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

<b>Sequence Item Pull Ports</b>	This section defines the port, export, and imp port classes for communicating sequence items between <code>uvm_sequencer #(REQ,RSP)</code> and <code>uvm_driver #(REQ,RSP)</code> .
<code>uvm_seq_item_pull_port #(REQ,RSP)</code>	UVM provides a port, export, and imp connector for use in sequencer-driver communication.
<code>uvm_seq_item_pull_export #(REQ,RSP)</code>	This export type is used in sequencer-driver communication.
<code>uvm_seq_item_pull_imp #(REQ,RSP,IMP)</code>	This imp type is used in sequencer-driver communication.

# 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

<b>uvm_seq_item_pull_port #(REQ,RSP)</b>
UVM provides a port, export, and imp connector for use in sequencer-driver communication.
<b>CLASS HIERARCHY</b>
<div>uvm_port_base#(uvm_sqr_if_base#(REQ,RSP))</div> <div><b>uvm_seq_item_pull_port#(REQ,RSP)</b></div>
<b>CLASS DECLARATION</b>
<pre>class uvm_seq_item_pull_port #(     type REQ = int,     type RSP = REQ ) extends uvm_port_base #(uvm_sqr_if_base #(REQ, RSP))</pre>

# 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

<a href="#">get_next_item</a>	Retrieves the next available item from a sequence.
<a href="#">try_next_item</a>	Retrieves the next available item from a sequence if one is available.
<a href="#">item_done</a>	Indicates that the request is completed to the sequencer.
<a href="#">wait_for_sequences</a>	Waits for a sequence to have a new item available.
<a href="#">has_do_available</a>	Indicates whether a sequence item is available for immediate processing.
<a href="#">get_peek</a>	Retrieves the next available item from a sequence. Returns the current request item if one is in the sequencer fifo.
<a href="#">put</a>	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 its 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 its sequence ID and transaction ID set to match the request. This can

be done using the `uvm_sequence_item::set_id_info` call:

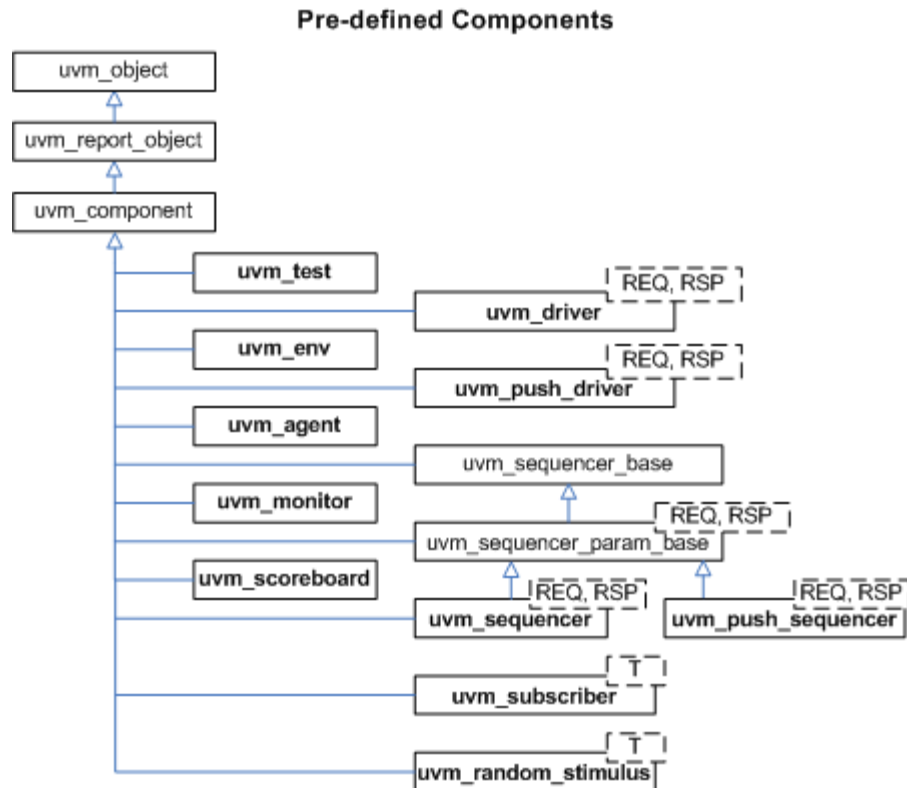
```
rsp.set_id_info(req);
```

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:

<i>Hierarchy</i>	provides methods for searching and traversing the component hierarchy.
<i>Phasing</i>	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
<i>Configuration</i>	provides methods for configuring component topology and other parameters ahead of and during component construction.
<i>Reporting</i>	provides a convenience interface to the <code>uvm_report_handler</code> . All messages, warnings, and errors are processed through this interface.
<i>Transaction recording</i>	provides methods for recording the transactions produced or consumed by the component to a transaction database (vendor specific).
<i>Factory</i>	provides a convenience interface to the <code>uvm_factory</code> . 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 its *parent*.

**HIERARCHY INTERFACE**

`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

<code>get_schedule</code>	Return handle to the phase schedule graph that applies to this component
<code>define_phase_schedule</code>	Builds and returns the required phase schedule subgraph for this component base
<code>set_phase_imp</code>	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 <code>exec</code> and <code>traverse</code> methods
<code>suspend</code>	Suspend this component.
<code>resume</code>	Resume this component.
<code>status</code>	Returns the status of this component.
<code>kill</code>	Kills the process tree associated with this component's currently running task-based phase, e.g., <code>run</code> .
<code>do_kill_all</code>	Recursively calls <code>kill</code> on this component and all its descendants, which abruptly ends the currently running task-based phase, e.g., <code>run</code> .
<code>stop</code>	The stop task is called when this component's <code>enable_stop_interrupt</code> bit is set and <code>global_stop_request</code> is called during a task-based phase, e.g., <code>run</code> .
<code>enable_stop_interrupt</code>	This bit allows a component to raise an objection to the stopping of the current phase.
<code>resolve_bindings</code>	Processes all port, export, and imp connections.
<b>CONFIGURATION INTERFACE</b>	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).
<code>set_config_int</code> <code>set_config_string</code> <code>set_config_object</code>	Calling <code>set_config_*</code> causes configuration settings to be created and placed in a table internal to this component.
<code>get_config_int</code> <code>get_config_string</code> <code>get_config_object</code>	These methods retrieve configuration settings made by previous calls to their <code>set_config_*</code> counterparts.
<code>check_config_usage</code>	Check all configuration settings in a components configuration table to determine if the setting has been used, overridden or not used.
<code>apply_config_settings</code>	Searches for all config settings matching this component's instance path.
<code>print_config_settings</code>	Called without arguments, <code>print_config_settings</code> prints all configuration information for this component, as set by previous calls to <code>set_config_*</code> .
<code>print_config</code>	<code>Print_config_settings</code> prints all configuration information for this component, as set by previous calls to <code>set_config_*</code> and exports to the resources pool.
<code>print_config_with_audit</code>	Operates the same as <code>print_config</code> except that the audit bit is forced to 1.
<code>print_config_matches</code>	Setting this static variable causes <code>get_config_*</code> to print info about matching configuration settings as they are being applied.
<b>OBJECTION INTERFACE</b>	These methods provide object level hooks into the <code>uvm_objection</code> mechanism.
<code>raised</code>	The <code>raised</code> callback is called when a descendant of the component instance raises the specified <i>objection</i> .
<code>dropped</code>	The <code>dropped</code> callback is called when a descendant of the component instance raises the specified <i>objection</i> .
<code>all_dropped</code>	The <code>all_dropped</code> callback is called when a descendant of the component instance raises the specified <i>objection</i> .

## 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.
<code>accept_tr</code>	This function marks the acceptance of a transaction, <i>tr</i> , by this component.
<code>do_accept_tr</code>	The <code>accept_tr</code> method calls this function to accommodate any user-defined post-accept action.
<code>begin_tr</code>	This function marks the start of a transaction, <i>tr</i> , by this component.
<code>begin_child_tr</code>	This function marks the start of a child transaction, <i>tr</i> , by this component.
<code>do_begin_tr</code>	The <code>begin_tr</code> and <code>begin_child_tr</code> methods call this function to accommodate any user-defined post-begin action.
<code>end_tr</code>	This function marks the end of a transaction, <i>tr</i> , by this component.
<code>do_end_tr</code>	The <code>end_tr</code> method calls this function to accommodate any user-defined post-end action.
<code>record_error_tr</code>	This function marks an error transaction by a component.
<code>record_event_tr</code>	This function marks an event transaction by a component.
<code>print_enabled</code>	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 <code>super.define_phase_schedule(domain,MYNAME);</code>

## new

```
function new (string      name,
             uvm_component parent)
```

Creates a new component with the given leaf instance *name* and handle 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 registered 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 persist until `global_stop_request()` is called. However, if a single phase objection is raised using `phase.raise_objection()`, then the phase will automatically end 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  
  ...  
endclass
```

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("CGOVR", 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 descendant of the component instance raises the specified *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 descendant of the component instance raises the specified *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 descendant of the component instance raises the specified *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 verboisities, 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

<b>Callbacks Classes</b>	This section defines the classes used for callback registration, management, and user-defined callbacks.
<a href="#">uvm_callbacks #(T,CB)</a>	The <i>uvm_callbacks</i> class provides a base class for implementing callbacks, which are typically used to modify or augment component behavior without changing the component class.
<a href="#">uvm_callback_iter</a>	The <i>uvm_callback_iter</i> class is an iterator class for iterating over callback queues of a specific callback type.
<a href="#">uvm_callback</a>	The <i>uvm_callback</i> class is the base class for user-defined callback classes.

## 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 particular hook. A demonstration of this methodology is provided in an example included in the kit.

## Summary

<b>uvm_callbacks #(T,CB)</b>	
The <i>uvm_callbacks</i> class provides a base class for implementing callbacks, which are typically used to modify or augment component behavior without changing the component class.	
<b>T</b>	This type parameter specifies the base object type with which the <b>CB</b> callback objects will be registered.

<b>CB</b>	This type parameter specifies the base callback type that will be managed by this callback class.
<b>Add/DELETE INTERFACE</b>	
<b>add</b>	Registers the given callback object, <i>cb</i> , with the given <i>obj</i> handle.
<b>add_by_name</b>	Registers the given callback object, <i>cb</i> , with one or more <i>uvm_components</i> .
<b>delete</b>	Deletes the given callback object, <i>cb</i> , from the queue associated with the given <i>obj</i> handle.
<b>delete_by_name</b>	Removes the given callback object, <i>cb</i> , associated with one or more <i>uvm_component</i> callback queues.
<b>ITERATOR INTERFACE</b>	
This set of functions provide an iterator interface for callback queues.	
<b>get_first</b>	returns the first enabled callback of type CB which resides in the queue for <i>obj</i> .
<b>get_last</b>	returns the last enabled callback of type CB which resides in the queue for <i>obj</i> .
<b>get_next</b>	returns the next enabled callback of type CB which resides in the queue for <i>obj</i> , using <i>itr</i> as the starting point.
<b>get_prev</b>	returns the previous enabled callback of type CB which resides in the queue for <i>obj</i> , using <i>itr</i> as the starting point.
<b>DEBUG</b>	
<b>display</b>	This function displays callback information for <i>obj</i> .

## 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 INTERFACE

### add

```
static function void add(T      obj,
                        uvm_callback cb,
                        uvm_append 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_apprepnd ordering = UVM_APPEND)
```

Registers the given callback object, *cb*, with one or more *uvm\_component*s. 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.



## 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

<code>new</code>	Creates a new callback iterator object.
<code>first</code>	Returns the first valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object.
<code>last</code>	Returns the last valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object.
<code>next</code>	Returns the next valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object.
<code>prev</code>	Returns the previous valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object.
<code>get_cb</code>	Returns the last callback accessed via a <code>first()</code> or <code>next()</code> 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.

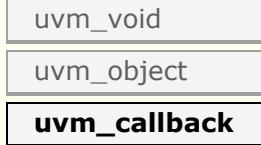
The prototypes for each hook method are completely application specific with no restrictions.

## Summary

### uvm\_callback

The *uvm\_callback* class is the base class for user-defined callback classes.

#### CLASS HIERARCHY



#### CLASS DECLARATION

```
class uvm_callback extends uvm_object
```

#### METHODS

<code>new</code>	Creates a new <i>uvm_callback</i> object, giving it an optional <i>name</i> .
<code>callback_mode</code>	Enable/disable callbacks (modeled like <i>rand_mode</i> and <i>constraint_mode</i> ).
<code>is_enabled</code>	Returns 1 if the callback is enabled, 0 otherwise.
<code>get_type_name</code>	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.

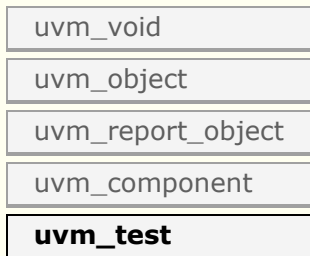
Deriving from `uvm_test` will allow you to distinguish tests from other component types that inherit from `uvm_component` directly. Such tests will automatically inherit features that may be added to `uvm_test` in the future.

## Summary

### uvm\_test

This class is the virtual base class for the user-defined tests.

#### CLASS HIERARCHY



#### 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.

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



#### 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



#### CLASS DECLARATION

```
virtual class uvm_agent extends uvm_component
```

#### METHODS

<code>new</code>	Creates and initializes an instance of this class using the normal constructor arguments for <code>uvm_component</code> : <i>name</i> is the name of the instance, and <i>parent</i> is the handle to the hierarchical parent, if any.
<code>get_is_active</code>	Returns UVM_ACTIVE if 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 if 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.

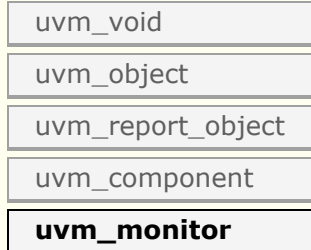
Deriving from `uvm_monitor` allows you to distinguish monitors from generic component types inheriting from `uvm_component`. Such monitors will automatically inherit features that may be added to `uvm_monitor` in the future.

## Summary

### uvm\_monitor

This class should be used as the base class for user-defined monitors.

#### CLASS HIERARCHY



#### 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.

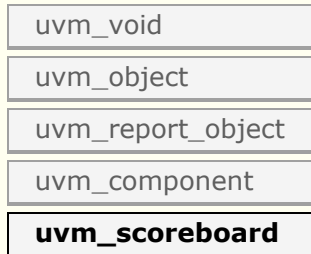
Deriving from uvm\_scoreboard will allow you to distinguish scoreboards from other component types inheriting directly from uvm\_component. Such scoreboards will automatically inherit and benefit from features that may be added to uvm\_scoreboard in the future.

## Summary

### uvm\_scoreboard

The uvm\_scoreboard virtual class should be used as the base class for user-defined scoreboards.

#### CLASS HIERARCHY



#### 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.

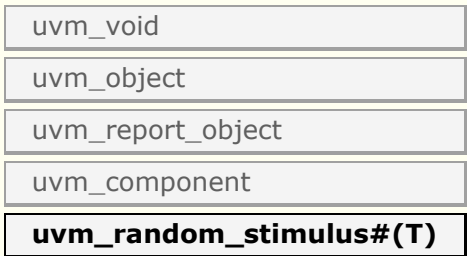
While very useful in its own right, this component can also be used as a template for defining other stimulus generators, or it can be extended to add additional stimulus generation methods and to simplify test writing.

## Summary

### uvm\_random\_stimulus #(T)

A general purpose unidirectional random stimulus class.

#### CLASS HIERARCHY



#### 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.



### 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



#### 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)
```

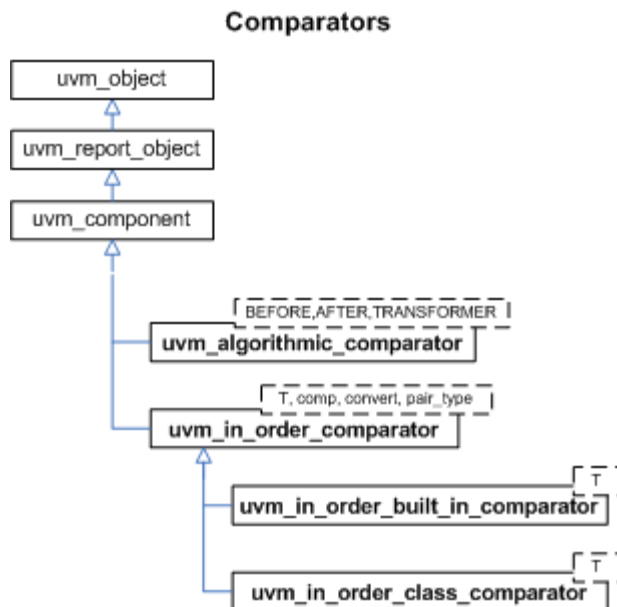
A pure virtual method that must be defined in each subclass. Access to this method by outside components should be done via the `analysis_export`.

# 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

### Comparators

The following classes define comparators for objects and built-in types.

<a href="#">uvm_in_order_comparator</a> <a href="#">#(T,comp_type,convert,pair_type)</a>	Compares two streams of data objects of the type parameter, T.
<a href="#">uvm_in_order_built_in_comparator</a> <a href="#">#(T)</a>	This class uses the <a href="#">uvm_built_in_*</a> comparison, converter, and pair classes.
<a href="#">uvm_in_order_class_comparator</a> <a href="#">#(T)</a>	This class uses the <a href="#">uvm_class_*</a> comparison, converter, and pair classes.

## 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

<i>T</i>	Specifies the type of transactions to be compared.
<i>comp_type</i>	A policy class to compare the two transaction streams. It must provide the method "function bit comp(T a, T b)" which returns <i>TRUE</i> if <i>a</i> and <i>b</i> are the same.
<i>convert</i>	A policy class to convert the transactions being compared to a string. It must provide the method "function string convert2string(T a)".
<i>pair_type</i>	A policy class to allow pairs of transactions to be handled as a single <a href="#">uvm_object</a> 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

<code>before_export</code>	The export to which one stream of data is written.
<code>after_export</code>	The export to which the other stream of data is written.
<code>pair_ap</code>	The comparator sends out pairs of transactions across this analysis port.

#### METHODS

<code>flush</code>	This method sets <code>m_matches</code> and <code>m_mismatches</code> 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 mismatches 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

<b>uvm_pair classes</b>	This section defines container classes for handling value pairs.
<code>uvm_pair #(T1,T2)</code>	Container holding handles to two objects whose types are specified by the type parameters, T1 and T2.
<code>uvm_built_in_pair #(T1,T2)</code>	Container holding two variables of built-in types (int, string, etc.)

# 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.

<b>VARIABLES</b>	
T1 first	The handle to the first object in the pair
T2 second	The second variable in the pair
<b>METHODS</b>	
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

### Policy Classes

Policy classes are used to implement polymorphic operations that differ between built-in types and class-based types.

<code>uvm_built_in_comp #(T)</code>	This policy class is used to compare built-in types.
<code>uvm_built_in_converter #(T)</code>	This policy class is used to convert built-in types to strings.
<code>uvm_built_in_clone #(T)</code>	This policy class is used to clone built-in types via the <code>=</code> operator.
<code>uvm_class_comp #(T)</code>	This policy class is used to compare two objects of the same type.
<code>uvm_class_converter #(T)</code>	This policy class is used to convert a class object to a string.
<code>uvm_class_clone #(T)</code>	This policy class is used to clone class objects.

## 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

<code>`uvm_info</code>	Calls <code>uvm_report_info</code> if <code>VERBOSITY</code> is lower than the configured verbosity of the associated reporter.
<code>`uvm_warning</code>	Calls <code>uvm_report_warning</code> with a verbosity of <code>UVM_NONE</code> .
<code>`uvm_error</code>	Calls <code>uvm_report_error</code> with a verbosity of <code>UVM_NONE</code> .
<code>`uvm_fatal</code>	Calls <code>uvm_report_fatal</code> with a verbosity of <code>UVM_NONE</code> .
<code>`uvm_info_context</code>	Operates identically to <code>`uvm_info</code> but requires that the context in which the message is printed be



<code>`uvm_warning_context</code>	explicitly supplied as a macro argument. Operates identically to <code>`uvm_warning</code> but requires that the context in which the message is printed be explicitly supplied as a macro argument.
<code>`uvm_error_context</code>	Operates identically to <code>`uvm_error</code> but requires that the context in which the message is printed be explicitly supplied as a macro argument.
<code>`uvm_fatal_context</code>	Operates identically to <code>`uvm_fatal</code> 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

<code>`uvm_field_sarray_object</code>	integrals. Implements the data operations for a one-dimensional static array of <b>uvm_object</b> -based objects.
<code>`uvm_field_sarray_string</code>	Implements the data operations for a one-dimensional static array of strings.
<code>`uvm_field_sarray_enum</code>	Implements the data operations for a one-dimensional static array of enums.
<code>`UVM_FIELD_ARRAY_* MACROS</code>	Macros that implement data operations for one-dimensional dynamic array properties.
<code>`uvm_field_array_int</code>	Implements the data operations for a one-dimensional dynamic array of integrals.
<code>`uvm_field_array_object</code>	Implements the data operations for a one-dimensional dynamic array of <b>uvm_object</b> -based objects.
<code>`uvm_field_array_string</code>	Implements the data operations for a one-dimensional dynamic array of strings.
<code>`uvm_field_array_enum</code>	Implements the data operations for a one-dimensional dynamic array of enums.
<code>`UVM_FIELD_QUEUE_* MACROS</code>	Macros that implement data operations for dynamic queues.
<code>`uvm_field_queue_int</code>	Implements the data operations for a queue of integrals.
<code>`uvm_field_queue_object</code>	Implements the data operations for a queue of <b>uvm_object</b> -based objects.
<code>`uvm_field_queue_string</code>	Implements the data operations for a queue of strings.
<code>`uvm_field_queue_enum</code>	Implements the data operations for a one-dimensional queue of enums.
<code>`UVM_FIELD_AA_*_STRING MACROS</code>	Macros that implement data operations for associative arrays indexed by <i>string</i> .
<code>`uvm_field_aa_int_string</code>	Implements the data operations for an associative array of integrals indexed by <i>string</i> .
<code>`uvm_field_aa_object_string</code>	Implements the data operations for an associative array of <b>uvm_object</b> -based objects indexed by <i>string</i> .
<code>`uvm_field_aa_string_string</code>	Implements the data operations for an associative array of strings indexed by <i>string</i> .
<code>`UVM_FIELD_AA_*_INT MACROS</code>	Macros that implement data operations for associative arrays indexed by an integral type.
<code>`uvm_field_aa_object_int</code>	Implements the data operations for an associative array of <b>uvm_object</b> -based objects indexed by the <i>int</i> data type.
<code>`uvm_field_aa_int_int</code>	Implements the data operations for an associative array of integral types indexed by the <i>int</i> data type.
<code>`uvm_field_aa_int_int_unsigned</code>	Implements the data operations for an associative array of integral types indexed by the <i>int unsigned</i> data type.
<code>`uvm_field_aa_int_integer</code>	Implements the data operations for an associative array of integral types indexed by the <i>integer</i> data type.
<code>`uvm_field_aa_int_integer_unsigned</code>	Implements the data operations for

	an associative array of integral types indexed by the <i>integer unsigned</i> data type.
<code>`uvm_field_aa_int_byte</code>	Implements the data operations for an associative array of integral types indexed by the <i>byte</i> data type.
<code>`uvm_field_aa_int_byte_unsigned</code>	Implements the data operations for an associative array of integral types indexed by the <i>byte unsigned</i> data type.
<code>`uvm_field_aa_int_shortint</code>	Implements the data operations for an associative array of integral types indexed by the <i>shortint</i> data type.
<code>`uvm_field_aa_int_shortint_unsigned</code>	Implements the data operations for an associative array of integral types indexed by the <i>shortint unsigned</i> data type.
<code>`uvm_field_aa_int_longint</code>	Implements the data operations for an associative array of integral types indexed by the <i>longint</i> data type.
<code>`uvm_field_aa_int_longint_unsigned</code>	Implements the data operations for an associative array of integral types indexed by the <i>longint unsigned</i> data type.
<code>`uvm_field_aa_int_key</code>	Implements the data operations for an associative array of integral types indexed by any integral key data type.
<code>`uvm_field_aa_int_enumkey</code>	Implements the data operations for an associative array of integral types indexed by any enumeration key data type.
<b>RECORDING MACROS</b>	The recording macros assist users who implement the <code>uvm_object::do_record</code> method.
<code>`uvm_record_attribute</code>	Vendor-independent macro for recording attributes (fields) to a vendor-specific transaction database.
<code>`uvm_record_field</code>	Macro for recording name-value pairs into a transaction recording database.
<b>PACKING MACROS</b>	The packing macros assist users who implement the <code>uvm_object::do_pack</code> method.
<b>PACKING - WITH SIZE INFO</b>	
<code>`uvm_pack_intN</code>	Pack an integral variable.
<code>`uvm_pack_enumN</code>	Pack an integral variable.
<code>`uvm_pack_sarrayN</code>	Pack a static array of integrals.
<code>`uvm_pack_arrayN</code>	Pack a dynamic array of integrals.
<code>`uvm_pack_queueN</code>	Pack a queue of integrals.
<b>PACKING - No SIZE INFO</b>	
<code>`uvm_pack_int</code>	Pack an integral variable without having to also specify the bit size.
<code>`uvm_pack_enum</code>	Pack an enumeration value.
<code>`uvm_pack_string</code>	Pack a string variable.
<code>`uvm_pack_real</code>	Pack a variable of type real.
<code>`uvm_pack_sarray</code>	Pack a static array without having to also specify the bit size of its elements.
<code>`uvm_pack_array</code>	Pack a dynamic array without having to also specify the bit size of its elements.
<code>`uvm_pack_queue</code>	Pack a queue without having to also specify the bit size of its elements.
<b>UNPACKING MACROS</b>	The unpacking macros assist users

who implement the `uvm_object::do_unpack` method.

