



Open Verification Methodology

# OVM Class Reference

Version 2.1.2  
June 2011

© 2008–2011 Cadence Design Systems, Inc. (Cadence). All rights reserved.  
Cadence Design Systems, Inc., 2655 Seely Ave., San Jose, CA 95134, USA.

© 2008–2011 Mentor Graphics, Corp. (Mentor). All rights reserved.  
Mentor Graphics, Corp., 8005 SW Boeckman Rd., Wilsonville, OR 97070, USA

This product is licensed under the Apache Software Foundation's Apache License, Version 2.0, January 2004. The full license is available at: <http://www.apache.org/licenses/>

**Trademarks:** Trademarks and service marks of Cadence Design Systems, Inc. and Mentor Graphics, Corp. contained in this document are attributed to Cadence and Mentor with the appropriate symbol. For queries regarding Cadence's or Mentor's trademarks, contact the corporate legal department at the address shown above. All other trademarks are the property of their respective holders.

**Restricted Permission:** This publication is protected by copyright law. Cadence and Mentor grant permission to print hard copy of this publication subject to the following conditions:

1. The publication may not be modified in any way.
2. Any authorized copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement.

**Disclaimer:** Information in this publication is provided as is and subject to change without notice and does not represent a commitment on the part of Cadence or Mentor. Cadence and Mentor do not make, and expressly disclaim, any representations or warranties as to the completeness, accuracy, or usefulness of the information contained in this document. Cadence and Mentor do not warrant that use of such information will not infringe any third party rights, nor does Cadence or Mentor assume any liability for damages or costs of any kind that may result from use of such information.

**Restricted Rights:** Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor

# OVM Class Reference

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

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

We divide the OVM 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.

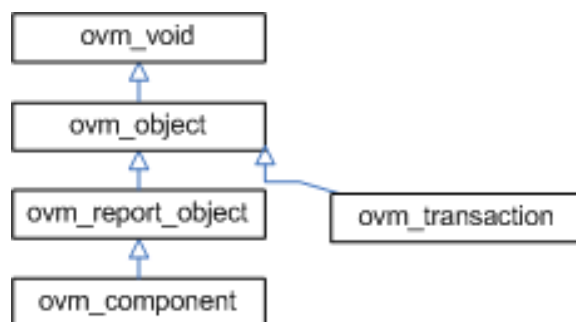
<i>Base</i>	This basic building blocks for all environments are components, which do the actual work, transactions, which convey information between components, and ports, which provide the interfaces used to convey transactions. The OVM's core <i>base</i> classes provide these building blocks. See <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 OVM factory is used to manufacture (create) OVM 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>Synchronization</i>	The OVM provides event and barrier synchronization classes for process synchronization. See <a href="#">Synchronization Classes</a> for more information.
<i>Policies</i>	Each of OVM's policy classes perform a specific task for ovm_object-based objects: printing, comparing, recording, packing, and unpacking. They are implemented separately from ovm_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 <a href="#">Policy Classes</a> for more information.
<i>TLM</i>	The OVM 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 <a href="#">TLM Interfaces, Ports, and Exports</a> for details.
<i>Components</i>	Components form the foundation of the OVM. They encapsulate behavior of drivers, scoreboards, and other objects in a testbench. The OVM library provides a set of predefined component types, all derived directly or indirectly from <a href="#">ovm_component</a> . See <a href="#">Predefined Component Classes</a> for more information.

<i>Sequencers</i>	The sequencer serves as an arbiter for controlling transaction flow from multiple stimulus generators. More specifically, the sequencer controls the flow of <a href="#">ovm_sequence_item</a> -based transactions generated by one or more <a href="#">ovm_sequence #(REQ,RSP)</a> -based sequences. See <a href="#">Sequencer Classes</a> for more information.
<i>Sequences</i>	Sequences encapsulate user-defined procedures that generate multiple <a href="#">ovm_sequence_item</a> -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 <a href="#">Sequence Classes</a> for more information.
<i>Macros</i>	The OVM provides several macros to help increase user productivity. See <Utility and Field Macros> and <a href="#">Sequence and Do Action Macros</a> for a complete list.
<i>Globals</i>	This category defines a small list of types, variables, functions, and tasks defined in <i>ovm_pkg</i> scope. These items are accessible from any scope that imports the <i>ovm_pkg</i> . See <a href="#">Types and Enumerations</a> and <a href="#">Globals</a> for details.

## Core Base Classes

The OVM 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 OVM provides base classes for these, as shown below.



- [ovm\\_object](#) - All components and transactions derive from *ovm\_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.
- [ovm\\_component](#) - The *ovm\_component* class is the root base class for all OVM 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 *ovm\_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 *ovm\_component* also defines configuration, reporting, transaction recording, and factory interfaces.
- [ovm\\_transaction](#) - The *ovm\_transaction* is the root base class for OVM transactions, which, unlike *ovm\_components*, are transient in nature. It extends [ovm\\_object](#) to include a timing and recording interface. Simple transactions can derive directly from *ovm\_transaction*, while sequence-enabled transactions derive from *ovm\_sequence\_item*.
- [ovm\\_root](#) - The *ovm\_root* class is special *ovm\_component* that serves as the top-level component for all OVM components, provides phasing control for all OVM components, and other global services.

## ovm\_void

The *ovm\_void* class is the base class for all OVM 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 *ovm\_void* inherit none of the OVM functionality, but such classes may be placed in *ovm\_void*-typed containers along with other OVM objects.

# ovm\_object

The `ovm_object` class is the base class for all OVM 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 `ovm_object` must implement the pure virtual methods such as [create](#) and [get\\_type\\_name](#).

## Summary

### ovm\_object

The `ovm_object` class is the base class for all OVM data and hierarchical classes.

#### Class Declaration

```
virtual class ovm_object extends ovm_void
```

<a href="#">new</a>	Creates a new <code>ovm_object</code> with the given instance <i>name</i> .
<b>Seeding</b>	
<a href="#">use_ovm_seeding</a>	This bit enables or disables the OVM seeding mechanism.
<a href="#">reseed</a>	Calls <i>srandom</i> on the object to reseed the object using the OVM seeding mechanism, which sets the seed based on type name and instance name instead of based on instance position in a thread.
<b>Identification</b>	
<a href="#">set_name</a>	Sets the instance name of this object, overwriting any previously given name.
<a href="#">get_name</a>	Returns the name of the object, as provided by the <i>name</i> argument in the <a href="#">new</a> constructor or <a href="#">set_name</a> method.
<a href="#">get_full_name</a>	Returns the full hierarchical name of this object.
<a href="#">get_inst_id</a>	Returns the object's unique, numeric instance identifier.
<a href="#">get_inst_count</a>	Returns the current value of the instance counter, which represents the total number of <code>ovm_object</code> -based objects that have been allocated in simulation.
<a href="#">get_type</a>	Returns the type-proxy (wrapper) for this object.
<a href="#">get_object_type</a>	Returns the type-proxy (wrapper) for this object.
<a href="#">get_type_name</a>	This function returns the type name of the object, which is typically the type identifier enclosed in quotes.
<b>Creation</b>	
<a href="#">create</a>	The <code>create</code> method allocates a new object of the same type as this object and returns it via a base <code>ovm_object</code> handle.
<a href="#">clone</a>	The <code>clone</code> method creates and returns an exact copy of this object.
<b>Printing</b>	
<a href="#">print</a>	The <code>print</code> method deep-prints this object's properties in a format and manner governed by the given <i>printer</i> argument; if the <i>printer</i> argument is not provided, the global <a href="#">ovm_default_printer</a> is used.
<a href="#">sprint</a>	The <i>sprint</i> method works just like the <a href="#">print</a> method, except the output is returned in a string rather than displayed.

<code>do_print</code>	The <i>do_print</i> method is the user-definable hook called by <code>print</code> and <code>sprint</code> that allows users to customize what gets printed or sprinted beyond the field information provided by the <code>&lt;`ovm_field_*&gt;</code> macros.
<code>convert2string</code>	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.
Fields declared in <code>&lt;`ovm_field_*&gt;</code> macros, if used, will not	automatically appear in calls to <code>convert2string</code> .
<b>Recording</b>	
<code>record</code>	The <code>record</code> 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 <code>record</code> method.
<b>Copying</b>	
<code>copy</code>	The <code>copy</code> 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 <code>copy</code> method.
<b>Comparing</b>	
<code>compare</code>	The <code>compare</code> method deep compares this data object with the object provided in the <i>rhs</i> (right-hand side) argument.
<code>do_compare</code>	The <code>do_compare</code> method is the user-definable hook called by the <code>compare</code> method.
<b>Packing</b>	
<code>pack</code>	
<code>pack_bytes</code>	
<code>pack_ints</code>	
<code>do_pack</code>	The <code>pack</code> methods bitwise-concatenate this object's properties into an array of bits, bytes, or ints. The <code>do_pack</code> method is the user-definable hook called by the <code>pack</code> methods.
<b>Unpacking</b>	
<code>unpack</code>	
<code>unpack_bytes</code>	
<code>unpack_ints</code>	The <code>unpack</code> 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 <code>unpack</code> 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 ovm_object-based properties indexed by a <i>field_name</i> string.

## new

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

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



is unnamed.

## Seeding

---

### use\_ovm\_seeding

---

```
static bit use_ovm_seeding = 1
```

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

When enabled, OVM-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 [ovm\\_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 OVM 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\\_ovm\\_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 ovm\_objects do not inherently possess hierarchy.

Objects possessing hierarchy, such as [ovm\\_components](#), override the default implementation. Other objects might be associated with component hierarchy but are not themselves components. For example, [ovm\\_sequence #\(REQ,RSP\)](#) classes are typically associated with a [ovm\\_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 ovm\_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 ovm_object_wrapper get_type ()
```

Returns the type-proxy (wrapper) for this object. The [ovm\\_factory](#)'s type-based override and creation methods take arguments of [ovm\\_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 ovm_object;
  typedef ovm_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 ``ovm*_utils` macros, if employed.

## get\_object\_type

---

```
virtual function ovm_object_wrapper get_object_type ()
```

Returns the type-proxy (wrapper) for this object. The [ovm\\_factory](#)'s type-based override and creation methods take arguments of [ovm\\_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 ovm_object;
  typedef ovm_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 ``ovm_*_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 ovm_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 ovm_object create (string name = " ")
```

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

### A typical implementation is as follows

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

```
endfunction
```

## clone

---

```
virtual function ovm_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 (ovm_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 [ovm\\_default\\_printer](#) is used. See [ovm\\_printer](#) for more information on printer output formatting. See also [ovm\\_line\\_printer](#), [ovm\\_tree\\_printer](#), and [ovm\\_table\\_printer](#) for details on the pre-defined printer "policies," or formatters, provided by the OVM.

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 (ovm_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 (ovm\_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 `<`ovm_field_*>` macros.

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 ovm_object;
  data_obj data;
  int f1;
  virtual function void do_print (ovm_printer printer);
    super.do_print(printer);
    printer.print_field("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();
ovm_report_info("Received",t.sprint());
```

See [ovm\\_printer](#) for information about the printer API.

## 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 [ovm\\_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 `<`ovm_field_*>` macros, if used, will not

---

automatically appear in calls to `convert2string`.

An example implementation of `convert2string` follows.

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

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

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

**Then, to display an object, you could write**

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

**The output will look similar to**

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

## Recording

---

## record

---

```
function void record (ovm_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 `ovm_default_recorder` policy is used. See `ovm_recorder` for information.

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

## do\_record

---

```
virtual function void do_record (ovm_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 `ovm_recorder` for more information.

**A typical implementation is as follows**

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

# Copying

---

## copy

---



```
function void copy (ovm_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 (ovm_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 ovm_object;
...
int f1;
function void do_copy (ovm_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 (ovm_object    rhs,
                      ovm_comparer  comparer = null
                      )
```

The compare method deep compares this data object with the object provided in the *rhs* (right-hand side) argument.

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 *ovm\_default\_comparer* policy is used. See [ovm\\_comparer](#) for more information.

## do\_compare

---

```
virtual function bit do_compare (ovm_object  rhs,
                                ovm_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.

**A typical implementation is as follows**

```
class mytype extends ovm_object;
...
int f1;
virtual function bit do_compare (ovm_object rhs,ovm_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 `ovm_object`. Thus, you must `$cast` it to the type of this object before comparing.

The actual comparison should be implemented using the `ovm_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 [ovm\\_comparer](#) for more details.

## Packing

---

### pack

---

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

## pack\_bytes

---

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

## pack\_ints

---

```
function int pack_ints (ref int unsigned   intstream[],
                       input ovm_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 [ovm\\_default\\_packer](#) policy is used. See [ovm\\_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 (ovm_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 (ovm_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 ``ovm_*_field` macros are used, the above meta information is included provided the `ovm_packer`'s `<use_metadata>` variable is set.

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 ovm_packer packer      = null      )
```

### unpack\_bytes

---

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

### unpack\_ints

---

```
function int unpack_ints (ref int unsigned  intstream[],
                          input ovm_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 *ovm\_default\_packer* policy is used. See [ovm\\_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 (ovm_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 [ovm\\_packer](#) for more information.

The following implementation corresponds to the example given in `do_pack`.

```
function void do_unpack (ovm_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,
                                     ovm_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,
                                         ovm_object value,
                                         bit          clone      = 1,
                                         bit          recurse      = 1      )
```

These methods provide write access to integral, string, and ovm\_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 [ovm\\_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 ovm_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, ovm_bitstream_t value);
    if (ovm_is_match (field_name, "myint"))
        myint = value;
    else if (ovm_is_match (field_name, "mybyte"))
        mybyte = value;
endfunction

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

// provide access to sub-objects
function void set_object_local(string field_name, ovm_object value,
                               bit clone=1);
    if (ovm_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
                myobj = clone ? tmp.clone() : tmp;
        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.

# ovm\_transaction

The ovm\_transaction class is the root base class for OVM transactions. Inheriting all the methods of ovm\_object, ovm\_transaction adds a timing and recording interface.

## Summary

### ovm\_transaction

The ovm\_transaction class is the root base class for OVM transactions.

#### Class Hierarchy

ovm\_object

**ovm\_transaction**

#### Class Declaration

```
virtual class ovm_transaction extends ovm_object
```

#### Methods

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



## Methods

---

### new

---

```
function new (string      name      = "",
              ovm_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 `ovm_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 `ovm_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 `ovm_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 ovm\_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 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 ovm_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. See `ovm_event_pool` for more information.

## set\_initiator

---

```
function void set_initiator (ovm_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 ovm_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 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.

# ovm\_component

The `ovm_component` class is the root base class for OVM components. In addition to the features inherited from `ovm_object` and `ovm_report_object`, `ovm_component` provides the following interfaces:

<i>Hierarchy</i>	provides methods for searching and traversing the component hierarchy.
<i>Configuration</i>	provides methods for configuring component topology and other parameters ahead of and during component construction.
<i>Phasing</i>	defines a phased test flow that all components follow. Derived components implement one or more of the predefined phase callback methods to perform their function. During simulation, all components' callbacks are executed in precise order. Phasing is controlled by <code>ovm_top</code> , the singleton instance of <code>ovm_root</code> .
<i>Reporting</i>	provides a convenience interface to the <code>ovm_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>ovm_factory</code> . The factory is used to create new components and other objects based on type-wide and instance-specific configuration.

The `ovm_component` is automatically seeded during construction using OVM seeding, if enabled. All other objects must be manually reseeded, if appropriate. See `ovm_object::reseed` for more information.

## Summary

### ovm\_component

The `ovm_component` class is the root base class for OVM components.

#### Class Hierarchy



#### Class Declaration

```
virtual class ovm_component extends ovm_report_object
```

<b>new</b>	Creates a new component with the given leaf instance <i>name</i> and handle to its <i>parent</i> .
<b>Hierarchy Interface</b>	These methods provide user access to information about the component hierarchy, i.e., topology.
<code>get_parent</code>	Returns a handle to this component's parent, or null if it has no parent.
<code>get_full_name</code>	Returns the full hierarchical name of this object.
<code>get_child</code>	
<code>get_next_child</code>	
<code>get_first_child</code>	These methods are used to iterate through this component's children, if any.
<code>get_num_children</code>	Returns the number of this component's children.
<code>has_child</code>	Returns 1 if this component has a child with the given <i>name</i> , 0 otherwise.
<code>set_name</code>	Renames this component to <i>name</i> and recalculates all descendants' full names.
<code>lookup</code>	Looks for a component with the given hierarchical <i>name</i> relative to this component.
<b>Phasing Interface</b>	Components execute their behavior in strictly ordered, pre-defined phases.
<code>build</code>	The build phase callback is the first of several methods automatically called during the course of simulation.
<code>connect</code>	The connect phase callback is one of several methods automatically called during the course of simulation.
<code>end_of_elaboration</code>	The end_of_elaboration phase callback is one of several methods automatically called during the course of simulation.
<code>start_of_simulation</code>	The start_of_simulation phase callback is one of several methods automatically called during the course of simulation.

run	The run phase callback is the only predefined phase that is time-consuming, i.e., task-based.
extract	The extract phase callback is one of several methods automatically called during the course of simulation.
check	The check phase callback is one of several methods automatically called during the course of simulation.
report	The report phase callback is the last of several predefined phase methods automatically called during the course of simulation.
suspend	Suspends the process tree spawned from this component's currently executing task-based phase, e.g.
resume	Resumes the process tree spawned from this component's currently executing task-based phase, e.g.
status	Returns the status of the parent process associated with the currently running task-based phase, e.g., <b>run</b> .
kill	Kills the process tree associated with this component's currently running task-based phase, e.g., <b>run</b> .
do_kill_all	Recursively calls <b>kill</b> on this component and all its descendants, which abruptly ends the currently running task-based phase, e.g., <b>run</b> .
stop	The stop task is called when this component's <b>enable_stop_interrupt</b> bit is set and <b>global_stop_request</b> is called during a task-based phase, e.g., <b>run</b> .
enable_stop_interrupt	This bit allows a component to raise an objection to the stopping of the current phase.
resolve_bindings	Processes all port, export, and imp connections.
<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).
set_config_int	Calling set_config_* causes configuration settings to be created and placed in a table internal to this component.
set_config_string	
set_config_object	
get_config_int	These methods retrieve configuration settings made by previous calls to their set_config_* counterparts.
get_config_string	
get_config_object	
check_config_usage	Check all configuration settings in a components configuration table to determine if the setting has been used, overridden or not used.
apply_config_settings	Searches for all config settings matching this component's instance path.
print_config_settings	Called without arguments, print_config_settings prints all configuration information for this component, as set by previous calls to set_config_*.
print_config_matches	Setting this static variable causes get_config_* to print info about matching configuration settings as they are being applied.
<b>Objection Interface</b>	These methods provide object level hooks into the <b>ovm_objection</b> mechanism.
raised	The raised callback is called when a descendant of the component instance raises the specified <i>objection</i> .
dropped	The dropped callback is called when a descendant of the component instance raises the specified <i>objection</i> .
all_dropped	The all_dropped callback is called when a descendant of the component instance raises the specified <i>objection</i> .
<b>Factory Interface</b>	The factory interface provides convenient access to a portion of OVM's <b>ovm_factory</b> interface.
create_component	A convenience function for <b>ovm_factory::create_component_by_name</b> , this method calls upon the factory to create a new child component whose type corresponds to the preregistered type name, <i>requested_type_name</i> , and instance name, <i>name</i> .
create_object	A convenience function for <b>ovm_factory::create_object_by_name</b> , this method calls upon the factory to create a new object whose type corresponds to the preregistered type name, <i>requested_type_name</i> , and instance name, <i>name</i> .
set_type_override_by_type	A convenience function for <b>ovm_factory::set_type_override_by_type</b> , 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 <b>ovm_factory::set_inst_override_by_type</b> , this method registers a factory override for components and objects created at this level of hierarchy or below.



<code>set_type_override</code>	A convenience function for <code>ovm_factory::set_type_override_by_name</code> , this method configures the factory to create an object of type <i>override_type_name</i> whenever the factory is asked to produce a type represented by <i>original_type_name</i> .
<code>set_inst_override</code>	A convenience function for <code>ovm_factory::set_inst_override_by_type</code> , this method registers a factory override for components created at this level of hierarchy or below.
<code>print_override_info</code>	This factory debug method performs the same lookup process as <code>create_object</code> and <code>create_component</code> , but instead of creating an object, it prints information about what type of object would be created given the provided arguments.
<b>Hierarchical Reporting Interface</b>	This interface provides versions of the <code>set_report_*</code> methods in the <code>ovm_report_object</code> base class that are applied recursively to this component and all its children.
<code>set_report_severity_action_hier</code> <code>set_report_id_action_hier</code> <code>set_report_severity_id_action_hier</code>	These methods recursively associate the specified action with reports of the given <i>severity</i> , <i>id</i> , or <i>severity-id</i> pair.
<code>set_report_default_file_hier</code> <code>set_report_severity_file_hier</code> <code>set_report_id_file_hier</code> <code>set_report_severity_id_file_hier</code>	These methods recursively associate the specified FILE descriptor with reports of the given <i>severity</i> , <i>id</i> , or <i>severity-id</i> pair.
<code>set_report_verbosity_level_hier</code>	This method recursively sets the maximum verbosity level for reports for this component and all those below it.
<b>Recording Interface</b>	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.

## new

```
function new (string      name,
              ovm_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, *ovm\_top*.

All classes derived from `ovm_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 ovm_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 [ovm\\_object::get\\_name](#).

### [get\\_child](#)

---

```
function ovm_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 [ovm\\_object::print](#) for each child:

```
string name;
ovm_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 ovm_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.

## Phasing Interface

---

Components execute their behavior in strictly ordered, pre-defined phases. Each phase is defined by its own method, which derived components can override to incorporate component-specific behavior. During simulation, the phases are executed one by one, where one phase must complete before the next phase begins. The following briefly describe each phase:

<i>new</i>	Also known as the <i>constructor</i> , the component does basic initialization of any members not subject to configuration.
<i>build</i>	The component constructs its children. It uses the <code>get_config</code> interface to obtain any configuration for itself, the <code>set_config</code> interface to set any configuration for its own children, and the factory interface for actually creating the children and other objects it might need.
<i>connect</i>	The component now makes connections (binds TLM ports and exports) from child-to-child or from child-to-self (i.e. to promote a child port or export up the hierarchy for external access. Afterward, all connections are checked via <a href="#">resolve_bindings</a> before entering the <a href="#">end_of_elaboration</a> phase.
<i>end_of_elaboration</i>	At this point, the entire testbench environment has been built and connected. No new components and connections may be created from this point forward. Components can do final checks for proper connectivity, and it can initiate communication with other tools that require stable, quasi-static component structure..
<i>start_of_simulation</i>	The simulation is about to begin, and this phase can be used to perform any pre-run activity such as displaying banners, printing final testbench topology and configuration information.
<i>run</i>	This is where verification takes place. It is the only predefined, time-consuming phase. A component's primary function is implemented in the <a href="#">run</a> task. Other processes may be forked if desired. When a component returns from its run task, it does not signify completion of its run phase. Any processes that it may have forked <i>continue to run</i> . The run phase terminates in one of four ways:

<i>stop</i>	When a component's <a href="#">enable_stop_interrupt</a> bit is set and <a href="#">global_stop_request</a> is called, the component's <a href="#">stop</a> task is called. Components can implement stop to allow completion of in-progress transactions, <flush> queues, etc. Upon return from stop() by all enabled components, a <a href="#">do_kill_all</a> is issued. If the <a href="#">ovm_test_done_objection</a> is being used, this stopping procedure is deferred until all outstanding objections on <a href="#">ovm_test_done</a> have been dropped.
<i>objections dropped</i>	The <a href="#">ovm_test_done_objection</a> will implicitly call <a href="#">global_stop_request</a> when all objections to ending the phase are dropped. The stop procedure described above is then allowed to proceed normally.
<i>kill</i>	When called, all component's <a href="#">run</a> processes are killed immediately. While kill can be called directly, it is recommended that components use the stopping mechanism, which affords a more ordered and safe shut-down.
<i>timeout</i>	If a timeout was set, then the phase ends if it expires before either of the above occur. Without a stop, kill, or timeout, simulation can continue "forever", or the simulator may end simulation prematurely if it determines that all processes are waiting.
<i>extract</i>	This phase can be used to extract simulation results from coverage collectors and scoreboards, collect status/error counts, statistics, and other information from components in bottom-up order. Being a separate phase, extract ensures all relevant data from potentially independent sources (i.e. other components) are collected before being checked in the next phase.
<i>check</i>	Having extracted vital simulation results in the previous phase, the check phase can be used to validate such data and determine the overall simulation outcome. It too executes bottom-up.
<i>report</i>	Finally, the report phase is used to output results to files and/or the screen.