#### UNPACKING - WITH SIZE INFO

<code>`uvm_unpack_intN</code>	Unpack into an integral variable.
<code>`uvm_unpack_enumN</code>	Unpack enum of type <i>TYPE</i> into <i>VAR</i> .
<code>`uvm_unpack_sarrayN</code>	Unpack a static (fixed) array of integrals.
<code>`uvm_unpack_arrayN</code>	Unpack into a dynamic array of integrals.
<code>`uvm_unpack_queueN</code>	Unpack into a queue of integrals.

#### UNPACKING - No SIZE INFO

<code>`uvm_unpack_int</code>	Unpack an integral variable without having to also specify the bit size.
<code>`uvm_unpack_enum</code>	Unpack an enumeration value, which requires its type be specified.
<code>`uvm_unpack_string</code>	Pack a string variable.
<code>`uvm_unpack_real</code>	Unpack a variable of type real.
<code>`uvm_unpack_sarray</code>	Unpack a static array without having to also specify the bit size of its elements.
<code>`uvm_unpack_array</code>	Unpack a dynamic array without having to also specify the bit size of its elements.
<code>`uvm_unpack_queue</code>	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 el;
  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, el, 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 its 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).

<code>UVM_ALL_ON</code>	Set all operations on (default).
<code>UVM_DEFAULT</code>	Use the default flag settings.
<code>UVM_NOCOPY</code>	Do not copy this field.
<code>UVM_NOCOMPARE</code>	Do not compare this field.
<code>UVM_NOPRINT</code>	Do not print this field.
<code>UVM_NODEFPRINT</code>	Do not print the field if it is the same as its
<code>UVM_NOPACK</code>	Do not pack or unpack this field.
<code>UVM_PHYSICAL</code>	Treat as a physical field. Use physical setting in policy class for this field.
<code>UVM_ABSTRACT</code>	Treat as an abstract field. Use the abstract setting in the policy class for this field.
<code>UVM_READONLY</code>	Do not allow setting of this field from the <code>set_*_local</code> methods.

A radix for printing and recording can be specified by OR'ing one of the following constants in the *FLAG* argument

<i>UVM_BIN</i>	Print / record the field in binary (base-2).
<i>UVM_DEC</i>	Print / record the field in decimal (base-10).
<i>UVM_UNSIGNED</i>	Print / record the field in unsigned decimal (base-10).
<i>UVM_OCT</i>	Print / record the field in octal (base-8).
<i>UVM_HEX</i>	Print / record the field in hexadecimal (base-16).
<i>UVM_STRING</i>	Print / record the field in string format.
<i>UVM_TIME</i>	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 sequence 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 sequence item or sequence is

``uvm_rand_send`

executed with the priority specified in the argument.

``uvm_rand_send_pri`

This macro processes the item or sequence that has been already been allocated (possibly with ``uvm_create`). 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 ``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

```

## Summary

### uvm\_callback\_defines.svh

#### CALLBACK MACROS

<code>`uvm_register_cb</code>	Registers the given <i>CB</i> callback type with the given <i>T</i> object type.
<code>`uvm_set_super_type</code>	Defines the super type of <i>T</i> to be <i>ST</i> .
<code>`uvm_do_callbacks</code>	Calls the given <i>METHOD</i> of all callbacks of type <i>CB</i> registered with the calling object (i.e.
<code>`uvm_do_obj_callbacks</code>	Calls the given <i>METHOD</i> of all callbacks based on type <i>CB</i> registered with the given object, <i>OBJ</i> , which is or is based on type <i>T</i> .
<code>`uvm_do_callbacks_exit_on</code>	Calls the given <i>METHOD</i> of all callbacks of type <i>CB</i> registered with the calling object (i.e.
<code>`uvm_do_obj_callbacks_exit_on</code>	Calls the given <i>METHOD</i> of all callbacks of type <i>CB</i> registered with the given object <i>OBJ</i> , which must be or be based on type <i>T</i> , and returns upon the first callback that returns the bit value given by <i>VAL</i> .

## 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(myobj, 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 of 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 implementation declaration macros provide a way for an implementer to provide multiple implementation 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 implementation 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 an interface. This set of declarations allow you to easily create a new implementation class to allow for multiple implementations. Although the new implementation 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 coming into put_imp1
  ...
endfunction
  function void put_2(input T t);
    //puts coming 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 separate package to avoid collisions and to allow sharing of the definitions.

## Summary

### TLM Implementation Port Declaration Macros

The TLM implementation declaration macros provide a way for an implementer to provide multiple implementation 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.
<code>`uvm_put_imp_decl</code>	Define the class uvm_put_impSFX for providing both blocking and non-blocking put implementations.
<code>`uvm_blocking_get_imp_decl</code>	Define the class uvm_blocking_get_impSFX for providing blocking get implementations.
<code>`uvm_nonblocking_get_imp_decl</code>	Define the class uvm_nonblocking_get_impSFX for providing non-blocking get implementations.
<code>`uvm_get_imp_decl</code>	Define the class uvm_get_impSFX for providing both blocking and non-blocking get implementations.
<code>`uvm_blocking_peek_imp_decl</code>	Define the class uvm_blocking_peek_impSFX for providing blocking peek implementations.
<code>`uvm_nonblocking_peek_imp_decl</code>	Define the class uvm_nonblocking_peek_impSFX for providing non-blocking peek implementations.
<code>`uvm_peek_imp_decl</code>	Define the class uvm_peek_impSFX for providing both blocking and non-blocking peek implementations.
<code>`uvm_blocking_get_peek_imp_decl</code>	Define the class uvm_blocking_get_peek_impSFX for providing the blocking get_peek implementation.
<code>`uvm_nonblocking_get_peek_imp_decl</code>	Define the class uvm_nonblocking_get_peek_impSFX for providing non-blocking get_peek implementation.
<code>`uvm_get_peek_imp_decl</code>	Define the class uvm_get_peek_impSFX for providing both blocking and non-blocking get_peek implementations.
<code>`uvm_blocking_master_imp_decl</code>	Define the class uvm_blocking_master_impSFX for providing the blocking master implementation.
<code>`uvm_nonblocking_master_imp_decl</code>	Define the class uvm_nonblocking_master_impSFX for providing the non-blocking master implementation.
<code>`uvm_master_imp_decl</code>	Define the class uvm_master_impSFX for providing both blocking and non-blocking master implementations.
<code>`uvm_blocking_slave_imp_decl</code>	Define the class uvm_blocking_slave_impSFX for providing the blocking slave implementation.
<code>`uvm_nonblocking_slave_imp_decl</code>	Define the class uvm_nonblocking_slave_impSFX for providing the non-blocking slave implementation.
<code>`uvm_slave_imp_decl</code>	Define the class uvm_slave_impSFX for providing both blocking and non-blocking slave implementations.
<code>`uvm_blocking_transport_imp_decl</code>	Define the class uvm_blocking_transport_impSFX for providing the blocking transport implementation.
<code>`uvm_nonblocking_transport_imp_decl</code>	Define the class uvm_nonblocking_transport_impSFX

```
`uvm_transport_imp_decl
```

for providing the non-blocking transport implementation. Define the class `uvm_transport_impSFX` for providing both blocking and non-blocking transport implementations. Define the class `uvm_analysis_impSFX` for providing an analysis implementation.

```
`uvm_analysis_imp_decl
```

## 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` implementation.

## **``uvm_nonblocking_get_peek_imp_decl`**

---

Define the class `uvm_nonblocking_get_peek_impSFX` for providing non-blocking `get_peek` implementation.

## **``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 implementation.

## **``uvm_nonblocking_master_imp_decl`**

---

Define the class `uvm_nonblocking_master_impSFX` for providing the non-blocking master implementation.

## **``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 implementation.

## ``uvm_nonblocking_slave_imp_decl`

---

Define the class `uvm_nonblocking_slave_impSFX` for providing the non-blocking slave implementation.

## ``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 implementation.

## ``uvm_nonblocking_transport_imp_decl`

---

Define the class `uvm_nonblocking_transport_impSFX` for providing the non-blocking transport implementation.

## ``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 implementation 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
```

## Summary

### uvm\_reg\_defines.svh

#### REGISTER DEFINES

<code>`UVM_REG_ADDR_WIDTH</code>	Maximum address width in bits
<code>`UVM_REG_DATA_WIDTH</code>	Maximum data width in bits
<code>`UVM_REG_BYTENABLE_WIDTH</code>	Maximum number of byte enable bits
<code>`UVM_REG_CVR_WIDTH</code>	Maximum number of bits in a <code>uvm_reg_cvr_t</code> 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 govern 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 mismatches 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

<code>uvm_printer</code>	The <code>uvm_printer</code> class provides an interface for printing <code>uvm_objects</code> in various formats.
<code>uvm_table_printer</code>	The table printer prints output in a tabular format.
<code>uvm_tree_printer</code>	By overriding various methods of the <code>uvm_printer</code> super class, the tree printer prints output in a tree format.
<code>uvm_line_printer</code>	The line printer prints output in a line format.
<code>uvm_printer_knobs</code>	The <code>uvm_printer_knobs</code> class defines the printer settings available to all printer subtypes.

## 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_NORADIX,
        byte scope_separator = ".",
        string type_name = ""

```

Prints an integral field.

<i>name</i>	The name of the field.
<i>value</i>	The value of the field.
<i>size</i>	The number of bits of the field (maximum is 4096).
<i>radix</i>	The radix to use for printing the printer knob for radix is used if no radix is specified.
<i>scope_separator</i>	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 is 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**



**CLASS DECLARATION**

```
class uvm_table_printer extends uvm_printer
```

#### VARIABLES

##### new

Creates a new instance of *uvmm\_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 *uvmm\_table\_printer*.

## METHODS

---

### emit

```
virtual function string emit()
```

Formats the collected information from prior calls to *print\_\** into table format.

## uvmm\_tree\_printer

By overriding various methods of the [uvmm\\_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

### uvmm\_tree\_printer

By overriding various methods of the [uvmm\\_printer](#) super class, the tree printer prints output in a tree format.

#### CLASS HIERARCHY

```
uvmm_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

<b>header</b>	Indicates whether the <print_header> function should be called when printing an object.
<b>footer</b>	Indicates whether the <print_footer> function should be called when printing an object.
<b>full_name</b>	Indicates whether <adjust_name> should print the full name of an identifier or just the leaf name.
<b>identifier</b>	Indicates whether <adjust_name> should print the identifier.
<b>type_name</b>	Controls whether to print a field's type name.
<b>size</b>	Controls whether to print a field's size.
<b>depth</b>	Indicates how deep to recurse when printing objects.
<b>reference</b>	Controls whether to print a unique reference ID for object handles.
<b>begin_elements</b>	Defines the number of elements at the head of a list to print.
<b>end_elements</b>	This defines the number of elements at the end of a list that should be printed.
<b>prefix</b>	Specifies the string prepended to each output line
<b>indent</b>	This knob specifies the number of spaces to use for level indentation.
<b>show_root</b>	This setting indicates whether or not the initial object that is printed (when current depth is 0) prints the full path

	name.
<code>mcd</code>	This is a file descriptor, or multi-channel descriptor, that specifies where the print output should be directed.
<code>separator</code>	For tree printers only, determines the opening and closing separators used for nested objects.
<code>show_radix</code>	Indicates whether the radix string ('h, and so on) should be prepended to an integral value when one is printed.
<code>default_radix</code>	This knob sets the default radix to use for integral values when no radix enum is explicitly supplied to the <code>print_int()</code> method.
<code>dec_radix</code>	This string should be prepended to the value of an integral type when a radix of <code>UVM_DEC</code> is used for the radix of the integral object.
<code>bin_radix</code>	This string should be prepended to the value of an integral type when a radix of <code>UVM_BIN</code> is used for the radix of the integral object.
<code>oct_radix</code>	This string should be prepended to the value of an integral type when a radix of <code>UVM_OCT</code> is used for the radix of the integral object.
<code>unsigned_radix</code>	This is the string which should be prepended to the value of an integral type when a radix of <code>UVM_UNSIGNED</code> is used for the radix of the integral object.
<code>hex_radix</code>	This string should be prepended to the value of an integral type when a radix of <code>UVM_HEX</code> is used for the radix of the integral object.
<b>METHODS</b>	
<code>get_radix_str</code>	Converts the radix from an enumerated to a printable radix according to the radix printing knobs ( <code>bin_radix</code> , 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 mismatches 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

<code>policy</code>	Determines whether comparison is UVM_DEEP, UVM_REFERENCE, or UVM_SHALLOW.
<code>show_max</code>	Sets the maximum number of messages to send to the messenger for mismatches of an object.
<code>verbosity</code>	Sets the verbosity for printed messages.
<code>sev</code>	Sets the severity for printed messages.
<code>mismatches</code>	This string is reset to an empty string when a comparison is started.
<code>physical</code>	This bit provides a filtering mechanism for fields.
<code>abstract</code>	This bit provides a filtering mechanism for fields.
<code>check_type</code>	This bit determines whether the type, given by <code>uvm_object::get_type_name</code> , is used to verify that the types of two objects are the same.
<code>result</code>	This bit stores the number of mismatches for a given compare operation.

#### METHODS

<code>compare_field</code>	Compares two integral values.
<code>compare_field_int</code>	This method is the same as <code>compare_field</code> except that the arguments are small integers, less than or equal to 64 bits.
<code>compare_field_real</code>	This method is the same as <code>compare_field</code> except that the arguments are real numbers.
<code>compare_object</code>	Compares two class objects using the <code>policy</code> knob to determine whether the comparison should be deep, shallow, or reference.
<code>compare_string</code>	Compares two string variables.
<code>print_msg</code>	Causes the error count to be incremented and the message, <code>msg</code> , to be appended to the <code>mismatches</code> 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 messenger 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 mismatches for a given compare operation. You can use the result to determine the number of mismatches 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 mismatch.

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

<code>tr_handle</code>	This is an integral handle to a transaction object.
<code>default_radix</code>	This is the default radix setting if <code>record_field</code> is called without a radix.
<code>physical</code>	This bit provides a filtering mechanism for fields.
<code>abstract</code>	This bit provides a filtering mechanism for fields.
<code>identifier</code>	This bit is used to specify whether or not an object's reference should be recorded when the object is recorded.
<code>recursion_policy</code>	Sets the recursion policy for recording objects.

#### METHODS

<code>record_field</code>	Records an integral field (less than or equal to 4096 bits).
<code>record_field_real</code>	Records an real field.
<code>record_object</code>	Records an object field.
<code>record_string</code>	Records a string field.
<code>record_time</code>	Records a time value.
<code>record_generic</code>	Records the <i>name-value</i> pair, where <i>value</i> 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

<code>pack_field</code>	Packs an integral value (less than or equal to 4096 bits) into the packed array.
<code>pack_field_int</code>	Packs the integral value (less than or equal to 64 bits) into the pack array.
<code>pack_string</code>	Packs a string value into the pack array.
<code>pack_time</code>	Packs a time <i>value</i> as 64 bits into the pack array.
<code>pack_real</code>	Packs a real <i>value</i> as 64 bits into the pack array.
<code>pack_object</code>	Packs an object value into the pack array.

#### UNPACKING

<code>is_null</code>	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.
<code>unpack_field_int</code>	Unpacks bits from the pack array and returns the bit-stream that was unpacked.
<code>unpack_field</code>	Unpacks bits from the pack array and returns the bit-stream that was unpacked.
<code>unpack_string</code>	Unpacks a string.
<code>unpack_time</code>	Unpacks the next 64 bits of the pack array and places them into a time variable.
<code>unpack_real</code>	Unpacks the next 64 bits of the pack array and places them into a real variable.
<code>unpack_object</code>	Unpacks an object and stores the result into <i>value</i> .
<code>get_packed_size</code>	Returns the number of bits that were packed.

#### VARIABLES

<code>physical</code>	This bit provides a filtering mechanism for fields.
<code>abstract</code>	This bit provides a filtering mechanism for fields.
<code>use_metadata</code>	This flag indicates whether to encode metadata when packing dynamic data, or to decode metadata when unpacking.
<code>big_endian</code>	This bit determines the order that integral data is packed (using <code>pack_field</code> , <code>pack_field_int</code> , <code>pack_time</code> , or <code>pack_real</code> ) and how the data is unpacked from the pack array (using <code>unpack_field</code> , <code>unpack_field_int</code> , <code>unpack_time</code> , or <code>unpack_real</code> ).

## 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.

## 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
```

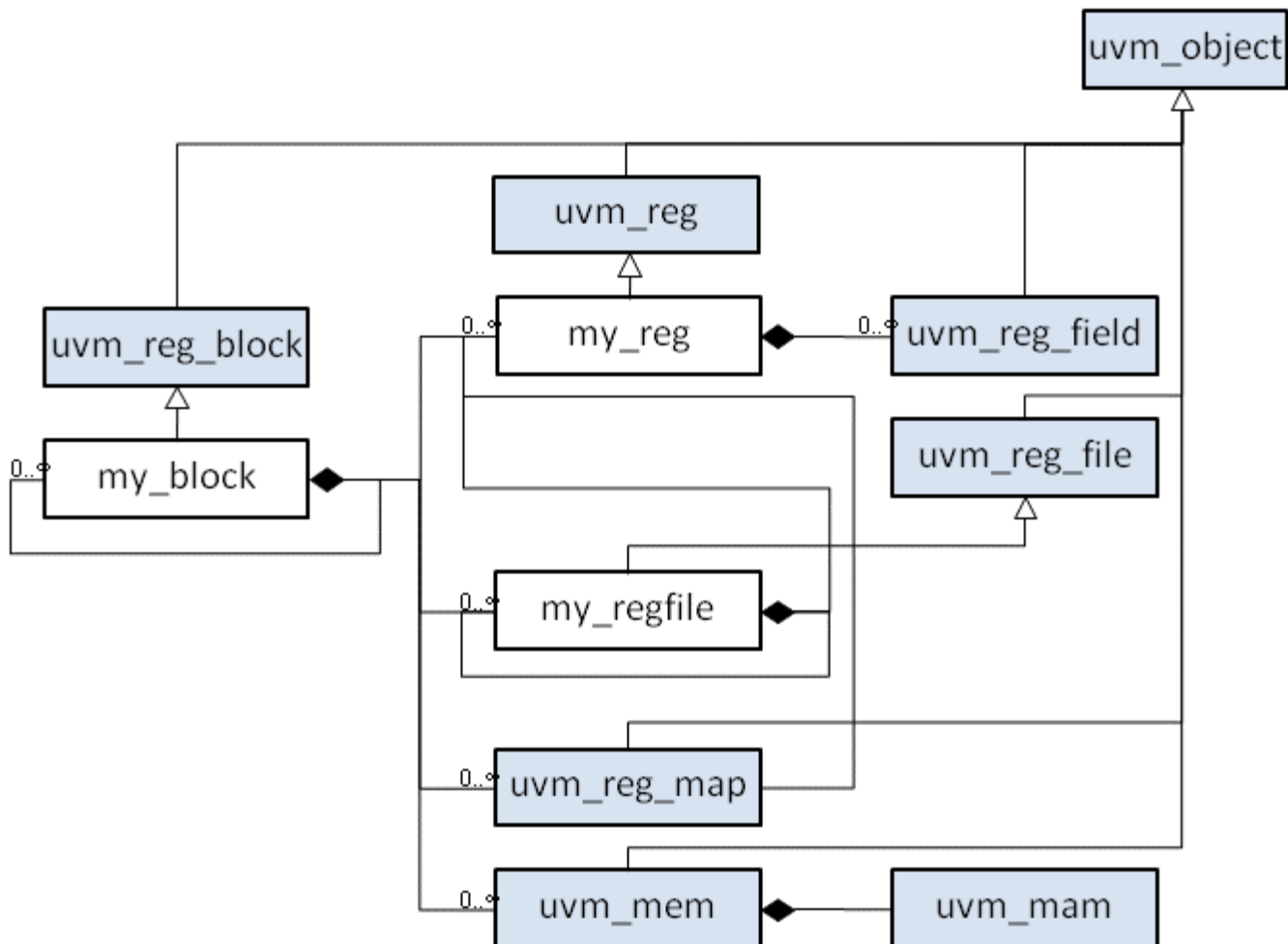
# Register Layer

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

### Global Declarations for the Register Layer

This section defines globally available types, enums, and utility classes.

#### TYPES

<a href="#">uvm_reg_data_t</a>	2-state data value with <code>`UVM_REG_DATA_WIDTH</code> bits
<a href="#">uvm_reg_data_logic_t</a>	4-state data value with <code>`UVM_REG_DATA_WIDTH</code> bits
<a href="#">uvm_reg_addr_t</a>	2-state address value with <code>`UVM_REG_ADDR_WIDTH</code> bits
<a href="#">uvm_reg_addr_logic_t</a>	4-state address value with <code>`UVM_REG_ADDR_WIDTH</code> bits
<a href="#">uvm_reg_byte_en_t</a>	2-state byte_enable value with <code>`UVM_REG_BYTENABLE_WIDTH</code> bits
<a href="#">uvm_reg_cvr_t</a>	Coverage model value set with <code>`UVM_REG_CVR_WIDTH</code> bits.
<a href="#">uvm_hdl_path_slice</a>	Slice of an HDL path

#### ENUMERATIONS

<a href="#">uvm_status_e</a>	Return status for register operations
<a href="#">uvm_path_e</a>	Path used for register operation
<a href="#">uvm_check_e</a>	Read-only or read-and-check
<a href="#">uvm_endianness_e</a>	Specifies byte ordering
<a href="#">uvm_elem_kind_e</a>	Type of element being read or written
<a href="#">uvm_access_e</a>	Type of operation begin performed
<a href="#">uvm_hier_e</a>	Whether to provide the requested information from a hierarchical context.
<a href="#">uvm_predict_e</a>	How the mirror is to be updated
<a href="#">uvm_coverage_model_e</a>	Coverage models available or desired.
<a href="#">uvm_reg_mem_tests_e</a>	Select which pre-defined test sequence to execute.

#### UTILITY CLASSES

<a href="#">uvm_hdl_path_concat</a>	Concatenation of HDL variables
<a href="#">uvm_utils</a>	This class contains useful template functions.

## 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.

<i>path</i>	Path to the HDL variable.
<i>offset</i>	Offset of the LSB in the register that this variable implements
<i>size</i>	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

<code>UVM_IS_OK</code>	Operation completed successfully
------------------------	----------------------------------

<i>UVM_NOT_OK</i>	Operation completed with error
<i>UVM_HAS_X</i>	Operation completed successfully bit had unknown bits.

## uvm\_path\_e

---

Path used for register operation

<i>UVM_FRONTDOOR</i>	Use the front door
<i>UVM_BACKDOOR</i>	Use the back door
<i>UVM_PREDICT</i>	Operation derived from observations by a bus monitor via the <a href="#">uvm_reg_predictor</a> class.
<i>UVM_DEFAULT_PATH</i>	Operation specified by the context

## uvm\_check\_e

---

Read-only or read-and-check

<i>UVM_NO_CHECK</i>	Read only
<i>UVM_CHECK</i>	Read and check

## uvm\_endianness\_e

---

Specifies byte ordering

<i>UVM_NO_ENDIAN</i>	Byte ordering not applicable
<i>UVM_LITTLE_ENDIAN</i>	Least-significant bytes first in consecutive addresses
<i>UVM_BIG_ENDIAN</i>	Most-significant bytes first in consecutive addresses
<i>UVM_LITTLE_FIFO</i>	Least-significant bytes first at the same address
<i>UVM_BIG_FIFO</i>	Most-significant bytes first at the same address

## uvm\_elem\_kind\_e

---

Type of element being read or written

<i>UVM_REG</i>	Register
<i>UVM_FIELD</i>	Field
<i>UVM_MEM</i>	Memory location

## uvm\_access\_e

---

Type of operation begin performed

<i>UVM_READ</i>	Read operation
<i>UVM_WRITE</i>	Write operation

## uvm\_hier\_e

---

Whether to provide the requested information from a hierarchical context.

<i>UVM_NO_HIER</i>	Provide info from the local context
<i>UVM_HIER</i>	Provide info based on the hierarchical context

## uvm\_predict\_e

---

How the mirror is to be updated

<i>UVM_PREDICT_DIRECT</i>	Predicted value is as-is
<i>UVM_PREDICT_READ</i>	Predict based on the specified value having been read
<i>UVM_PREDICT_WRITE</i>	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.

<i>UVM_NO_COVERAGE</i>	None
<i>UVM_CVR_REG_BITS</i>	Individual register bits
<i>UVM_CVR_ADDR_MAP</i>	Individual register and memory addresses
<i>UVM_CVR_FIELD_VALS</i>	Field values
<i>UVM_CVR_ALL</i>	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.

<i>UVM_DO_REG_HW_RESET</i>	Run <a href="#">uvm_reg_hw_reset_seq</a>
<i>UVM_DO_REG_BIT_BASH</i>	Run <a href="#">uvm_reg_bit_bash_seq</a>
<i>UVM_DO_REG_ACCESS</i>	Run <a href="#">uvm_reg_access_seq</a>
<i>UVM_DO_MEM_ACCESS</i>	Run <a href="#">uvm_mem_access_seq</a>
<i>UVM_DO_SHARED_ACCESS</i>	Run <a href="#">uvm_reg_mem_shared_access_seq</a>
<i>UVM_DO_MEM_WALK</i>	Run <a href="#">uvm_mem_walk_seq</a>
<i>UVM_DO_ALL_REG_MEM_TESTS</i>	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](#) specifying 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 exist in the concatenation if portions of the registers are not implemented.

For example, the following register

```
Bits:  1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
       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 implemented 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

<code>find_all</code>	Recursively finds all component instances of the parameter type <i>TYPE</i> , starting with the component given by <i>start</i> .
<code>get_config</code>	This method gets the object config of type <i>TYPE</i> associated with component <i>comp</i> .

## 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

<b>new</b>	Create a new instance and type-specific configuration
<b>configure</b>	Instance-specific configuration
<b>create_map</b>	Create an address map in this block
<b>check_data_width</b>	Check that the specified data width (in bits) is less than or equal to the value of `UVM_REG_DATA_WIDTH
<b>set_default_map</b>	Defines the default address map
<b>default_map</b>	Default address map
<b>lock_model</b>	Lock a model and build the address map.
<b>is_locked</b>	Return TRUE if the model is locked.

#### INTROSPECTION

<b>get_name</b>	Get the simple name
<b>get_full_name</b>	Get the hierarchical name
<b>get_parent</b>	Get the parent block
<b>get_root_blocks</b>	Get the all root blocks
<b>find_blocks</b>	Find the blocks whose hierarchical names match the specified <i>name</i> glob.
<b>find_block</b>	Find the first block whose hierarchical names match the specified <i>name</i> glob.
<b>get_blocks</b>	Get the sub-blocks
<b>get_maps</b>	Get the address maps
<b>get_registers</b>	Get the registers
<b>get_fields</b>	Get the fields
<b>get_virtual_registers</b>	Get the virtual registers
<b>get_virtual_fields</b>	Get the virtual fields
<b>get_block_by_name</b>	Finds a sub-block with the specified simple name.
<b>get_map_by_name</b>	Finds an address map with the specified simple name.
<b>get_reg_by_name</b>	Finds a register with the specified simple name.
<b>get_field_by_name</b>	Finds a field with the specified simple name.
<b>get_mem_by_name</b>	Finds a memory with the specified simple name.
<b>get_vreg_by_name</b>	Finds a virtual register with the specified simple



	name.
<code>get_vfield_by_name</code>	Finds a virtual field with the specified simple name.
<b>COVERAGE</b>	
<code>build_coverage</code>	Check if all of the specified coverage model must be built.
<code>add_coverage</code>	Specify that additional coverage models are available.
<code>has_coverage</code>	Check if block has coverage model(s)
<code>set_coverage</code>	Turns on coverage measurement.
<code>get_coverage</code>	Check if coverage measurement is on.
<code>sample</code>	Functional coverage measurement method
<code>sample_values</code>	Functional coverage measurement method for field values
<b>ACCESS</b>	
<code>get_default_path</code>	Default access path
<code>reset</code>	Reset the mirror for this block.
<code>needs_update</code>	Check if DUT registers need to be written
<code>update</code>	Batch update of register.
<code>mirror</code>	Update the mirrored values
<code>write_reg_by_name</code>	Write the named register
<code>read_reg_by_name</code>	Read the named register
<code>write_mem_by_name</code>	Write the named memory
<code>read_mem_by_name</code>	Read the named memory
<b>BACKDOOR</b>	
<code>get_backdoor</code>	Get the user-defined backdoor for all registers in this block
<code>set_backdoor</code>	Set the user-defined backdoor for all registers in this block
<code>clear_hdl_path</code>	Delete HDL paths
<code>add_hdl_path</code>	Add an HDL path
<code>has_hdl_path</code>	Check if a HDL path is specified
<code>get_hdl_path</code>	Get the incremental HDL path(s)
<code>get_full_hdl_path</code>	Get the full hierarchical HDL path(s)
<code>set_default_hdl_path</code>	Set the default design abstraction
<code>get_default_hdl_path</code>	Get the default design abstraction
<code>set_hdl_path_root</code>	Specify a root HDL path
<code>is_hdl_path_root</code>	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 implicit address map for a block with a single, unnamed 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 hierarchical 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_HIER)
```

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    = UVM_HIER)
```

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 identifiers. 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 successfully 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 least 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_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 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_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 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_DEFAULT,
                                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_DEFAULT,
                               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 overridden 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 overridden 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 abstraction 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 abstraction 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 abstraction 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 prepended 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 abstraction is specified, the default design abstraction for this block is used.

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

<a href="#">new</a>	Create a new instance
<a href="#">configure</a>	Instance-specific configuration
<a href="#">add_reg</a>	Add a register
<a href="#">add_mem</a>	Add a memory
<a href="#">add_submap</a>	Add an address map
<a href="#">set_sequencer</a>	Set the sequencer and adapter associated with this map.
<a href="#">set_submap_offset</a>	Set the offset of the given <i>submap</i> to <i>offset</i> .
<a href="#">get_submap_offset</a>	Return the offset of the given <i>submap</i> .
<a href="#">set_base_addr</a>	Set the base address of this map.
<a href="#">reset</a>	Reset the mirror for all registers in this address map.

#### INTROSPECTION

<a href="#">get_name</a>	Get the simple name
<a href="#">get_full_name</a>	Get the hierarchical name
<a href="#">get_root_map</a>	Get the externally-visible address map
<a href="#">get_parent</a>	Get the parent block
<a href="#">get_parent_map</a>	Get the higher-level address map
<a href="#">get_base_addr</a>	Get the base offset address for this map.
<a href="#">get_n_bytes</a>	Get the width in bytes of the bus associated with this map.
<a href="#">get_base_addr</a>	Gets the endianness of the bus associated with this map.
<a href="#">get_sequencer</a>	Gets the sequencer for the bus associated with this map.
<a href="#">get_adapter</a>	Gets the bus adapter for the bus associated with this map.
<a href="#">get_submaps</a>	Get the address sub-maps
<a href="#">get_registers</a>	Get the registers
<a href="#">get_fields</a>	Get the fields
<a href="#">get_virtual_registers</a>	Get the virtual registers
<a href="#">get_virtual_fields</a>	Get the virtual fields
<a href="#">get_physical_addresses</a>	Translate a local address into external addresses

<code>get_reg_by_offset</code>	Get register mapped at offset
<code>get_mem_by_offset</code>	Get memory mapped at offset

#### Bus Access

<code>set_auto_predict</code>	Sets the auto-predict mode for this map.
<code>get_auto_predict</code>	Gets the auto-predict mode setting for this map.
<code>do_bus_write</code>	Perform a bus write operation.
<code>do_bus_read</code>	Perform a bus read operation.
<code>do_write</code>	Perform a write operation.
<code>do_read</code>	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.

<i>parent</i>	the block in which this map is created and applied
<i>base_addr</i>	the base address for this map. All registers, memories, and sub-blocks will be at offsets to this address
<i>n_bytes</i>	the byte-width of the bus on which this map is used
<i>endian</i>	the endian format. See <a href="#">uvm_endianness_e</a> for possible values
<i>byte_addressing</i>	specifies whether the address increment is on a per-byte basis. For example, consecutive memory locations with $\sim n\_bytes \sim = 4$ (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 memories 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      = 0 )
```

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 this 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 abstraction 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 abstraction 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 abstraction 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 abstraction for the nearest register file or block ancestor. Returns "" if no default design abstraction has been specified.

Register abstraction base class

A register represents a set of fields that are accessible as a single entity.

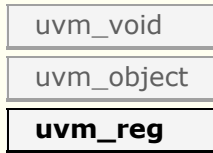
A register may be mapped to one or more address maps, each with different access rights and policy.

## Summary

### uvm\_reg

Register abstraction base class

#### CLASS HIERARCHY



#### CLASS DECLARATION

```
virtual class uvm_reg extends uvm_object
```

#### INITIALIZATION

<code>new</code>	Create a new instance and type-specific configuration
<code>configure</code>	Instance-specific configuration
<code>set_offset</code>	Modify the offset of the register

#### INTROSPECTION

<code>get_name</code>	Get the simple name
<code>get_full_name</code>	Get the hierarchical name
<code>get_parent</code>	Get the parent block
<code>get_regfile</code>	Get the parent register file
<code>get_n_maps</code>	Returns the number of address maps this register is mapped in
<code>is_in_map</code>	Returns 1 if this register is in the specified address <i>map</i>
<code>get_maps</code>	Returns all of the address <i>maps</i> where this register is mapped
<code>get_rights</code>	Returns the access rights of this register.
<code>get_n_bits</code>	Returns the width, in bits, of this register.
<code>get_n_bytes</code>	Returns the width, in bytes, of this register.
<code>get_max_size</code>	Returns the maximum width, in bits, of all registers.
<code>get_fields</code>	Return the fields in this register
<code>get_field_by_name</code>	Return the named field in this register
<code>get_offset</code>	Returns the offset of this register
<code>get_address</code>	Returns the base external physical address of this register
<code>get_addresses</code>	Identifies the external physical address(es) of this register

#### ACCESS

<code>set</code>	Set the desired value for this register
<code>get</code>	Return the desired value of the fields in the register.
<code>needs_update</code>	Returns 1 if any of the fields need updating
<code>reset</code>	Reset the desired/mirrored value for this register.
<code>get_reset</code>	Get the specified reset value for this register
<code>has_reset</code>	Check if any field in the register has a reset value specified for the specified reset <i>kind</i> .
<code>set_reset</code>	Specify or modify the reset value for this register
<code>write</code>	Write the specified value in this register
<code>read</code>	Read the current value from this register

<code>poke</code>	Deposit the specified value in this register
<code>peek</code>	Read the current value from this register
<code>update</code>	Updates the content of the register in the design to match the desired value
<code>mirror</code>	Read the register and update/check its mirror value
<code>predict</code>	Update the mirrored value for this register.
<code>is_busy</code>	Returns 1 if register is currently being read or written.

#### FRONTDOOR

<code>set_frontend</code>	Set a user-defined frontend for this register
<code>get_frontend</code>	Returns the user-defined frontend for this register

#### BACKDOOR

<code>set_backdoor</code>	Set a user-defined backdoor for this register
<code>get_backdoor</code>	Returns the user-defined backdoor for this register
<code>clear_hdl_path</code>	Delete HDL paths
<code>add_hdl_path</code>	Add an HDL path
<code>add_hdl_path_slice</code>	Append the specified HDL slice to the HDL path of the register instance for the specified design abstraction.
<code>has_hdl_path</code>	Check if a HDL path is specified
<code>get_hdl_path</code>	Get the incremental HDL path(s)
<code>get_hdl_path_kinds</code>	Get design abstractions for which HDL paths have been defined
<code>get_full_hdl_path</code>	Get the full hierarchical HDL path(s)
<code>backdoor_read</code>	User-define backdoor read access
<code>backdoor_write</code>	User-defined backdoor read access
<code>backdoor_read_func</code>	User-defined backdoor read access
<code>backdoor_watch</code>	User-defined DUT register change monitor

#### COVERAGE

<code>include_coverage</code>	Specify which coverage model that must be included in various block, register or memory abstraction class instances.
<code>build_coverage</code>	Check if all of the specified coverage models must be built.
<code>add_coverage</code>	Specify that additional coverage models are available.
<code>has_coverage</code>	Check if register has coverage model(s)
<code>set_coverage</code>	Turns on coverage measurement.
<code>get_coverage</code>	Check if coverage measurement is on.
<code>sample</code>	Functional coverage measurement method
<code>sample_values</code>	Functional coverage measurement method for field values

#### CALLBACKS

<code>pre_write</code>	Called before register write.
<code>post_write</code>	Called after register write.
<code>pre_read</code>	Called before register read.
<code>post_read</code>	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 addresses are specified in little endian order. Returns the number of bytes transferred 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 abstraction 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 address *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

```
Bits:  1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
        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 implemented 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 abstraction 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 abstraction 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 identifiers. 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 address map use the access the register (thus the field).

## Summary

### uvm\_reg\_field

Field abstraction class

#### CLASS HIERARCHY



#### 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

<code>pre_write</code>	Called before field write.
<code>post_write</code>	Called after field write.
<code>pre_read</code>	Called before field read.
<code>post_read</code>	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 <i>HARD</i> reset is as-is, other W have no effects, R: no effect
"W01"	W: first one after <i>HARD</i> 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 specific access value is valid or not. To help catch typing errors, user-defined access values must be defined using this method to avoid being 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 abstraction 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 field 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 abstraction 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.

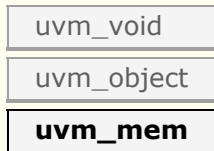
Unlike registers, memories are not mirrored because of the potentially large data space: tests that walk the entire memory space would negate any benefit from sparse memory modelling techniques. Rather than relying on a mirror, it is recommended that backdoor access be used instead.

## Summary

### uvm\_mem

Memory abstraction base class

#### CLASS HIERARCHY



#### CLASS DECLARATION

```
class uvm_mem extends uvm_object
```

#### INITIALIZATION

<code>new</code>	Create a new instance and type-specific configuration
<code>configure</code>	Instance-specific configuration
<code>set_offset</code>	Modify the offset of the memory
<code>Modifying the offset of a memory will make the abstract model</code>	diverge from the specification that was used to create it.
<code>mam</code>	Memory allocation manager

#### INTROSPECTION

<code>get_name</code>	Get the simple name
<code>get_full_name</code>	Get the hierarchical name
<code>get_parent</code>	Get the parent block
<code>get_n_maps</code>	Returns the number of address maps this memory is mapped in
<code>is_in_map</code>	Return TRUE if this memory is in the specified address <i>map</i>
<code>get_maps</code>	Returns all of the address <i>maps</i> where this memory is mapped
<code>get_rights</code>	Returns the access rights of this memory.
<code>get_access</code>	Returns the access policy of the memory when written and read via an address map.
<code>get_size</code>	Returns the number of unique memory locations in this memory.
<code>get_n_bytes</code>	Return the width, in number of bytes, of each memory location
<code>get_n_bits</code>	Returns the width, in number of bits, of each memory location
<code>get_max_size</code>	Returns the maximum width, in number of bits, of all memories
<code>get_virtual_registers</code>	Return the virtual registers in this memory
<code>get_virtual_fields</code>	Return the virtual fields in the memory
<code>get_vreg_by_name</code>	Find the named virtual register



<code>get_vfield_by_name</code>	Find the named virtual field
<code>get_vreg_by_offset</code>	Find the virtual register implemented at the specified offset
<code>get_offset</code>	Returns the base offset of a memory location
<code>get_address</code>	Returns the base external physical address of a memory location
<code>get_addresses</code>	Identifies the external physical address(es) of a memory location

#### **HDL ACCESS**

<code>write</code>	Write the specified value in a memory location
<code>read</code>	Read the current value from a memory location
<code>burst_write</code>	Write the specified values in memory locations
<code>burst_read</code>	Read values from memory locations
<code>poke</code>	Deposit the specified value in a memory location
<code>peek</code>	Read the current value from a memory location

#### **FRONTDOOR**

<code>set_frontdoor</code>	Set a user-defined frontdoor for this memory
<code>get_frontdoor</code>	Returns the user-defined frontdoor for this memory

#### **BACKDOOR**

<code>set_backdoor</code>	Set a user-defined backdoor for this memory
<code>get_backdoor</code>	Returns the user-defined backdoor for this memory
<code>clear_hdl_path</code>	Delete HDL paths
<code>add_hdl_path</code>	Add an HDL path
<code>add_hdl_path_slice</code>	Add the specified HDL slice to the HDL path for the specified design abstraction.
<code>has_hdl_path</code>	Check if a HDL path is specified
<code>get_hdl_path</code>	Get the incremental HDL path(s)
<code>get_full_hdl_path</code>	Get the full hierarchical HDL path(s)
<code>get_hdl_path_kinds</code>	Get design abstractions for which HDL paths have been defined
<code>backdoor_read</code>	User-define backdoor read access
<code>backdoor_write</code>	User-defined backdoor read access
<code>backdoor_read_func</code>	User-defined backdoor read access

#### **CALLBACKS**

<code>pre_write</code>	Called before memory write.
<code>post_write</code>	Called after memory write.
<code>pre_read</code>	Called before memory read.
<code>post_read</code>	Called after memory read.