All task-based phases ([run](#) is the only pre-defined task phase) will run forever until killed or stopped via [kill](#) or [global\\_stop\\_request](#). The latter causes each component's [stop](#) task to get called back if its [enable\\_stop\\_interrupt](#) bit is set. After all components' stop tasks return, the OVM will end the phase.

Note- the [post\\_new](#), [export\\_connections](#), [import\\_connections](#), [configure](#), and [pre\\_run](#) phases are deprecated. [build](#) replaces [post\\_new](#), [connect](#) replaces both [import\\_](#) and [export\\_connections](#), and [start\\_of\\_simulation](#) replaces [pre\\_run](#).

## build

```
virtual function void build ()
```

The build phase callback is the first of several methods automatically called during the course of simulation. The build phase is the second of a two-pass construction process (the first is the built-in new method).

The build phase can add additional hierarchy based on configuration information not available at time of initial construction. Any override should call [super.build\(\)](#).

Starting after the initial construction phase ([new](#) method) has completed, the build phase consists of calling all components' build methods recursively top-down, i.e., parents' build are executed before the children. This is the only phase that executes top-down.

The build phase of the [ovm\\_component](#) class executes 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\(\)](#) in the subtype's build method.

See [ovm\\_phase](#) for more information on phases.

## connect

---

```
virtual function void connect ()
```

The connect phase callback is one of several methods automatically called during the course of simulation.

Starting after the [build](#) phase has completed, the connect phase consists of calling all components' connect methods recursively in depth-first, bottom-up order, i.e., children are executed before their parents.

Generally, derived classes should override this method to make port and export connections via the similarly-named [ovm\\_port\\_base #\(IF\)::connect](#) method. Any override should call `super.connect()`.

This method should never be called directly.

See [ovm\\_phase](#) for more information on phases.

---

## end\_of\_elaboration

```
virtual function void end_of_elaboration ()
```

The end\_of\_elaboration phase callback is one of several methods automatically called during the course of simulation.

Starting after the [connect](#) phase has completed, this phase consists of calling all components' end\_of\_elaboration methods recursively in depth-first, bottom-up order, i.e., children are executed before their parents.

Generally, derived classes should override this method to perform any checks on the elaborated hierarchy before the simulation phases begin. Any override should call `super.end_of_elaboration()`.

This method should never be called directly.

See [ovm\\_phase](#) for more information on phases.

---

## start\_of\_simulation

```
virtual function void start_of_simulation ()
```

The start\_of\_simulation phase callback is one of several methods automatically called during the course of simulation.

Starting after the [end\\_of\\_elaboration](#) phase has completed, this phase consists of calling all components' start\_of\_simulation methods recursively in depth-first, bottom-up order, i.e. children are executed before their parents.

Generally, derived classes should override this method to perform component- specific pre-run operations, such as discovery of the elaborated hierarchy, printing banners, etc. Any override should call `super.start_of_simulation()`.

This method should never be called directly.

See [ovm\\_phase](#) for more information on phases.

---

```
virtual task run ()
```

The run phase callback is the only predefined phase that is time-consuming, i.e., task-based. It executes after the [start\\_of\\_simulation](#) phase has completed. Derived classes should override this method to perform the bulk of its functionality, forking additional processes if needed.

In the run phase, all components' run tasks are forked as independent processes. Returning from its run task does not signify completion of a component's run phase; any processes forked by run continue to run.

The run phase terminates in one of four ways.

- 1explicit call to [global\\_stop\\_request](#) - When [global\\_stop\\_request](#) is called, an ordered shut-down for the currently running phase begins. First, all enabled components' status tasks are called bottom-up, i.e., childrens' [stop](#) tasks are called before the parent's. A component is enabled by its `enable_stop_interrupt` bit. Each component can implement `stop` to allow completion of in-progress transactions, flush queues, and other shut-down activities. Upon return from `stop` by all enabled components, the recursive `do_kill_all` is called on all top-level component(s). If the [ovm\\_test\\_done](#) objection> is being used, this stopping procedure is deferred until all outstanding objections on [ovm\\_test\\_done](#) have been dropped.
- 2all objections to [ovm\\_test\\_done](#) have been dropped - When all objections on the [ovm\\_test\\_done](#) objection have been dropped, [global\\_stop\\_request](#) is called automatically, thus kicking off the stopping procedure described above. See [ovm\\_objection](#) for details on using the objection mechanism.
- 3explicit call to [kill](#) or [do\\_kill\\_all](#) - When [kill](#) is called, this component's run processes are killed immediately. The [do\\_kill\\_all](#) methods applies to this 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.
- 4timeout - The phase ends if the timeout expires before an explicit call to [global\\_stop\\_request](#) or [kill](#). By default, the timeout is set to near the maximum simulation time possible. You may override this via [set\\_global\\_timeout](#), but you cannot disable the timeout completely.

If the default 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](#).

The run task should never be called directly.

See [ovm\\_phase](#) for more information on phases.

---

## extract

```
virtual function void extract ()
```

The extract phase callback is one of several methods automatically called during the course of simulation.

Starting after the [run](#) phase has completed, the extract phase consists of calling all components' extract methods recursively in depth-first, bottom-up order, i.e., children are executed before their parents.

Generally, derived classes should override this method to collect information for the subsequent [check](#) phase when such information needs to be collected in a hierarchical, bottom-up manner. Any override should call `super.extract()`.

This method should never be called directly.

See [ovm\\_phase](#) for more information on phases.

## check

---

```
virtual function void check ()
```

The check phase callback is one of several methods automatically called during the course of simulation.

Starting after the [extract](#) phase has completed, the check phase consists of calling all components' check methods recursively in depth-first, bottom-up order, i.e., children are executed before their parents.

Generally, derived classes should override this method to perform component specific, end-of-test checks. Any override should call `super.check()`.

This method should never be called directly.

See [ovm\\_phase](#) for more information on phases.

## report

---

```
virtual function void report ()
```

The report phase callback is the last of several predefined phase methods automatically called during the course of simulation.

Starting after the [check](#) phase has completed, the report phase consists of calling all components' report methods recursively in depth-first, bottom-up order, i.e., children are executed before their parents.

Generally, derived classes should override this method to perform component-specific reporting of test results. Any override should call `super.report()`.

This method should never be called directly.

See [ovm\\_phase](#) for more information on phases.

## suspend

---

```
virtual task suspend ()
```

Suspends the process tree spawned from this component's currently executing task-based phase, e. g. [run](#).

## resume

---

```
virtual task resume ()
```

Resumes the process tree spawned from this component's currently executing task-based phase, e. g. [run](#).

## status

---

```
function string status ()
```

Returns the status of the parent process associated with the currently running task-based phase, e.g., [run](#).

---

## 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 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](#) 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

```
protected 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,
                                     ovm_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,
                                         ovm_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 `ovm_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 `ovm_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 ovm 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 ovm_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 ovm_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 `ovm_object`, you must provide an `ovm_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 `ovm_object`.

```
class mycomponent extends ovm_component;

    local myobj_t data;

    function void build();
        ovm_object tmp;
        super.build();
        if(get_config_object("data", tmp))
            if (!$cast(data, tmp))
                $display("error! config setting for 'data' not of type myobj_t");
        endfunction
    ...
```

The above example overrides the `build` method. If you want to retain any base functionality, you must call `super.build()`.

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();
    set_report_id_action_hier(CFGOVR, OVM_DISPLAY);
    set_report_id_action_hier(CFGSET, OVM_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 `ovm_component::build`.

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 ``ovm_*_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()` method should be overloaded and you should not call `super.build()`. If this case, you must also set the `m_build_done` bit. 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   = "",
                                   ovm_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.

### `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 `ovm_objection` mechanism.

### `raised`

---

```
virtual function void raised (ovm_objection objection,
                             ovm_object    source_obj,
                             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 (ovm_objection objection,
                              ovm_object    source_obj,
                              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 (ovm_objection objection,
                          ovm_object    source_obj,
                          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 OVM's [ovm\\_factory](#) interface. For creating new objects and components, the preferred method of accessing the factory is via the object or component wrapper (see [ovm\\_component\\_registry #\(T,Tname\)](#) and [ovm\\_object\\_registry #\(T,Tname\)](#)). The wrapper also provides functions for setting type and instance overrides.

## create\_component

---

```
function ovm_component create_component (string requested_type_name,
                                         string name                )
```

A convenience function for [ovm\\_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 [ovm\\_factory](#) for details on factory operation.

## create\_object

```
function ovm_object create_object (string requested_type_name,
                                   string name                    = "" )
```

A convenience function for [ovm\\_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 [ovm\\_factory](#) for details on factory operation.

## set\_type\_override\_by\_type

```
static function void set_type_override_by_type (
    ovm_object_wrapper original_type,
    ovm_object_wrapper override_type,
    bit                  replace      = 1
)
```

A convenience function for [ovm\\_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 [ovm\\_factory::create\\_object\\_by\\_type](#) or [ovm\\_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,
                                         ovm_object_wrapper original_type,
                                         ovm_object_wrapper override_type )
```

A convenience function for [ovm\\_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 [ovm\\_factory::create\\_object\\_by\\_type](#) or [ovm\\_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, or by directly calling `type::type_id::get()`, where `type` is the user type and `type_id` is the name of the typedef to [ovm\\_object\\_registry #\(T, Tname\)](#) or [ovm\\_component\\_registry #\(T, Tname\)](#).

If you are employing the ``ovm_*_utils` macros, the typedef and the `get_type` method will be implemented for you.

The following example shows ``ovm_*_utils` usage

```
class comp extends ovm_component;
  `ovm_component_utils(comp)
  ...
endclass

class mycomp extends ovm_component;
  `ovm_component_utils(mycomp)
  ...
endclass

class block extends ovm_component;
  `ovm_component_utils(block)
  comp c_inst;
  virtual function void build();
    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 [ovm\\_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 `ovm_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 `ovm_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\_severity\_action\_hier

---

```
function void set_report_severity_action_hier (ovm_severity severity,
                                                ovm_action action )
```

### set\_report\_id\_action\_hier

---



```
function void set_report_id_action_hier (string    id,
                                       ovm_action action)
```

### [set\\_report\\_severity\\_id\\_action\\_hier](#)

---

```
function void set_report_severity_id_action_hier(ovm_severity severity,
                                                string      id,
                                                ovm_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 action associated with a severity.

For a list of severities and their default actions, refer to [ovm\\_report\\_handler](#).

### [set\\_report\\_default\\_file\\_hier](#)

---

```
function void set_report_default_file_hier (OVM_FILE file)
```

### [set\\_report\\_severity\\_file\\_hier](#)

---

```
function void set_report_severity_file_hier (ovm_severity severity,
                                             OVM_FILE      file      )
```

### [set\\_report\\_id\\_file\\_hier](#)

---

```
function void set_report_id_file_hier (string  id,
                                       OVM_FILE file)
```

### [set\\_report\\_severity\\_id\\_file\\_hier](#)

---

```
function void set_report_severity_id_file_hier(ovm_severity severity,
                                                string      id,
                                                OVM_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 [ovm\\_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 [ovm\\_report\\_handler](#) for a list of predefined message verbosity levels and their meaning.

## 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 (ovm_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 [ovm\\_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 (ovm_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 (ovm_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 [ovm\\_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 != OVM_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 (ovm_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 (ovm_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 (ovm_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 `ovm_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 ``ovm_*_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 (ovm_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",
                                ovm_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 `ovm_object`, *info*, as implemented in its `<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",
                                ovm_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

---

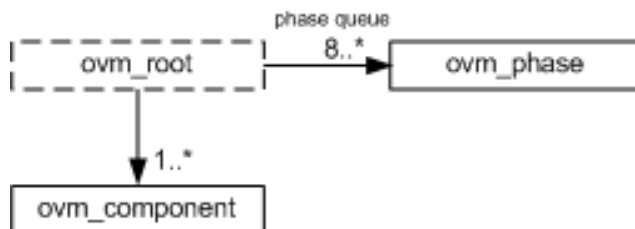
```
bit print_enabled = 1
```

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.

## ovm\_root

The *ovm\_root* class serves as the implicit top-level and phase controller for all OVM components. Users do not directly instantiate *ovm\_root*. The OVM automatically creates a single instance of *ovm\_root* that users can access via the global (ovm\_pkg-scope) variable, *ovm\_top*.



The *ovm\_top* instance of *ovm\_root* plays several key roles in the OVM.

- Implicit top-level*     The *ovm\_top* serves as an implicit top-level component. Any component whose parent is specified as NULL becomes a child of *ovm\_top*. Thus, all OVM components in simulation are descendants of *ovm\_top*.
- Phase control*        *ovm\_top* manages the phasing for all components. There are eight phases predefined in every component: build, connect, end\_of\_elaboration, start\_of\_simulation, run, extract, check, and report. Of these, only the run phase is a task. All others are functions. OVM's flexible phasing mechanism allows users to insert any number of custom function and task-based phases. See [run\\_test](#), [insert\\_phase](#), and [stop\\_request](#), and others.
- Search*                Use *ovm\_top* to search for components based on their hierarchical name. See [find](#) and [find\\_all](#).
- Report configuration* Use *ovm\_top* to globally configure report verbosity, log files, and actions. For example, *ovm\_top.set\_report\_verbosity\_level\_hier(OVM\_FULL)* would set full verbosity for all components in simulation.
- Global reporter*      Because *ovm\_top* is globally accessible (in ovm\_pkg scope), OVM's reporting mechanism is accessible from anywhere outside *ovm\_component*, such as in modules and sequences. See [ovm\\_report\\_error](#), [ovm\\_report\\_warning](#), and other global methods.

## Summary

### ovm\_root

The *ovm\_root* class serves as the implicit top-level and phase controller for all OVM components.

#### Class Hierarchy

ovm_object
ovm_report_object
ovm_component
<b>ovm_root</b>

## Class Declaration

```
class ovm_root extends ovm_component
```

## Methods

<a href="#">run_test</a>	Phases all components through all registered phases.
<a href="#">stop_request</a>	Calling this function triggers the process of shutting down the currently running task-based phase.
<a href="#">in_stop_request</a>	This function returns 1 if a stop request is currently active, and 0 otherwise.
<a href="#">insert_phase</a>	Inserts a new phase given by <code>new_phase</code> <u>after</u> the existing phase given by <code>exist_phase</code> .
<a href="#">find</a>	
<a href="#">find_all</a>	Returns the component handle ( <code>find</code> ) or list of components handles ( <code>find_all</code> ) matching a given string.
<a href="#">get_current_phase</a>	Returns the handle of the currently executing phase.
<a href="#">get_phase_by_name</a>	Returns the handle of the phase having the given <i>name</i> .

## Variables

<a href="#">phase_timeout</a>	
<a href="#">stop_timeout</a>	These set watchdog timers for task-based phases and stop tasks.
<a href="#">enable_print_topology</a>	If set, then the entire testbench topology is printed just after completion of the <code>end_of_elaboration</code> phase.
<a href="#">finish_on_completion</a>	If set, then <code>run_test</code> will call <code>\$finish</code> after all phases are executed.
<a href="#">ovm_top</a>	This is the top-level that governs phase execution and provides component search interface.

## Methods

[raised](#)  
[all\\_dropped](#)

# 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, `+OVM_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.

## 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 ovm\_top will then begin execution of the next phase, if any.

## in\_stop\_request

---

```
function bit in_stop_request()
```

This function returns 1 if a stop request is currently active, and 0 otherwise.

## insert\_phase

---

```
function void insert_phase (ovm_phase new_phase,
                           ovm_phase exist_phase)
```

Inserts a new phase given by new\_phase after the existing phase given by exist\_phase. The ovm\_top maintains a queue of phases executed in consecutive order. If exist\_phase is null, then new\_phase is inserted at the head of the queue, i.e., it becomes the first phase.

## find

---

```
function ovm_component find (string comp_match)
```

## find\_all

---

```
function void find_all (      string      comp_match,
                           ref ovm_component comps[$],
                           input ovm_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).



## get\_current\_phase

---

```
function ovm_phase get_current_phase ()
```

Returns the handle of the currently executing phase.

## get\_phase\_by\_name

---

```
function ovm_phase get_phase_by_name (string name)
```

Returns the handle of the phase having the given *name*.

# Variables

---

## phase\_timeout

---

```
time phase_timeout = 0
```

## 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.

## 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.

## ovm\_top

---

```
`const ovm_root ovm_top = ovm_root::get()
```

This is the top-level that governs phase execution and provides component search interface. See [ovm\\_root](#) for more information.

## Methods

---

### raised

---

```
function void ovm_root::raised (ovm_objection objection,
                                ovm_object    source_obj,
                                int            count     )
```

### all\_dropped

---

```
task ovm_root::all_dropped (ovm_objection objection,
                             ovm_object    source_obj,
                             int            count     )
```

# ovm\_phase

The ovm\_phase class is used for defining phases for ovm\_component and its subclasses. For a list of predefined phases see [ovm\\_component::Phasing Interface](#)

## Summary

### ovm\_phase

The ovm\_phase class is used for defining phases for ovm\_component and its subclasses.

#### Class Declaration

```
virtual class ovm_phase
```

#### Methods

<a href="#">new</a>	Creates a phase object.
<a href="#">get_name</a>	Returns the name of the phase object as supplied in the constructor.
<a href="#">is_task</a>	Returns 1 if the phase is time consuming and 0 if not.
<a href="#">is_top_down</a>	Returns 1 if the phase executes top-down (executes the parent's phase callback before executing the children's callback) and 0 otherwise.
<a href="#">get_type_name</a>	Derived classes should override this method to return the phase type name.
<a href="#">wait_start</a>	Waits until the phase has been started.
<a href="#">wait_done</a>	Waits until the phase has been completed.
<a href="#">is_in_progress</a>	Returns 1 if the phase is currently in progress (active), 0 otherwise.
<a href="#">is_done</a>	Returns 1 if the phase has completed, 0 otherwise.
<a href="#">reset</a>	Resets phase state such that is_done and is_in_progress both return 0.
<a href="#">call_task</a>	Calls the task-based phase of the component given by parent, which must be derived from ovm_component.
<a href="#">call_func</a>	Calls the function-based phase of the component given by parent.

## Methods

### new

```
function new (string name,
             bit    is_top_down,
             bit    is_task    )
```

Creates a phase object.

The name is the name of the phase. When is\_top\_down is set, the parent is phased before its children. is\_task indicates whether the phase callback is a task (1) or function (0). Only tasks may consume simulation time and execute blocking statements.

## get\_name

---

```
function string get_name ()
```

Returns the name of the phase object as supplied in the constructor.

## is\_task

---

```
function bit is_task ()
```

Returns 1 if the phase is time consuming and 0 if not.

## is\_top\_down

---

```
function bit is_top_down ()
```

Returns 1 if the phase executes top-down (executes the parent's phase callback before executing the children's callback) and 0 otherwise.

## get\_type\_name

---

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

Derived classes should override this method to return the phase type name.

## wait\_start

---

```
task wait_start ()
```

Waits until the phase has been started.

## wait\_done

---

```
task wait_done ()
```

Waits until the phase has been completed.

## is\_in\_progress

---

```
function bit is_in_progress ()
```

Returns 1 if the phase is currently in progress (active), 0 otherwise.

## is\_done

---

```
function bit is_done ()
```

Returns 1 if the phase has completed, 0 otherwise.

## reset

---

```
function void reset ()
```

Resets phase state such that is\_done and is\_in\_progress both return 0.

## call\_task

---

```
virtual task call_task (ovm_component parent)
```

Calls the task-based phase of the component given by parent, which must be derived from ovm\_component. A task-based phase is defined by subtyping ovm\_phase and overriding this method. The override must \$cast the base parent handle to the actual component type that defines the phase callback, and then call the phase callback.

## call\_func

---

```
virtual function void call_func (ovm_component parent)
```

Calls the function-based phase of the component given by parent. A function-based phase is defined by subtyping ovm\_phase and overriding this method. The override must \$cast the base parent handle to the actual component type that defines the phase callback, and then call that phase callback.

## Usage

---

Phases are a synchronizing mechanism for the environment. They are represented by callback methods. A set of predefined phases and corresponding callbacks are provided in ovm\_component. Any class deriving from ovm\_component may implement any or all of these callbacks, which are executed in a particular order. Depending on the properties of any given

phase, the corresponding callback is either a function or task, and it is executed in top-down or bottom-up order.

The OVM provides the following predefined phases for all `ovm_component`s.

<i>build</i>	Depending on configuration and factory settings, create and configure additional component hierarchies.
<i>connect</i>	Connect ports, exports, and implementations (imps).
<i>end_of_elaboration</i>	Perform final configuration, topology, connection, and other integrity checks.
<i>start_of_simulation</i>	Do pre-run activities such as printing banners, pre-loading memories, etc.
<i>run</i>	Most verification is done in this time-consuming phase. May fork other processes. Phase ends when <code>global_stop_request</code> is called explicitly.
<i>extract</i>	Collect information from the run in preparation for checking.
<i>check</i>	Check simulation results against expected outcome.
<i>report</i>	Report simulation results.

A phase is defined by an instance of an *ovm\_phase* subtype. If a phase is to be shared among several component types, the instance must be accessible from a common scope, such as a package.

To have a user-defined phase get called back during simulation, the phase object must be registered with the top-level OVM phase controller, `ovm_top`.

### Inheriting from the *ovm\_phase* Class

When creating a user-defined phase, you must do the following.

1. Define a new phase class, which must extend *ovm\_phase*. To enable use of the phase by any component, we recommend this class be parameterized. The easiest way to define a new phase is to invoke a predefined macro. For example:

```
`ovm_phase_func_topdown_decl( preload )
```

This convenient phase declaration macro is described below.

2. Create a single instance of the phase in a convenient place in a package, or in the same scope as the component classes that will use the phase.

```
typedef class my_memory;
preload_phase #(my_memory) preload_ph = new;
```

3. Register the phase object with `ovm_top`.