#### **COVERAGE**

<code>build_coverage</code>	Check if all of the specified coverage model must be built.
<code>add_coverage</code>	Specify that additional coverage models are available.
<code>has_coverage</code>	Check if memory has coverage model(s)
<code>set_coverage</code>	Turns on coverage measurement.
<code>get_coverage</code>	Check if coverage measurement is on.
<code>sample</code>	Functional coverage measurement method

## INITIALIZATION

## new

---

```
function new (      string    name,  
                 longint    size,  
                 int        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.

### 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 addresses are specified in little endian order. Returns the number of bytes transferred 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_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 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 specified *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 abstraction 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, memories are mapped linearly into the address space of the address maps that instantiate them. If memories are accessed using a different mechanism, a user-defined access mechanism must be defined and associated with the corresponding



memory abstraction class

If the memory is mapped in multiple address maps, an address *map* must be specified.

## get\_frontend

---

```
function uvm_reg_frontend get_frontend(uvm_reg_map map = null)
```

Returns the user-defined frontend for this memory

If null, no user-defined frontend has been defined. A user-defined frontend is defined by using the [uvm\\_mem::set\\_frontend\(\)](#) 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 abstraction 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 abstraction 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 identifiers. 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 successfully 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.

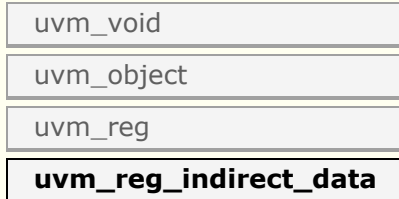
This class should not be instantiated directly. A type-specific class extension should be used to provide a factory-enabled constructor and specify the *n\_bits* and coverage models.

## Summary

### uvm\_reg\_indirect\_data

Indirect data access abstraction class

#### CLASS HIERARCHY



#### CLASS DECLARATION

```
class uvm_reg_indirect_data extends uvm_reg
```

#### METHODS

<code>new</code>	Create an instance of this class
<code>configure</code>	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



#### 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-prediction 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

<b>uvm_vreg</b>	A virtual register is a collection of fields, overlaid on top of a memory, usually in an array.
<a href="#">uvm_vreg</a>	Virtual register abstraction base class
<a href="#">uvm_vreg_cbs</a>	Pre/post read/write callback facade class

# 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

<b>uvm_vreg</b>	
Virtual register abstraction base class	
<b>CLASS HIERARCHY</b>	
<div>uvm_void</div> <div>uvm_object</div> <div><b>uvm_vreg</b></div>	
<b>CLASS DECLARATION</b>	
<div>class uvm_vreg extends uvm_object</div>	
<b>INITIALIZATION</b>	
<a href="#">new</a>	Create a new instance and type-specific configuration
<a href="#">configure</a>	Instance-specific configuration
<a href="#">implement</a>	Dynamically implement, resize or relocate a virtual register array
<a href="#">allocate</a>	Randomly implement, resize or relocate a virtual register array
<a href="#">get_region</a>	Get the region where the virtual register array is implemented
<a href="#">release_region</a>	Dynamically un-implement a virtual register array
<b>INTROSPECTION</b>	
<a href="#">get_name</a>	Get the simple name

<code>get_full_name</code>	Get the hierarchical name
<code>get_parent</code>	Get the parent block
<code>get_memory</code>	Get the memory where the virtual register array is implemented
<code>get_n_maps</code>	Returns the number of address maps this virtual register array is mapped in
<code>is_in_map</code>	Return TRUE if this virtual register array is in the specified address <i>map</i>
<code>get_maps</code>	Returns all of the address <i>maps</i> where this virtual register array is mapped
<code>get_rights</code>	Returns the access rights of this virtual register array
<code>get_access</code>	Returns the access policy of the virtual register array when written and read via an address map.
<code>get_size</code>	Returns the size of the virtual register array.
<code>get_n_bytes</code>	Returns the width, in bytes, of a virtual register.
<code>get_n_memlocs</code>	Returns the number of memory locations used by a single virtual register.
<code>get_incr</code>	Returns the number of memory locations between two individual virtual registers in the same array.
<code>get_fields</code>	Return the virtual fields in this virtual register
<code>get_field_by_name</code>	Return the named virtual field in this virtual register
<code>get_offset_in_memory</code>	Returns the offset of a virtual register
<code>get_address</code>	Returns the base external physical address of a virtual register

#### HDL Access

<code>write</code>	Write the specified value in a virtual register
<code>read</code>	Read the current value from a virtual register
<code>poke</code>	Deposit the specified value in a virtual register
<code>peek</code>	Sample the current value in a virtual register
<code>reset</code>	Reset the access semaphore

#### CALLBACKS

<code>pre_write</code>	Called before virtual register write.
<code>post_write</code>	Called after virtual register write.
<code>pre_read</code>	Called before virtual register read.
<code>post_read</code>	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 register 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 register 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 register 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_PATH,
                  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_PATH)
```

```

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

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\_voiduvm\_objectuvm\_callbackuvm\_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

<b>Virtual Register Field Classes</b>	This section defines the virtual field and callback classes.
<code>uvm_vreg_field</code>	Virtual field abstraction class
<code>uvm_vreg_field_cbs</code>	Pre/post read/write callback facade class

# 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**

<code>new</code>	Create a new virtual field instance
<code>configure</code>	Instance-specific configuration

**INTROSPECTION**

<code>get_name</code>	Get the simple name
<code>get_full_name</code>	Get the hierarchical name
<code>get_parent</code>	Get the parent virtual register
<code>get_lsb_pos_in_register</code>	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.
<code>get_n_bits</code>	Returns the width, in bits, of the virtual field.
<code>get_access</code>	Returns the access policy of the virtual field register when written and read via an address map.

**HDL ACCESS**

<code>write</code>	Write the specified value in a virtual field
--------------------	--

<code>read</code>	Read the current value from a virtual field
<code>poke</code>	Deposit the specified value in a virtual field
<code>peek</code>	Sample the current value from a virtual field

#### **CALLBACKS**

<code>pre_write</code>	Called before virtual field write.
<code>post_write</code>	Called after virtual field write
<code>pre_read</code>	Called before virtual field read.
<code>post_read</code>	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_PATH,
                    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 implementing 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_PATH,
                  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



#### CLASS DECLARATION

```
class uvm_vreg_field_cbs extends uvm_callback
```

#### METHODS

<a href="#">pre_write</a>	Callback called before a write operation.
<a href="#">post_write</a>	Called after a write operation
<a href="#">pre_read</a>	Called before a virtual field read.
<a href="#">post_read</a>	Called after a virtual field read.

#### TYPES

<a href="#">uvm_vreg_field_cb</a>	Convenience callback type declaration
<a href="#">uvm_vreg_field_cb_iter</a>	Convenience callback iterator type declaration

### 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

<b>Register Callbacks</b>	This section defines the base class used for all register callback extensions.
<a href="#">uvm_reg_cbs</a>	Facade class for field, register, memory and backdoor access callback methods.
<b>Typedefs</b>	
<a href="#">uvm_reg_cb</a>	Convenience callback type declaration for registers
<a href="#">uvm_reg_cb_iter</a>	Convenience callback iterator type declaration for registers
<a href="#">uvm_reg_bd_cb</a>	Convenience callback type declaration for backdoor
<a href="#">uvm_reg_bd_cb_iter</a>	Convenience callback iterator type declaration for backdoor
<a href="#">uvm_mem_cb</a>	Convenience callback type declaration for memories
<a href="#">uvm_mem_cb_iter</a>	Convenience callback iterator type declaration for memories
<a href="#">uvm_reg_field_cb</a>	Convenience callback type declaration for fields
<a href="#">uvm_reg_field_cb_iter</a>	Convenience callback iterator type declaration for fields
<b>PREDEFINED EXTENSIONS</b>	
<a href="#">uvm_reg_read_only_cbs</a>	Pre-defined register callback method for read-only registers that will issue an error if a write() operation is attempted.
<a href="#">uvm_reg_write_only_cbs</a>	Pre-defined register callback method for write-only registers that will issue an error if a read() operation is attempted.

# uvm\_reg\_cbs

Facade class for field, register, memory and backdoor access callback methods.

## Summary

<b>uvm_reg_cbs</b>
Facade class for field, register, memory and backdoor access callback methods.
<b>CLASS HIERARCHY</b>
<div>uvm_void</div> <div>uvm_object</div> <div>uvm_callback</div> <div><b>uvm_reg_cbs</b></div>
<b>CLASS DECLARATION</b>
<div>virtual class uvm_reg_cbs extends uvm_callback</div>

## METHODS

<code>pre_write</code>	Called before a write operation.
<code>post_write</code>	Called after user-defined backdoor register write.
<code>pre_read</code>	Callback called before a read operation.
<code>post_read</code>	Callback called after a read operation.
<code>post_predict</code>	Called by the <code>uvm_reg_field::predict()</code> method after a successful UVM_PREDICT_READ or UVM_PREDICT_WRITE prediction.
<code>encode</code>	Data encoder
<code>decode</code>	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`.

<i>Backdoor</i>	<code>uvm_reg_backdoor::pre_write</code> , <code>uvm_reg_cbs::pre_write</code> cbs for backdoor.
<i>Register</i>	<code>uvm_reg::pre_write</code> , <code>uvm_reg_cbs::pre_write</code> cbs for reg, then foreach field: <code>uvm_reg_field::pre_write</code> , <code>uvm_reg_cbs::pre_write</code> cbs for field
<i>RegField</i>	<code>uvm_reg_field::pre_write</code> , <code>uvm_reg_cbs::pre_write</code> cbs for field
<i>Memory</i>	<code>uvm_mem::pre_write</code> , <code>uvm_reg_cbs::pre_write</code> 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

<i>Backdoor</i>	<a href="#">uvm_reg_cbs::post_write</a> cbs for backdoor, <a href="#">uvm_reg_backdoor::post_write</a>
<i>Register</i>	<a href="#">uvm_reg_cbs::post_write</a> cbs for reg, <a href="#">uvm_reg::post_write</a> , then foreach field: <a href="#">uvm_reg_cbs::post_write</a> cbs for field, <a href="#">uvm_reg_field::post_read</a>
<i>RegField</i>	<a href="#">uvm_reg_cbs::post_write</a> cbs for field, <a href="#">uvm_reg_field::post_write</a>
<i>Memory</i>	<a href="#">uvm_reg_cbs::post_write</a> cbs for mem, <a href="#">uvm_mem::post_write</a>

The *rw* argument holds information about the operation.

- Modifying the *status* member modifies the returned status.
- Modifying 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](#).

<i>Backdoor</i>	<a href="#">uvm_reg_backdoor::pre_read</a> , <a href="#">uvm_reg_cbs::pre_read</a> cbs for backdoor
<i>Register</i>	<a href="#">uvm_reg::pre_read</a> , <a href="#">uvm_reg_cbs::pre_read</a> cbs for reg, then foreach field: <a href="#">uvm_reg_field::pre_read</a> , <a href="#">uvm_reg_cbs::pre_read</a> cbs for field
<i>RegField</i>	<a href="#">uvm_reg_field::pre_read</a> , <a href="#">uvm_reg_cbs::pre_read</a> cbs for field
<i>Memory</i>	<a href="#">uvm_mem::pre_read</a> , <a href="#">uvm_reg_cbs::pre_read</a> 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](#).

<i>Backdoor</i>	<a href="#">uvm_reg_cbs::post_read</a> cbs for backdoor, <a href="#">uvm_reg_backdoor::post_read</a>
<i>Register</i>	<a href="#">uvm_reg_cbs::post_read</a> cbs for reg, <a href="#">uvm_reg::post_read</a> , then foreach field: <a href="#">uvm_reg_cbs::post_read</a> cbs for field, <a href="#">uvm_reg_field::post_read</a>
<i>RegField</i>	<a href="#">uvm_reg_cbs::post_read</a> cbs for field, <a href="#">uvm_reg_field::post_read</a>
<i>Memory</i>	<a href="#">uvm_reg_cbs::post_read</a> cbs for mem, <a href="#">uvm_mem::post_read</a>

The *rw* argument holds information about the operation.

- Modifying the readback *value* or *status* modifies the actual returned value and status.
- Modifying 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

<a href="#">uvm_reg_cb</a>	Convenience callback type declaration for registers
<a href="#">uvm_reg_cb_iter</a>	Convenience callback iterator type declaration for registers
<a href="#">uvm_reg_bd_cb</a>	Convenience callback type declaration for backdoor
<a href="#">uvm_reg_bd_cb_iter</a>	Convenience callback iterator type declaration for backdoor
<a href="#">uvm_mem_cb</a>	Convenience callback type declaration for memories
<a href="#">uvm_mem_cb_iter</a>	Convenience callback iterator type declaration for memories
<a href="#">uvm_reg_field_cb</a>	Convenience callback type declaration for fields
<a href="#">uvm_reg_field_cb_iter</a>	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

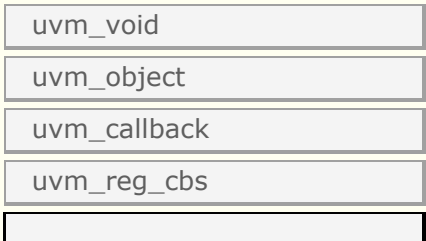
Pre-defined register callback method for read-only registers that will issue an error if a write() operation is attempted.

### 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\_reg\_read\_only\_cbs

### CLASS DECLARATION

```
class uvm_reg_read_only_cbs extends uvm_reg_cbs
```

### METHODS

<code>pre_write</code>	Produces an error message and sets status to <code>UVM_NOT_OK</code> .
<code>add</code>	Add this callback to the specified register and its contained fields.
<code>remove</code>	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](#)

[add](#)

[remove](#)

Produces an error message and sets status to **UVM\_NOT\_OK**.

Add this callback to the specified register and its contained fields.

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

### Memory Allocation Manager

Manages the exclusive allocation of consecutive memory locations called *regions*.

<code>uvm_mem_mam</code>	Memory allocation manager
<code>uvm_mem_region</code>	Allocated memory region descriptor
<code>uvm_mem_mam_policy</code>	An instance of this class is randomized to determine the starting offset of a randomly allocated memory region.
<code>uvm_mem_mam_cfg</code>	Specifies the memory managed by an instance of a <code>uvm_mem_mam</code> memory allocation manager class.

## 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

<code>alloc_mode_e</code>	Memory allocation mode
<code>locality_e</code>	Location of memory regions
<code>default_alloc</code>	Region allocation policy
<code>new</code>	Create a new manager instance
<code>reconfigure</code>	Reconfigure the manager

#### MEMORY MANAGEMENT

<code>reserve_region</code>	Reserve a specific memory region
<code>request_region</code>	Request and reserve a memory region
<code>release_region</code>	Release the specified region
<code>release_all_regions</code>	Forcibly release all allocated memory regions.

#### INTROSPECTION

<code>convert2string</code>	Image of the state of the manager
<code>for_each</code>	Iterate over all currently allocated regions
<code>get_memory</code>	Get the managed memory implementation

## INITIALIZATION

---

### alloc\_mode\_e

---

Memory allocation mode

Specifies how to allocate a memory region

<i>GREEDY</i>	Consume new, previously unallocated memory
<i>THRIFTY</i>	Reused previously released memory as much as possible (not yet implemented)

### locality\_e

---

Location of memory regions

Specifies where to locate new memory regions

<i>BROAD</i>	Locate new regions randomly throughout the address space
<i>NEARBY</i>	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

<a href="#">get_start_offset</a>	Get the start offset of the region
<a href="#">get_end_offset</a>	Get the end offset of the region
<a href="#">get_len</a>	Size of the memory region
<a href="#">get_n_bytes</a>	Number of bytes in the region
<a href="#">release_region</a>	Release this region
<a href="#">get_memory</a>	Get the memory where the region resides
<a href="#">get_virtual_registers</a>	Get the virtual register array in this region
<a href="#">write</a>	Write to a memory location in the region.
<a href="#">read</a>	Read from a memory location in the region.
<a href="#">burst_write</a>	Write to a set of memory location in the region.
<a href="#">burst_read</a>	Read from a set of memory location in the region.
<a href="#">poke</a>	Deposit in a memory location in the region.
<a href="#">peek</a>	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

<code>len</code>	Number of addresses required
<code>start_offset</code>	The starting offset of the region
<code>min_offset</code>	Minimum address offset in the managed address space
<code>max_offset</code>	Maximum address offset in the managed address space
<code>in_use</code>	Regions already allocated in the managed address space



### 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

<code>n_bytes</code>	Number of bytes in each memory location
<code>end_offset</code>	Last address of managed space
<code>mode</code>	Region allocation mode
<code>locality</code>	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 abstract 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

<b>Generic Register Operation Descriptors</b>	This section defines the abstract register transaction item.
<code>uvm_reg_item</code>	Defines an abstract register transaction item.
<code>uvm_reg_bus_op</code>	Struct that defines a generic bus transaction for register and memory accesses, having <i>kind</i> (read or write), <i>address</i> , <i>data</i> , and <i>byte enable</i> information.

# 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

<b>uvm_reg_item</b>	
Defines an abstract register transaction item.	
<b>CLASS HIERARCHY</b>	
<code>uvm_void</code>	
<code>uvm_object</code>	
<code>uvm_transaction</code>	
<code>uvm_sequence_item</code>	
<b>uvm_reg_item</b>	
<b>CLASS DECLARATION</b>	
<code>class uvm_reg_item extends uvm_sequence_item</code>	
<b>VARIABLES</b>	
<code>element_kind</code>	Kind of element being accessed: REG, MEM, or FIELD.
<code>element</code>	A handle to the RegModel model element associated with this transaction.
<code>kind</code>	Kind of access: READ or WRITE.
<code>value</code>	The value to write to, or after completion, the value read from the DUT.
<code>offset</code>	For memory accesses, the offset address.
<code>status</code>	The result of the transaction: IS_OK, HAS_X, or ERROR.
<code>local_map</code>	The local map used to obtain addresses.
<code>map</code>	The original map specified for the operation.

<code>path</code>	The path being used: <code>UVM_FRONTDOOR</code> or <code>UVM_BACKDOOR</code> .
<code>parent</code>	The sequence from which the operation originated.
<code>prior</code>	The priority requested of this transfer, as defined by <code>uvm_sequence_base::start_item</code> .
<code>extension</code>	Handle to optional user data, as conveyed in the call to write, read, mirror, or update call.
<code>bd_kind</code>	If path is <code>UVM_BACKDOOR</code> , this member specifies the abstraction kind for the backdoor access, e.g.
<code>fname</code>	The file name from where this transaction originated, if provided at the call site.
<code>lineno</code>	The file name from where this transaction originated, if provided at the call site.
<b>METHODS</b>	
<code>new</code>	Create a new instance of this type, giving it the optional <i>name</i> .
<code>convert2string</code>	Returns a string showing the contents of this transaction.
<code>do_copy</code>	Copy the <i>rhs</i> 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

<code>info</code>	The bus-independent read/write information.
<code>kind</code>	Kind of access: READ or WRITE.
<code>addr</code>	The bus address.
<code>data</code>	The data to write.
<code>n_bits</code>	The number of bits of <code>uvm_reg_item::value</code> being transferred by this transaction.
<code>byte_en</code>	Enables for the byte lanes on the bus.
<code>status</code>	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

<b>Classes for Adapting Between Register and Bus Operations</b>	This section defines classes used to convert transaction streams between generic register address/data reads and writes and physical bus accesses.
<code>uvm_reg_adapter</code>	This class defines an interface for converting between <code>uvm_reg_bus_op</code> and a specific bus transaction.
<code>uvm_reg_tlm_adapter</code>	For converting between <code>uvm_reg_bus_op</code> and <code>uvm_tlm_gp</code> items.

## uvm\_reg\_adapter

This class defines an interface for converting between `uvm_reg_bus_op` and a specific bus transaction.

## Summary

<b>uvm_reg_adapter</b>	
This class defines an interface for converting between <code>uvm_reg_bus_op</code> and a specific bus transaction.	
<b>CLASS HIERARCHY</b>	
<div>uvm_void</div> <div>uvm_object</div> <div><b>uvm_reg_adapter</b></div>	
<b>CLASS DECLARATION</b>	
<pre>virtual class uvm_reg_adapter extends uvm_object</pre>	
<code>new</code>	Create a new instance of this type, giving it the optional <i>name</i> .
<code>supports_byte_enable</code>	Set this bit in extensions of this class if the bus protocol supports byte enables.
<code>provides_responses</code>	Set this bit in extensions of this class if the bus driver provides separate response items.
<code>reg2bus</code>	Extensions of this class <i>must</i> implement this method to convert a <code>uvm_reg_item</code> to the <code>uvm_sequence_item</code> subtype that defines the bus transaction.
<code>bus2reg</code>	Extensions of this class <i>must</i> implement this method to copy members of the given <i>bus_item</i> to corresponding members of the provided <i>bus_rw</i> instance.
<b>EXAMPLE</b>	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(rreg2apb_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



##### CLASS DECLARATION

```
class uvm_reg_tlm_adapter extends uvm_reg_adapter
```

##### METHODS

<a href="#">reg2bus</a>	Converts a <a href="#">uvm_reg_bus_op</a> struct to a <a href="#">uvm_tlm_gp</a> item.
<a href="#">bus2reg</a>	Converts a <a href="#">uvm_tlm_gp</a> item to a <a href="#">uvm_reg_bus_op</a> .

## 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

<b>Register Sequence and Predictor Classes</b>	This section defines the base classes used for register stimulus generation.
<a href="#">uvm_reg_sequence</a>	This class provides base functionality for both user-defined RegModel test sequences and "register translation sequences".
<a href="#">uvm_reg_frontdoor</a>	Facade class for register and memory frontdoor access.
<a href="#">uvm_reg_predictor</a>	Updates the register model mirror based on observed bus transactions

## 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 sequencer which 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

BASE

**uvm\_reg\_sequence**

#### CLASS DECLARATION

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

[BASE](#) Specifies the sequence type to extend from.

<code>model</code>	Block abstraction this sequence executes on, defined only when this sequence is a user-defined test sequence.
<code>adapter</code>	Adapter to use for translating between abstract register transactions and physical bus transactions, defined only when this sequence is a translation sequence.
<code>reg_seqr</code>	Layered upstream “register” sequencer.
<code>new</code>	Create a new instance, giving it the optional <i>name</i> .
<code>body</code>	Continually gets a register transaction from the configured upstream sequencer, <code>reg_seqr</code> , and executes the corresponding bus transaction via <code>&lt;do_rw_access&gt;</code> .
<code>do_reg_item</code>	Executes the given register transaction, <i>rw</i> , via the sequencer on which this sequence was started (i.e.
<b>CONVENIENCE WRITE/ READ API</b>	The following methods delegate to the corresponding method in the register or memory element.
<code>write_reg</code>	Writes the given register <i>rg</i> using <code>uvm_reg::write</code> , supplying ‘this’ as the <i>parent</i> argument.
<code>read_reg</code>	Reads the given register <i>rg</i> using <code>uvm_reg::read</code> , supplying ‘this’ as the <i>parent</i> argument.
<code>poke_reg</code>	Pokes the given register <i>rg</i> using <code>uvm_reg::poke</code> , supplying ‘this’ as the <i>parent</i> argument.
<code>peek_reg</code>	Peeks the given register <i>rg</i> using <code>uvm_reg::peek</code> , supplying ‘this’ as the <i>parent</i> argument.
<code>update_reg</code>	Updates the given register <i>rg</i> using <code>uvm_reg::update</code> , supplying ‘this’ as the <i>parent</i> argument.
<code>mirror_reg</code>	Mirrors the given register <i>rg</i> using <code>uvm_reg::mirror</code> , supplying ‘this’ as the <i>parent</i> argument.
<code>write_mem</code>	Writes the given memory <i>mem</i> using <code>uvm_mem::write</code> , supplying ‘this’ as the <i>parent</i> argument.
<code>read_mem</code>	Reads the given memory <i>mem</i> using <code>uvm_mem::read</code> , supplying ‘this’ as the <i>parent</i> argument.
<code>poke_mem</code>	Pokes the given memory <i>mem</i> using <code>uvm_mem::poke</code> , supplying ‘this’ as the <i>parent</i> argument.
<code>peek_mem</code>	Peeks the given memory <i>mem</i> using <code>uvm_mem::peek</code> , supplying ‘this’ as the <i>parent</i> 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

<code>rw_info</code>	Holds information about the register being read or written
<code>sequencer</code>	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

<code>bus_in</code>	Observed bus transactions of type <i>BUSTYPE</i> are received from this port and processed.
<code>reg_ap</code>	Analysis output port that publishes <code>uvm_reg_item</code> transactions converted from bus transactions received on <i>bus_in</i> .
<code>map</code>	The map used to convert a bus address to the corresponding register or memory handle.
<code>adapter</code>	The adapter used to convey the parameters of a bus operation in terms of a canonical <code>uvm_reg_bus_op</code> datum.

#### METHODS

<code>new</code>	Create a new instance of this type, giving it the optional <i>name</i> and <i>parent</i> .
<code>pre_predict</code>	Override this method to change the value or re-direct the target register
<code>check_phase</code>	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.

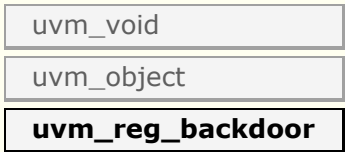
This class can be extended by users to provide user-specific back-door access to registers and memories that are not implemented in pure SystemVerilog or that are not accessible using the default DPI backdoor mechanism.

## Summary

### uvm\_reg\_backdoor

Base class for user-defined back-door register and memory access.

#### CLASS HIERARCHY



#### 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

<code>UVM_HDL_MAX_WIDTH</code>	Sets the maximum size bit vector for backdoor access.
--------------------------------	---

#### METHODS

<code>uvm_hdl_check_path</code>	Checks that the given HDL <i>path</i> exists.
<code>uvm_hdl_deposit</code>	Sets the given HDL <i>path</i> to the specified <i>value</i> .
<code>uvm_hdl_force</code>	Forces the <i>value</i> on the given <i>path</i> .
<code>uvm_hdl_force_time</code>	Forces the <i>value</i> on the given <i>path</i> for the specified amount of <i>force_time</i> .
<code>uvm_hdl_release_and_read</code>	Releases a value previously set with <code>uvm_hdl_force</code> .
<code>uvm_hdl_release</code>	Releases a value previously set with <code>uvm_hdl_force</code> .
<code>uvm_hdl_read()</code>	Gets the value at the given <i>path</i> .

## 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 been 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

### Bit Bashing Test Sequences

This section defines classes that test individual bits of the registers defined in a register model.

**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.

**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.

## 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 during 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_reg_item)  
{  
    uvm_sequence #(uvm_reg_item)  
}
```

#### VARIABLES

<code>model</code>	The block to be tested.
<code>reg_seq</code>	The sequence used to test one register

#### METHODS

<code>body</code>	Executes the Register Bit Bash sequence.
<code>do_block</code>	Test all of the registers in a given <i>block</i>
<code>reset_blk</code>	Reset the DUT that corresponds to the specified block abstraction class.

### 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

### 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.

#### `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.

#### `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.

#### `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.

## `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

<code>body</code>	Executes the Register Access sequence.
<code>do_block</code>	Test all of the registers in a block
<code>reset_blk</code>	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

### 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.

#### `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.

#### `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.

#### `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.

## `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

<code>model</code>	The block to be tested
<code>reg_seq</code>	The sequence used to test one register
<code>mem_seq</code>	The sequence used to test one memory

## METHODS

<code>body</code>	Executes the Shared Register and Memory sequence
<code>do_block</code>	Test all of the registers and memories in a block
<code>reset_blk</code>	Reset the DUT that corresponds to the specified block abstraction class.

# 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

### Memory Access Test Sequence

<code>uvm_mem_single_access_seq</code>	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.
<code>uvm_mem_access_seq</code>	Verify the accessibility of all memories in a block by executing the <code>uvm_mem_single_access_seq</code> sequence on every memory within it.

## 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.

### 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

<b>Memory Walking-Ones Test Sequences</b>	This section defines sequences for applying a “walking-ones” algorithm on one or more memories.
<a href="#">uvm_mem_single_walk_seq</a>	Runs the walking-ones algorithm on the memory given by the <b>mem</b> property, which must be assigned prior to starting this sequence.
<a href="#">uvm_mem_walk_seq</a>	Verifies the all memories in a block by executing the <b>uvm_mem_single_walk_seq</b> sequence on every memory within it.

## 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 specified 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 specified 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

### HDL Paths Checking Test Sequence

## 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' verbosity 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 capabilities 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.

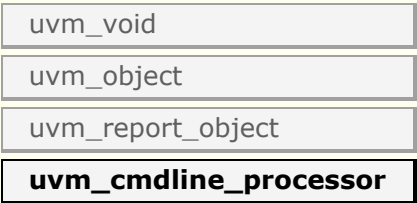
The *uvm\_cmdline\_processor* class also provides support for setting various UVM variables from the command line such as components' verbosity and configuration settings for integral types and strings. Each of these capabilities is described in the Built-in UVM Aware Command Line Arguments section.

## Summary

### uvm\_cmdline\_processor

This class provides an interface to the command line arguments that were provided for the given simulation.

#### CLASS HIERARCHY



#### 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 simulation.

*get\_tool\_version* Returns the version of the simulation tool that is executing the simulation.

#### 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 individual 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
fool23.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 formatting 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 simulation. 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 simulation. 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.

### ``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.

<code>UVM_BIN</code>	Selects binary (%b) format
<code>UVM_DEC</code>	Selects decimal (%d) format
<code>UVM_UNSIGNED</code>	Selects unsigned decimal (%u) format
<code>UVM_OCT</code>	Selects octal (%o) format
<code>UVM_HEX</code>	Selects hexadecimal (%h) format
<code>UVM_STRING</code>	Selects string (%s) format
<code>UVM_TIME</code>	Selects time (%t) format
<code>UVM_ENUM</code>	Selects enumeration value (name) format

### `uvm_recursion_policy_enum`

---

Specifies the policy for copying objects.

<code>UVM_DEEP</code>	Objects are deep copied (object must implement copy method)
<code>UVM_SHALLOW</code>	Objects are shallow copied using default SV copy.
<code>UVM_REFERENCE</code>	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.

<code>UVM_DEFAULT</code>	All field operations turned on
<code>UVM_COPY</code>	Field will participate in <code>uvm_object::copy</code>
<code>UVM_COMPARE</code>	Field will participate in <code>uvm_object::compare</code>



<i>UVM_PRINT</i>	Field will participate in <a href="#">uvm_object::print</a>
<i>UVM_RECORD</i>	Field will participate in <a href="#">uvm_object::record</a>
<i>UVM_PACK</i>	Field will participate in <a href="#">uvm_object::pack</a>
<i>UVM_NOCOPY</i>	Field will not participate in <a href="#">uvm_object::copy</a>
<i>UVM_NOCOMPARE</i>	Field will not participate in <a href="#">uvm_object::compare</a>
<i>UVM_NOPRINT</i>	Field will not participate in <a href="#">uvm_object::print</a>
<i>UVM_NORECORD</i>	Field will not participate in <a href="#">uvm_object::record</a>
<i>UVM_NOPACK</i>	Field will not participate in <a href="#">uvm_object::pack</a>
<i>UVM_DEEP</i>	Object field will be deep copied
<i>UVM_SHALLOW</i>	Object field will be shallow copied
<i>UVM_REFERENCE</i>	Object field will copied by reference
<i>UVM_READONLY</i>	Object field will NOT be automatically configured.

## REPORTING

---

### [uvm\\_severity](#)

---

Defines all possible values for report severity.

<i>UVM_INFO</i>	Informative message.
<i>UVM_WARNING</i>	Indicates a potential problem.
<i>UVM_ERROR</i>	Indicates a real problem. Simulation continues subject to the configured message action.
<i>UVM_FATAL</i>	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.

<i>UVM_NO_ACTION</i>	No action is taken
<i>UVM_DISPLAY</i>	Sends the report to the standard output
<i>UVM_LOG</i>	Sends the report to the file(s) for this (severity,id) pair
<i>UVM_COUNT</i>	Counts the number of reports with the COUNT attribute. When this value reaches max_quit_count, the simulation terminates
<i>UVM_EXIT</i>	Terminates the simulation immediately.
<i>UVM_CALL_HOOK</i>	Callback the report hook methods
<i>UVM_STOP</i>	Causes \$stop to be executed, putting the simulation into interactive mode.

### [uvm\\_verbosity](#)

---

Defines standard verbosity levels for reports.

<i>UVM_NONE</i>	Report is always printed. Verbosity level setting can not disable it.
<i>UVM_LOW</i>	Report is issued if configured verbosity is set to UVM_LOW or above.
<i>UVM_MEDIUM</i>	Report is issued if configured verbosity is set to UVM_MEDIUM or above.
<i>UVM_HIGH</i>	Report is issued if configured verbosity is set to UVM_HIGH or above.
<i>UVM_FULL</i>	Report is issued if configured verbosity is set to UVM_FULL or above.

## PORT TYPE

---

### uvm\_port\_type\_e

---

Specifies the type of port

<i>UVM_PORT</i>	The port requires the interface that is its type parameter.
<i>UVM_EXPORT</i>	The port provides the interface that is its type parameter via a connection to some other export or implementation.
<i>UVM_IMPLEMENTATION</i>	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

<i>SEQ_ARB_FIFO</i>	Requests are granted in FIFO order (default)
<i>SEQ_ARB_WEIGHTED</i>	Requests are granted randomly by weight
<i>SEQ_ARB_RANDOM</i>	Requests are granted randomly
<i>SEQ_ARB_STRICT_FIFO</i>	Requests at highest priority granted in fifo order
<i>SEQ_ARB_STRICT_RANDOM</i>	Requests at highest priority granted in randomly
<i>SEQ_ARB_USER</i>	Arbitration is delegated to the user-defined function, <code>user_priority_arbitration</code> . That function will specify the next sequence to grant.

### uvm\_sequence\_state\_enum

---

Defines current sequence state

<i>CREATED</i>	The sequence has been allocated.
<i>PRE_BODY</i>	The sequence is started and the <code>pre_body</code> task is being

	executed.
<i>BODY</i>	The sequence is started and the body task is being executed.
<i>POST_BODY</i>	The sequence is started and the post_body task is being executed.
<i>ENDED</i>	The sequence has ended by the completion of the body task.
<i>STOPPED</i>	The sequence has been forcibly ended by issuing a kill() on the sequence.
<i>FINISHED</i>	The sequence is completely finished executing.

## uvm\_sequence\_lib\_mode

---

Specifies the random selection mode of a sequence library

<i>UVM_SEQ_LIB_RAND</i>	Random sequence selection
<i>UVM_SEQ_LIB_RANDC</i>	Random cyclic sequence selection
<i>UVM_SEQ_LIB_ITEM</i>	Emit only items, no sequence execution
<i>UVM_SEQ_LIB_USER</i>	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](#).

<i>UVM_PHASE_TASK</i>	The phase is a task-based phase, a fork is done for each participating component and so the traversal order is arbitrary
<i>UVM_PHASE_TOPDOWN</i>	The phase is a function phase, components are traversed from top-down, allowing them to add to the component tree as they go.
<i>UVM_PHASE_BOTTOMUP</i>	The phase is a function phase, components are traversed from the bottom up, allowing roll-up / consolidation functionality.
<i>UVM_PHASE_SCHEDULE_NODE</i>	The phase is not an imp, but a dummy phase graph node representing the beginning of a VIP schedule of phases.
<i>UVM_PHASE_ENDSCHEDULE_NODE</i>	The phase is not an imp, but a dummy phase graph node representing the end of a VIP schedule of phases
<i>UVM_PHASE_DOMAIN_NODE</i>	The phase is not an imp, but a dummy phase graph node representing an entire domain branch with schedules beneath

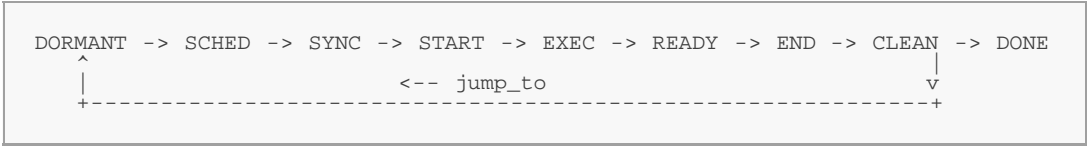
### uvm\_phase\_state

---

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

<i>UVM_PHASE_DORMANT</i>	Nothing has happened with the phase in this domain.
<i>UVM_PHASE_SCHEDULED</i>	At least one immediate predecessor has completed. Scheduled phases block until all predecessors complete or until a jump is executed.
<i>UVM_PHASE_SYNCING</i>	All predecessors complete, checking that all synced phases (e.g. across domains) are at or beyond this point
<i>UVM_PHASE_STARTED</i>	phase ready to execute, running phase_started() callback
<i>UVM_PHASE_EXECUTING</i>	An executing phase is one where the phase callbacks are being executed. It's process is tracked by the phaser.
<i>UVM_PHASE_READY_TO_END</i>	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.
<i>UVM_PHASE_ENDED</i>	phase completed execution, now running phase_ended() callback
<i>UVM_PHASE_CLEANUP</i>	all processes related to phase are being killed
<i>UVM_PHASE_DONE</i>	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

<i>UVM_COMPLETED</i>	the phase completed normally
<i>UVM_FORCED_STOP</i>	the phase was forced to terminate prematurely
<i>UVM_SKIPPED</i>	the phase was in the path of a forward jump
<i>UVM_RERUN</i>	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`.

<i>UVM_EQ</i>	equal
<i>UVM_NE</i>	not equal
<i>UVM_LT</i>	less than
<i>UVM_LTE</i>	less than or equal to
<i>UVM_GT</i>	greater than
<i>UVM_GTE</i>	greater than or equal to

## OBJECTIONS

---

### uvm\_objection\_event

---

Enumerated the possible objection events one could wait on. See [uvm\\_objection::wait\\_for](#).

<i>UVM_RAISED</i>	an objection was raised
<i>UVM_DROPPED</i>	an objection was raised
<i>UVM_ALL_DROPPED</i>	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.

## Summary

### Globals

#### SIMULATION CONTROL

<code>run_test</code>	Convenience function for <code>uvm_top.run_test()</code> .
<code>uvm_test_done</code>	An instance of the <code>uvm_test_done_objection</code> class, this object is used by components to coordinate when to end the currently running task-based phase.
<code>global_stop_request</code>	Convenience function for <code>uvm_test_done.stop_request()</code> .
<code>set_global_timeout</code>	Convenience function for <code>uvm_top.set_timeout()</code> .
<code>set_global_stop_timeout</code>	Convenience function for <code>uvm_test_done.stop_timeout = timeout</code> .

#### REPORTING

<code>uvm_report_enabled</code>	Returns 1 if the configured verbosity in <code>uvm_top</code> is greater than <code>verbosity</code> and the action associated with the given <code>severity</code> and <code>id</code> is not <code>UVM_NO_ACTION</code> , else returns 0.
<code>uvm_report_info</code> <code>uvm_report_warning</code> <code>uvm_report_error</code> <code>uvm_report_fatal</code>	These methods, defined in package scope, are convenience functions that delegate to the corresponding component methods in <code>uvm_top</code> .

#### CONFIGURATION

<code>set_config_int</code>	This is the global version of <code>set_config_int</code> in <code>uvm_component</code> .
<code>set_config_object</code>	This is the global version of <code>set_config_object</code> in <code>uvm_component</code> .
<code>set_config_string</code>	This is the global version of <code>set_config_string</code> in <code>uvm_component</code> .

#### MISCELLANEOUS

<code>uvm_is_match</code>	Returns 1 if the two strings match, 0 otherwise.
<code>uvm_string_to_bits</code>	Converts an input string to its bit-vector equivalent.
<code>uvm_bits_to_string</code>	Converts an input bit-vector to its string equivalent.
<code>uvm_wait_for_nba_region</code>	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.
<code>uvm_split_string</code>	Returns a queue of strings, <i>values</i> , that is the result of the <i>str</i> split based on the <i>sep</i> .

## 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_component`-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 a 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>.

## \$#!

- +UVM\_DUMP\_CMDLINE\_ARGS**  
[uvm\\_cmdline\\_processor](#)
- +UVM\_MAX\_QUIT\_COUNT**  
[uvm\\_cmdline\\_processor](#)
- +UVM\_OBJECTION\_TRACE**  
[uvm\\_cmdline\\_processor](#)
- +UVM\_PHASE\_TRACE**  
[uvm\\_cmdline\\_processor](#)
- +uvm\_set\_action**  
[uvm\\_cmdline\\_processor](#)
- +uvm\_set\_config\_int,+uvm\_set\_config\_string**  
[uvm\\_cmdline\\_processor](#)
- +uvm\_set\_inst\_override,+uvm\_set\_type\_override**  
[uvm\\_cmdline\\_processor](#)
- +uvm\_set\_severity**  
[uvm\\_cmdline\\_processor](#)
- +uvm\_set\_verbosity**  
[uvm\\_cmdline\\_processor](#)
- +UVM\_TESTNAME**  
[uvm\\_cmdline\\_processor](#)
- +UVM\_TIMEOUT**  
[uvm\\_cmdline\\_processor](#)
- +UVM\_VERBOSITY**  
[uvm\\_cmdline\\_processor](#)
- `uvm\_add\_to\_sequence\_library**
- `uvm\_analysis\_imp\_decl**
- `uvm\_blocking\_get\_imp\_decl**
- `uvm\_blocking\_get\_peek\_imp\_decl**
- `uvm\_blocking\_master\_imp\_decl**
- `uvm\_blocking\_peek\_imp\_decl**
- `uvm\_blocking\_put\_imp\_decl**
- `uvm\_blocking\_slave\_imp\_decl**
- `uvm\_blocking\_transport\_imp\_decl**
- `uvm\_component\_end**
- `uvm\_component\_param\_utils**
- `uvm\_component\_param\_utils\_begin**
- `uvm\_component\_registry**
- `uvm\_component\_utils**
- `uvm\_component\_utils\_begin**
- `uvm\_create**
- `uvm\_create\_on**
- `uvm\_declare\_p\_sequencer**
- `uvm\_do**
- `uvm\_do\_callbacks**
- `uvm\_do\_callbacks\_exit\_on**
- `uvm\_do\_obj\_callbacks**
- `uvm\_do\_obj\_callbacks\_exit\_on**
- `uvm\_do\_on**

- `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

- `uvm\_unpack\_sarray
- `uvm\_unpack\_sarrayN
- `uvm\_unpack\_string
- `uvm\_warning
- `uvm\_warning\_context

## A

### **abstract**

- uvm\_comparer
- uvm\_packer
- uvm\_recorder

### **abstractions**

- uvm\_reg\_mem\_hdl\_paths\_seq

### **accept\_tr**

- uvm\_component
- uvm\_transaction

### **Access**

- uvm\_reg
- uvm\_reg\_block
- uvm\_reg\_field
- uvm\_reg\_fifo

### **accessors**

- uvm\_tlm\_generic\_payload

### **adapter**

- uvm\_reg\_predictor
- uvm\_reg\_sequence

### **add**

- uvm\_callbacks#(T,CB)
- uvm\_heartbeat
- uvm\_pool#(KEY,T)
- uvm\_reg\_read\_only\_cbs
- uvm\_reg\_write\_only\_cbs

### **Add/delete interface**

- uvm\_callbacks#(T,CB)

### **add\_by\_name**

- uvm\_callbacks#(T,CB)

### **add\_callback**

- uvm\_event

### **add\_coverage**

- uvm\_mem
- uvm\_reg
- uvm\_reg\_block

### **add\_hdl\_path**

- uvm\_mem
- uvm\_reg
- uvm\_reg\_block
- uvm\_reg\_file

### **add\_hdl\_path\_slice**

- uvm\_mem
- uvm\_reg

### **add\_mem**

- uvm\_reg\_map



- add\_path**
  - uvm\_hdl\_path\_concat
- add\_phase**
  - uvm\_phase
- add\_reg**
  - uvm\_reg\_map
- add\_schedule**
  - uvm\_phase
- add\_slice**
  - uvm\_hdl\_path\_concat
- add\_submap**
  - uvm\_reg\_map
- addr**
  - uvm\_reg\_bus\_op
- adjust\_name**
  - uvm\_printer
- after\_export**
  - uvm\_algorithmic\_comparator#(BEFORE,AFTER,TRANSFORMER)
  - uvm\_in\_order\_comparator#(T,comp\_type,convert,pair\_type)
- all\_dropped**
  - uvm\_callbacks\_objection
  - uvm\_component
  - uvm\_objection
  - uvm\_objection\_callback
  - uvm\_test\_done\_objection
- alloc\_mode\_e**
  - uvm\_mem\_mam
- allocate**
  - uvm\_vreg
- Analysis**
  - Global
  - uvm\_tlm\_if\_base#(T1,T2)
- Analysis Ports**
- analysis\_export**
  - uvm\_subscriber
- analysis\_export#(T)**
  - uvm\_tlm\_analysis\_fifo
- apply\_config\_settings**
  - uvm\_component
- Argument Values**
  - uvm\_cmdline\_processor
- Audit Trail**
  - uvm\_resource\_base

## B

- b\_transport**
  - uvm\_tlm\_if
- Backdoor**
  - uvm\_mem
  - uvm\_reg

uvm\_reg\_block  
uvm\_reg\_file

### **backdoor\_read**

uvm\_mem  
uvm\_reg

### **backdoor\_read\_func**

uvm\_mem  
uvm\_reg

### **backdoor\_watch**

uvm\_reg

### **backdoor\_write**

uvm\_mem  
uvm\_reg

### **BASE**

uvm\_reg\_sequence

### **Basic Arguments**

uvm\_cmdline\_processor

### **bd\_kind**

uvm\_reg\_item

### **before\_export**

uvm\_algorithmic\_comparator#(BEFORE,AFTER,TRANSFORMER)  
uvm\_in\_order\_comparator#(T,comp\_type,convert,pair\_type)

### **begin\_child\_tr**

uvm\_component  
uvm\_transaction

### **begin\_elements**

uvm\_printer\_knobs

### **begin\_event**

uvm\_transaction

### **BEGIN\_REQ**

### **BEGIN\_RESP**

### **begin\_tr**

uvm\_component  
uvm\_transaction

### **Bidirectional Interfaces&Ports**

### **big\_endian**

uvm\_packer

### **bin\_radix**

uvm\_printer\_knobs

### **Bit Bashing Test Sequences**

### **Blocking get**

uvm\_tlm\_if\_base#(T1,T2)

### **Blocking peek**

uvm\_tlm\_if\_base#(T1,T2)

### **Blocking put**

uvm\_tlm\_if\_base#(T1,T2)