```
class my_memory extends ovm_component;
function new(string name, ovm_component parent);
    super.new(name, parent);
```

```

    ovm_top.insert_phase(preload_ph, start_of_simulation_ph);
endfunction
virtual function void preload(); // our new phase
...
endfunction
endclass

```

## Phase Macros (Optional)

The following macros simplify the process of creating a user-defined phase. They create a phase type that is parameterized to the component class that uses the phase.

## Summary

<b>Usage</b>	Phases are a synchronizing mechanism for the environment.
<b>Macros</b>	
<code>`ovm_phase_func_decl</code>	<code>`ovm_phase_func_decl (PHASE_NAME, TOP_DOWN)</code>
<code>`ovm_phase_task_decl</code>	
<code>`ovm_phase_func_topdown_decl</code>	
<code>`ovm_phase_func_bottomup_decl</code>	
<code>`ovm_phase_task_topdown_decl</code>	
<code>`ovm_phase_task_bottomup_decl</code>	These alternative macros have a single phase name argument.

## Macros

### ``ovm_phase_func_decl`

``ovm_phase_func_decl (PHASE_NAME, TOP_DOWN)`

The *PHASE\_NAME* argument is used to define the name of the phase, the name of the component method that is called back during phase execution, and the prefix of the type-name of the phase class that gets generated.

The above macro creates the following class definition.

```

class PHASE_NAME`_phase #(type PARENT=int) extends ovm_phase;

    PARENT m_parent;

    function new();
        super.new(`"NAME`",TOP_DOWN,1);
    endfunction
    virtual function void call_func();
        m_parent.NAME(); // call the component's phase callback
    endtask

```

```

    virtual task execute(ovm_component parent);
        assert($cast(m_parent,parent));
        call_func();
    endtask
endclass

```

## ``ovm_phase_task_decl`

---

```
`ovm_phase_task_decl (PHASE_NAME, TOP_DOWN)
```

The above macro creates the following class definition.

```

class PHASE_NAME`_phase #(type PARENT=int) extends ovm_phase;
    PARENT m_parent;
    function new();
        super.new(`"NAME`",TOP_DOWN,1);
    endfunction
    virtual task call_task();
        m_parent.NAME(); // call the component's phase callback
    endtask
    virtual task execute(ovm_component parent);
        assert($cast(m_parent,parent));
        call_task();
    endtask
endclass

```

## ``ovm_phase_func_topdown_decl`

---

## ``ovm_phase_func_bottomup_decl`

---

## ``ovm_phase_task_topdown_decl`

---

## ``ovm_phase_task_bottomup_decl`

---

These alternative macros have a single phase name argument. The top-down or bottom-up selection is specified in the macro name, which makes them more self-documenting than those with a 0 or 1 2nd argument.



```
`define ovm_phase_func_topdown_decl `ovm_phase_func_decl (PHASE_NAME,1)
`define ovm_phase_func_bottomup_decl `ovm_phase_func_decl (PHASE_NAME,0)
`define ovm_phase_task_topdown_decl `ovm_phase_task_decl (PHASE_NAME,1)
`define ovm_phase_task_bottomup_decl `ovm_phase_task_decl (PHASE_NAME,0)
```

## ovm\_port\_base #(IF)

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

The ovm\_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 ovm\_port\_base.

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

The OVM 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 `tlm_if_base #(T1,T2)`.

Just before `ovm_component::end_of_elaboration`, an internal `ovm_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.

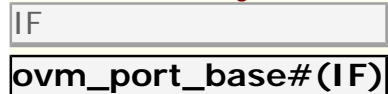
ovm\_port\_base possesses the properties of components in that they have a hierarchical instance path and parent. Because SystemVerilog does not support multiple inheritance, ovm\_port\_base can not extend both the interface it implements and `ovm_component`. Thus, ovm\_port\_base contains a local instance of ovm\_component, to which it delegates such commands as `get_name`, `get_full_name`, and `get_parent`.

### Summary

## ovm\_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 ovm_port_base #(
    type    IF  =  ovm_void
) extends IF
  
```

### Methods

<code>new</code>	The first two arguments are the normal <code>ovm_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.

<a href="#">min_size</a>	Returns the minimum number of implementation ports that must be connected to this port by the end_of_elaboration phase.
<a href="#">max_size</a>	Returns the maximum number of implementation ports that must be connected to this port by the end_of_elaboration phase.
<a href="#">is_unbounded</a>	Returns 1 if this port has no maximum on the number of implementation (imp) ports this port can connect to.
<a href="#">is_port</a>	
<a href="#">is_export</a>	
<a href="#">is_imp</a>	Returns 1 if this port is of the type given by the method name, 0 otherwise.
<a href="#">size</a>	Gets the number of implementation ports connected to this port.
<a href="#">set_default_index</a>	Sets the default implementation port to use when calling an interface method.
<a href="#">connect</a>	Connects this port to the given <i>provider</i> port.
<a href="#">debug_connected_to</a>	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).
<a href="#">debug_provided_to</a>	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).
<a href="#">resolve_bindings</a>	This callback is called just before entering the end_of_elaboration phase.
<a href="#">get_if</a>	Returns the implementation (imp) port at the given index from the array ofimps this port is connected to.

## Methods

---

### new

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

The first two arguments are the normal [ovm\\_component](#) constructor arguments.

The *port\_type* can be one of [OVM\\_PORT](#), [OVM\\_EXPORT](#), or [OVM\\_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 OVM\_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 ovm_component get_parent()
```

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

## get\_comp

---

```
virtual function ovm_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 [ovm\\_component](#). Instead, they contain an instance of `<ovm_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 "ovm\_port", "ovm\_export" or "ovm\_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 (imp) ports this port can connect to. A port is unbounded when the *max\_size* argument in the constructor is specified as OVM\_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 OVM\_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 ovm\_blocking\_put\_port #(T) is compatible with an ovm\_put\_export #(T) and ovm\_blocking\_put\_imp #(T) because the export and imp provide the interface required by the ovm\_blocking\_put\_port.
- Ports of type **OVM\_EXPORT** can only connect to other exports or imps.
- Ports of type **OVM\_IMPLEMENTATION** can not be connected, as they are bound to the component that implements the interface at time of construction.

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

### Relationships, when enabled, are checked are as follows

- If this port is an OVM\_PORT type, the *provider* can be a parent port, or a sibling export or implementation port.
- If this port is an **OVM\_EXPORT** type, the provider can be a child export or implementation port.

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

Note- the `ovm_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 ovm_port_base #(IF) get_if(int index=0)
```

Returns the implementation (imp) port at the given index from the array ofimps 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.

# ovm\_barrier\_pool

## Summary

### ovm\_barrier\_pool

#### Class Hierarchy

```
ovm_object
```

```
ovm_barrier_pool
```

#### Class Declaration

```
class ovm_barrier_pool extends ovm_object
```

#### Methods

<a href="#">new</a>	Creates a new barrier pool with the given <i>name</i> .
<a href="#">get_global_pool</a>	Returns the singleton global barrier pool.
<a href="#">get</a>	Returns the barrier with the given <i>name</i> .
<a href="#">num</a>	Returns the number of uniquely named barriers stored in the pool.
<a href="#">delete</a>	Removes the barrier with the given <i>name</i> from the pool.
<a href="#">exists</a>	Returns 1 if a barrier with the given <i>name</i> exists in the pool, 0 otherwise.
<a href="#">first</a>	Returns the name of the first barrier stored in the pool.
<a href="#">last</a>	Returns the name of the last barrier stored in the pool.
<a href="#">next</a>	Returns the name of the next barrier in the pool.
<a href="#">prev</a>	Returns the name of the previous barrier in the pool.

## Methods

### [new](#)

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

Creates a new barrier pool with the given *name*.

### [get\\_global\\_pool](#)

```
static function ovm_barrier_pool get_global_pool ()
```

Returns the singleton global barrier pool.

This allows barriers to be shared amongst components throughout the verification environment.



## get

---

```
virtual function ovm_barrier get (string name)
```

Returns the barrier with the given *name*.

If no barrier exists by that name, a new barrier is created with that name and returned.

## num

---

```
virtual function int num ()
```

Returns the number of uniquely named barriers stored in the pool.

## delete

---

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

Removes the barrier with the given *name* from the pool.

## exists

---

```
virtual function int exists (string name)
```

Returns 1 if a barrier with the given *name* exists in the pool, 0 otherwise.

## first

---

```
virtual function int first (ref string name)
```

Returns the name of the first barrier stored in the pool.

If the pool is empty, then *name* is unchanged and 0 is returned.

If the pool is not empty, then *name* is name of the first barrier and 1 is returned.

## last

---

```
virtual function int last (ref string name)
```

Returns the name of the last barrier stored in the pool.

If the pool is empty, then 0 is returned and *name* is unchanged.

If the pool is not empty, then *name* is set to the last name in the pool and 1 is returned.

## next

---

```
virtual function int next (ref string name)
```

Returns the name of the next barrier in the pool.

If the input *name* is the last name in the pool, then *name* is left unchanged and 0 is returned.

If a next name is found, then *name* is updated with that name and 1 is returned.

## prev

---

```
virtual function int prev (ref string name)
```

Returns the name of the previous barrier in the pool.

If the input *name* is the first name in the pool, then *name* is left unchanged and 0 is returned.

If a previous name is found, then *name* is updated with that name and 1 is returned.

# ovm\_event\_pool

The ovm\_event\_pool is essentially an associative array of [ovm\\_event](#) objects indexed by the string name of the event.

## Summary

### ovm\_event\_pool

The ovm\_event\_pool is essentially an associative array of [ovm\\_event](#) objects indexed by the string name of the event.

#### Class Hierarchy

```
ovm_object
```

```
ovm_event_pool
```

#### Class Declaration

```
class ovm_event_pool extends ovm_object
```

#### Methods

<a href="#">new</a>	Creates a new event pool with the given <i>name</i> .
<a href="#">get_global_pool</a>	Returns the singleton global event pool.
<a href="#">get</a>	Returns the event with the given <i>name</i> .
<a href="#">num</a>	Returns the number of uniquely named events stored in the pool.
<a href="#">delete</a>	Removes the event with the given <i>name</i> from the pool.
<a href="#">exists</a>	Returns 1 if an event with the given <i>name</i> exists in the pool, 0 otherwise.
<a href="#">first</a>	Returns the name of the first event stored in the pool.
<a href="#">last</a>	Returns the name of the last event stored in the pool.
<a href="#">next</a>	Returns the name of the next event in the pool.
<a href="#">prev</a>	Returns the name of the previous event in the pool.

## Methods

### [new](#)

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

Creates a new event pool with the given *name*.

### [get\\_global\\_pool](#)

```
static function ovm_event_pool get_global_pool ()
```

Returns the singleton global event pool.

This allows events to be shared between components throughout the verification environment.

## get

---

```
virtual function ovm_event get (string name)
```

Returns the event with the given *name*.

If no event exists by that name, a new event is created with that name and returned.

## num

---

```
virtual function int num ()
```

Returns the number of uniquely named events stored in the pool.

## delete

---

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

Removes the event with the given *name* from the pool.

## exists

---

```
virtual function int exists (string name)
```

Returns 1 if an event with the given *name* exists in the pool, 0 otherwise.

## first

---

```
virtual function int first (ref string name)
```

Returns the name of the first event stored in the pool.

If the pool is empty, then *name* is unchanged and 0 is returned.

If the pool is not empty, then *name* is name of the first event and 1 is returned.

## last

---

```
virtual function int last (ref string name)
```

Returns the name of the last event stored in the pool.

If the pool is empty, then 0 is returned and *name* is unchanged.

If the pool is not empty, then *name* is set to the last name in the pool and 1 is returned.

## next

---

```
virtual function int next (ref string name)
```

Returns the name of the next event in the pool.

If the input *name* is the last name in the pool, then *name* is unchanged and 0 is returned.

If a next name is found, then *name* is updated with that name and 1 is returned.

## prev

---

```
virtual function int prev (ref string name)
```

Returns the name of the previous event in the pool.

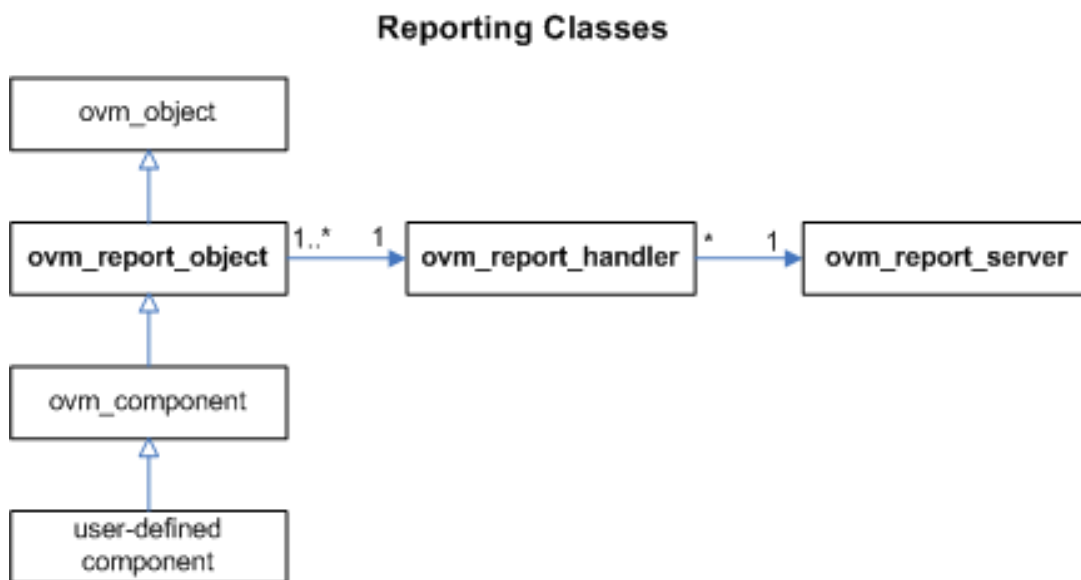
If the input *name* is the first name in the pool, then *name* is left unchanged and 0 is returned.

If a previous name is found, then *name* is updated with that name and 1 is returned.

## 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 OVM reporting facility is the [ovm\\_report\\_object](#) from which all [ovm\\_components](#) extend. The `ovm_report_object` delegates most tasks to its internal [ovm\\_report\\_handler](#). If the report handler determines the report is not filtered based the configured verbosity setting, it sends the report to the central [ovm\\_report\\_server](#) for formatting and processing.



## ovm\_report\_object

The `ovm_report_object` provides an interface to the OVM 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 `ovm_report_object` are delegated to an internal instance of an `ovm_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 `ovm_report_server`.

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

*Actions* can be set for (in increasing priority) severity, id, and (severity,id) pair. They include output to the screen `OVM_DISPLAY`, whether the message counters should be incremented `OVM_COUNT`, and whether a \$finish should occur `OVM_EXIT`.

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

<code>OVM_INFO</code>	-	<code>OVM_DISPLAY</code>	
<code>OVM_WARNING</code>	-	<code>OVM_DISPLAY</code>	
<code>OVM_ERROR</code>	-	<code>OVM_DISPLAY</code>	<code>OVM_COUNT</code>
<code>OVM_FATAL</code>	-	<code>OVM_DISPLAY</code>	<code>OVM_EXIT</code>

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

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

## Summary

### ovm\_report\_object

The `ovm_report_object` provides an interface to the OVM reporting facility.

#### Class Hierarchy

ovm\_object

**ovm\_report\_object****Class Declaration**

virtual class ovm\_report\_object extends ovm\_object

new	Creates a new report object with the given name.
-----	--

**Reporting**

ovm\_report\_info

ovm\_report\_warning

ovm\_report\_error

ovm\_report\_fatal

These are the primary reporting methods in the OVM.

**Callbacks**

report\_info\_hook

report\_error\_hook

report\_warning\_hook

report\_fatal\_hook

report\_hook

These hook methods can be defined in derived classes to perform additional actions when reports are issued.

report\_header

Prints version and copyright information.

report\_summarize

Outputs statistical information on the reports issued by the central report server.

die

This method is called by the report server if a report reaches the maximum quit count or has an OVM\_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\_severity\_action

set\_report\_id\_action

set_report_severity_id_action	These methods associate the specified action or actions with reports of the given <i>severity</i> , <i>id</i> , or <i>severity-id</i> pair.
-------------------------------	---

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*.

ovm\_report\_enabled

Returns 1 if the configured verbosity for this object is greater than *verbosity* and the action associated with the given *severity* and *id* is not OVM\_NO\_ACTION, else returns 0.

set\_report\_max\_quit\_count

Sets the maximum quit count in the report handler to *max\_count*.**Setup**

set\_report\_handler

Sets the report handler, overwriting the default instance.

get\_report\_handler

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

reset\_report\_handler

Resets the underlying report handler to its default settings.

get\_report\_server

Returns the **ovm\_report\_server** instance associated with this report object.

dump\_report\_state

This method dumps the internal state of the report handler.

new	Creates a new reporter instance with the given name.
-----	--



## new

---

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

Creates a new report object with the given name. This method also creates a new [ovm\\_report\\_handler](#) object to which most tasks are delegated.

# Reporting

---

## ovm\_report\_info

---

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

## ovm\_report\_warning

---

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

## ovm\_report\_error

---

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

## ovm\_report\_fatal

---

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

These are the primary reporting methods in the OVM. Using these instead of *\$display* and other ad hoc approaches ensures consistent output and central control over where output is directed and any actions that result. All reporting methods have the same arguments, although each has a different default verbosity:

*id* a unique id for the report or report group that can be used for identification and therefore targeted filtering. You can configure an individual report's actions and output file(s) using this id string.

*message* the message body, preformatted if necessary to a single string.

*verbosity* the verbosity of the message, indicating its relative importance. If this number is less than or equal to the effective verbosity level (see `<set_report_verbosity_level>`), then the report is issued, subject to the configured action and file descriptor settings. Verbosity is ignored for warnings, errors, and fatals to ensure users do not inadvertently filter them out. It remains in the methods for backward compatibility.

*filename/line*(Optional) The location from which the report was issued. Use the predefined macros, `__FILE__` and `__LINE__`. If specified, it is displayed in the output.

## Callbacks

---

### report\_info\_hook

---

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

### report\_error\_hook

---

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

## report\_warning\_hook

---

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

## report\_fatal\_hook

---

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

## report\_hook

---

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

These hook methods can be defined in derived classes to perform additional actions when reports are issued. They are called only if the OVM\_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(OVM_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 `ovm_root::run_test` task calls this method just before it component phasing begins.

## report\_summarize

---

```
virtual function void report_summarize(OVM_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 ovm\_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 OVM\_EXIT action associated with it, e.g., as with fatal errors.

If this report object is an [ovm\\_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\_severity\_action

---

```
function void set_report_severity_action (ovm_severity severity,
                                          ovm_action   action   )
```

## set\_report\_id\_action

---

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

## set\_report\_severity\_id\_action

---

```
function void set_report_severity_id_action (ovm_severity severity,
                                             string          id,
                                             ovm_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 take precedence over an action associated with a *severity*.

The *action* argument can take the value [OVM\\_NO\\_ACTION](#), or it can be a bitwise OR of any combination of [OVM\\_DISPLAY](#), [OVM\\_LOG](#), [OVM\\_COUNT](#), [<OVM\\_STOP>](#), [OVM\\_EXIT](#), and [OVM\\_CALL\\_HOOK](#).

## set\_report\_default\_file

---

```
function void set_report_default_file (OVM_FILE file)
```

## set\_report\_severity\_file

---

```
function void set_report_severity_file (ovm_severity severity,
                                       OVM_FILE      file      )
```

## set\_report\_id\_file

---

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

## set\_report\_severity\_id\_file

---

```
function void set_report_severity_id_file (ovm_severity severity,
                                           string          id,
                                           OVM_FILE      file      )
```

These methods configure the report handler to direct some or all of its output to the given file

descriptor. The *file* argument must be a multi-channel descriptor (mcd) or file id compatible with \$fdisplay.

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

When a report is issued and its associated action has the OVM\_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()
```

Gets the verbosity level in effect for this object. Reports issued with verbosity greater than this will be filtered out.

### get\_report\_action

---

```
function int get_report_action(ovm_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(ovm_severity severity,
                                     string          id       )
```

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

### ovm\_report\_enabled

---

```
function int ovm_report_enabled(int          verbosity,
                                ovm_severity severity = OVM_INFO,
                                string         id      = " ")
```

Returns 1 if the configured verbosity for this object is greater than *verbosity* and the action associated with the given *severity* and *id* is not OVM\_NO\_ACTION, else returns 0.

See also [get\\_report\\_verbosity\\_level](#) and [get\\_report\\_action](#), and the global version of [ovm\\_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 OVM\_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 OVM\_COUNT reports.

## Setup

---

### [set\\_report\\_handler](#)

---

```
function void set_report_handler(ovm_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 ovm_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 ovm_report_server get_report_server()
```

Returns the [ovm\\_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.

## new

---

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

Creates a new reporter instance with the given name.



# ovm\_report\_handler

The `ovm_report_handler` is the class to which most methods in `ovm_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 `ovm_report_object` for information on the OVM reporting mechanism.

The relationship between `ovm_report_object` (a base class for `ovm_component`) and `ovm_report_handler` is typically one to one, but it can be many to one if several `ovm_report_objects` are configured to use the same `ovm_report_handler_object`. See `ovm_report_object::set_report_handler`.

The relationship between `ovm_report_handler` and `ovm_report_server` is many to one.

## Summary

### ovm\_report\_handler

The `ovm_report_handler` is the class to which most methods in `ovm_report_object` delegate.

#### Class Declaration

```
class ovm_report_handler
```

#### Methods

<code>new</code>	Creates and initializes a new <code>ovm_report_handler</code> object.
<code>run_hooks</code>	The <code>run_hooks</code> method is called if the <code>OVM_CALL_HOOK</code> action is set for a report.
<code>get_verbosity_level</code>	Returns the configured maximum verbosity level.
<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>ovm_report_error</code> ) in <code>ovm_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 `ovm_report_handler` object.

### run\_hooks

---

```
virtual function bit run_hooks(ovm_report_object client,
                              ovm_severity      severity,
                              string             id,
                              string             message,
                              int                verbosity,
                              string             filename,
                              int                line      )
```

The `run_hooks` method is called if the [OVM\\_CALL\\_HOOK](#) action is set for a report. It first calls the client's `<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()
```

Returns the configured maximum verbosity level.

---

### [get\\_action](#)

```
function ovm_action get_action(ovm_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 OVM_FILE get_file_handle(ovm_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(ovm_severity    severity,  
                             string         name,  
                             string         id,  
                             string         message,  
                             int            verbosity_level,  
                             string         filename,  
                             int            line,  
                             ovm_report_object client      )
```

This is the common handler method used by the four core reporting methods (e.g., `ovm_report_error`) in [ovm\\_report\\_object](#).

## **`format_action`**

---

```
function string format_action(ovm_action action)
```

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

# ovm\_report\_server

ovm\_report\_server is a global server that processes all of the reports generated by an ovm\_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\_ovm\_info may be overloaded in a subclass.

## Summary

### ovm\_report\_server

ovm\_report\_server is a global server that processes all of the reports generated by an ovm\_report\_handler.

#### Class Declaration

```
class ovm_report_server
```

#### 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\_max\_quit\_count**

**get\_max\_quit\_count** Get or set the maximum number of COUNT actions that can be tolerated before an OVM\_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 ovm\_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\_max\_quit\_count

---

```
function void set_max_quit_count(int count)
```

### 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 OVM\_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(ovm_severity severity,  
                                int          count   )
```

---

### **get\_severity\_count**

---

```
function int get_severity_count(ovm_severity severity)
```

---

### **incr\_severity\_count**

---

```
function void incr_severity_count(ovm_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(ovm_severity    severity,
                                     string           name,
                                     string           id,
                                     string           message,
                                     ovm_action       action,
                                     OVM_FILE         file,
                                     string           filename,
                                     int              line,
                                     string           composed_message,
                                     int              verbosity_level,
                                     ovm_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(ovm_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(OVM_FILE file = )
```

See ovm\_report\_object::report\_summarize method.

## dump\_server\_state

---

```
function void dump_server_state()
```

Dumps server state information.

## get\_server

---

```
function ovm_report_server get_server()
```

Returns a handle to the central report server.

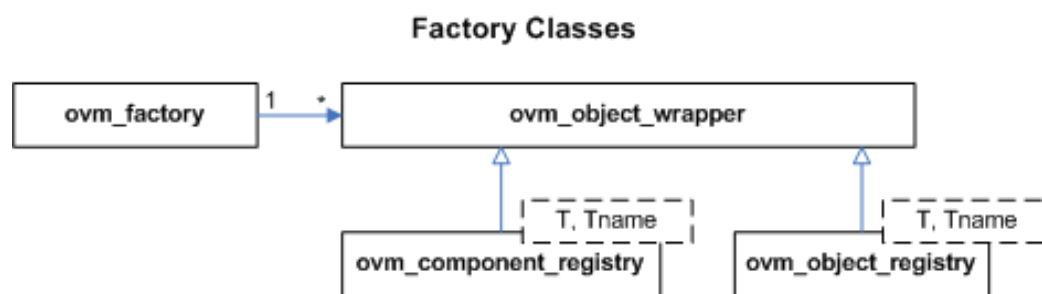


## Factory Classes

As the name implies, the `ovm_factory` is used to manufacture (create) OVM 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 `ovm_factory::Usage`. The factory generates and stores lightweight proxies to the user-defined objects and components: `ovm_object_registry #(T,Tname)` for objects and `ovm_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. `ovm_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.



## ovm\_component\_registry #(T,Tname)

The `ovm_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 [ovm\\_factory](#). Without it, registration would require an instance of the component itself.

See Usage section below for information on using `ovm_component_registry`.

### Summary

#### ovm\_component\_registry #(T,Tname)

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

##### Class Hierarchy

```
ovm_object_wrapper
```

```
ovm_component_registry#(T,Tname)
```

##### Class Declaration

```
class ovm_component_registry #(
    type    T      = ovm_component,
    string  Tname = "<unknown>"
) extends ovm_object_wrapper
```

##### Methods

[create\\_component](#) Creates a component of type *T* having the provided *name* and *parent*.

[get\\_type\\_name](#) Returns the value given by the string parameter, *Tname*.

[get](#) Returns the singleton instance of this type.

[create](#) Returns an instance of the component type, *T*, represented by this proxy, subject to any factory overrides based on the context provided by the *parent's* full name.

[set\\_type\\_override](#) Configures the factory to create an object of the type represented by *override\_type* whenever a request is made to create an object of the type, *T*, represented by this proxy, provided no instance override applies.

[set\\_inst\\_override](#) Configures the factory to create a component of the type represented by *override\_type* whenever a request is made to create an object of the type, *T*, represented by this proxy, with matching instance paths.

## Methods

### create\_component

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

Creates a component of type *T* having the provided *name* and *parent*. This is an override of the method in [ovm\\_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 [ovm\\_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,
                        ovm_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 (ovm_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(ovm_object_wrapper override_type,
                                       string                inst_path,
                                       ovm_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.

## ovm\_object\_registry #(T,Tname)

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

See Usage section below for information on using `ovm_component_registry`.

### Summary

#### ovm\_object\_registry #(T,Tname)

The `ovm_object_registry` serves as a lightweight proxy for an `ovm_object` of type *T* and type name *Tname*, a string.

##### Class Hierarchy

```
ovm_object_wrapper
```

```
ovm_object_registry#(T,Tname)
```

##### Class Declaration

```
class ovm_object_registry #(
    type    T        = ovm_object,
    string  Tname    = "<unknown>"
) extends ovm_object_wrapper
```

<code>create_object</code>	Creates an object of type <i>T</i> and returns it as a handle to an <code>ovm_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 represented by <i>override_type</i> whenever a request is made to create an object of the type represented by this proxy, provided no instance override applies.
<code>set_inst_override</code>	Configures the factory to create an object of the type represented by <i>override_type</i> whenever a request is made to create an object of the type represented by this proxy, with matching instance paths.
<b>Usage</b>	This section describes usage for the <code>ovm_*_registry</code> classes.

### create\_object

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

Creates an object of type *T* and returns it as a handle to an `ovm_object`. This is an override of the method in `ovm_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 [ovm\\_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      = " ",
                          ovm_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 (ovm_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(ovm_object_wrapper override_type,
                                       string                inst_path,
                                       ovm_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 `ovm_*_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 OVM component of type *mycomp*

```
class mycomp extends ovm_component;
  typedef ovm_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 OVM component of type *mycomp* in a vendor-independent way, you would write instead:

```
class mycomp extends ovm_component;
  `ovm_component_utils(mycomp);
  ...
endclass
```

The ``ovm_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 OVM component of type driver `#(T)`, you would write:

```
class driver #(type T=int) extends ovm_component;
  `ovm_component_param_utils(driver #(T));
  ...
endclass
```

The ``ovm_component_param_utils` and ``ovm_object_param_utils` macros are used to register

parameterized classes with the factory. Unlike the the non-param versions, these macros do not specify the *Tname* parameter in the underlying `ovm_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 [ovm\\_factory](#) is no longer recommended.

# OVM Factory

This page covers the following classes.

- [ovm\\_factory](#) - creates objects and components according to user-defined type and instance-based overrides.
- [ovm\\_object\\_wrapper](#) - a lightweight substitute (proxy) representing a user-defined object or component.

## Summary

### OVM Factory

This page covers the following classes.

## ovm\_factory

As the name implies, `ovm_factory` is used to manufacture (create) OVM 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 [ovm\\_object\\_registry #\(T,Tname\)](#) and [ovm\\_component\\_registry #\(T,Tname\)](#) class are used to proxy [ovm\\_objects](#) and [ovm\\_components](#).

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

*type-based* The type-based interface is far less prone to errors in usage. When errors do occur, they are caught at compile-time.

*name-based* The name-based interface is dominated by string arguments that can be misspelled and provided in the wrong order. Errors in name-based requests might only be caught at the time of the call, if at all. Further, the name-based interface is not portable across simulators when used with parameterized classes.

See [Usage](#) section for details on configuring and using the factory.

## Summary

### ovm\_factory

As the name implies, `ovm_factory` is used to manufacture (create) OVM objects and components.

#### Class Declaration

```
class ovm_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



<a href="#">debug_create_by_type</a>	These methods perform the same search algorithm as the <code>create_*</code> methods, but they do not create new objects.
<a href="#">debug_create_by_name</a>	
<a href="#">find_override_by_type</a>	These methods return the proxy to the object that would be created given the arguments.
<a href="#">find_override_by_name</a>	
<a href="#">print</a>	Prints the state of the <code>ovm_factory</code> , including registered types, instance overrides, and type overrides.
<b>Usage</b>	Using the factory involves three basic operations

## Registering Types

---

### [register](#)

---

```
function void register (ovm_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 (ovm_object_wrapper original_type,
                                         ovm_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 (ovm_object_wrapper original_type,
                                       ovm_object_wrapper override_type,
                                       bit replace = 1)
```

### 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 ovm_object create_object_by_type (ovm_object_wrapper requested_type,
                                       string parent_inst_path = "",
                                       string name = "")
```

### create\_component\_by\_type

---

```
function ovm_component create_component_by_type (
    ovm_object_wrapper requested_type,
    string               parent_inst_path = "",
    string               name,
    ovm_component        parent
)
```

### create\_object\_by\_name

---

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

### create\_component\_by\_name

---

```
function ovm_component create_component_by_name (string requested_type_name,
                                                  string parent_inst_path   = "",
                                                  string name,
                                                  ovm_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 [ovm\\_component](#) base class, and a requested object must be derived from the [ovm\\_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 [ovm\\_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 (ovm_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 ovm_object_wrapper find_override_by_type (
    ovm_object_wrapper requested_type,
    string              full_inst_path
)
```

### find\_override\_by\_name

---

```
function ovm_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 `ovm_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 OVM types (prefixed with `ovm_`) are included in the list of registered types.

## Usage

---

Using the factory involves three basic operations

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

We'll briefly cover each of these steps here. More reference information can be found at [Utility Macros](#), [ovm\\_component\\_registry #\(T,Tname\)](#), [ovm\\_object\\_registry #\(T,Tname\)](#), [ovm\\_component](#).

## 1 -- Registering objects and component types with the factory

When defining [ovm\\_object](#) and [ovm\\_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 ovm_object;
  `ovm_object_utils(packet)
endclass

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

Objects that are parameterized are declared as

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

Components that are not parameterized are declared as

```
class comp extends ovm_component;
  `ovm_component_utils(comp)
endclass
```

Components that are parameterized are declared as

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

The ``ovm_*_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 [ovm\\_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 ``ovm_*_param_utils` macros for parameterized classes differ from ``ovm_*_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 ``ovm_*_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=ovm_object) extends ovm_driver;

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

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

    // standard component constructor
    function new(string name, ovm_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();
        pkt = packet::type_id::create("pkt",this);
    endfunction

    // print the packet so we can confirm its type when printing
    virtual function void do_print(ovm_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 `ovm_object::get_type_name`, which we recommend writing in terms of a static string constant.

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

    `ovm_component_param_utils(driverD1 #(T))

    function new(string name, ovm_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=ovm_object) extends driverB #(T);

    `ovm_component_param_utils(driverD2 #(T))

    function new(string name, ovm_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 ovm_agent;

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

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

  virtual function void build();

    // 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 ovm_env;

  `ovm_component_utils(env)

  agent agent0;
  agent agent1;

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

  virtual function void build();

    // 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
endclass

```

```

endfunction

// at end_of_elaboration, print topology and factory state to verify
virtual function void end_of_elaboration();
    ovm_top.print_topology();
endfunction

virtual task run();
    #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 `<ovm_top.print_topology>` in env's `ovm_component::end_of_elaboration` method) is similar to the following:

```

# OVM_INFO @ 0 [RNTST] Running test ...
# OVM_INFO @ 0 [OVMTOP] OVM testbench topology:
# -----
# Name                Type                Size      Value
# -----
# env0                env                -          env0@2
#   agent0            agent                -          agent0@4
#     driver0         driverB #(packet)  -          driver0@8
#       pkt           packet                -          pkt@21
#     driver1         driverD #(packet)  -          driver1@14
#       pkt           packet                -          pkt@23
#   agent1            agent                -          agent1@6

```



```
# driver0 driverD #(packet) - driver0@24
# pkt packet - pkt@37
# driver1 driverD2 #(packet) - driver1@30
# pkt packet - pkt@39
# -----
```

## ovm\_object\_wrapper

The `ovm_object_wrapper` provides an abstract interface for creating object and component proxies. Instances of these lightweight proxies, representing every `ovm_object`-based and `ovm_component`-based object available in the test environment, are registered with the `ovm_factory`. When the factory is called upon to create an object or component, it finds and delegates the request to the appropriate proxy.

### Summary

#### ovm\_object\_wrapper

The `ovm_object_wrapper` provides an abstract interface for creating object and component proxies.

##### Class Declaration

```
virtual class ovm_object_wrapper
```

##### Methods

`create_object` Creates a new object with the optional *name*.

`create_component` Creates a new component, passing to its constructor the given *name* and *parent*.

`get_type_name` Derived classes implement this method to return the type name of the object created by `create_component` or `create_object`.

## Methods

### create\_object

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

Creates a new object with the optional *name*. An object proxy (e.g., `ovm_object_registry #(T, Tname)`) implements this method to create an object of a specific type, *T*.

### create\_component

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

Creates a new component, passing to its constructor the given *name* and *parent*. A component proxy (e.g. `ovm_component_registry #(T,Tname)`) implements this method to create a component of a specific type, *T*.

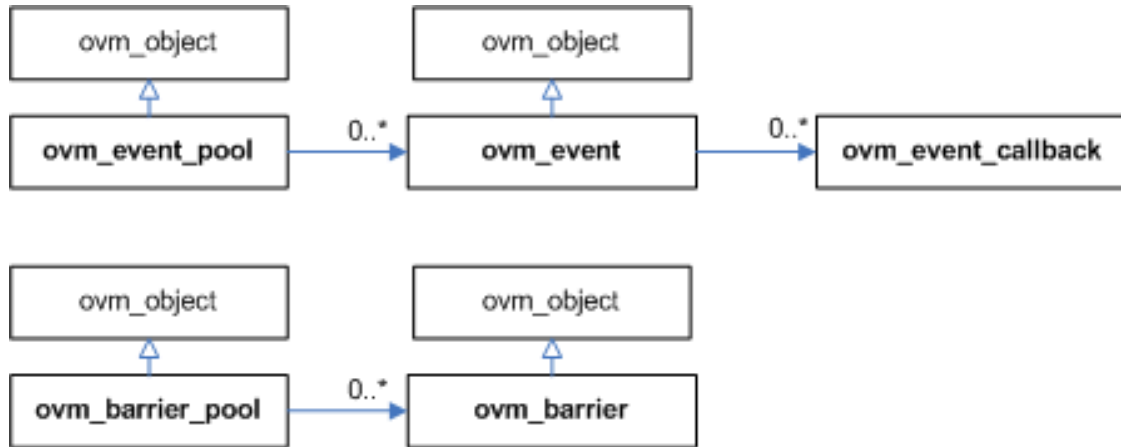
### get\_type\_name

```
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.

# Synchronization Classes



The OVM provides event and barrier synchronization classes for managing concurrent processes.

- [ovm\\_event](#) - OVM's event class augments the SystemVerilog event datatype with such services as setting callbacks and data delivery.
- [ovm\\_barrier](#) - A barrier is used to prevent a pre-configured number of processes from continuing until all have reached a certain point in simulation.
- [ovm\\_event\\_pool](#) and [ovm\\_barrier\\_pool](#) - The event and barrier pool classes are used to store collections of events and barriers, all indexed by string name. Each pool class contains a static, "global" pool instance for sharing across all processes.

# ovm\_event

The ovm\_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

### ovm\_event

The ovm\_event class is a wrapper class around the SystemVerilog event construct.

#### Class Hierarchy

```
ovm_object
```

```
ovm_event
```

#### Class Declaration

```
class ovm_event extends ovm_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_ptrigger</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_ptrigger_data</code>	This method calls <code>wait_ptrigger</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, <i>cb</i> , with this event.
<code>delete_callback</code>	Unregisters the given callback, <i>cb</i> , 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 ovm_object data)
```

This method calls [wait\\_trigger](#) followed by [get\\_trigger\\_data](#).

## wait\_ptrigger\_data

---

```
virtual task wait_ptrigger_data (output ovm_object data)
```

This method calls [wait\\_ptrigger](#) followed by [get\\_trigger\\_data](#).

## trigger

---

```
virtual function void trigger (ovm_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 ovm_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 (ovm_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 (ovm_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.



# ovm\_event\_callback

The `ovm_event_callback` class is an abstract class that is used to create callback objects which may be attached to [ovm\\_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

### ovm\_event\_callback

The `ovm_event_callback` class is an abstract class that is used to create callback objects which may be attached to [ovm\\_events](#).

#### Class Hierarchy

```
ovm_object
```

```
ovm_event_callback
```

#### Class Declaration

```
virtual class ovm_event_callback extends ovm_object
```

#### Methods

[new](#) Creates a new callback object.

[pre\\_trigger](#) This callback is called just before triggering the associated event.

[post\\_trigger](#) This callback is called after triggering the associated event.

## Methods

### new

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

Creates a new callback object.

### pre\_trigger

```
virtual function bit pre_trigger (ovm_event e,
                                  ovm_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 [ovm\\_event](#) that is being triggered, and *data* is the optional data associated with the event trigger.

## post\_trigger

---

```
virtual function void post_trigger (ovm_event  e,  
                                   ovm_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 [ovm\\_event](#) that is being triggered, and *data* is the optional data associated with the event trigger.

# ovm\_barrier

The ovm\_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

### ovm\_barrier

The ovm\_barrier class provides a multiprocess synchronization mechanism.

#### Class Hierarchy

```
ovm_object
```

```
ovm_barrier
```

#### Class Declaration

```
class ovm_barrier extends ovm_object
```

#### Methods

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

# ovm\_objection

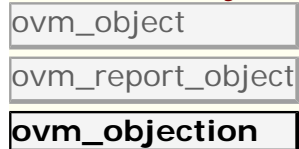
Objections provide a facility for coordinating status information between two or more participating components, objects, and even module-based IP. In particular, the *ovm\_test\_done* built-in objection provides a means for coordinating when to end a test, i.e. when to call [global\\_stop\\_request](#) to end the [ovm\\_component::run](#) phase. When all participating components have dropped their raised objections with *ovm\_test\_done*, an implicit call to *global\_stop\_request* is issued.

## Summary

### ovm\_objection

Objections provide a facility for coordinating status information between two or more participating components, objects, and even module-based IP.

#### Class Hierarchy



#### Class Declaration

```
class ovm_objection extends ovm_report_object
```

**new** Creates a new objection instance.

#### Objection Control

**raise\_objection** Raises the number of objections for the source *object* by *count*, which defaults to 1.  
**drop\_objection** Drops the number of objections for the source *object* by *count*, which defaults to 1.  
**set\_drain\_time** Sets the drain time on the given *object* to *drain*.

#### Callback Hooks

**raised** Objection callback that is called when a **raise\_objection** has reached *obj*.  
**dropped** Objection callback that is called when a **drop\_objection** has reached *obj*.  
**all\_dropped** Objection callback that is called when a **drop\_objection** has reached *obj*, and the total count for *obj* goes to zero.

#### Objection Status

**get\_objection\_count** Returns the current number of objections raised by the given *object*.  
**get\_objection\_total** Returns the current number of objections raised by the given *object* and all descendants.  
**get\_drain\_time** Returns the current drain time set for the given *object* (default: 0 ns).  
**display\_objections** Displays objection information about the given *object*.

#### new

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

Creates a new objection instance.

## Objection Control

## raise\_objection

---

```
function void raise_objection (ovm_object obj    = null,
                              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, *ovm\_top*, is chosen.

Raising an objection causes the following.

- The source and total objection counts for *object* are increased by *count*.
- The objection's [raised](#) virtual method is called, which calls the [ovm\\_component::raised](#) method for all of the components up the hierarchy.

## drop\_objection

---

```
function void drop_objection (ovm_object obj    = null,
                              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, *ovm\_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 [ovm\\_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 [ovm\\_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 [ovm\\_component::all\\_dropped](#) method (if *object* is a component).

- The process then waits for the *all\_dropped* callback to complete.
- After the drain time has elapsed and *all\_dropped* callback has completed, propagation of the dropped objection to the parent proceeds as described in [raise\\_objection](#), except as described below.

If a new objection for this *object* or any of its descendents is raised during the drain time or during execution of the *all\_dropped* callback at any point, the hierarchical chain described above is terminated and the dropped callback does not go up the hierarchy. The raised objection will propagate up the hierarchy, but the number of raised propagated up is reduced by the number of drops that were pending waiting for the *all\_dropped*/drain time completion. Thus, if exactly one objection caused the count to go to zero, and during the drain exactly one new objection comes in, no raises or drops are 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

---

```
function void set_drain_time (ovm_object obj,
                             time       drain)
```

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 (ovm_object obj,
                             ovm_object source_obj,
                             int       count      )
```

Objection callback that is called when a [raise\\_objection](#) has reached *obj*. The default implementation calls [ovm\\_component::raised](#).

### dropped

---



```
virtual function void dropped (ovm_object obj,
                             ovm_object source_obj,
                             int count )
```

Objection callback that is called when a [drop\\_objection](#) has reached *obj*. The default implementation calls [ovm\\_component::dropped](#).

## [all\\_dropped](#)

---

```
virtual task all_dropped (ovm_object obj,
                         ovm_object source_obj,
                         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 [ovm\\_component::all\\_dropped](#).

# Objection Status

---

## [get\\_objection\\_count](#)

---

```
function int get_objection_count (ovm_object obj)
```

Returns the current number of objections raised by the given *object*.

## [get\\_objection\\_total](#)

---

```
function int get_objection_total (ovm_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 (ovm_object obj)
```

Returns the current drain time set for the given *object* (default: 0 ns).

## [display\\_objections](#)

---

```
protected function string m_display_objections(ovm_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, <ovm\_top>, is chosen. The *show\_header* argument allows control of whether a header is output.

## ovm\_test\_done\_objection

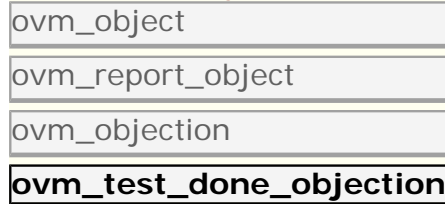
Built-in end-of-test coordination

### Summary

#### ovm\_test\_done\_objection

Built-in end-of-test coordination

##### Class Hierarchy



##### Class Declaration

```
class ovm_test_done_objection extends ovm_objection
```

##### Methods

- qualify** Checks that the given *object* is derived from either *ovm\_component* or *ovm\_sequence\_base*.
- all\_dropped** This callback is called when the given *object's* objection count reaches zero; if the *object* is the implicit top-level, <ovm\_top>, then it means there are no more objections raised for the *ovm\_test\_done* objection.
- raise\_objection** Calls *ovm\_objection::raise\_objection* after calling *qualify*.
- drop** Calls *ovm\_objection::drop\_objection* after calling *qualify*.
- force\_stop**

## Methods

### qualify

```
virtual function void qualify(ovm_object obj = null,
                             bit is_raise )
```

Checks that the given *object* is derived from either *ovm\_component* or *ovm\_sequence\_base*.

## all\_dropped

---

```
virtual task all_dropped (ovm_object obj,
                        ovm_object source_obj,
                        int count )
```

This callback is called when the given *object*'s objection count reaches zero; if the *object* is the implicit top-level, <ovm\_top> then it means there are no more objections raised for the *ovm\_test\_done* objection. Thus, after calling [ovm\\_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 (ovm_object obj = null,
                                     int count = 1 )
```

Calls [ovm\\_objection::raise\\_objection](#) after calling [qualify](#). If the *object* is not provided or is *null*, then the implicit top-level component, *ovm\_top*, is chosen.

## drop

---

```
virtual function void drop_objection (ovm_object obj = null,
                                     int count = 1 )
```

Calls [ovm\\_objection::drop\\_objection](#) after calling [qualify](#). If the *object* is not provided or is *null*, then the implicit top-level component, *ovm\_top*, is chosen.

## force\_stop

---

```
virtual task force_stop(ovm_object obj = null )
```

## ovm\_pool #(T)

Implements a class-based dynamic associative array. Allows sparse arrays to be allocated on demand, and passed and stored by reference.

### Summary

#### ovm\_pool #(T)

Implements a class-based dynamic associative array.