### **Blocking transport**

uvm\_tlm\_if\_base#(T1,T2)

### **blocking\_put\_port**

uvm\_random\_stimulus#(T)

### **body**

uvm\_mem\_access\_seq  
uvm\_mem\_single\_walk\_seq

- uvm\_mem\_walk\_seq
- uvm\_reg\_access\_seq
- uvm\_reg\_bit\_bash\_seq
- uvm\_reg\_hw\_reset\_seq
- uvm\_reg\_mem\_built\_in\_seq
- uvm\_reg\_mem\_shared\_access\_seq
- uvm\_reg\_sequence
- uvm\_sequence\_base

## **BODY**

### **build\_coverage**

- uvm\_mem
- uvm\_reg
- uvm\_reg\_block

### **build\_ph**

- Pre-Defined Phases

### **build\_phase**

- uvm\_component

## **Built-in UVM Aware Command Line Arguments**

- uvm\_cmdline\_processor

### **burst\_read**

- uvm\_mem
- uvm\_mem\_region

### **burst\_write**

- uvm\_mem
- uvm\_mem\_region

## **Bus Access**

- uvm\_reg\_map

### **bus\_in**

- uvm\_reg\_predictor

### **bus2reg**

- uvm\_reg\_adapter
- uvm\_reg\_tlm\_adapter

### **byte\_en**

- uvm\_reg\_bus\_op

# **C**

## **Callback Hooks**

- uvm\_objection

## **Callback Interface**

- uvm\_report\_catcher

## **Callback Macros**

### **callback\_mode**

- uvm\_callback

## **Callbacks**

- uvm\_mem
- uvm\_phase
- uvm\_reg
- uvm\_reg\_field
- uvm\_report\_object
- uvm\_vreg
- uvm\_vreg\_field

## 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

- compare**
  - uvm\_object
- compare\_field**
  - uvm\_comparer
- compare\_field\_int**
  - uvm\_comparer
- compare\_field\_real**
  - uvm\_comparer
- compare\_object**
  - uvm\_comparer
- compare\_string**
  - uvm\_comparer
- Comparing**
  - uvm\_object
- compose\_message**
  - uvm\_report\_server
- Configuration**
  - Global
  - uvm\_object
  - uvm\_report\_object
- Configuration and Resource Classes**
- Configuration Interface**
  - uvm\_component
- 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\_ph**
  - Pre-Defined Phases
- 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\_ph**
  - Pre-Defined Phases
- connect\_phase**
  - uvm\_component
- Construction**
  - uvm\_phase
- Container Classes**
- Convenience Write/Read API**
  - uvm\_reg\_sequence

**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

### **data**

uvm\_reg\_bus\_op

### **Debug**

uvm\_callbacks#(T,CB)

uvm\_factory

uvm\_report\_catcher

uvm\_resource\_pool

### **debug\_connected\_to**

uvm\_port\_base#(IF)

### **debug\_create\_by\_name**

uvm\_factory

### **debug\_create\_by\_type**

uvm\_factory

### **debug\_provided\_to**

uvm\_port\_base#(IF)

### **dec\_radix**

uvm\_printer\_knobs

### **decode**

uvm\_reg\_cbs

### **decr**

uvm\_tlm\_time

### **Default Policy Classes**

#### **default\_alloc**

uvm\_mem\_mam

#### **default\_map**

uvm\_reg\_block

#### **default\_path**

uvm\_reg\_block

#### **default\_precedence**

uvm\_resource\_base

#### **default\_radix**

uvm\_printer\_knobs

uvm\_recorder

#### **define\_access**

uvm\_reg\_field

#### **define\_phase\_schedule**

uvm\_component

### **delete**

uvm\_callbacks#(T,CB)

uvm\_object\_string\_pool#(T)

uvm\_pool#(KEY,T)

uvm\_queue#(T)

#### **delete\_by\_name**

uvm\_callbacks#(T,CB)

#### **delete\_callback**

- uvm\_event
- depth**
  - uvm\_printer\_knobs
- die**
  - uvm\_report\_object
- disable\_recording**
  - uvm\_transaction
- display**
  - uvm\_callbacks#(T,CB)
- display\_objections**
  - uvm\_objection
- do\_accept\_tr**
  - uvm\_component
  - uvm\_transaction
- do\_begin\_tr**
  - uvm\_component
  - uvm\_transaction
- do\_block**
  - uvm\_mem\_access\_seq
  - uvm\_mem\_walk\_seq
  - uvm\_reg\_access\_seq
  - uvm\_reg\_bit\_bash\_seq
  - 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**
  - uvm\_resource\_db
  - uvm\_resource\_pool
- dump\_report\_state**
  - uvm\_report\_object
- dump\_server\_state**
  - uvm\_report\_server

## E

- element**
  - uvm\_reg\_item
- element\_kind**
  - uvm\_reg\_item
- emit**
  - uvm\_printer
  - uvm\_table\_printer
  - uvm\_tree\_printer
- enable\_print\_topology**
  - uvm\_root
- enable\_recording**
  - uvm\_transaction
- enable\_stop\_interrupt**
  - uvm\_component
- encode**
  - uvm\_reg\_cbs
- end\_elements**
  - uvm\_printer\_knobs
- end\_event**
  - uvm\_transaction

- end\_of\_elaboration\_ph**  
Pre-Defined Phases
- end\_of\_elaboration\_phase**  
uvm\_component
- end\_offset**  
uvm\_mem\_mam\_cfg
- END\_REQ**
- END\_RESP**
- end\_tr**  
uvm\_component  
uvm\_transaction
- ENDED**
- Enumerations**
- events**  
uvm\_transaction
- Example**  
uvm\_reg\_adapter
- exec\_func**  
uvm\_phase
- exec\_task**  
uvm\_phase
- execute**  
uvm\_bottomup\_phase  
uvm\_task\_phase  
uvm\_topdown\_phase
- execute\_item**  
uvm\_sequencer\_base
- exists**  
uvm\_config\_db#(T)  
uvm\_pool#(KEY,T)
- extension**  
uvm\_reg\_item
- Extensions Mechanism**  
uvm\_tlm\_generic\_payload
- extract\_ph**  
Pre-Defined Phases
- extract\_phase**  
uvm\_component

## F

- Factory Classes**
- Factory Component and Object Wrappers**
- Factory Interface**  
uvm\_component
- Field automation**
- Field Macros**
- fifo**  
uvm\_reg\_fifo
- final\_ph**  
Pre-Defined Phases

- final\_phase**
  - uvm\_component
- find**
  - uvm\_phase
  - uvm\_root
- find\_all**
  - uvm\_root
  - uvm\_utils
- find\_block**
  - 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
- finish\_on\_completion**
  - uvm\_root
- FINISHED**
- first**
  - uvm\_callback\_iter
  - uvm\_pool#(KEY,T)
- flush**
  - uvm\_in\_order\_comparator#(T,comp\_type,convert,pair\_type)
  - uvm\_tlm\_fifo
- fname**
  - uvm\_reg\_item
- footer**
  - uvm\_printer\_knobs
- for\_each**
  - uvm\_mem\_mam
- force\_stop**
  - uvm\_test\_done\_objection
- format\_action**
  - uvm\_report\_handler
- format\_header**
  - uvm\_printer
- format\_row**
  - uvm\_printer
- Frontdoor**
  - uvm\_mem
  - uvm\_reg
- full\_name**
  - uvm\_printer\_knobs

## G

### **generate\_stimulus**

uvm\_random\_stimulus#(T)

### **Generic Payload**

### **Generic Register Operation Descriptors**

#### **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 and Peek**

#### **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\_ap**

uvm\_tlm\_fifo\_base#(T)

#### **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\_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)

**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

**get\_vfield\_by\_name**

uvm\_mem

uvm\_reg\_block

**get\_virtual\_fields**

uvm\_mem

uvm\_reg\_block

uvm\_reg\_map

**get\_virtual\_registers**

uvm\_mem

uvm\_mem\_region

uvm\_reg\_block

uvm\_reg\_map

**get\_vreg\_by\_name**

uvm\_mem

uvm\_reg\_block

**get\_vreg\_by\_offset**

uvm\_mem

**Global Declarations for the Register Layer****global\_stop\_request****Globals**

base/uvm\_globals.svh

tlm2/tlm2\_generic\_payload.svh

tlm2/tlm2\_ifs.svh

**grab**

uvm\_sequence\_base

uvm\_sequencer\_base

**H****has\_child**

uvm\_component

**has\_coverage**

uvm\_mem

uvm\_reg

uvm\_reg\_block

**has\_do\_available**

uvm\_sequencer\_base

uvm\_sqr\_if\_base#(REQ,RSP)

**has\_hdl\_path**

uvm\_mem

uvm\_reg

uvm\_reg\_block

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

## **ID**

uvm\_tlm\_extension

## **id\_count**

uvm\_report\_server

## **Identification**

uvm\_object

## **identifier**

uvm\_printer\_knobs

uvm\_recorder

## **if overriding this method,always follow this pattern**

uvm\_component

## **IMP binding classes**

## **IMP binding macros**

## **implement**

uvm\_vreg

## **in\_use**

uvm\_mem\_mam\_policy

## **include\_coverage**

uvm\_reg

## **incr**

uvm\_tlm\_time

## **incr\_id\_count**

uvm\_report\_server

## **incr\_quit\_count**

uvm\_report\_server

## **incr\_severity\_count**

uvm\_report\_server

## **indent**

uvm\_printer\_knobs

## **info**

uvm\_reg\_bus\_op

## **init\_access\_record**

uvm\_resource\_base

## **initialization**

uvm\_vreg\_field

## **Initialization**

uvm\_mem

uvm\_mem\_mam

uvm\_reg

uvm\_reg\_block

uvm\_reg\_field

uvm\_reg\_fifo

uvm\_reg\_file

uvm\_reg\_map

uvm\_vreg

## **insert**

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



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)

**Iterator interface**

uvm\_callbacks#(T,CB)

**J**

**jump**

uvm\_phase

**jump\_all**  
uvm\_phase

**Jumping**  
uvm\_phase

## K

**kill**  
uvm\_component  
uvm\_sequence\_base

**kind**  
uvm\_reg\_bus\_op  
uvm\_reg\_item

**knobs**  
uvm\_printer

## 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)

**len**  
uvm\_mem\_mam\_policy

**lineno**  
uvm\_reg\_item

**local\_map**  
uvm\_reg\_item

**locality**  
uvm\_mem\_mam\_cfg

**locality\_e**  
uvm\_mem\_mam

**lock**  
uvm\_resource\_base  
uvm\_sequence\_base  
uvm\_sequencer\_base

**lock\_model**  
uvm\_reg\_block

**Locking Interface**  
uvm\_resource#(T)  
uvm\_resource\_base

**lookup**  
uvm\_component

**Lookup**  
uvm\_resource\_pool

**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

**m\_address**  
 uvm\_tlm\_generic\_payload

**m\_byte\_enable**  
 uvm\_tlm\_generic\_payload

**m\_byte\_enable\_length**  
 uvm\_tlm\_generic\_payload

**m\_command**  
 uvm\_tlm\_generic\_payload

**m\_data**  
 uvm\_tlm\_generic\_payload

**m\_dmi**  
 uvm\_tlm\_generic\_payload

**m\_length**  
 uvm\_tlm\_generic\_payload

**m\_response\_status**  
 uvm\_tlm\_generic\_payload

**m\_set\_hier\_mode**  
 uvm\_objection

**m\_streaming\_width**  
 uvm\_tlm\_generic\_payload

**Macros**  
 macros/uvm\_message\_defines.svh  
 macros/uvm\_tlm\_defines.svh  
 tlm2/tlm2\_defines.svh

**main\_ph**  
 Pre-Defined Phases

**main\_phase**  
 uvm\_component

**mam**  
 uvm\_mem

**map**  
 uvm\_reg\_item  
 uvm\_reg\_predictor

**Master and Slave**

**master\_export**  
 uvm\_tlm\_req\_rsp\_channel#(REQ,RSP)

**match\_scope**

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

## Methods for printer subtyping

uvm\_printer

## Methods for printer usage

uvm\_printer

## mid\_do

uvm\_sequence\_base

## min\_offset

uvm\_mem\_mam\_policy

## min\_size

uvm\_port\_base#(IF)

## mirror

uvm\_reg

uvm\_reg\_block

uvm\_reg\_field

uvm\_reg\_fifo

## mirror\_reg

uvm\_reg\_sequence

## Miscellaneous

## miscompares

uvm\_comparer

## mode

uvm\_mem\_mam\_cfg

## model

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\_built\_in\_seq

uvm\_reg\_mem\_shared\_access\_seq

uvm\_reg\_sequence

## Modifying the offset of a memory will make the abstract model

uvm\_mem

# N

## n\_bits

uvm\_reg\_bus\_op

## n\_bytes

uvm\_mem\_mam\_cfg

## nb\_transport

uvm\_tlm\_if\_base#(T1,T2)

## nb\_transport\_bw

uvm\_tlm\_if

## nb\_transport\_fw

uvm\_tlm\_if

## needs\_update

uvm\_reg

uvm\_reg\_block

uvm\_reg\_field

## new

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)

- uvm\_sequencer\_base
- uvm\_sequencer\_param\_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\_generic\_payload
- uvm\_tlm\_nb\_initiator\_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\_time
- uvm\_tlm\_transport\_channel#(REQ,RSP)
- uvm\_topdown\_phase
- uvm\_transaction
- uvm\_tree\_printer
- uvm\_vreg
- uvm\_vreg\_field

#### **next**

- uvm\_callback\_iter
- uvm\_pool#(KEY,T)

#### **Non-blocking get**

- uvm\_tlm\_if\_base#(T1,T2)

#### **Non-blocking peek**

- uvm\_tlm\_if\_base#(T1,T2)

#### **Non-blocking put**

- uvm\_tlm\_if\_base#(T1,T2)

#### **Non-blocking transport**

- uvm\_tlm\_if\_base#(T1,T2)

#### **Notification**

- uvm\_resource\_base

#### **num**

- uvm\_pool#(KEY,T)

## O

#### **Objection Control**

- uvm\_objection

#### **Objection Interface**

- uvm\_component

#### **Objection Mechanism**

#### **Objection Status**

- uvm\_objection

#### **Objections**

#### **oct\_radix**



uvm\_printer\_knobs

**offset**

uvm\_reg\_item

## P

### **pack**

uvm\_object

### **pack\_bytes**

uvm\_object

### **pack\_field**

uvm\_packer

### **pack\_field\_int**

uvm\_packer

### **pack\_ints**

uvm\_object

### **pack\_object**

uvm\_packer

### **pack\_real**

uvm\_packer

### **pack\_string**

uvm\_packer

### **pack\_time**

uvm\_packer

### **Packing**

uvm\_object

uvm\_packer

### **Packing Macros**

#### **Packing-No Size Info**

#### **Packing-With Size Info**

### **pair\_ap**

uvm\_in\_order\_comparator#(T,comp\_type,convert,pair\_type)

### **parent**

uvm\_reg\_item

### **path**

uvm\_reg\_item

### **peek**

uvm\_mem

uvm\_mem\_region

uvm\_reg

uvm\_reg\_field

uvm\_sqr\_if\_base#(REQ,RSP)

uvm\_tlm\_if\_base#(T1,T2)

uvm\_vreg

uvm\_vreg\_field

### **peek\_mem**

uvm\_reg\_sequence

### **peek\_reg**

uvm\_reg\_sequence

### **phase\_ended**

uvm\_component

uvm\_phase

## **phase\_started**

uvm\_component  
uvm\_phase

## **Phasing**

base/uvm\_object\_globals.svh  
base/uvm\_phases.svh

## **Phasing Implementation**

### **Phasing Interface**

uvm\_component

## **physical**

uvm\_comparer  
uvm\_packer  
uvm\_recorder

## **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

## **policy**

uvm\_comparer

## **Policy Classes**

comps/uvm\_policies.svh  
policies.txt

## **Pool Classes**

### **pop\_back**

uvm\_queue#(T)

### **pop\_front**

uvm\_queue#(T)

## **Port Base Classes**

### **Port Type**

### **Ports**

uvm\_algorithmic\_comparator#(BEFORE,AFTER,TRANSFORMER)  
uvm\_driver#(REQ,RSP)  
uvm\_in\_order\_comparator#(T,comp\_type,convert,pair\_type)  
uvm\_push\_driver#(REQ,RSP)  
uvm\_push\_sequencer#(REQ,RSP)  
uvm\_random\_stimulus#(T)  
uvm\_subscriber  
uvm\_tlm\_analysis\_fifo  
uvm\_tlm\_fifo\_base#(T)  
uvm\_tlm\_req\_rsp\_channel#(REQ,RSP)  
uvm\_tlm\_transport\_channel#(REQ,RSP)

## **Ports,Exports,and Imps**

### **post\_body**

uvm\_sequence\_base

## **POST\_BODY**

### **post\_configure\_ph**

Pre-Defined Phases

### **post\_configure\_phase**

uvm\_component

## **post\_do**

uvm\_sequence\_base

## **post\_main\_ph**

Pre-Defined Phases

## **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\_ph**

Pre-Defined Phases

## **post\_reset\_phase**

uvm\_component

## **post\_shutdown\_ph**

Pre-Defined Phases

## **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-Defined Phases**

### **pre\_abort**

uvm\_component

### **pre\_body**

uvm\_sequence\_base

### **PRE\_BODY**

### **pre\_configure\_ph**

Pre-Defined Phases

### **pre\_configure\_phase**

uvm\_component

### **pre\_do**

uvm\_sequence\_base

### **pre\_main\_ph**

Pre-Defined Phases

### **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\_ph**

Pre-Defined Phases

### **pre\_reset\_phase**

uvm\_component

### **pre\_shutdown\_ph**

Pre-Defined Phases

### **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

### **precedence**

uvm\_resource\_base

## **Predefined Component Classes**

## **Predefined Extensions**

### **predict**

uvm\_reg  
uvm\_reg\_field

### **prefix**

uvm\_printer\_knobs

### **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\_matches**
  - uvm\_component
- print\_config\_settings**
  - uvm\_component
- print\_config\_with\_audit**
  - uvm\_component
- print\_enabled**
  - 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
- Printing**
  - uvm\_object
- prior**
  - uvm\_reg\_item
- Priority**
  - uvm\_resource#(T)
  - uvm\_resource\_base
- process\_report**
  - uvm\_report\_server
- provides\_responses**
  - uvm\_reg\_adapter
- push\_back**
  - uvm\_queue#(T)
- push\_front**
  - uvm\_queue#(T)
- put**

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

## **raise\_objection**

- uvm\_objection
- uvm\_phase
- uvm\_test\_done\_objection

## **raised**

- uvm\_callbacks\_objection
- uvm\_component
- uvm\_objection
- uvm\_objection\_callback

## **read**

- uvm\_mem
- uvm\_mem\_region
- uvm\_reg
- uvm\_reg\_backdoor
- uvm\_reg\_field
- uvm\_reg\_fifo
- uvm\_resource#(T)
- uvm\_vreg
- uvm\_vreg\_field

## **Read-only Interface**

- uvm\_resource\_base

## **Read/Write Interface**

- uvm\_resource#(T)

## **read\_by\_name**

- uvm\_resource\_db

## **read\_by\_type**

- uvm\_resource\_db

## **read\_func**

- uvm\_reg\_backdoor

## **read\_mem**

- uvm\_reg\_sequence

## **read\_mem\_by\_name**

- uvm\_reg\_block

## **read\_reg**

- uvm\_reg\_sequence

## **read\_reg\_by\_name**

- uvm\_reg\_block

## **read\_with\_loc;**

- uvm\_resource#(T)

## **reconfigure**

- uvm\_mem\_mam

## **record**

- uvm\_object

## **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
- Recording**
  - uvm\_object
- Recording Interface**
  - uvm\_component
- Recording Macros**
- recursion\_policy**
  - uvm\_recorder
- reference**
  - uvm\_printer\_knobs
- reg\_ap**
  - uvm\_reg\_predictor
- reg\_seq**
  - uvm\_reg\_access\_seq
  - uvm\_reg\_bit\_bash\_seq
  - uvm\_reg\_mem\_shared\_access\_seq
- reg\_seqr**
  - uvm\_reg\_sequence
- reg2bus**
  - uvm\_reg\_adapter
  - uvm\_reg\_tlm\_adapter
- register**
  - uvm\_factory
- Register Access Test Sequences**
- Register Callbacks**
- Register Defines**
- Register Layer**
- Register Sequence and Predictor Classes**
- Registering Types**
  - 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 Macros

### 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\_ph

Pre-Defined Phases

### report\_phase

uvm\_component

### report\_summarize

uvm\_report\_object

### report\_warning\_hook

uvm\_report\_object

## Reporting

Global

base/uvm\_globals.svh

base/uvm\_object\_globals.svh

uvm\_report\_catcher

uvm\_report\_object

## Reporting Classes

### Reporting Interface

uvm\_sequence\_item

### req\_export

uvm\_push\_driver#(REQ,RSP)

### req\_port

uvm\_push\_sequencer#(REQ,RSP)

### request\_ap

uvm\_tlm\_req\_rsp\_channel#(REQ,RSP)

### request\_region

uvm\_mem\_mam

## Requests

uvm\_sequencer\_param\_base#(REQ,RSP)

### 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\_ph**

- Pre-Defined Phases

## **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)

## **Resources**

### **Response API**

- uvm\_sequence\_base

## **response\_ap**

- uvm\_tlm\_req\_rsp\_channel#(REQ,RSP)

## **response\_handler**

- uvm\_sequence\_base

## **Responses**

- uvm\_sequencer\_param\_base#(REQ,RSP)

## **result**

- uvm\_comparer

## **resume**

- uvm\_component

## **rg**

- uvm\_reg\_shared\_access\_seq
- uvm\_reg\_single\_access\_seq
- uvm\_reg\_single\_bit\_bash\_seq

## **rsp\_export**

- uvm\_sequencer\_param\_base#(REQ,RSP)

## **rsp\_port**

- uvm\_driver#(REQ,RSP)
- uvm\_push\_driver#(REQ,RSP)

## **Run-Time Schedule Global Variables**

- Pre-Defined Phases

## **run\_hooks**

- uvm\_report\_handler

## **run\_ph**

- Pre-Defined Phases

## **run\_phase**

- uvm\_component
- uvm\_push\_sequencer#(REQ,RSP)

## **run\_test**

- Global



uvm\_root

**rw\_info**

uvm\_reg\_frontdoor

# S

## **sample**

- uvm\_mem
- uvm\_reg
- uvm\_reg\_block

## **sample\_values**

- uvm\_reg
- uvm\_reg\_block

## **Schedule**

- uvm\_phase

## **Scope Interface**

- uvm\_resource\_base

## **Seeding**

- uvm\_object

## **send\_request**

- uvm\_sequence#(REQ,RSP)
- uvm\_sequence\_base
- uvm\_sequencer\_base
- uvm\_sequencer\_param\_base#(REQ,RSP)

## **separator**

- uvm\_printer\_knobs

## **SEQ\_ARB\_FIFO**

## **SEQ\_ARB\_RANDOM**

## **SEQ\_ARB\_STRICT\_FIFO**

## **SEQ\_ARB\_STRICT\_RANDOM**

## **SEQ\_ARB\_USER**

## **SEQ\_ARB\_WEIGHTED**

## **seq\_item\_export**

- uvm\_sequencer#(REQ,RSP)

## **seq\_item\_port**

- uvm\_driver#(REQ,RSP)

## **Sequence Action Macros**

## **Sequence Action Macros for Pre-Existing Sequences**

## **Sequence Classes**

## **Sequence Control**

- uvm\_sequence\_base

## **Sequence Execution**

- uvm\_sequence\_base

## **Sequence Item Execution**

- uvm\_sequence\_base

## **Sequence Item Pull Ports**

## **Sequence Library**

## **Sequence on Sequencer Action Macros**

## **Sequence-Related Macros**

## **sequencer**

- uvm\_reg\_frontdoor

## **Sequencer Classes**

## **Sequencer Port**

## Sequencer Subtypes

### Sequences

#### set

- uvm\_config\_db#(T)
- uvm\_hdl\_path\_concat
- uvm\_reg
- uvm\_reg\_field
- uvm\_reg\_fifo
- uvm\_resource#(T)
- uvm\_resource\_db
- uvm\_resource\_pool

#### Set

- uvm\_resource\_pool

#### set priority

- uvm\_resource#(T)
- uvm\_resource\_base

#### Set Priority

- uvm\_resource\_pool

#### Set/Get Interface

- uvm\_resource#(T)

#### set\_abstime

- uvm\_tlm\_time

#### set\_access

- uvm\_reg\_field

#### set\_action

- uvm\_report\_catcher

#### set\_address

- uvm\_tlm\_generic\_payload

#### set\_anonymous

- uvm\_resource\_db

#### set\_arbitration

- uvm\_sequencer\_base

#### set\_auto\_predict

- uvm\_reg\_map

#### set\_auto\_reset

- uvm\_barrier

#### set\_backdoor

- uvm\_mem
- uvm\_reg
- uvm\_reg\_block

#### set\_base\_addr

- uvm\_reg\_map

#### 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

**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**

**show\_max**

uvm\_comparer

**show\_radix**

uvm\_printer\_knobs

**show\_root**

uvm\_printer\_knobs

**shutdown\_ph**

Pre-Defined Phases

**shutdown\_phase**

uvm\_component

**Simulation Control****Singleton**

uvm\_cmdline\_processor

**size**

uvm\_port\_base#(IF)

uvm\_printer\_knobs

uvm\_queue#(T)

uvm\_reg\_fifo

uvm\_tlm\_fifo

**slave\_export**

uvm\_tlm\_req\_rsp\_channel#(REQ,RSP)

**slices**

uvm\_hdl\_path\_concat

**Special Overrides**

uvm\_reg\_fifo

**spell\_check**

uvm\_resource\_pool

**sprint**

uvm\_object

**start**

uvm\_heartbeat

uvm\_sequence\_base

**start\_item**

uvm\_sequence\_base

uvm\_sequence\_item

**start\_of\_simulation\_ph**

Pre-Defined Phases

**start\_of\_simulation\_phase**

uvm\_component

**start\_offset**

uvm\_mem\_mam\_policy

**start\_phase\_sequence**

uvm\_sequencer\_base

**starting\_phase**

uvm\_sequence\_base

**State**

uvm\_phase

**status**

uvm\_component

uvm\_reg\_bus\_op

uvm\_reg\_item

**stop**

- 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



# U

## **ungrab**

uvm\_sequence\_base  
uvm\_sequencer\_base

## **Unidirectional Interfaces&Ports**

## **UNINITIALIZED\_PHASE**

## **unlock**

uvm\_resource\_base  
uvm\_sequence\_base  
uvm\_sequencer\_base

## **unpack**

uvm\_object

## **unpack\_bytes**

uvm\_object

## **unpack\_field**

uvm\_packer

## **unpack\_field\_int**

uvm\_packer

## **unpack\_ints**

uvm\_object

## **unpack\_object**

uvm\_packer

## **unpack\_real**

uvm\_packer

## **unpack\_string**

uvm\_packer

## **unpack\_time**

uvm\_packer

## **Unpacking**

uvm\_object  
uvm\_packer

## **Unpacking Macros**

## **Unpacking-No Size Info**

## **Unpacking-With Size Info**

## **unsigned\_radix**

uvm\_printer\_knobs

## **unsync**

uvm\_phase

## **update**

uvm\_reg  
uvm\_reg\_block  
uvm\_reg\_fifo

## **update\_reg**

uvm\_reg\_sequence

## **Usage**

Global  
uvm\_factory

uvm\_object\_registry#(T,Tname)

## **use\_metadata**

uvm\_packer

## **use\_response\_handler**

uvm\_sequence\_base

## **use\_uvm\_seeding**

uvm\_object

## **used**

uvm\_tlm\_fifo

## **User-Defined Phases**

### **user\_priority\_arbitration**

uvm\_sequencer\_base

## **Utility and Field Macros for Components and Objects**

### **Utility Classes**

### **Utility Functions**

uvm\_resource\_base

### **Utility Macros**

### **UVM Class Reference**

### **UVM Factory**

### **UVM HDL Backdoor Access support routines**

uvm\*\_export#(REQ,RSP)

uvm\*\_export#(T)

uvm\*\_imp ports

uvm\*\_imp#(REQ,RSP,IMP,REQ\_IMP,RSP\_IMP)

uvm\*\_imp#(T,IMP)

uvm\*\_port#(REQ,RSP)

uvm\*\_port#(T)

uvm\_access\_e

uvm\_action

uvm\_active\_passive\_enum

uvm\_agent

uvm\_algorithmic\_comparator#(BEFORE,AFTER,TRANSFORMER)

uvm\_algorithmic\_comparator.svh

UVM\_ALL\_DROPPED

uvm\_analysis\_export

uvm\_analysis\_imp

uvm\_analysis\_port

UVM\_BACKDOOR

uvm\_barrier

UVM\_BIG\_ENDIAN

UVM\_BIG\_FIFO

UVM\_BIN

uvm\_bits\_to\_string

uvm\_bitstream\_t

uvm\_bottomup\_phase

uvm\_built\_in\_clone#(T)

uvm\_built\_in\_comp#(T)

uvm\_built\_in\_converter#(T)

uvm\_built\_in\_pair#(T1,T2)

UVM\_CALL\_HOOK

uvm\_callback

uvm\_callback\_defines.svh

uvm\_callback\_iter

uvm\_callbacks#(T,CB)

uvm\_callbacks\_objection

UVM\_CHECK

uvm\_check\_e

uvm\_class\_clone#(T)



uvm\_class\_comp#(T)  
 uvm\_class\_converter#(T)  
 uvm\_cmdline\_processor  
 uvm\_comparer  
 UVM\_COMPLETED  
 uvm\_component  
 uvm\_component\_registry#(T,Tname)  
 uvm\_config\_db#(T)  
 UVM\_COUNT  
 uvm\_coverage\_model\_e  
 UVM\_CVR\_ADDR\_MAP  
 UVM\_CVR\_ALL  
 UVM\_CVR\_FIELD\_VALS  
 UVM\_CVR\_REG\_BITS  
 UVM\_DEC  
 UVM\_DEEP  
 uvm\_default\_comparer  
 uvm\_default\_line\_printer  
 uvm\_default\_packer  
 UVM\_DEFAULT\_PATH  
 uvm\_default\_printer  
 uvm\_default\_recorder  
 uvm\_default\_table\_printer  
 uvm\_default\_tree\_printer  
 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\_domain  
 uvm\_driver#(REQ,RSP)  
 UVM\_DROPPED  
 uvm\_elem\_kind\_e  
 uvm\_endianness\_e  
 UVM\_ENUM  
 uvm\_env  
 UVM\_EQ  
 UVM\_ERROR  
 uvm\_event  
 uvm\_event\_callback  
 UVM\_EXIT  
 UVM\_EXPORT  
 uvm\_factory  
 UVM\_FATAL  
 UVM\_FIELD  
 UVM\_FORCED\_STOP  
 UVM\_FRONTDOOR  
 UVM\_FULL  
 UVM\_GT  
 UVM\_GTE  
 UVM\_HAS\_X  
 uvm\_hdl\_check\_path  
 uvm\_hdl\_deposit  
 uvm\_hdl\_force  
 uvm\_hdl\_force\_time  
 UVM\_HDL\_MAX\_WIDTH  
 uvm\_hdl\_path\_concat

uvm\_hdl\_path\_slice  
 uvm\_hdl\_read  
 uvm\_hdl\_release  
 uvm\_hdl\_release\_and\_read  
 uvm\_heartbeat  
 UVM\_HEX  
 UVM\_HIER  
 uvm\_hier\_e  
 UVM\_HIGH  
 UVM\_IMPLEMENTATION  
 uvm\_in\_order\_built\_in\_comparator#(T)  
 uvm\_in\_order\_class\_comparator#(T)  
 uvm\_in\_order\_comparator#(T,comp\_type,convert,pair\_type)  
 UVM\_INFO  
 uvm\_is\_match  
 UVM\_IS\_OK  
 uvm\_line\_printer  
 UVM\_LITTLE\_ENDIAN  
 UVM\_LITTLE\_FIFO  
 UVM\_LOG  
 UVM\_LOW  
 UVM\_LT  
 UVM\_LTE  
 UVM\_MEDIUM  
 uvm\_mem  
 UVM\_MEM  
 uvm\_mem\_access\_seq  
 uvm\_mem\_cb  
 uvm\_mem\_cb\_iter  
 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\_NE  
 UVM\_NO\_ACTION  
 UVM\_NO\_CHECK  
 UVM\_NO\_COVERAGE  
 UVM\_NO\_ENDIAN  
 UVM\_NO\_HIER  
 UVM\_NONE  
 UVM\_NOT\_OK  
 uvm\_object  
 uvm\_object\_registry#(T,Tname)  
 uvm\_object\_string\_pool#(T)  
 uvm\_object\_wrapper  
 uvm\_objection  
 uvm\_objection\_callback  
 uvm\_objection\_event  
 UVM\_OCT  
 uvm\_packer  
 uvm\_pair classes  
 uvm\_pair#(T1,T2)  
 uvm\_path\_e  
 uvm\_phase  
 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\_state  
**UVM\_PHASE\_SYNCING**  
**UVM\_PHASE\_TASK**  
**UVM\_PHASE\_TOPDOWN**  
uvm\_phase\_transition  
uvm\_phase\_type  
uvm\_pool#(KEY,T)  
**UVM\_PORT**  
uvm\_port\_base#(IF)  
uvm\_port\_component#(PORT)  
uvm\_port\_component\_base  
uvm\_port\_type\_e  
**UVM\_PREDICT**  
**UVM\_PREDICT\_DIRECT**  
uvm\_predict\_e  
**UVM\_PREDICT\_READ**  
**UVM\_PREDICT\_WRITE**  
uvm\_printer  
uvm\_printer\_knobs  
uvm\_push\_driver#(REQ,RSP)  
uvm\_push\_sequencer#(REQ,RSP)  
uvm\_queue#(T)  
uvm\_radix\_enum  
**UVM\_RAISED**  
uvm\_random\_stimulus#(T)  
**UVM\_READ**  
uvm\_recorder  
uvm\_recursion\_policy\_enum  
**UVM\_REFERENCE**  
uvm\_reg  
**UVM\_REG**  
uvm\_reg\_access\_seq  
uvm\_reg\_adapter  
uvm\_reg\_addr\_logic\_t  
uvm\_reg\_addr\_t  
uvm\_reg\_backdoor  
uvm\_reg\_bd\_cb  
uvm\_reg\_bd\_cb\_iter  
uvm\_reg\_bit\_bash\_seq  
uvm\_reg\_block  
uvm\_reg\_bus\_op  
uvm\_reg\_byte\_en\_t  
uvm\_reg\_cb  
uvm\_reg\_cb\_iter  
uvm\_reg\_cbs  
uvm\_reg\_cvr\_t  
uvm\_reg\_data\_logic\_t  
uvm\_reg\_data\_t  
uvm\_reg\_defines.svh

- 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

### value

- uvm\_reg\_field
- uvm\_reg\_item

### Variables

- Global
- uvm\_comparer
- uvm\_hdl\_path\_concat
- uvm\_line\_printer
- uvm\_mem\_access\_seq
- uvm\_mem\_mam\_cfg
- uvm\_mem\_mam\_policy
- uvm\_mem\_shared\_access\_seq
- uvm\_mem\_single\_access\_seq
- uvm\_mem\_single\_walk\_seq
- uvm\_mem\_walk\_seq
- uvm\_packer
- uvm\_pair#(T1,T2)
- uvm\_printer\_knobs
- uvm\_recorder
- uvm\_reg\_access\_seq
- uvm\_reg\_bit\_bash\_seq
- uvm\_reg\_bus\_op
- uvm\_reg\_frontdoor
- uvm\_reg\_hw\_reset\_seq
- uvm\_reg\_item
- uvm\_reg\_mem\_built\_in\_seq
- uvm\_reg\_mem\_hdl\_paths\_seq
- uvm\_reg\_mem\_shared\_access\_seq
- uvm\_reg\_predictor
- uvm\_reg\_shared\_access\_seq
- uvm\_reg\_single\_access\_seq
- uvm\_reg\_single\_bit\_bash\_seq
- uvm\_report\_server
- uvm\_root
- uvm\_sequencer#(REQ,RSP)
- uvm\_table\_printer
- uvm\_test\_done\_objection
- uvm\_transaction
- uvm\_tree\_printer

### verbosity

- uvm\_comparer

### Virtual Register Field Classes

## W

### wait\_for

- uvm\_barrier
- uvm\_objection

- 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

[\\$#!](#) · [0-9](#) · [A](#) · [B](#) · [C](#) · [D](#) · [E](#) · [F](#) · [G](#) · [H](#) · [I](#) · [J](#) · [K](#) · [L](#) · [M](#) · [N](#) · [O](#) · [P](#) · [Q](#) · [R](#) · [S](#) · [T](#) · [U](#) · [V](#) · [W](#) · [X](#) · [Y](#) · [Z](#)

## P

**Phasing Implementation**  
**Pre-Defined Phases**

## U

**User-Defined Phases**  
**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\_imp**  
**uvm\_analysis\_port**  
**uvm\_barrier**  
**uvm\_bottomup\_phase**  
**uvm\_built\_in\_clone#(T)**  
**uvm\_built\_in\_comp#(T)**  
**uvm\_built\_in\_converter#(T)**  
**uvm\_built\_in\_pair#(T1,T2)**  
**uvm\_callback**  
**uvm\_callback\_iter**  
**uvm\_callbacks#(T,CB)**  
**uvm\_callbacks\_objection**  
**uvm\_class\_clone#(T)**  
**uvm\_class\_comp#(T)**  
**uvm\_class\_converter#(T)**  
**uvm\_cmdline\_processor**  
**uvm\_comparer**  
**uvm\_component**  
**uvm\_component\_registry#(T,Tname)**  
**uvm\_config\_db#(T)**  
**uvm\_domain**  
**uvm\_driver#(REQ,RSP)**  
**uvm\_env**  
**uvm\_event**  
**uvm\_event\_callback**  
**uvm\_factory**  
**uvm\_hdl\_path\_concat**  
**uvm\_heartbeat**  
**uvm\_in\_order\_built\_in\_comparator#(T)**  
**uvm\_in\_order\_class\_comparator#(T)**  
**uvm\_in\_order\_comparator#(T,comp\_type,convert,pair\_type)**  
**uvm\_line\_printer**  
**uvm\_mem**  
**uvm\_mem\_access\_seq**

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

```
uvm_tlm_transport_channel#(REQ,RSP)
uvm_topdown_phase
uvm_transaction
uvm_tree_printer
uvm_utils
uvm_vreg_cbs
uvm_vreg_field
uvm_vreg_field_cbs
```

# File Index

[\\$#!](#) · [0-9](#) · [A](#) · [B](#) · [C](#) · [D](#) · [E](#) · [F](#) · [G](#) · [H](#) · [I](#) · [J](#) · [K](#) · [L](#) · [M](#) · [N](#) · [O](#) · [P](#) · [Q](#) · [R](#) · [S](#) · [T](#) · [U](#) · [V](#) · [W](#) · [X](#) · [Y](#) · [Z](#)

## U

	<a href="#">uvm_algorithmic_comparator.svh</a>
	<a href="#">uvm_callback_defines.svh</a>
	<a href="#">uvm_reg_defines.svh</a>

# Macro Index

[\\$#!](#) · [0-9](#) · [A](#) · [B](#) · [C](#) · [D](#) · [E](#) · [F](#) · [G](#) · [H](#) · [I](#) · [J](#) · [K](#) · [L](#) · [M](#) · [N](#) · [O](#) · [P](#) · [Q](#) · [R](#) · [S](#) · [T](#) · [U](#) · [V](#) · [W](#) · [X](#) · [Y](#) · [Z](#)

## \$#!

- ``uvm_add_to_sequence_library`
- ``uvm_analysis_imp_decl`
- ``uvm_blocking_get_imp_decl`
- ``uvm_blocking_get_peek_imp_decl`
- ``uvm_blocking_master_imp_decl`
- ``uvm_blocking_peek_imp_decl`
- ``uvm_blocking_put_imp_decl`
- ``uvm_blocking_slave_imp_decl`
- ``uvm_blocking_transport_imp_decl`
- ``uvm_component_end`
- ``uvm_component_param_utils`
- ``uvm_component_param_utils_begin`
- ``uvm_component_registry`
- ``uvm_component_utils`
- ``uvm_component_utils_begin`
- ``uvm_create`
- ``uvm_create_on`
- ``uvm_declare_p_sequencer`
- ``uvm_do`
- ``uvm_do_callbacks`
- ``uvm_do_callbacks_exit_on`
- ``uvm_do_obj_callbacks`
- ``uvm_do_obj_callbacks_exit_on`
- ``uvm_do_on`
- ``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_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_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\_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**



- ` 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
- ` uvm\_unpack\_sarray
- ` uvm\_unpack\_sarrayN
- ` uvm\_unpack\_string
- ` uvm\_warning
- ` uvm\_warning\_context

# Method Index

\$#! · 0-9 · [A](#) · [B](#) · [C](#) · [D](#) · [E](#) · [F](#) · [G](#) · [H](#) · [I](#) · [J](#) · [K](#) · [L](#) · [M](#) · [N](#) · [O](#) · [P](#) · [Q](#) · [R](#) · [S](#) · [T](#) · [U](#) · [V](#) · [W](#) · [X](#) · [Y](#) · [Z](#)

## A

### **accept\_tr**

[uvm\\_component](#)  
[uvm\\_transaction](#)

### **add**

[uvm\\_callbacks#\(T,CB\)](#)  
[uvm\\_heartbeat](#)  
[uvm\\_pool#\(KEY,T\)](#)  
[uvm\\_reg\\_read\\_only\\_cbs](#)  
[uvm\\_reg\\_write\\_only\\_cbs](#)

### **add\_by\_name**

[uvm\\_callbacks#\(T,CB\)](#)

### **add\_callback**

[uvm\\_event](#)

### **add\_coverage**

[uvm\\_mem](#)  
[uvm\\_reg](#)  
[uvm\\_reg\\_block](#)

### **add\_hdl\_path**

[uvm\\_mem](#)  
[uvm\\_reg](#)  
[uvm\\_reg\\_block](#)  
[uvm\\_reg\\_file](#)

### **add\_hdl\_path\_slice**

[uvm\\_mem](#)  
[uvm\\_reg](#)

### **add\_mem**

[uvm\\_reg\\_map](#)

### **add\_path**

[uvm\\_hdl\\_path\\_concat](#)

### **add\_phase**

[uvm\\_phase](#)

### **add\_reg**

[uvm\\_reg\\_map](#)

### **add\_schedule**

[uvm\\_phase](#)

### **add\_slice**

[uvm\\_hdl\\_path\\_concat](#)

### **add\_submap**

[uvm\\_reg\\_map](#)

### **adjust\_name**

[uvm\\_printer](#)

### **all\_dropped**

[uvm\\_callbacks\\_objection](#)  
[uvm\\_component](#)  
[uvm\\_objection](#)  
[uvm\\_objection\\_callback](#)

uvm\_test\_done\_objection

### **allocate**

uvm\_vreg

### **apply\_config\_settings**

uvm\_component

## **B**

### **b\_transport**

uvm\_tlm\_if

### **backdoor\_read**

uvm\_mem

uvm\_reg

### **backdoor\_read\_func**

uvm\_mem

uvm\_reg

### **backdoor\_watch**

uvm\_reg

### **backdoor\_write**

uvm\_mem

uvm\_reg

### **begin\_child\_tr**

uvm\_component

uvm\_transaction

### **begin\_tr**

uvm\_component

uvm\_transaction

### **body**

uvm\_mem\_access\_seq

uvm\_mem\_single\_walk\_seq

uvm\_mem\_walk\_seq

uvm\_reg\_access\_seq

uvm\_reg\_bit\_bash\_seq

uvm\_reg\_mem\_built\_in\_seq

uvm\_reg\_mem\_shared\_access\_seq

uvm\_reg\_sequence

uvm\_sequence\_base

### **build\_coverage**

uvm\_mem

uvm\_reg

uvm\_reg\_block

### **build\_phase**

uvm\_component

### **burst\_read**

uvm\_mem

uvm\_mem\_region

### **burst\_write**

uvm\_mem

uvm\_mem\_region

### **bus2reg**

uvm\_reg\_adapter

uvm\_reg\_tlm\_adapter

## C

### **callback\_mode**

uvm\_callback

### **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

### **check\_config\_usage**

uvm\_component

### **check\_data\_width**

uvm\_reg\_block

### **check\_phase**

uvm\_component

uvm\_reg\_predictor

### **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

### **compare**

uvm\_object

### **compare\_field**

uvm\_comparer

### **compare\_field\_int**

uvm\_comparer

### **compare\_field\_real**

uvm\_comparer

### **compare\_object**

uvm\_comparer

### **compare\_string**

uvm\_comparer

**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

**create\_object\_by\_type**  
uvm\_factory

**current\_grabber**  
uvm\_sequencer\_base

## D

**debug\_connected\_to**  
uvm\_port\_base#(IF)

**debug\_create\_by\_name**  
uvm\_factory

**debug\_create\_by\_type**  
uvm\_factory

**debug\_provided\_to**  
uvm\_port\_base#(IF)

**decode**  
uvm\_reg\_cbs

**decr**  
uvm\_tlm\_time

**define\_access**  
uvm\_reg\_field

**define\_phase\_schedule**  
uvm\_component

**delete**  
uvm\_callbacks#(T,CB)  
uvm\_object\_string\_pool#(T)  
uvm\_pool#(KEY,T)  
uvm\_queue#(T)

**delete\_by\_name**  
uvm\_callbacks#(T,CB)

**delete\_callback**  
uvm\_event

**die**  
uvm\_report\_object

**disable\_recording**  
uvm\_transaction

**display**  
uvm\_callbacks#(T,CB)

**display\_objections**  
uvm\_objection

**do\_accept\_tr**  
uvm\_component  
uvm\_transaction

**do\_begin\_tr**  
uvm\_component  
uvm\_transaction

**do\_block**  
uvm\_mem\_access\_seq  
uvm\_mem\_walk\_seq  
uvm\_reg\_access\_seq  
uvm\_reg\_bit\_bash\_seq

- 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**

- uvm\_resource\_db
- uvm\_resource\_pool

### **dump\_report\_state**

- uvm\_report\_object

### **dump\_server\_state**

- uvm\_report\_server

## **E**

### **emit**

- uvm\_printer
- uvm\_table\_printer
- uvm\_tree\_printer

### **enable\_recording**

- uvm\_transaction

### **encode**

- uvm\_reg\_cbs

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

- uvm\_component

### **end\_tr**

- uvm\_component
- uvm\_transaction

### **exec\_func**

- uvm\_phase

### **exec\_task**

- uvm\_phase

### **execute**

- uvm\_bottomup\_phase
- uvm\_task\_phase
- uvm\_topdown\_phase

### **execute\_item**

- uvm\_sequencer\_base

### **exists**

- uvm\_config\_db#(T)
- uvm\_pool#(KEY,T)