#### Class Hierarchy

```
ovm_object
```

```
ovm_pool#(T)
```

#### Class Declaration

```
class ovm_pool #(type KEY = int,
                  T       = ovm_void
                  ) extends ovm_object
```

#### Methods

<code>new</code>	Creates a new pool with the given <i>name</i> .
<code>get_global_pool</code>	Returns the singleton global pool for the item type, T.
<code>get_global</code>	Returns the specified item instance from the global item pool.
<code>get</code>	Returns the item with the given <i>key</i> .
<code>add</code>	Adds the given ( <i>key</i> , <i>item</i> ) pair to the pool.
<code>num</code>	Returns the number of uniquely keyed items stored in the pool.
<code>delete</code>	Removes the item with the given <i>key</i> from the pool.
<code>exists</code>	Returns 1 if a item with the given <i>key</i> exists in the pool, 0 otherwise.
<code>first</code>	Returns the key of the first item stored in the pool.
<code>last</code>	Returns the key of the last item stored in the pool.
<code>next</code>	Returns the key of the next item in the pool.
<code>prev</code>	Returns the key of the previous item in the pool.

## Methods

#### new

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

Creates a new pool with the given *name*.

#### get\_global\_pool

```
static function this_type get_global_pool ()
```

Returns the singleton global pool for the item type, T.

This allows items to be shared amongst components throughout the verification environment.

## get\_global

---

```
static function T get_global (KEY key)
```

Returns the specified item instance from the global item pool.

## get

---

```
virtual function T get (KEY key)
```

Returns the item with the given *key*.

If no item exists by that key, a new item is created with that key and returned.

## add

---

```
virtual function void add (KEY key,  
                           T    item)
```

Adds the given (*key*, *item*) pair to the pool.

## 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.

## ovm\_object\_string\_pool #(T)

This provides a specialization of the generic <ovm\_pool #(KEY,T) class for an associative

array of [ovm\\_object](#)-based objects indexed by string. Specializations of this class include the *ovm\_event\_pool* and *ovm\_barrier\_pool* classes.

## Summary

### ovm\_object\_string\_pool #(T)

This provides a specialization of the generic `<ovm_pool #(KEY,T)` class for an associative array of [ovm\\_object](#)-based objects indexed by string.

#### Class Hierarchy

```
ovm_pool#(string,T)
```

```
ovm_object_string_pool#(T)
```

#### Class Declaration

```
class ovm_object_string_pool #(
    type    T =    ovm_object
) extends ovm_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.



# ovm\_queue #(T)

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

## Summary

### ovm\_queue #(T)

Implements a class-based dynamic queue.

#### Class Hierarchy

```
ovm_object
```

```
ovm_queue#(T)
```

#### Class Declaration

```
class ovm_queue #(type T = int ) extends ovm_object
```

#### Methods

<b>new</b>	Creates a new queue with the given <i>name</i> .
<b>get_global_queue</b>	Returns the singleton global queue for the item type, T.
<b>get_global</b>	Returns the specified item instance from the global item queue.
<b>get</b>	Returns the item at the given <i>index</i> .
<b>size</b>	Returns the number of items stored in the queue.
<b>insert</b>	Inserts the item at the given <i>index</i> in the queue.
<b>delete</b>	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.
<b>pop_front</b>	Returns the first element in the queue (index=0), or <i>null</i> if the queue is empty.
<b>pop_back</b>	Returns the last element in the queue (index=size()-1), or <i>null</i> if the queue is empty.
<b>push_front</b>	Inserts the given <i>item</i> at the front of the queue.
<b>push_back</b>	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.

## ovm\_callbacks #(T,CB)

The *ovm\_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.

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

#### ovm\_callbacks #(T,CB)

The *ovm\_callbacks* class provides a base class for implementing callbacks, which are typically used to modify or augment component behavior without changing the component class.

##### Class Hierarchy

```
ovm_pool#(T,ovm_queue#(CB))
```

```
ovm_callbacks#(T,CB)
```

##### Class Declaration

```
class ovm_callbacks #(
    type      T      =    int,
              CB     =    int
) extends ovm_pool #(T,ovm_queue #(CB))
```

##### Parameters

<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.

##### Methods

<b>new</b>	Creates a new <i>ovm_callbacks</i> object, giving it an optional <i>name</i> .
<b>get_global_cbs</b>	Returns the global callback pool for this type.
<b>add_cb</b>	Registers the given callback object, <i>cb</i> , with the given <i>obj</i> handle.
<b>delete_cb</b>	Removes a previously registered callback, <i>cb</i> , for the given object, <i>obj</i> .
<b>trace_mode</b>	This function takes a single argument to turn on (1) or off (0) tracing.
<b>display_cbs</b>	Displays information about all registered callbacks for the given <i>obj</i> handle.

## Parameters

---

### T

---

This type parameter specifies the base object type with which the [CB](#) callback objects will be registered.

### CB

---

This type parameter specifies the base callback type that will be managed by this callback class. The callback type is typically a interface class, which defines one or more virtual method prototypes that users can override in subtypes.

## Methods

---

### new

---

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

Creates a new ovm\_callbacks object, giving it an optional *name*.

### get\_global\_cbs

---

Returns the global callback pool for this type.

This allows items to be shared amongst components throughout the verification environment.

### add\_cb

---

```
virtual function void add_cb(T    obj,
                             CB   cb,
                             bit  append = 1 )
```

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 *append* is 1 (default), the callback will be executed after previously added callbacks, else the callback will be executed ahead of previously added callbacks.

## delete\_cb

---

```
virtual function void delete_cb(T  obj,
                               CB cb  )
```

Removes a previously registered callback, *cb*, for the given object, *obj*.

## trace\_mode

---

```
function void trace_mode(bit mode)
```

This function takes a single argument to turn on (1) or off (0) tracing. The default is to turn tracing on.

## display\_cbs

---

```
function void display_cbs(T obj = null )
```

Displays information about all registered callbacks for the given *obj* handle. If *obj* is not provided or is null, then information about all callbacks for all objects is displayed.

# ovm\_callback

The *ovm\_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

## ovm\_callback

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

### Class Hierarchy

```
ovm_object
```

```
ovm_callback
```

### Class Declaration

```
class ovm_callback extends ovm_object
```

### Methods

<code>new</code>	Creates a new <i>ovm_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 = "ovm_callback" )
```

Creates a new *ovm\_callback* object, giving it an optional *name*.

### `callback_mode`

---

```
function void callback_mode(bit on)
```

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.



## Policy Classes

Each of OVM's policy classes perform a specific task for *ovm\_object*-based objects: printing, comparing, recording, packing, and unpacking. They are implemented separately from *ovm\_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.

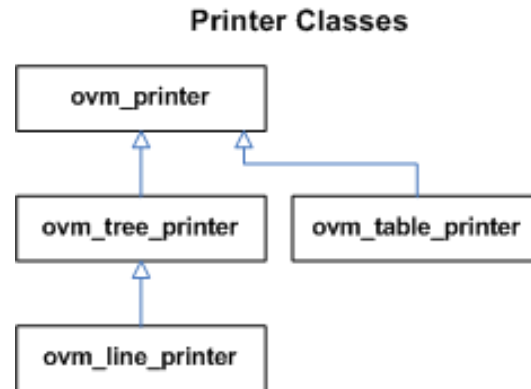
Each policy class includes several user-configurable parameters that control the operation. Users may also customize operations by deriving new policy subtypes from these base types. For example, the OVM provides four different *ovm\_printer*-based policy classes, each of which print objects in a different format.

- *ovm\_printer* - performs deep printing of *ovm\_object*-based objects. The OVM provides several subtypes to *ovm\_printer* that print objects in a specific format: *ovm\_table\_printer*, *ovm\_tree\_printer*, and *ovm\_line\_printer*. Each such printer has many configuration options that govern what and how object members are printed.
- *ovm\_comparer* - performs deep comparison of *ovm\_object*-based objects. Users may configure what is compared and how mismatches are reported.
- *ovm\_recorder* - performs the task of recording *ovm\_object*-based objects to a transaction data base. The implementation is vendor-specific.
- *ovm\_packer* - used to pack (serialize) and unpack *ovm\_object*-based properties into bit, byte, or int arrays and back again.

## ovm\_printer

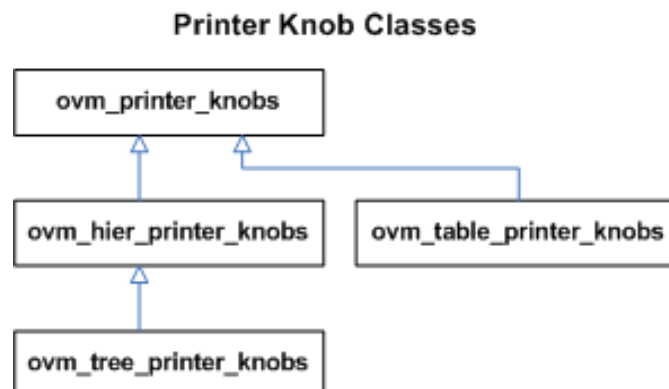
The `ovm_printer` class provides an interface for printing `ovm_objects` in various formats. Subtypes of `ovm_printer` implement different print formats, or policies.

A user-defined printer format can be created, or one of the following four built-in printers can be used:



- `ovm_printer` - provides raw, essentially un-formatted output
- `ovm_table_printer` - prints the object in a tabular form.
- `ovm_tree_printer` - prints the object in a tree form.
- `ovm_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 separate knob classes:



- `ovm_printer_knobs` - common printer settings
- `ovm_hier_printer_knobs` - settings for printing hierarchically
- `ovm_table_printer_knobs` - settings specific to the table printer
- `ovm_tree_printer_knobs` - settings specific to the tree printer

For convenience, global instances of each printer type are available for direct reference in your testbenches.

- [ovm\\_default\\_tree\\_printer](#)
- [ovm\\_default\\_line\\_printer](#)
- [ovm\\_default\\_table\\_printer](#)
- [ovm\\_default\\_printer](#) (set to [default\\_table\\_printer](#) by default)

The [ovm\\_default\\_printer](#) is used by [ovm\\_object::print](#) and [ovm\\_object::sprint](#) when the optional [ovm\\_printer](#) argument to these methods is not provided.

## Summary

### ovm\_printer

The [ovm\\_printer](#) class provides an interface for printing [ovm\\_objects](#) in various formats.

#### Class Declaration

```
class ovm_printer
```

<a href="#">knobs</a>	The knob object provides access to the variety of knobs associated with a specific printer instance.
-----------------------	--

#### Methods for printer usage

<a href="#">print_field</a>	Prints an integral field.
<a href="#">print_object_header</a>	Prints the header of an object.
<a href="#">print_object</a>	Prints an object.
<a href="#">print_string</a>	Prints a string field.
<a href="#">print_time</a>	Prints a time value.

#### Methods for printer subtyping

<a href="#">print_header</a>	Prints header information.
<a href="#">print_footer</a>	Prints footer information.
<a href="#">print_id</a>	Prints a field's name, or <i>id</i> , which is the full instance name.
<a href="#">print_type_name</a>	Prints a field's type name.
<a href="#">print_size</a>	Prints a field's size.
<a href="#">print_newline</a>	Prints a newline character.
<a href="#">print_value</a>	Prints an integral field's value.
<a href="#">print_value_object</a>	Prints a unique handle identifier for the given object.
<a href="#">print_value_string</a>	Prints a string field's value.
<a href="#">print_value_array</a>	Prints an array's value.
<a href="#">print_array_header</a>	Prints the header of an array.
<a href="#">print_array_range</a>	Prints a range using ellipses for values.
<a href="#">print_array_footer</a>	Prints the header of a footer.

### knobs

```
ovm_printer_knobs knobs = new
```

The knob object provides access to the variety of knobs associated with a specific printer instance.

Each derived printer class overwrites the [knobs](#) variable with the a derived knob class that extends [ovm\\_printer\\_knobs](#). The derived knobs class adds more knobs to the base knobs.

## Methods for printer usage

---

### print\_field

---

```
virtual function void print_field (string      name,
                                   ovm_bitstream_t value,
                                   int          size,
                                   ovm_radix_enum radix      = OVM_NORADIX,
                                   byte         scope_separator = ".",
                                   string       type_name      = " ")
```

Prints an integral field.

*name*            The name of the field.

*value*           The value of the field.

*size*            The number of bits of the field (maximum is 4096).

*radix*           The radix to use for printing the printer knob for radix is used if no radix is specified.

*scope\_separator* is used to find the leaf name since many printers only print the leaf name of a field.  
Typical values for the separator are . (dot) or [ (open bracket).

### print\_object\_header

---

```
virtual function void print_object_header (string      name,
                                           ovm_object value,
                                           byte        scope_separator = ".")
```

Prints the header of an object.

This function is called when an object is printed by reference. For this function, the object will not be recursed.

### print\_object

---

```
virtual function void print_object (string      name,
                                    ovm_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 [ovm\\_components](#) are printed. To turn this behavior off, you must set the [ovm\\_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.

## Methods for printer subtyping

---

### [print\\_header](#)

---

```
virtual function void print_header ()
```

Prints header information. It is called when the current depth is 0, before any fields have been printed.

### [print\\_footer](#)

---

```
virtual function void print_footer ()
```

Prints footer information. It is called when the current depth is 0, after all fields have been printed.

### [print\\_id](#)

---

```
virtual protected function void print_id (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\\_type\\_name](#)

---

```
virtual protected function void print_type_name (string name,
                                                bit      is_object = )
```

Prints a field's type name.

The *is\_object* bit indicates that the item being printed is an object derived from [ovm\\_object](#).

### [print\\_size](#)

---

```
virtual protected function void print_size (int size = -1 )
```

Prints a field's size. A size of -1 indicates that no size is available, in which case the printer inserts the appropriate white space if the format requires it.

### [print\\_newline](#)

---

```
virtual protected function void print_newline (bit do_global_indent = 1 )
```

Prints a newline character. It is up to the printer to determine how or whether to display new lines. The *do\_global\_indent* bit indicates whether the call to `print_newline()` should honor the indent knob.

### [print\\_value](#)

---

```
virtual protected function void print_value (ovm_bitstream_t value,
                                             int              size,
                                             ovm_radix_enum  radix  = OVM_NORADIX)
```

Prints an integral field's value.

The *value* vector is up to 4096 bits, and the *size* input indicates the number of bits to actually print.

The *radix* input is the radix that should be used for printing the value.

### [print\\_value\\_object](#)

---

```
virtual protected function void print_value_object (ovm_object value)
```

Prints a unique handle identifier for the given object.

### [print\\_value\\_string](#)

---

```
virtual protected function void print_value_string (string value)
```

Prints a string field's value.

## print\_value\_array

---

```
virtual function void print_value_array (string value = "",
                                       int    size  = 0
                                       )
```

Prints an array's value.

This only prints the header value of the array, which means that it implements the printer-specific `print_array_header()`.

*value* is the value to be printed for the array. It is generally the string representation of *size*, but it may be any string. *size* is the number of elements in the array.

## 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, [ovm\\_printer\\_knobs::begin\\_elements](#) and [ovm\\_printer\\_knobs::end\\_elements](#).

This function should be called after `begin_elements` have been printed and after `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.

## ovm\_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

### ovm\_table\_printer

The table printer prints output in a tabular format.

#### Class Hierarchy

```
ovm_printer
```

```
ovm_table_printer
```

#### Class Declaration

```
class ovm_table_printer extends ovm_printer
```

#### Variables

**new** Creates a new instance of *ovm\_table\_printer*.

**knobs** An instance of *ovm\_table\_printer\_knobs*, which govern the content and format of the printed table.

## Variables

### new

```
function new()
```

Creates a new instance of *ovm\_table\_printer*.

### knobs

```
ovm_table_printer_knobs knobs = new
```

An instance of *ovm\_table\_printer\_knobs*, which govern the content and format of the printed table.



## ovm\_tree\_printer

By overriding various methods of the [ovm\\_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

### ovm\_tree\_printer

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

#### Class Hierarchy

```
ovm_printer
```

```
ovm_tree_printer
```

#### Class Declaration

```
class ovm_tree_printer extends ovm_printer
```

#### Variables

**new** Creates a new instance of *ovm\_tree\_printer*.

**knobs** An instance of *ovm\_tree\_printer\_knobs*, which govern the content and format of the printed tree.

## Variables

### new

```
function new()
```

Creates a new instance of *ovm\_tree\_printer*.

### knobs

```
ovm_tree_printer_knobs knobs = new
```

An instance of [ovm\\_tree\\_printer\\_knobs](#), which govern the content and format of the printed tree.

# ovm\_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

### ovm\_line\_printer

The line printer prints output in a line format.

#### Class Hierarchy

```
ovm_printer
```

```
ovm_tree_printer
```

```
ovm_line_printer
```

#### Class Declaration

```
class ovm_line_printer extends ovm_tree_printer
```

#### Variables

**new** Creates a new instance of *ovm\_line\_printer*.

#### Methods

**print\_newline** Overrides `ovm_printer::print_newline` to not print a newline, effectively making everything appear on a single line.

## Variables

### new

```
function new()
```

Creates a new instance of *ovm\_line\_printer*.

## Methods

### print\_newline

```
virtual function void print_newline (bit do_global_indent = 1 )
```

Overrides `ovm_printer::print_newline` to not print a newline, effectively making everything appear on a single line.

## ovm\_printer\_knobs

The `ovm_printer_knobs` class defines the printer settings available to all printer subtypes. Printer subtypes may subtype this class to provide additional knobs for their specific format. For example, the `ovm_table_printer` uses the `ovm_table_printer_knobs`, which defines knobs for setting table column widths.

### Summary

#### ovm\_printer\_knobs

The `ovm_printer_knobs` class defines the printer settings available to all printer subtypes.

##### Class Declaration

```
class ovm_printer_knobs
```

##### Variables

<code>max_width</code>	The maximum width of a field.
<code>truncation</code>	Specifies the character to use to indicate a field was truncated.
<code>header</code>	Indicates whether the <code>&lt;print_header&gt;</code> function should be called when printing an object.
<code>footer</code>	Indicates whether the <code>&lt;print_footer&gt;</code> function should be called when printing an object.
<code>global_indent</code>	Specifies the number of spaces of indentation to add whenever a newline is printed.
<code>full_name</code>	Indicates whether <code>&lt;print_id&gt;</code> should print the full name of an identifier or just the leaf name.
<code>identifier</code>	Indicates whether <code>&lt;print_id&gt;</code> should print the identifier.
<code>depth</code>	Indicates how deep to recurse when printing objects.
<code>reference</code>	Controls whether to print a unique reference ID for object handles.
<code>type_name</code>	Controls whether to print a field's type name.
<code>size</code>	Controls whether to print a field's size.
<code>begin_elements</code>	Defines the number of elements at the head of a list to print.
<code>end_elements</code>	This defines the number of elements at the end of a list that should be printed.
<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>prefix</code>	Specifies the string prepended to each output line
<code>mcd</code>	This is a file descriptor, or multi-channel descriptor, that specifies where the print output should be directed.
<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_field()</code> method.
<code>dec_radix</code>	This string should be prepended to the value of an integral type when a radix of <code>OVM_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>OVM_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>OVM_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>OVM_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>OVM_HEX</code> is used for the radix of the integral object.

##### Methods

<code>get_radix_str</code>	Converts the radix from an enumerated to a printable radix according to the radix printing knobs (bin_radix, and so on).
----------------------------	--

# Variables

---

## max\_width

---

```
int max_width = 999
```

The maximum width of a field. Any field that requires more characters will be truncated.

## truncation

---

```
string truncation = "+"
```

Specifies the character to use to indicate a field was truncated.

## 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.

## global\_indent

---

```
int global_indent = 0
```

Specifies the number of spaces of indentation to add whenever a newline is printed.

## full\_name

---

```
bit full_name = 1
```

Indicates whether <print\_id> should print the full name of an identifier or just the leaf name. The line, table, and tree printers ignore this bit and always print only the leaf name.

## identifier

---

```
bit identifier = 1
```

Indicates whether <print\_id> should print the identifier. This is useful in cases where you just want the values of an object, but no identifiers.

## 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.

## 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.

## 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.

## 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.

## prefix

---

```
string prefix = ""
```

Specifies the string prepended to each output line

## mcd

---

```
int mcd = OVM_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.

## default\_radix

---

```
ovm_radix_enum default_radix = OVM_HEX
```

This knob sets the default radix to use for integral values when no radix enum is explicitly supplied to the `print_field()` method.

## dec\_radix

---

```
string dec_radix = "'d"
```

This string should be prepended to the value of an integral type when a radix of [OVM\\_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 [OVM\\_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 [OVM\\_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 [OVM\\_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 [OVM\\_HEX](#) is used for the radix of the integral object.

# Methods

---

## get\_radix\_str

---

```
function string get_radix_str (ovm_radix_enum radix)
```

Converts the radix from an enumerated to a printable radix according to the radix printing knobs (bin\_radix, and so on).

## ovm\_hier\_printer\_knobs

The *ovm\_hier\_printer\_knobs* is a simple container class that extends `<ovm_printer::ovm_printer_knobs>` with settings for printing information hierarchically.

### Summary

### ovm\_hier\_printer\_knobs

The *ovm\_hier\_printer\_knobs* is a simple container class that extends `<ovm_printer::ovm_printer_knobs>` with settings for printing information hierarchically.

#### Class Hierarchy

```
ovm_printer_knobs
```

```
ovm_hier_printer_knobs
```

### Class Declaration

```
class ovm_hier_printer_knobs extends ovm_printer_knobs
```

### Variables

[indent\\_str](#) This knob specifies the string to use for level indentation.

[show\\_root](#) This setting indicates whether or not the initial object that is printed (when current depth is 0) prints the full path name.

## Variables

### [indent\\_str](#)

```
string indent_str = " "
```

This knob specifies the string 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.

## ovm\_table\_printer\_knobs

The *ovm\_table\_printer\_knobs* is a simple container class that extends <ovm\_printer::ovm\_hier\_printer\_knobs> with settings specific to printing in table format.

### Summary

### ovm\_table\_printer\_knobs

The *ovm\_table\_printer\_knobs* is a simple container class that extends <ovm\_printer::ovm\_hier\_printer\_knobs> with settings specific to printing in table format.

#### Class Hierarchy

```
ovm_printer_knobs
```

```
ovm_hier_printer_knobs
```

```
ovm_table_printer_knobs
```

### Class Declaration

```
class ovm_table_printer_knobs extends ovm_hier_printer_knobs
```



**Variables**

**name\_width** Sets the width of the *name* column.

**type\_width** Sets the width of the *type* column.

**size\_width** Sets the width of the *size* column.

**value\_width** Sets the width of the *value* column.

## Variables

---

### name\_width

---

```
int name_width = 25
```

Sets the width of the *name* column. If set to 0, the column is not printed.

### type\_width

---

```
int type_width = 20
```

Sets the width of the *type* column. If set to 0, the column is not printed.

### size\_width

---

```
int size_width = 5
```

Sets the width of the *size* column. If set to 0, the column is not printed.

### value\_width

---

```
int value_width = 20
```

Sets the width of the *value* column. If set to 0, the column is not printed.

## ovm\_tree\_printer\_knobs

The *ovm\_tree\_printer\_knobs* is a simple container class that extends <ovm\_printer::ovm\_hier\_printer\_knobs> with settings specific to printing in tree format.

### Summary

### ovm\_tree\_printer\_knobs

The *ovm\_tree\_printer\_knobs* is a simple container class that extends `<ovm_printer::ovm_hier_printer_knobs>` with settings specific to printing in tree format.

### Class Hierarchy

```
ovm_printer_knobs
```

```
ovm_hier_printer_knobs
```

```
ovm_tree_printer_knobs
```

### Class Declaration

```
class ovm_tree_printer_knobs extends ovm_hier_printer_knobs
```

### Variables

**separator** Determines the opening and closing separators used for nested objects.

## Variables

---

### separator

---

```
string separator = "{}"
```

Determines the opening and closing separators used for nested objects.

## ovm\_comparer

The ovm\_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 `ovm_object::compare` and `ovm_object::do_compare` methods are passed an ovm\_comparer policy object.

### Summary

#### ovm\_comparer

The ovm\_comparer class provides a policy object for doing comparisons.

##### Class Declaration

```
class ovm_comparer
```

##### Variables

<code>policy</code>	Determines whether comparison is OVM_DEEP, OVM_REFERENCE, or OVM_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>ovm_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, <i>msg</i> , to be appended to the <code>mismatches</code> string (a newline is used to separate messages).

## Variables

### policy

```
ovm_recursion_policy_enum policy = OVM_DEFAULT_POLICY
```

Determines whether comparison is OVM\_DEEP, OVM\_REFERENCE, or OVM\_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 = OVM_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

---

```
ovm_severity sev = OVM_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 [ovm\\_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 [ovm\\_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 [ovm\\_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,
                                     ovm_bitstream_t lhs,
                                     ovm_bitstream_t rhs,
                                     int              size,
                                     ovm_radix_enum   radix = OVM_NORADIX )
```

Compares two integral values.

The *name* input is used for purposes of storing and printing a miscompare.

The left-hand-side *lhs* and right-hand-side *rhs* objects are the two objects used for comparison.

The size variable indicates the number of bits to compare; size must be less than or equal to 4096.

The radix is used for reporting purposes, the default radix is hex.

## compare\_field\_int

---

```
virtual function bit compare_field_int (string      name,
                                       logic[63:0]  lhs,
                                       logic[63:0]  rhs,
                                       int          size,
                                       ovm_radix_enum radix = OVM_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,
                                     ovm_object lhs,
                                     ovm_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.

# ovm\_recorder

The ovm\_recorder class provides a policy object for recording [ovm\\_objects](#). The policies determine how recording should be done.

A default recorder instance, [ovm\\_default\\_recorder](#), is used when the [ovm\\_object::record](#) is called without specifying a recorder.

## Summary

### ovm\_recorder

The ovm\_recorder class provides a policy object for recording [ovm\\_objects](#).

#### Class Declaration

```
class ovm_recorder
```

#### Variables

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

#### Methods

<a href="#">record_field</a>	Records an integral field (less than or equal to 4096 bits).
<a href="#">record_field_real</a>	Records an real field.
<a href="#">record_object</a>	Records an object field.
<a href="#">record_string</a>	Records a string field.
<a href="#">record_time</a>	Records a time value.
<a href="#">record_generic</a>	Records the name-value pair, where value has been converted to a string, e.g.

## 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

---

```
ovm_radix_enum default_radix = OVM_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 [ovm\\_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 [ovm\\_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

---

```
ovm_recursion_policy_enum policy = OVM_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,
                                   ovm_bitstream_t value,
                                   int          size,
                                   ovm_radix_enum radix  = OVM_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 [ovm\\_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,
                                     ovm_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, e.g. via `$psprintf` (`"%<format>", <some variable>`);

# ovm\_packer

The `ovm_packer` class provides a policy object for packing and unpacking `ovm_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 ``ovm_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

### ovm\_packer

The `ovm_packer` class provides a policy object for packing and unpacking `ovm_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 (ovm_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 OVM.

### [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 (ovm_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 OVM.

## Unpacking

---

### is\_null

---

```
virtual function bit is_null ()
```

This method is used during unpack operations to peek at the next 4-bit chunk of the pack data and determine if it is 0.

If the next four bits are all 0, then the return value is a 1; otherwise it is 0.

This is useful when unpacking objects, to decide whether a new object needs to be allocated or not.

### unpack\_field\_int

---

```
virtual function logic[63:0] unpack_field_int (int size)
```

Unpacks bits from the pack array and returns the bit-stream that was unpacked.

*size* is the number of bits to unpack; the maximum is 64 bits. This is a more efficient variant than `unpack_field` when unpacking into smaller vectors.

### unpack\_field

---

```
virtual function ovm_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 (ovm_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 [ovm\\_object::do\\_pack](#) and [ovm\\_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 [ovm\\_object::do\\_pack](#) and [ovm\\_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 `<do_pack>` and `<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 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 ovm_object;

  logic[15:0] value = 'h1234;

  function void do_pack (ovm_packer packer);
    packer.pack_field_int(value, 16);
  endfunction

  function void do_unpack (ovm_packer packer);
    value = packer.unpack_field_int(16);
  endfunction
endclass

mydata d = new;
bit bits[];

initial begin
  d.pack(bits); // 'b0001001000110100
  ovm_default_packer.big_endian = 0;
  d.pack(bits); // 'b0010110001001000
end
```

# TLM Interfaces, Ports, and Exports

The OVM 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.

## Interface Overview

The TLM standard specifies the required behavior (semantic) of each interface method. Classes (components) that implement a TLM interface must meet the specified semantic.

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

*blocking* A blocking interface conveys transactions in blocking fashion; its methods do not return until the transaction has been successfully sent or retrieved. Because delivery may consume time to complete, the methods in such an interface are declared as tasks.

*non-blocking* A non-blocking interface attempts to convey a transaction without consuming simulation time. Its methods are declared as functions. Because delivery may fail (e.g. the target component is busy and can not accept the request), the methods may return with failed status.

*combination* A combination interface contains both the blocking and non-blocking variants. In SystemC, combination interfaces are defined through multiple inheritance. Because SystemVerilog does not support multiple inheritance, the OVM emulates hierarchical interfaces via a common base class and interface mask.

Like their SystemC counterparts, the OVM's TLM port and export implementations allow connections between ports whose interfaces are not an exact match. For example, an *ovm\_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 *ovm\_get\_\** ports and exports, *ovm\_blocking\_get\_peek\_\** ports and exports, and *ovm\_get\_peek\_\** ports and exports.

The sections below provide an overview of the unidirectional and bidirectional TLM interfaces, ports, and exports.

## Summary

### TLM Interfaces, Ports, and Exports

The OVM TLM library defines several abstract, transaction-level interfaces and the ports and exports that facilitate their use.

**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.

**Analysis** The *analysis* interface is used to perform non-blocking broadcasts of transactions to connected components.

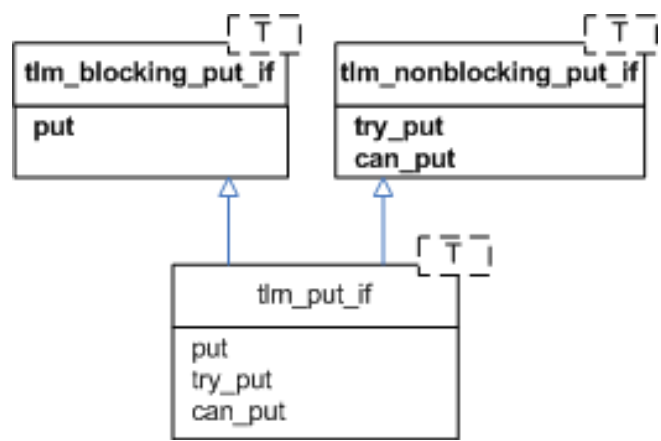
Ports, Exports, and Imps	The OVM provides unidirectional ports, exports, and implementation ports for connecting your components via the TLM interfaces.
Bidirectional Interfaces & Ports	The bidirectional interfaces consist of blocking, non-blocking, and combined blocking and non-blocking variants of the <i>transport</i> , <i>master</i> , and <i>slave</i> interfaces.
Transport	The <i>transport</i> interface sends a request transaction and returns a response transaction in a single task call, thereby enforcing an in-order execution semantic.
Master and Slave	The primitive, unidirectional <i>put</i> , <i>get</i> , and <i>peek</i> interfaces are combined to form bidirectional master and slave interfaces.
Ports, Exports, and Imps	The OVM provides bidirectional ports, exports, and implementation ports for connecting your components via the TLM interfaces.
Usage	We provide an example to illustrate 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

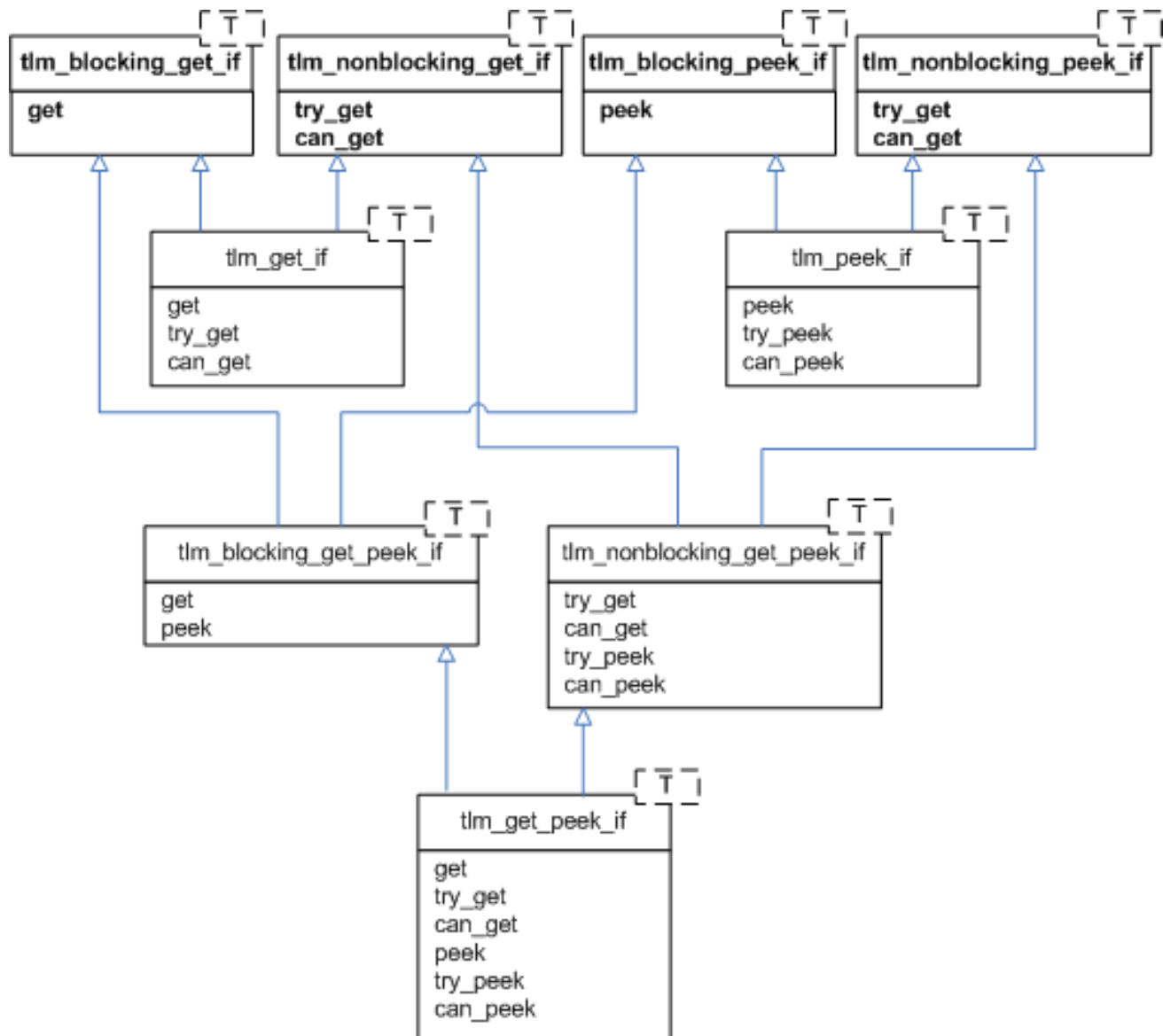
The *put* interfaces are used to send, or *put*, transactions to other components. Successful completion of a put guarantees its delivery, not execution.



### Get and Peek

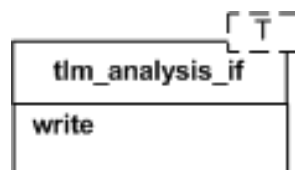
The *get* interfaces are used to retrieve transactions from other components. The *peek* interfaces are used for the same purpose, except the retrieved transaction is not consumed; successive calls to *peek* will return the same object. Combined *get\_peek* interfaces are also

defined.



## 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.



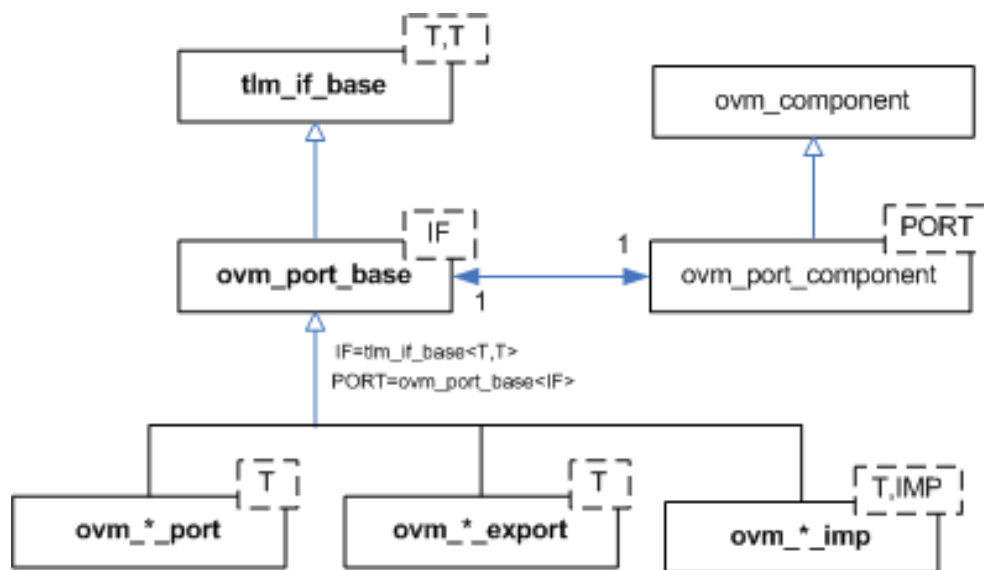
## Ports, Exports, and Imps

The OVM 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 ovm_*_export #(type T=int)
    extends ovm_port_base #(tlm_if_base #(T,T));

class ovm_*_port #(type T=int)
    extends ovm_port_base #(tlm_if_base #(T,T));

class ovm_*_imp #(type T=int)
    extends ovm_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

```

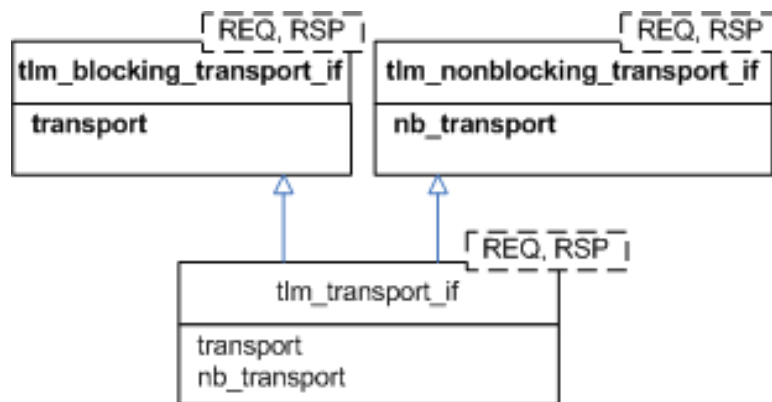
## Bidirectional Interfaces & Ports

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

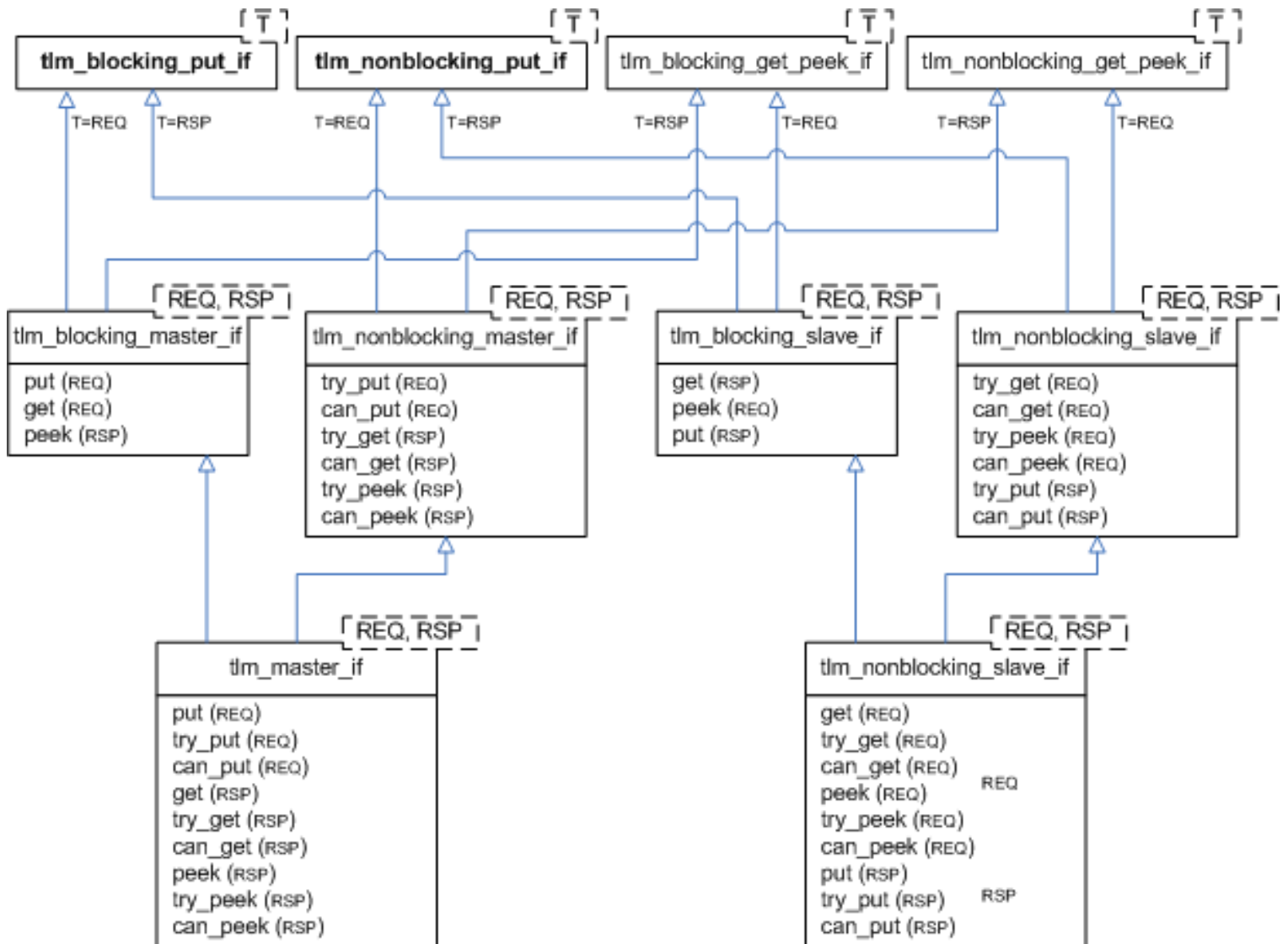
The *transport* interface sends a request transaction and returns a response transaction in a single task call, thereby enforcing an in-order execution semantic. The request and response transactions can be different types.



### 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.



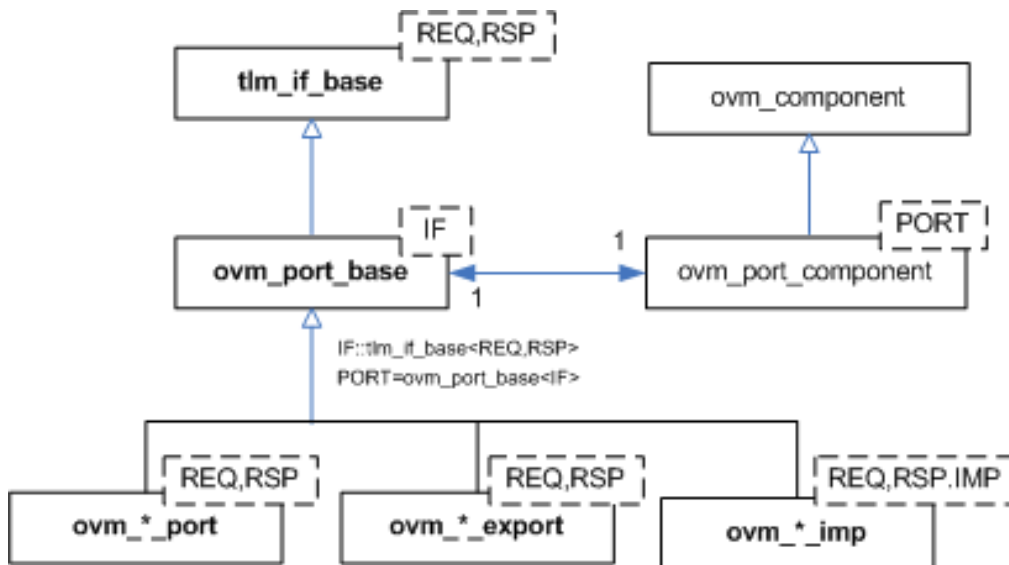
## Ports, Exports, and Imps

The OVM 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 ovm_*_port #(type REQ=int, RSP=int)
    extends ovm_port_base #(tlm_if_base #(REQ, RSP));

class ovm_*_export #(type REQ=int, RSP=int)
    extends ovm_port_base #(tlm_if_base #(REQ, RSP));

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

where the asterisk can be any of

```

transport
blocking_transport
nonblocking_transport

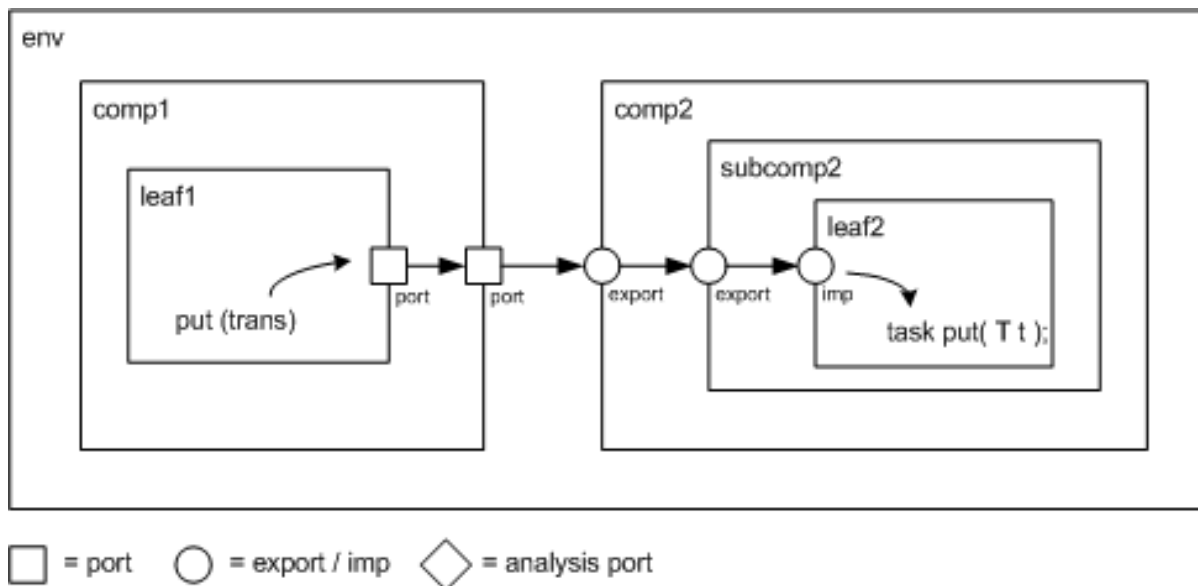
blocking_master
nonblocking_master
master

blocking_slave
nonblocking_slave
slave
  
```

## Usage

We provide an example to illustrate basic TLM connectivity using the blocking put interface.





*port-to-port*

leaf1's *out* port is connected to its parent's (comp1) *out* port

*port-to-export*

comp1's *out* port is connected to comp2's *in* export

*export-to-export*

comp2's *in* export is connected to its child's (subcomp2) *in* export

*export-to-imp*

subcomp2's *in* export is connected leaf2's *in* imp port.

*imp-to-implementation* leaf2's *in* imp port is connected to its implementation, leaf2

Hierarchical port connections are resolved and optimized just before the `ovm_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 "ovm_pkg.sv"
import ovm_pkg::*;

class trans extends ovm_transaction;
  rand int addr;
  rand int data;
  rand bit write;
endclass

class leaf1 extends ovm_component;

  `ovm_component_utils(leaf1)

  ovm_blocking_put_port #(trans) out;

  function new(string name, ovm_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
```

```

endclass

class compl extends ovm_component;

  `ovm_component_utils(compl)

  ovm_blocking_put_port #(trans) out;

  leaf1 leaf;

  function new(string name, ovm_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 ovm_component;

  `ovm_component_utils(leaf2)

  ovm_blocking_put_imp #(trans,leaf2) in;

  function new(string name, ovm_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 ovm_component;

  `ovm_component_utils(subcomp2)

  ovm_blocking_put_export #(trans) in;

  leaf2 leaf;

  function new(string name, ovm_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 ovm_component;

    `ovm_component_utils(comp2)

    ovm_blocking_put_export #(trans) in;

    subcomp2 subcomp;

    function new(string name, ovm_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 ovm_component;

    `ovm_component_utils(comp1)

    comp1 comp1_i;
    comp2 comp2_i;

    function new(string name, ovm_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 ovm_top.stop_request();
endmodule

```

## 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 OVM 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

#### tlm\_if\_base #(T1,T2)

This class declares all of the methods of the TLM API.