### **extract\_phase**

- uvm\_component

## **F**

### **final\_phase**

- uvm\_component

### **find**

- uvm\_phase
- uvm\_root

### **find\_all**

- uvm\_root
- uvm\_utils

### **find\_block**



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

**get\_vfield\_by\_name**

uvm\_mem  
uvm\_reg\_block

**get\_virtual\_fields**

uvm\_mem  
uvm\_reg\_block  
uvm\_reg\_map

**get\_virtual\_registers**

uvm\_mem  
uvm\_mem\_region  
uvm\_reg\_block  
uvm\_reg\_map

**get\_vreg\_by\_name**

uvm\_mem  
uvm\_reg\_block

**get\_vreg\_by\_offset**

uvm\_mem

**global\_stop\_request****grab**

uvm\_sequence\_base  
uvm\_sequencer\_base

## H

**has\_child**

uvm\_component

**has\_coverage**

uvm\_mem  
uvm\_reg  
uvm\_reg\_block

**has\_do\_available**

uvm\_sequencer\_base  
uvm\_sqr\_if\_base#(REQ,RSP)

**has\_hdl\_path**

uvm\_mem  
uvm\_reg  
uvm\_reg\_block  
uvm\_reg\_file

**has\_lock**

uvm\_sequence\_base  
uvm\_sequencer\_base

**has\_reset**

uvm\_reg  
uvm\_reg\_field

# I

## **ID**

uvm\_tlm\_extension

## **implement**

uvm\_vreg

## **include\_coverage**

uvm\_reg

## **incr**

uvm\_tlm\_time

## **incr\_id\_count**

uvm\_report\_server

## **incr\_quit\_count**

uvm\_report\_server

## **incr\_severity\_count**

uvm\_report\_server

## **init\_access\_record**

uvm\_resource\_base

## **insert**

uvm\_queue#(T)

## **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  
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

### **m\_set\_hier\_mode**

uvm\_objection

### **main\_phase**

uvm\_component

### **match\_scope**

uvm\_resource\_base

### **max\_size**

uvm\_port\_base#(IF)

### **mid\_do**

uvm\_sequence\_base

### **min\_size**

uvm\_port\_base#(IF)

### **mirror**

uvm\_reg

uvm\_reg\_block

uvm\_reg\_field

uvm\_reg\_fifo

### **mirror\_reg**

uvm\_reg\_sequence

## N

### **nb\_transport**

uvm\_tlm\_if\_base#(T1,T2)

### **nb\_transport\_bw**

uvm\_tlm\_if

### **nb\_transport\_fw**

uvm\_tlm\_if

### **needs\_update**

uvm\_reg

uvm\_reg\_block

uvm\_reg\_field

### **new**

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\_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



- uvm\_tlm\_nb\_transport\_bw\_port
- uvm\_tlm\_req\_rsp\_channel#(REQ,RSP)
- uvm\_tlm\_time
- uvm\_tlm\_transport\_channel#(REQ,RSP)
- uvm\_topdown\_phase
- uvm\_transaction
- uvm\_vreg
- uvm\_vreg\_field

#### **next**

- uvm\_callback\_iter
- uvm\_pool#(KEY,T)

#### **num**

- uvm\_pool#(KEY,T)

## **P**

#### **pack**

- uvm\_object

#### **pack\_bytes**

- uvm\_object

#### **pack\_field**

- uvm\_packer

#### **pack\_field\_int**

- uvm\_packer

#### **pack\_ints**

- uvm\_object

#### **pack\_object**

- uvm\_packer

#### **pack\_real**

- uvm\_packer

#### **pack\_string**

- uvm\_packer

#### **pack\_time**

- uvm\_packer

#### **peek**

- uvm\_mem
- uvm\_mem\_region
- uvm\_reg
- uvm\_reg\_field
- uvm\_sqr\_if\_base#(REQ,RSP)
- uvm\_tlm\_if\_base#(T1,T2)
- uvm\_vreg
- uvm\_vreg\_field

#### **peek\_mem**

- uvm\_reg\_sequence

#### **peek\_reg**

- uvm\_reg\_sequence

#### **phase\_ended**

- uvm\_component
- uvm\_phase

#### **phase\_started**

- uvm\_component

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

### **qualify**

uvm\_test\_done\_objection

## R

### **raise\_objection**

uvm\_objection

uvm\_phase

uvm\_test\_done\_objection

### **raised**

uvm\_callbacks\_objection

uvm\_component

uvm\_objection

uvm\_objection\_callback

### **read**

uvm\_mem

uvm\_mem\_region

uvm\_reg

uvm\_reg\_backdoor

uvm\_reg\_field

uvm\_reg\_fifo

uvm\_resource#(T)

uvm\_vreg

uvm\_vreg\_field

### **read\_by\_name**

uvm\_resource\_db

### **read\_by\_type**

uvm\_resource\_db

### **read\_func**

uvm\_reg\_backdoor

### **read\_mem**

uvm\_reg\_sequence

### **read\_mem\_by\_name**

uvm\_reg\_block

### **read\_reg**

uvm\_reg\_sequence

### **read\_reg\_by\_name**

uvm\_reg\_block

### **read\_with\_loc;**

uvm\_resource#(T)

### **reconfigure**

uvm\_mem\_mam

### **record**

uvm\_object

- 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

## **sample**

- uvm\_mem
- uvm\_reg
- uvm\_reg\_block

## **sample\_values**

- uvm\_reg
- uvm\_reg\_block

## **send\_request**

- uvm\_sequence#(REQ,RSP)
- uvm\_sequence\_base
- uvm\_sequencer\_base
- uvm\_sequencer\_param\_base#(REQ,RSP)

## **set**

- uvm\_config\_db#(T)
- uvm\_hdl\_path\_concat
- uvm\_reg
- uvm\_reg\_field
- uvm\_reg\_fifo
- uvm\_resource#(T)
- uvm\_resource\_db
- uvm\_resource\_pool

## **set\_priority**

- uvm\_resource#(T)
- uvm\_resource\_base

## **set\_abstime**

- uvm\_tlm\_time

## **set\_access**

- uvm\_reg\_field

## **set\_action**

- uvm\_report\_catcher

## **set\_address**

- uvm\_tlm\_generic\_payload

## **set\_anonymous**

- uvm\_resource\_db

## **set\_arbitration**

- uvm\_sequencer\_base

## **set\_auto\_predict**

- uvm\_reg\_map

## **set\_auto\_reset**

- uvm\_barrier

## **set\_backdoor**

- uvm\_mem
- uvm\_reg
- uvm\_reg\_block

## **set\_base\_addr**

- uvm\_reg\_map



**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

**set\_verbosity**  
 uvm\_report\_catcher

**set\_volatility**  
 uvm\_reg\_field

**set\_write**  
 uvm\_tlm\_generic\_payload

**shutdown\_phase**  
 uvm\_component

**size**  
 uvm\_port\_base#(IF)  
 uvm\_queue#(T)  
 uvm\_reg\_fifo  
 uvm\_tlm\_fifo

**spell\_check**  
 uvm\_resource\_pool

**sprint**  
 uvm\_object

**start**  
 uvm\_heartbeat  
 uvm\_sequence\_base

**start\_item**  
 uvm\_sequence\_base  
 uvm\_sequence\_item

**start\_of\_simulation\_phase**  
 uvm\_component

**start\_phase\_sequence**  
 uvm\_sequencer\_base

**status**  
 uvm\_component

**stop**  
 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)

**summarize**  
 uvm\_report\_server

**summarize\_report\_catcher**  
 uvm\_report\_catcher

**suspend**  
 uvm\_component

**sync**  
 uvm\_phase

**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

## **ungrab**

uvm\_sequence\_base  
uvm\_sequencer\_base

## **unlock**

uvm\_resource\_base  
uvm\_sequence\_base  
uvm\_sequencer\_base

## **unpack**

uvm\_object

## **unpack\_bytes**

uvm\_object

## **unpack\_field**

uvm\_packer

## **unpack\_field\_int**

uvm\_packer

## **unpack\_ints**

uvm\_object

## **unpack\_object**

uvm\_packer

## **unpack\_real**

uvm\_packer

## **unpack\_string**

uvm\_packer

## **unpack\_time**

uvm\_packer

## **unsync**

uvm\_phase

## **update**

uvm\_reg  
uvm\_reg\_block  
uvm\_reg\_fifo

## **update\_reg**

uvm\_reg\_sequence

## **use\_response\_handler**

uvm\_sequence\_base

## **used**

uvm\_tlm\_fifo

## **user\_priority\_arbitration**

uvm\_sequencer\_base

## **uvm\_bits\_to\_string**

## **uvm\_hdl\_check\_path**

## **uvm\_hdl\_deposit**

## **uvm\_hdl\_force**

## **uvm\_hdl\_force\_time**

## **uvm\_hdl\_read**

## **uvm\_hdl\_release**

- uvm\_hdl\_release\_and\_read**
- uvm\_is\_match**
- 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\_info**
  - Global
  - uvm\_report\_catcher
  - uvm\_report\_object
  - uvm\_sequence\_item
- uvm\_report\_warning**
  - Global
  - uvm\_report\_catcher
  - uvm\_report\_object
  - uvm\_sequence\_item
- uvm\_split\_string**
- uvm\_string\_to\_bits**
- uvm\_wait\_for\_nba\_region**

## W

- wait\_for**
  - uvm\_barrier
  - uvm\_objection
- 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
- 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

\$#! · 0-9 · [A](#) · [B](#) · [C](#) · [D](#) · [E](#) · [F](#) · [G](#) · [H](#) · [I](#) · [J](#) · [K](#) · [L](#) · [M](#) · [N](#) · [O](#) · [P](#) · [Q](#) · [R](#) · [S](#) · [T](#) · [U](#) · [V](#) · [W](#) · [X](#) · [Y](#) · [Z](#)

## 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

[\\$#!](#) · [0-9](#) · [A](#) · [B](#) · [C](#) · [D](#) · [E](#) · [F](#) · [G](#) · [H](#) · [I](#) · [J](#) · [K](#) · [L](#) · [M](#) · [N](#) · [O](#) · [P](#) · [Q](#) · [R](#) · [S](#) · [T](#) · [U](#) · [V](#) · [W](#) · [X](#) · [Y](#) · [Z](#)

## \$#!

- +UVM\_DUMP\_CMDLINE\_ARGS**
  - [uvm\\_cmdline\\_processor](#)
- +UVM\_MAX\_QUIT\_COUNT**
  - [uvm\\_cmdline\\_processor](#)
- +UVM\_OBJECTION\_TRACE**
  - [uvm\\_cmdline\\_processor](#)
- +UVM\_PHASE\_TRACE**
  - [uvm\\_cmdline\\_processor](#)
- +uvm\_set\_action**
  - [uvm\\_cmdline\\_processor](#)
- +uvm\_set\_config\_int,+uvm\_set\_config\_string**
  - [uvm\\_cmdline\\_processor](#)
- +uvm\_set\_inst\_override,+uvm\_set\_type\_override**
  - [uvm\\_cmdline\\_processor](#)
- +uvm\_set\_severity**
  - [uvm\\_cmdline\\_processor](#)
- +uvm\_set\_verbosity**
  - [uvm\\_cmdline\\_processor](#)
- +UVM\_TESTNAME**
  - [uvm\\_cmdline\\_processor](#)
- +UVM\_TIMEOUT**
  - [uvm\\_cmdline\\_processor](#)
- +UVM\_VERBOSITY**
  - [uvm\\_cmdline\\_processor](#)

## A

- abstract**
  - [uvm\\_comparer](#)
  - [uvm\\_packer](#)
  - [uvm\\_recorder](#)
- abstractions**
  - [uvm\\_reg\\_mem\\_hdl\\_paths\\_seq](#)
- adapter**
  - [uvm\\_reg\\_predictor](#)
  - [uvm\\_reg\\_sequence](#)
- addr**
  - [uvm\\_reg\\_bus\\_op](#)

## B

- bd\_kind**
  - uvm\_reg\_item
- begin\_elements**
  - uvm\_printer\_knobs
- begin\_event**
  - uvm\_transaction
- big\_endian**
  - uvm\_packer
- bin\_radix**
  - uvm\_printer\_knobs
- body**
  - uvm\_reg\_hw\_reset\_seq
- build\_ph**
  - Pre-Defined Phases
- bus\_in**
  - uvm\_reg\_predictor
- byte\_en**
  - uvm\_reg\_bus\_op

## C

- check\_ph**
  - Pre-Defined Phases
- check\_type**
  - uvm\_comparer
- configure\_ph**
  - Pre-Defined Phases
- connect\_ph**
  - Pre-Defined Phases

## D

- data**
  - uvm\_reg\_bus\_op
- dec\_radix**
  - uvm\_printer\_knobs
- default\_alloc**
  - uvm\_mem\_mam
- default\_map**
  - uvm\_reg\_block
- default\_path**
  - uvm\_reg\_block
- default\_precedence**
  - uvm\_resource\_base
- default\_radix**
  - uvm\_printer\_knobs
  - uvm\_recorder
- depth**

uvm\_printer\_knobs

## E

### **element**

uvm\_reg\_item

### **element\_kind**

uvm\_reg\_item

### **enable\_print\_topology**

uvm\_root

### **enable\_stop\_interrupt**

uvm\_component

### **end\_elements**

uvm\_printer\_knobs

### **end\_event**

uvm\_transaction

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

Pre-Defined Phases

### **end\_offset**

uvm\_mem\_mam\_cfg

### **events**

uvm\_transaction

### **extension**

uvm\_reg\_item

### **extract\_ph**

Pre-Defined Phases

## F

### **fifo**

uvm\_reg\_fifo

### **final\_ph**

Pre-Defined Phases

### **finish\_on\_completion**

uvm\_root

### **fname**

uvm\_reg\_item

### **footer**

uvm\_printer\_knobs

### **full\_name**

uvm\_printer\_knobs

## H

### **header**

uvm\_printer\_knobs

**hex\_radix**  
uvm\_printer\_knobs

## I

**id\_count**  
uvm\_report\_server

**identifier**  
uvm\_printer\_knobs  
uvm\_recorder

**in\_use**  
uvm\_mem\_mam\_policy

**indent**  
uvm\_printer\_knobs

**info**  
uvm\_reg\_bus\_op

## K

**kind**  
uvm\_reg\_bus\_op  
uvm\_reg\_item

**knobs**  
uvm\_printer

## L

**len**  
uvm\_mem\_mam\_policy

**lineno**  
uvm\_reg\_item

**local\_map**  
uvm\_reg\_item

**locality**  
uvm\_mem\_mam\_cfg

## M

**m\_address**  
uvm\_tlm\_generic\_payload

**m\_byte\_enable**  
uvm\_tlm\_generic\_payload

**m\_byte\_enable\_length**  
uvm\_tlm\_generic\_payload

**m\_command**

uvm\_tlm\_generic\_payload

### **m\_data**

uvm\_tlm\_generic\_payload

### **m\_dmi**

uvm\_tlm\_generic\_payload

### **m\_length**

uvm\_tlm\_generic\_payload

### **m\_response\_status**

uvm\_tlm\_generic\_payload

### **m\_streaming\_width**

uvm\_tlm\_generic\_payload

### **main\_ph**

Pre-Defined Phases

### **mam**

uvm\_mem

### **map**

uvm\_reg\_item

uvm\_reg\_predictor

### **max\_offset**

uvm\_mem\_mam\_policy

### **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

### **min\_offset**

uvm\_mem\_mam\_policy

### **miscompares**

uvm\_comparer

### **mode**

uvm\_mem\_mam\_cfg

### **model**

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\_built\_in\_seq

uvm\_reg\_mem\_shared\_access\_seq

uvm\_reg\_sequence

## **N**

### **n\_bits**

uvm\_reg\_bus\_op

### **n\_bytes**



uvm\_mem\_mam\_cfg

## **new**

uvm\_line\_printer  
uvm\_table\_printer  
uvm\_tree\_printer

# O

## **oct\_radix**

uvm\_printer\_knobs

## **offset**

uvm\_reg\_item

# P

## **parent**

uvm\_reg\_item

## **path**

uvm\_reg\_item

## **physical**

uvm\_comparer  
uvm\_packer  
uvm\_recorder

## **policy**

uvm\_comparer

## **post\_configure\_ph**

Pre-Defined Phases

## **post\_main\_ph**

Pre-Defined Phases

## **post\_reset\_ph**

Pre-Defined Phases

## **post\_shutdown\_ph**

Pre-Defined Phases

## **pre\_configure\_ph**

Pre-Defined Phases

## **pre\_main\_ph**

Pre-Defined Phases

## **pre\_reset\_ph**

Pre-Defined Phases

## **pre\_shutdown\_ph**

Pre-Defined Phases

## **precedence**

uvm\_resource\_base

## **prefix**

uvm\_printer\_knobs

## **print\_config\_matches**

uvm\_component

## **print\_enabled**

uvm\_component

**prior**

uvm\_reg\_item

**provides\_responses**

uvm\_reg\_adapter

## R

**recursion\_policy**

uvm\_recorder

**reference**

uvm\_printer\_knobs

**reg\_ap**

uvm\_reg\_predictor

**reg\_seq**

uvm\_reg\_access\_seq

uvm\_reg\_bit\_bash\_seq

uvm\_reg\_mem\_shared\_access\_seq

**reg\_seqr**

uvm\_reg\_sequence

**report\_ph**

Pre-Defined Phases

**reset\_ph**

Pre-Defined Phases

**result**

uvm\_comparer

**rg**

uvm\_reg\_shared\_access\_seq

uvm\_reg\_single\_access\_seq

uvm\_reg\_single\_bit\_bash\_seq

**run\_ph**

Pre-Defined Phases

**rw\_info**

uvm\_reg\_frontdoor

## S

**separator**

uvm\_printer\_knobs

**seq\_item\_export**

uvm\_sequencer#(REQ,RSP)

**sequencer**

uvm\_reg\_frontdoor

**sev**

uvm\_comparer

**show\_max**

uvm\_comparer

**show\_radix**

- uvm\_printer\_knobs
- show\_root**
  - uvm\_printer\_knobs
- shutdown\_ph**
  - Pre-Defined Phases
- size**
  - uvm\_printer\_knobs
- slices**
  - uvm\_hdl\_path\_concat
- start\_of\_simulation\_ph**
  - Pre-Defined Phases
- start\_offset**
  - uvm\_mem\_mam\_policy
- starting\_phase**
  - uvm\_sequence\_base
- status**
  - uvm\_reg\_bus\_op
  - uvm\_reg\_item
- stop\_timeout**
  - uvm\_test\_done\_objection
- supports\_byte\_enable**
  - uvm\_reg\_adapter

## T

- T1 first**
  - uvm\_pair#(T1,T2)
- T2 second**
  - uvm\_pair#(T1,T2)
- tests**
  - uvm\_reg\_mem\_built\_in\_seq
- top\_levels**
  - uvm\_root
- tr\_handle**
  - uvm\_recorder
- type\_name**
  - uvm\_printer\_knobs

## U

- unsigned\_radix**
  - uvm\_printer\_knobs
- use\_metadata**
  - uvm\_packer
- use\_uvm\_seeding**
  - uvm\_object
- uvm\_default\_comparer**
- uvm\_default\_line\_printer**

**uvm\_default\_packer**  
**uvm\_default\_printer**  
**uvm\_default\_recorder**  
**uvm\_default\_table\_printer**  
**uvm\_default\_tree\_printer**  
**UVM\_HDL\_MAX\_WIDTH**  
**uvm\_test\_done**  
**uvm\_top**  
    uvm\_root

## V

**value**  
    uvm\_reg\_field  
    uvm\_reg\_item  
**verbosity**  
    uvm\_comparer

# Constant Index

\$ # ! · 0 - 9 · A · **B** · **C** · D · **E** · **F** · G · H · I · J · K · L · M · N · O · **P** · Q · R · **S** · T · **U** · V · W · X · Y · Z

## 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

\$#! · 0-9 · [A](#) · [B](#) · C · D · E · F · [G](#) · H · I · J · K · L · [M](#) · N · O · [P](#) · Q · [R](#) · [S](#) · [T](#) · U · V · W · X · Y · Z

## A

### **after\_export**

[uvm\\_algorithmic\\_comparator#\(BEFORE,AFTER,TRANSFORMER\)](#)  
[uvm\\_in\\_order\\_comparator#\(T,comp\\_type,convert,pair\\_type\)](#)

### **analysis\_export**

[uvm\\_subscriber](#)

### **analysis\_export#(T)**

[uvm\\_tlm\\_analysis\\_fifo](#)

## B

### **before\_export**

[uvm\\_algorithmic\\_comparator#\(BEFORE,AFTER,TRANSFORMER\)](#)  
[uvm\\_in\\_order\\_comparator#\(T,comp\\_type,convert,pair\\_type\)](#)

### **blocking\_put\_port**

[uvm\\_random\\_stimulus#\(T\)](#)

## G

### **get\_ap**

[uvm\\_tlm\\_fifo\\_base#\(T\)](#)

### **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#\(REQ,RSP\)](#)

## M

### **master\_export**

[uvm\\_tlm\\_req\\_rsp\\_channel#\(REQ,RSP\)](#)

## P

### **pair\_ap**

[uvm\\_in\\_order\\_comparator#\(T,comp\\_type,convert,pair\\_type\)](#)

### **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)

## 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)