##### Class Declaration

```
virtual class 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.



## ovm\_\*\_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 *ovm\_\*\_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

analysis
```

Type parameters

*T* The type of transaction to be communicated by the export

Ports are connected to interface implementations directly via [ovm\\_\\*\\_imp #\(T,IMP\)](#) ports or indirectly via hierarchical connections to [ovm\\_\\*\\_port #\(T\)](#) and [ovm\\_\\*\\_export #\(T\)](#) ports.

## Summary

### ovm\_\*\_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 [ovm\\_component](#) constructor arguments.

## Methods

## new

The *name* and *parent* are the standard [ovm\\_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,
             ovm_component parent,
             int min_size=1,
             int max_size=1)
```

## ovm\_\*\_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 *ovm\_\*\_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 [ovm\\_\\*\\_imp #\(REQ,RSP,IMP,REQ\\_IMP,RSP\\_IMP\)](#) ports or indirectly via hierarchical connections to [ovm\\_\\*\\_port #\(REQ,RSP\)](#) and [ovm\\_\\*\\_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

## ovm\_\*\_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 **ovm\_component** constructor arguments.

## Methods

---

### new

---

The *name* and *parent* are the standard **ovm\_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, ovm\_component parent, int min\_size=1, int max\_size=1)

## ovm\_\*\_export #(T)

The unidirectional ovm\_\*\_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

analysis
```

Type parameters

*T* The type of transaction to be communicated by the export

Exports are connected to interface implementations directly via [ovm\\_\\*\\_imp #\(T,IMP\)](#) ports or indirectly via other [ovm\\_\\*\\_export #\(T\)](#) exports.

### Summary

#### ovm\_\*\_export #(T)

The unidirectional ovm\_\*\_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 [ovm\\_component](#) constructor arguments.

## Methods

## new

The *name* and *parent* are the standard [ovm\\_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,
             ovm_component parent,
             int min_size=1,
             int max_size=1)
```

## ovm\_\*\_export #(REQ,RSP)

The bidirectional ovm\_\*\_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 <ovm\_\*\_imp #(REQ,RSP,IMP) > ports or indirectly via other [ovm\\_\\*\\_export #\(REQ,RSP\)](#) exports.

### Summary

#### ovm\_\*\_export #(REQ,RSP)

The bidirectional ovm\_\*\_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 **ovm\_component** constructor arguments.

## Methods

---

### new

---

The *name* and *parent* are the standard **ovm\_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,  
             ovm_component parent,  
             int min_size=1,  
             int max_size=1)
```

## ovm\_\*\_imp ports

This page documents the following port classes

- `ovm_*_imp #(T,IMP)` - unidirectional implementation ports
- `ovm_*_imp #(REQ, RSP, IMP, REQ_IMP, RSP_IMP)` - bidirectional implementation ports

## Summary

### ovm\_\*\_imp ports

This page documents the following port classes

## ovm\_\*\_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 `ovm_*_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

analysis
```

### 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

### ovm\_\*\_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);
```

### ovm\_\*\_imp #(REQ, RSP, IMP, REQ\_IMP, RSP\_IMP)

Bidirectional implementation (imp) port classes--An imp port provides access to an implementation of the associated interface to all connected *ports* and *exports*. Each imp port instance *must* be connected to the component instance that implements the associated interface, typically the imp port's parent. All other connections-- e.g. to other ports and exports-- are prohibited.

The interface represented by the asterisk is any of the following

```
blocking_transport
nonblocking_transport
transport
```



```
blocking_master
nonblocking_master
master

blocking_slave
nonblocking_slave
slave
```

### Type parameters

*REQ* Request transaction type

*RSP* Response transaction type

*IMP* Component type that implements the interface methods, typically the the parent of this imp port.

*REQ\_IMP* Component type that implements the request side of the interface. Defaults to *IMP*. For master and slave imps only.

*RSP\_IMP* Component type that implements the response side of the interface. Defaults to *IMP*. For master and slave imps only.

The interface methods are implemented in a component of type *IMP*, a handle to which is passed in a constructor argument. The imp port delegates all interface calls to this component.

The master and slave imps have two modes of operation.

- A single component of type *IMP* implements the entire interface for both requests and responses.
- Two sibling components of type *REQ\_IMP* and *RSP\_IMP* implement the request and response interfaces, respectively. In this case, the *IMP* parent instantiates this imp port *and* the *REQ\_IMP* and *RSP\_IMP* components.

The second mode is needed when a component instantiates more than one imp port, as in the [tlm\\_req\\_rsp\\_channel #\(REQ,RSP\)](#) channel.

## Summary

### **ovm\*\_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)
```

## tlm\_fifo\_base #(T)

This class is the base for [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 [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

#### tlm\_fifo\_base #(T)

This class is the base for [tlm\\_fifo #\(T\)](#).

##### Class Hierarchy



##### Class Declaration

```

virtual class tlm_fifo_base #(
    type      T = int
) extends ovm_component
  
```

##### Ports

<a href="#">put_export</a>	The <i>put_export</i> provides both the blocking and non-blocking put interface methods to any attached port:
<a href="#">get_peek_export</a>	The <i>get_peek_export</i> provides all the blocking and non-blocking get and peek interface methods:
<a href="#">put_ap</a>	Transactions passed via <i>put</i> or <i>try_put</i> (via any port connected to the <a href="#">put_export</a> ) are sent out this port via its <i>write</i> method.
<a href="#">get_ap</a>	Transactions passed via <i>get</i> , <i>try_get</i> , <i>peek</i> , or <i>try_peek</i> (via any port connected to the <a href="#">get_peek_export</a> ) are sent out this port via its <i>write</i> method.

##### Methods

<a href="#">new</a>	The <i>name</i> and <i>parent</i> are the normal ovm_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 [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 [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 [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 [tlm\\_if\\_base #\(T1, T2\)](#) for more information on the *write* method.

## Methods

---

### new

---

```
function new(string      name,
              ovm_component parent = null
              )
```

The *name* and *parent* are the normal ovm\_component constructor arguments. The *parent* should be null if the 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\_fifo #(T)

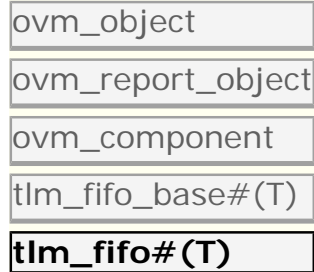
This class provides storage of transactions between two independently running processes. Transactions are put into the FIFO via the *put\_export*. transactions are fetched from the FIFO in the order they arrived via the *get\_peek\_export*. The *put\_export* and *get\_peek\_export* are inherited from the [tlm\\_fifo\\_base #\(T\)](#) super class, and the interface methods provided by these exports are defined by the [tlm\\_if\\_base #\(T1,T2\)](#) class.

### Summary

#### tlm\_fifo #(T)

This class provides storage of transactions between two independently running processes.

##### Class Hierarchy



##### Class Declaration

```

class tlm_fifo #(
    type      T = int
) extends tlm_fifo_base #(T)
  
```

##### Methods

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

## Methods

### new

```

function new(string      name,
              ovm_component parent = null,
              int         size    = 1
              )
  
```

The *name* and *parent* are the normal ovm\_component constructor arguments. The *parent*

should be null if the 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.

# tlm\_analysis\_fifo #(T)

An analysis\_fifo is a tlm\_fifo with an unbounded size and a write interface. It can be used any place an <ovm\_subscriber #(T)> is used. Typical usage is as a buffer between an analysis\_port in a monitor and an analysis component (e.g., a component derived from ovm\_subscriber).

## Summary

### tlm\_analysis\_fifo #(T)

An analysis\_fifo is a tlm\_fifo with an unbounded size and a write interface.

#### Class Hierarchy



#### Class Declaration

```

class tlm_analysis_fifo #(
    type      T =    int
) extends tlm_fifo #(T)
  
```

#### Ports

[analysis\\_port #\(T\)](#) The analysis\_export provides the write method to all connected analysis ports and parent exports:

#### Methods

[new](#) This is the standard ovm\_component constructor.

## Ports

### analysis\_port #(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 [tlm\\_if\\_base #\(T1,T2\)](#) for more information.



## Methods

---

### new

---

```
function new(string name          ,  
             ovm_component parent = null )
```

This is the standard ovm\_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 OVM component.

## tlm\_req\_rsp\_channel #(REQ,RSP)

The 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

## tlm\_req\_rsp\_channel #(REQ,RSP)

The tlm\_req\_rsp\_channel contains a request FIFO of type *REQ* and a response FIFO of type *RSP*.

### Class Hierarchy



### Class Declaration

```

class tlm_req_rsp_channel #(
    type    REQ    =    int,
    type    RSP    =    REQ
) extends ovm_component
  
```

### Ports

<a href="#">put_request_export</a>	The put_export provides both the blocking and non-blocking put interface methods to the request FIFO:
<a href="#">get_peek_response_export</a>	The get_peek_response_export provides all the blocking and non-blocking get and peek interface methods to the response FIFO:
<a href="#">get_peek_request_export</a>	The get_peek_export provides all the blocking and non-blocking get and peek interface methods to the response FIFO:
<a href="#">put_response_export</a>	The put_export provides both the blocking and non-blocking put interface methods to the response FIFO:
<a href="#">request_ap</a>	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.
<a href="#">response_ap</a>	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.
<a href="#">master_export</a>	Exports a single interface that allows a master to put requests and get or peek responses.
<a href="#">slave_export</a>	Exports a single interface that allows a slave to get or peek requests and to put responses.

### Methods

<a href="#">new</a>	The <i>name</i> and <i>parent</i> are the standard <a href="#">ovm_component</a> 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,
              ovm_component parent      = null,
              int         request_fifo_size = 1,
              int         response_fifo_size = 1 )
```

The *name* and *parent* are the standard `ovm_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.

## tlm\_transport\_channel #(REQ,RSP)

A `tlm_transport_channel` is a `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

#### tlm\_transport\_channel #(REQ,RSP)

A `tlm_transport_channel` is a `tlm_req_rsp_channel #(REQ,RSP)` that implements the transport interface.

#### Class Hierarchy

tlm\_req\_rsp\_channel #(REQ,RSP)

ovm\_object

ovm\_report\_object

ovm\_component

tlm\_req\_rsp\_channel #(REQ,RSP)

**tlm\_transport\_channel #(REQ,RSP)**

### Class Declaration

```
class tlm_transport_channel #(
    type      REQ    =    int,
    type      RSP    =    REQ
) extends 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 **ovm\_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

---

tlm\_req\_rsp\_channel #(REQ,RSP)

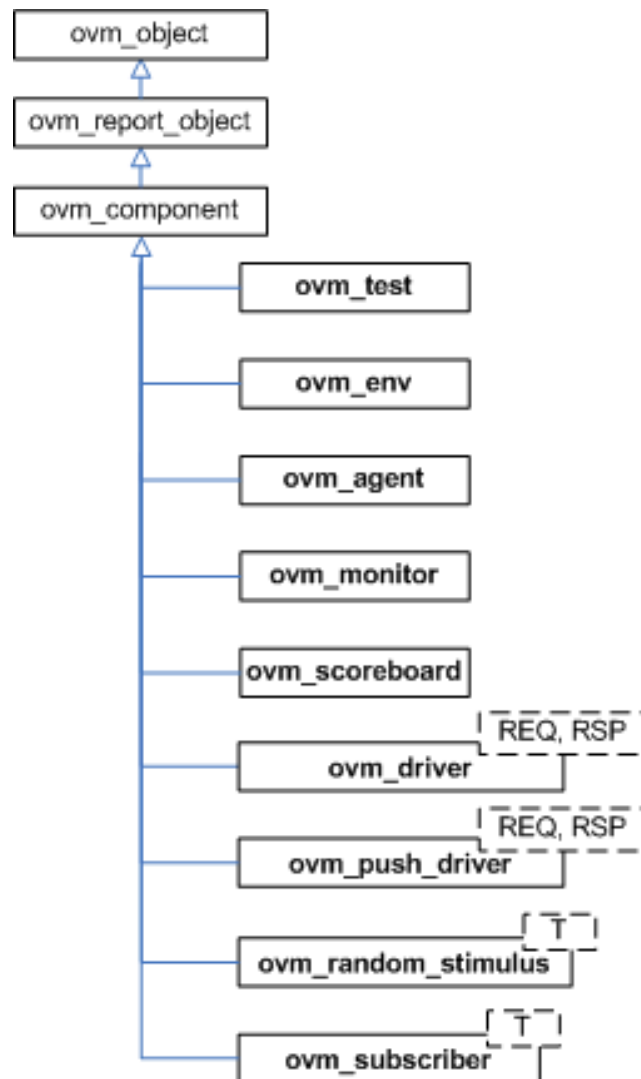
```
function new (string      name,  
              ovm_component parent = null  
              )
```

The *name* and *parent* are the standard [ovm\\_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.

# Predefined Component Classes

Components form the foundation of the OVM. They encapsulate behavior of drivers, scoreboards, and other objects in a testbench. The OVM library provides a set of predefined component types, all derived directly or indirectly from [ovm\\_component](#).

## Predefined Components





## ovm\_test

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

The ovm\_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 OVM\_TESTNAME command line or argument to the [ovm\\_root::run\\_test](#) task.

For example

```
prompt> SIM_COMMAND +OVM_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 +OVM\_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 [ovm\\_factory](#), then a fatal error occurs. If run\_test() is called without OVM\_TESTNAME being specified, then all components constructed before the call to run\_test will be cycled through their simulation phases.

Deriving from ovm\_test will allow you to distinguish tests from other component types that inherit from ovm\_component directly. Such tests will automatically inherit features that may be added to ovm\_test in the future.

### Summary

#### ovm\_test

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

#### Class Hierarchy

ovm_object
ovm_report_object
ovm_component
<b>ovm_test</b>

### Class Declaration

```
virtual class ovm_test extends ovm_component
```

### Methods

**new** Creates and initializes an instance of this class using the normal constructor arguments for **ovm\_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,
              ovm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for **ovm\_component**: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

## ovm\_env

The base class for hierarchical containers of other components that together comprise a complete environment. The environment may initially consist of the entire testbench. Later, it can be reused as a sub-environment in even larger system-level environments.

### Summary

#### ovm\_env

The base class for hierarchical containers of other components that together comprise a complete environment.

#### Class Hierarchy



#### Class Declaration

```
virtual class ovm_env extends ovm_component
```

#### Methods

**new** Creates and initializes an instance of this class using the normal constructor arguments for **ovm\_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",
              ovm_component parent = null)
```

Creates and initializes an instance of this class using the normal constructor arguments for **ovm\_component**: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

# ovm\_agent

The ovm\_agent virtual class should be used as the base class for the user- defined agents. Deriving from ovm\_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 ovm\_agent in the future.

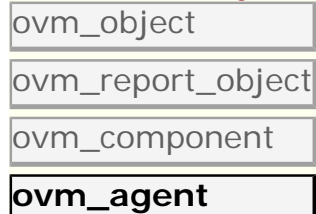
While an agent's build function, inherited from [ovm\\_component](#), can be implemented to define any agent topology, an agent typically contains three subcomponents: a driver, sequencer, and monitor. If the agent is active, subtypes should contain all three subcomponents. If the agent is passive, subtypes should contain only the monitor.

## Summary

### ovm\_agent

The ovm\_agent virtual class should be used as the base class for the user- defined agents.

#### Class Hierarchy



#### Class Declaration

```
virtual class ovm_agent extends ovm_component
```

#### Methods

**new** Creates and initializes an instance of this class using the normal constructor arguments for [ovm\\_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,
              ovm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for [ovm\\_component](#): *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

# ovm\_monitor

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

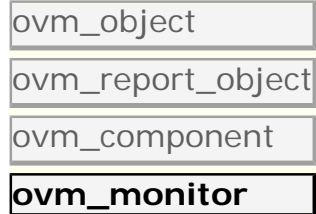
Deriving from ovm\_monitor allows you to distinguish monitors from generic component types inheriting from ovm\_component. Such monitors will automatically inherit features that may be added to ovm\_monitor in the future.

## Summary

### ovm\_monitor

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

#### Class Hierarchy



#### Class Declaration

```
virtual class ovm_monitor extends ovm_component
```

#### Methods

**new** Creates and initializes an instance of this class using the normal constructor arguments for **ovm\_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,
              ovm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for **ovm\_component**: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

# ovm\_scoreboard

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

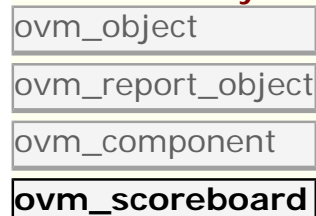
Deriving from ovm\_scoreboard will allow you to distinguish scoreboards from other component types inheriting directly from ovm\_component. Such scoreboards will automatically inherit and benefit from features that may be added to ovm\_scoreboard in the future.

## Summary

### ovm\_scoreboard

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

#### Class Hierarchy



#### Class Declaration

```
virtual class ovm_scoreboard extends ovm_component
```

#### Methods

**new** Creates and initializes an instance of this class using the normal constructor arguments for **ovm\_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,
              ovm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for **ovm\_component**: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

## ovm\_driver #(REQ,RSP)

The base class for drivers that initiate requests for new transactions via a `ovm_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

#### ovm\_driver #(REQ,RSP)

The base class for drivers that initiate requests for new transactions via a `ovm_seq_item_pull_port`.

##### Class Hierarchy

ovm\_object

ovm\_report\_object

ovm\_component

**ovm\_driver#(REQ,RSP)**

##### Class Declaration

```
class ovm_driver #(
    type REQ = ovm_sequence_item,
    type RSP = REQ
) extends ovm_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 **ovm\_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,  
             ovm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for [ovm\\_component](#): *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.



## ovm\_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

## ovm\_push\_driver #(REQ,RSP)

Base class for a driver that passively receives transactions, i.e.

### Class Hierarchy

ovm\_object

ovm\_report\_object

ovm\_component

**ovm\_push\_driver#(REQ,RSP)**

### Class Declaration

```
class ovm_push_driver #(
    type REQ = ovm_sequence_item,
    type RSP = REQ
) extends ovm_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 **ovm\_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,  
             ovm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for [ovm\\_component](#): *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

## ovm\_random\_stimulus #(T)

A general purpose unidirectional random stimulus class.

The ovm\_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.

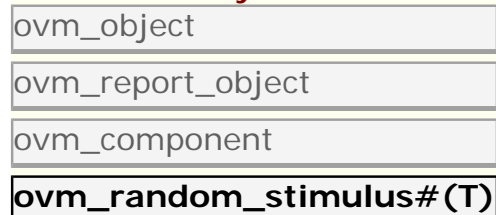
While very useful in its own right, this component can also be used as a template for defining other stimulus generators, or it can be extended to add additional stimulus generation methods and to simplify test writing.

### Summary

#### ovm\_random\_stimulus #(T)

A general purpose unidirectional random stimulus class.

##### Class Hierarchy



##### Class Declaration

```
class ovm_random_stimulus #(
    type T = ovm_transaction
) extends ovm_component
```

##### Ports

**blocking\_put\_port** The blocking\_put\_port is used to send the generated stimulus to the rest of the testbench.

##### Methods

**new** Creates a new instance of a specialization of this class.

**generate\_stimulus** Generate up to max\_count transactions of type T.

**stop\_stimulus\_generation** Stops the generation of stimulus.

## Ports

## blocking\_put\_port

---

The blocking\_put\_port is used to send the generated stimulus to the rest of the testbench.

## Methods

---

### new

---

```
function new(string      name,
             ovm_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

# ovm\_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.

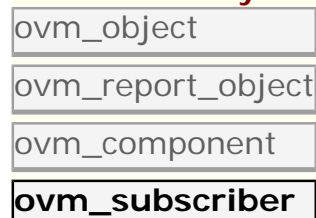
Subtypes of this class must define the write method to process the incoming transactions. This class is particularly useful when designing a coverage collector that attaches to a monitor.

## Summary

### ovm\_subscriber

This class provides an analysis export for receiving transactions from a connected analysis export.

#### Class Hierarchy



#### Class Declaration

```

virtual class ovm_subscriber #(
    type      T = int
) extends ovm_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 **ovm\_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,  
             ovm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for [ovm\\_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)
```

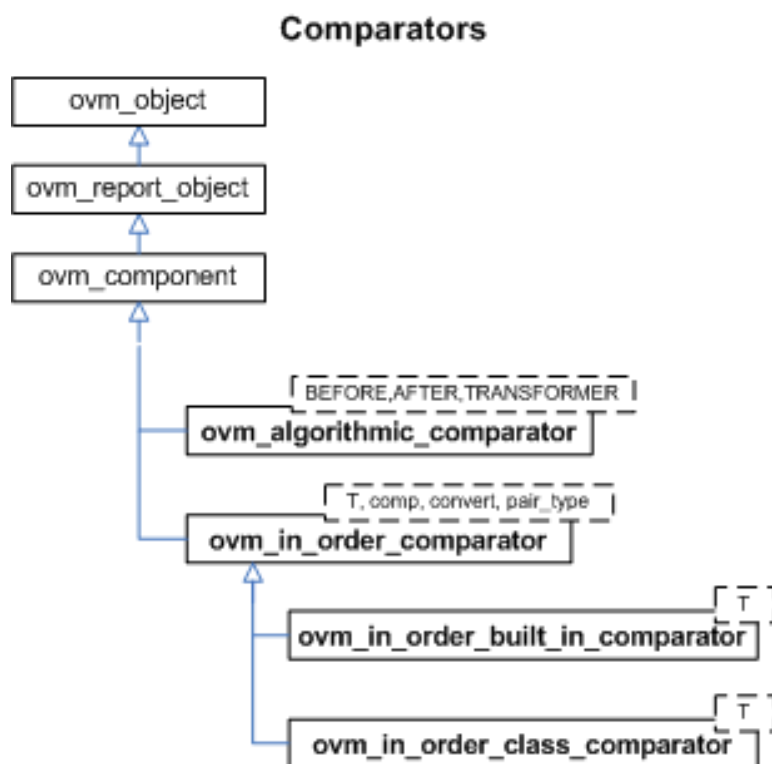
A pure virtual method that must be defined in each subclass. Access to this method by outside components should be done via the `analysis_export`.

# 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 OVM library provides a base class called *ovm\_in\_order\_comparator* and two derived classes: *ovm\_in\_order\_built\_in\_comparator* for comparing streams of built-in types and *ovm\_in\_order\_class\_comparator* for comparing streams of class objects.

The *ovm\_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.



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

Compares two streams of data objects of type *T*, a parameter to this class. These transactions may either be classes or built-in types. To be successfully compared, the two streams of data must be in the same order. Apart from that, there are no assumptions made about the relative timing of the two streams of data.

### Type parameters

- T* Specifies the type of transactions to be compared.
- comp* The type of the comparator to be used to compare the two transaction streams.
- convert* A policy class to allow `convert2string()` to be called on the transactions being compared. If *T* is an extension of `ovm_transaction`, then it uses `T::convert2string()`. If *T* is a built-in type, then the policy provides a `convert2string()` method for the comparator to call.
- pair\_type* A policy class to allow pairs of transactions to be handled as a single `ovm_transaction` 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, `<ovm_in_order_builtin_comparator #(T)>` for built-in types.

When *T* is a class, *T* must implement *comp* and `convert2string`, and you must specify class-based policy classes for *comp\_type*, *convert*, and *pair\_type*. In most cases, you can use the convenient subtype, `ovm_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

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

Compares two streams of data objects of type *T*, a parameter to this class.

#### Ports

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

#### Methods

- `flush` This method sets `m_matches` and `m_mismatches` back to zero.

## Ports



## before\_export

---

The export to which one stream of data is written. The port must be connected to an analysis port that will provide such data.

## after\_export

---

The export to which the other stream of data is written. The port must be connected to an analysis port that will provide such data.

## pair\_ap

---

The comparator sends out pairs of transactions across this analysis port. Both matched and unmatched pairs are published via a pair\_type objects. Any connected analysis export(s) will receive these transaction pairs.

# Methods

---

## flush

---

```
virtual function void flush()
```

This method sets m\_matches and m\_mismatches back to zero. The `tlm_fifo #(T)::flush` takes care of flushing the FIFOs.

## in\_order\_built\_in\_comparator #(T)

This class uses the ovm\_built\_in\_\* comparison, converter, and pair classes. Use this class for built-in types (int, bit, string, etc.)

### Summary

## in\_order\_built\_in\_comparator #(T)

This class uses the ovm\_built\_in\_\* comparison, converter, and pair classes.

**Class Hierarchy**

ovm\_in\_order\_comparator#(T)

**in\_order\_built\_in\_comparator#(T)****Class Declaration**

```
class ovm_in_order_built_in_comparator #(
    type      T      =      int
) extends ovm_in_order_comparator #(T)
```

**in\_order\_class\_comparator #(T)**

This class uses the ovm\_class\_\* comparison, converter, and pair classes. Use this class for comparing user-defined objects of type T, which must provide implementations of comp and convert2string.

## ovm\_algorithmic\_comparator.svh

### Summary

#### ovm\_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 OVM library provides a base class called `ovm_in_order_comparator` and two derived classes, which are `ovm_in_order_built_in_comparator` for comparing streams of built-in types and `ovm_in_order_class_comparator` for comparing streams of class objects.

The `ovm_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.

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

Compares two streams of data objects of different types, BEFORE and AFTER.

The algorithmic comparator is a wrapper around `ovm_in_order_class_comparator`. 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 `ovm_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 `ovm_in_order_comparator #(...)` class.

### Summary

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

Compares two streams of data objects of different types, BEFORE and AFTER.

#### Class Hierarchy

ovm_object
ovm_report_object
ovm_component
<b>ovm_algorithmic_comparator#(BEFORE,AFTER,TRANSFORMER)</b>

### Class Declaration

```
class ovm_algorithmic_comparator #(
    type    BEFORE      = int,
    type    AFTER       = int,
    type    TRANSFORMER = int
) extends ovm_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(
    TRANSFORMER transformer,
    string name      ,
    ovm_component parent )
```

Creates an instance of a specialization of this class. In addition to the standard `ovm_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.

## ovm\_pair #(T1,T2)

Container holding handles to two objects whose types are specified by the type parameters, T1 and T2.

### Summary

#### ovm\_pair #(T1,T2)

Container holding handles to two objects whose types are specified by the type parameters, T1 and T2.

#### Methods

**new** Creates an instance of ovm\_pair that holds a handle to two objects, as provided by the first two arguments.

## Methods

### new

```
function new (T1      f      = null,
              T2      s      = null,
              string name = " ")
```

Creates an instance of ovm\_pair that holds a handle to two objects, as provided by the first two arguments. The optional *name* argument gives a name to the new pair object.

## ovm\_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

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

Container holding two variables of built-in types (int, string, etc.)

#### Class Hierarchy

ovm\_object

ovm\_transaction

**ovm\_built\_in\_pair#(T1,T2)****Class Declaration**

```
class ovm_built_in_pair #(
    type    T1    =    int,
           T2    =    T1
) extends ovm_transaction
```

**Methods**

**new** Creates an instance of ovm\_pair that holds a handle to two elements, as provided by the first two arguments.

## Methods

---

**new**

```
function new (T1      f,
              T2      s,
              string name = " ")
```

Creates an instance of ovm\_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.

# ovm\_policies.svh

## Summary

### ovm\_policies.svh

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

## 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.

## ovm\_built\_in\_comp #(T)

This policy class is used to compare built-in types.

Provides a comp method that compares, AVM-style, the built-in type, T, for which the == operator is defined.

## Summary

### ovm\_built\_in\_comp #(T)

This policy class is used to compare built-in types.

#### Class Declaration

```
class ovm_built_in_comp #(type T = int )
```

## ovm\_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



## ovm\_built\_in\_converter #(T)

This policy class is used to convert built-in types to strings.

### Class Declaration

```
class ovm_built_in_converter #(type T = int )
```

## ovm\_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

## ovm\_built\_in\_clone #(T)

This policy class is used to clone built-in types via the = operator.

### Class Declaration

```
class ovm_built_in_clone #(type T = int )
```

## ovm\_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 implement the comp method, to which this class delegates the operation.

### Summary

## ovm\_class\_comp #(T)

This policy class is used to compare two objects of the same type.

### Class Declaration

```
class ovm_class_comp #(type T = int )
```

## ovm\_class\_converter #(T)

This policy class is used to convert a class object to a string.

Provides a `convert2string` method that converts the built-in type, `T`, to a string. The class `T` must implement the `convert2string` method, to which this class delegates the operation.

## Summary

### **ovm\_class\_converter #(T)**

This policy class is used to convert a class object to a string.

#### **Class Declaration**

```
class ovm_class_converter #(type T = int )
```

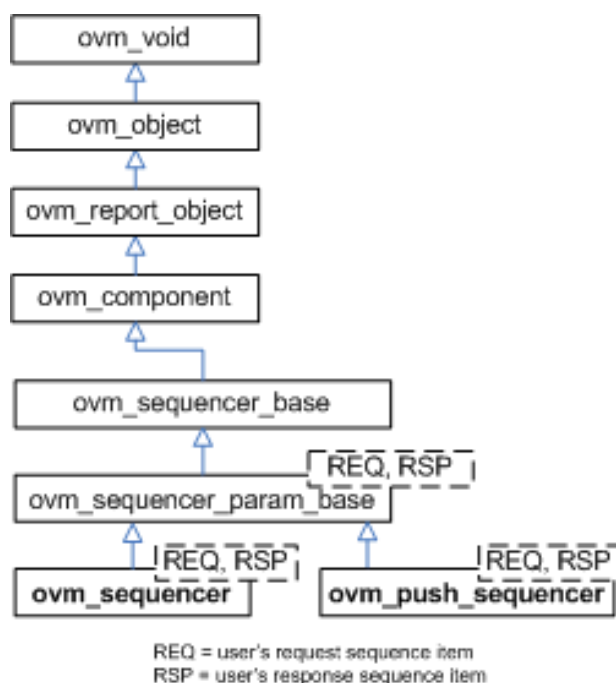
### **ovm\_class\_clone #(T)**

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.

# Sequencer Classes

The sequencer serves as an arbiter for controlling transaction flow from multiple stimulus generators. More specifically, the sequencer controls the flow of `ovm_sequence_item`-based transactions generated by one or more `ovm_sequence #(REQ,RSP)`-based sequences.



There are two sequencer variants available.

- `ovm_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 `ovm_driver #(REQ,RSP)`.
- `ovm_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 `ovm_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.

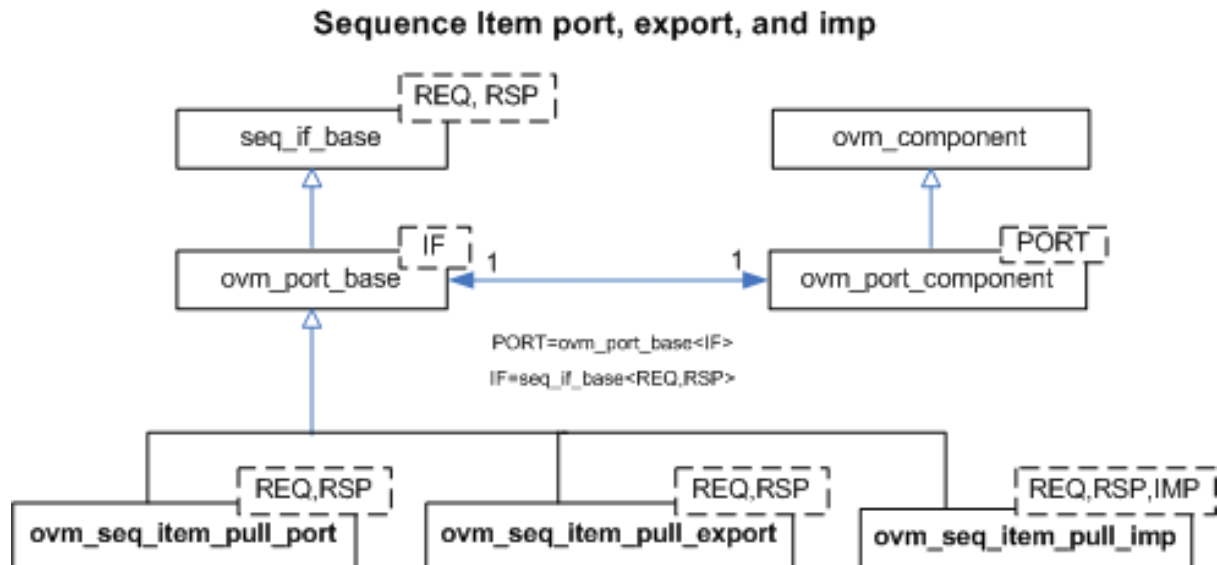
See [Sequence Classes](#) for an overview on sequences and sequence items.

## Sequence Item Ports

As with all OVM components, the sequencers and drivers described above use [TLM Interfaces](#),

Ports, and Exports to communicate transactions.

The `ovm_sequencer #(REQ,RSP)` and `ovm_driver #(REQ,RSP)` pair also uses a *sequence item pull port* to achieve the special execution semantic needed by the sequencer-driver pair.



sequencers and drivers use a *seq\_item\_port* specifically supports sequencer-driver communication. Connections to these ports are made in the same fashion as the TLM ports.

## 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

#### sqr\_if\_base #(REQ,RSP)

This class defines an interface for sequence drivers to communicate with sequencers.

##### Class Declaration

```
virtual class sqr_if_base #(type T1 = ovm_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</a>	Retrieves the next available item from a sequence.
<a href="#">peek</a>	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:

- 1Arbitrate 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.
- 2The chosen sequence will return from wait\_for\_grant
- 3The chosen sequence pre\_do is called
- 4The chosen sequence item is randomized
- 5The chosen sequence post\_do is called
- 6Return 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:

- 1Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, return null.
- 2The chosen sequence will return from wait\_for\_grant
- 3The chosen sequence pre\_do is called
- 4The chosen sequence item is randomized
- 5The chosen sequence post\_do is called
- 6Return 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 wait\_for\_item\_done calls made by a sequence for this item will return.

The current item is removed from the sequencer fifo.

If a response item is provided, then it will be sent back to the requesting sequence. The response item must have it's sequence ID and transaction ID set correctly, using the 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 pound\_zero\_count delta cycles. (This variable is defined in ovm\_sequencer\_base.) 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:

- 1Arbitrate 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.
- 2The chosen sequence will return from wait\_for\_grant
- 3The chosen sequence <pre\_do> is called
- 4The chosen sequence item is randomized
- 5The chosen sequence post\_do is called
- 6Indicate item\_done to the sequencer
- 7Return 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 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:

- 1Arbitrate 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.
- 2The chosen sequence will return from wait\_for\_grant
- 3The chosen sequence pre\_do is called
- 4The chosen sequence item is randomized
- 5The chosen sequence 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 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.



## ovm\_seq\_item\_pull\_port #(REQ,RSP)

OVM provides a port, export, and imp connector for use in sequencer-driver communication. All have standard port connector constructors, except that ovm\_seq\_item\_pull\_port's default min\_size argument is 0; it can be left unconnected.

### Summary

#### ovm\_seq\_item\_pull\_port #(REQ,RSP)

OVM provides a port, export, and imp connector for use in sequencer-driver communication.

##### Class Hierarchy

```
ovm_port_base#(sqr_if_base#(REQ,RSP))
```

```
ovm_seq_item_pull_port#(REQ,RSP)
```

##### Class Declaration

```
class ovm_seq_item_pull_port #(
    type      REQ      =    int,
    type      RSP      =    REQ
) extends ovm_port_base #(sqr_if_base #(REQ, RSP))
```

## ovm\_seq\_item\_pull\_export #(REQ,RSP)

This export type is used in sequencer-driver communication. It has the standard constructor for exports.

### Summary

#### ovm\_seq\_item\_pull\_export #(REQ,RSP)

This export type is used in sequencer-driver communication.

##### Class Hierarchy

```
ovm_port_base#(sqr_if_base#(REQ,RSP))
```

```
ovm_seq_item_pull_export#(REQ,RSP)
```

##### Class Declaration

```
class ovm_seq_item_pull_export #(
    type      REQ      =    int,
    type      RSP      =    REQ
) extends ovm_port_base #(sqr_if_base #(REQ, RSP))
```

## ovm\_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

### ovm\_seq\_item\_pull\_imp #(REQ,RSP,IMP)

This imp type is used in sequencer-driver communication.

#### Class Hierarchy

```
ovm_port_base#(sqr_if_base#(REQ,RSP))
```

```
ovm_seq_item_pull_imp#(REQ,RSP,IMP)
```

#### Class Declaration

```
class ovm_seq_item_pull_imp #(
    type      REQ      =    int,
    type      RSP      =    REQ,
    type      IMP      =    int
) extends ovm_port_base #(sqr_if_base #(REQ, RSP))
```

**end**

---

# ovm\_sequencer\_base

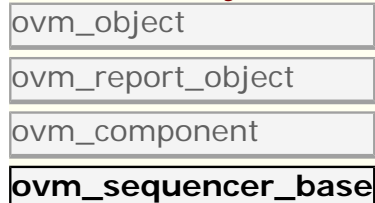
Controls the flow of sequences, which generate the stimulus (sequence item transactions) that is passed on to drivers for execution.

## Summary

### ovm\_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 ovm_sequencer_base extends ovm_component
```

#### Variables

<code>pound_zero_count</code>	Set this variable via <code>set_config_int</code> to set the number of delta cycles to insert in the <code>wait_for_sequences</code> task.
<code>count</code>	Sets the number of items to execute.
<code>max_random_count</code>	Set this variable via <code>set_config_int</code> to set the number of sequence items to generate, at the discretion of the derived sequence.
<code>max_random_depth</code>	Used for setting the maximum depth inside random sequences.
<code>default_sequence</code>	This property defines the sequence type (by name) that will be auto-started.

#### Methods

<code>new</code>	Creates and initializes an instance of this class using the normal constructor arguments for <code>ovm_component</code> : <code>name</code> is the name of the instance, and <code>parent</code> is the handle to the hierarchical parent.
<code>start_default_sequence</code>	Sequencers provide the <code>start_default_sequence</code> task to execute the default sequence in the run phase.
<code>user_priority_arbitration</code>	If the sequencer arbitration mode is set to <code>SEQ_ARB_USER</code> (via the <code>set_arbitration</code> method), then the sequencer will call this function each time that it needs to arbitrate among sequences.
<code>is_child</code>	Returns 1 if the child sequence is a child of the parent sequence, 0 otherwise.
<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>	Determines if a sequence is ready to supply a transaction.
<code>set_arbitration</code>	Specifies the arbitration mode for the sequencer.

<code>wait_for_sequences</code>	Waits for a sequence to have a new item available.
<code>add_sequence</code>	Adds a sequence of type specified in the <code>type_name</code> paramter to the sequencer's sequence library.
<code>get_seq_kind</code>	Returns an int <code>seq_kind</code> correlating to the sequence of type <code>type_name</code> in the sequencer's sequence library.
<code>get_sequence</code>	Returns a reference to a sequence specified by the <code>seq_kind</code> int.
<code>num_sequences</code>	Returns the number of sequences in the sequencer's sequence library.
<code>send_request</code>	Derived classes implement this function to send a request item to the sequencer, which will forward it to the driver.

## Variables

---

### `pound_zero_count`

---

```
int unsigned pound_zero_count = 6
```

Set this variable via `set_config_int` to set the number of delta cycles to insert in the `wait_for_sequences` task. The delta cycles are used to ensure that a sequence with back-to-back items has an opportunity to fill the action queue when the driver uses the non-blocking `try_get` interface.

### `count`

---

```
int count = -1
```

Sets the number of items to execute.

Supersedes the `max_random_count` variable for `ovm_random_sequence` class for backward compatibility.

### `max_random_count`

---

```
int unsigned max_random_count = 10
```

Set this variable via `set_config_int` to set the number of sequence items to generate, at the discretion of the derived sequence. The predefined `ovm_random_sequence` uses `count` to determine the number of random items to generate.

### `max_random_depth`

---

```
int unsigned max_random_depth = 4
```

Used for setting the maximum depth inside random sequences. (Beyond that depth, random

creates only simple sequences.)

## default\_sequence

---

```
protected string default_sequence = "ovm_random_sequence"
```

This property defines the sequence type (by name) that will be auto-started. The default sequence is initially set to `ovm_random_sequence`. It can be configured through the `ovm_component`'s `set_config_string` method using the field name "default\_sequence".

## Methods

---

### new

---

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

Creates and initializes an instance of this class using the normal constructor arguments for `ovm_component`: `name` is the name of the instance, and `parent` is the handle to the hierarchical parent.

### start\_default\_sequence

---

```
virtual task start_default_sequence()
```

Sequencers provide the `start_default_sequence` task to execute the default sequence in the run phase. This method is not intended to be called externally, but may be overridden in a derivative sequencer class if special behavior is needed when the default sequence is started. The user class `ovm_sequencer_param_base #(REQ,RSP)` implements this method.

### user\_priority\_arbitration

---

```
virtual function integer user_priority_arbitration(integer avail_sequences[$])
```

If the sequencer arbitration mode is set to `SEQ_ARB_USER` (via the `set_arbitration` method), then 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. Such an override must return one of the entries from the `avail_sequences` queue, which are int indexes into an internal queue, `arb_sequence_q`.

The default implementation behaves like `SEQ_ARB_FIFO`, which returns the entry at `avail_sequences[0]`.

If a user specifies that the sequencer is to use `user_priority_arbitration` through the call `set_arbitration(SEQ_ARB_USER)`, then the sequencer will call this function each time that it needs to arbitrate among sequences.

This function must return an int that matches one of the available sequences that is passed into the call through the `avail_sequences` parameter

Each int in `avail_sequences` points to an entry in the `arb_sequence_q`, which is a protected queue that may be accessed from this function.

To modify the operation of `user_priority_arbitration`, the function may arbitrarily choose any sequence among the list of `avail_sequences`. It is important to choose only an available sequence.

## is\_child

---

```
function bit is_child (ovm_sequence_base parent,
                      ovm_sequence_base child )
```

Returns 1 if the child sequence is a child of the parent sequence, 0 otherwise.

## wait\_for\_grant

---

```
virtual task wait_for_grant(ovm_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(ovm_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(ovm_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(ovm_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(ovm_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(ovm_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(ovm_sequence_base sequence_ptr)
```

Removes any locks and grabs obtained by the specified sequence\_ptr.

## ungrab

---

```
virtual function void ungrab(ovm_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 ovm_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()
```

Determines if a sequence is ready to supply a transaction. A sequence that obtains a transaction in pre-do must determine if the upstream object is ready to provide an item

Returns 1 if a sequence is ready to issue an operation. Returns 0 if no unblocked, relevant sequence is requesting.



## 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, <code>user_priority_arbitration</code> . That function will specify the next sequence to grant.

The default user function specifies FIFO order.

## 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 `pound_zero_count` delta cycles. (This variable is defined in `ovm_sequencer_base`.) User-derived sequencers may override its `wait_for_sequences` implementation to perform some other application-specific implementation.

## add\_sequence

---

```
function void add_sequence(string type_name)
```

Adds a sequence of type specified in the `type_name` paramter to the sequencer's sequence library.

## get\_seq\_kind

---

```
function int get_seq_kind(string type_name)
```

Returns an int `seq_kind` correlating to the sequence of type `type_name` in the sequencer's sequence library. If the named sequence is not registered a SEQNF warning is issued and -1 is returned.

## get\_sequence

---

```
function ovm_sequence_base get_sequence(int req_kind)
```

Returns a reference to a sequence specified by the `seq_kind` int. The `seq_kind` int may be obtained using the `get_seq_kind()` method.

## num\_sequences

---

```
function int num_sequences()
```

Returns the number of sequences in the sequencer's sequence library.

## send\_request

---

```
virtual function void send_request(ovm_sequence_base sequence_ptr,  
                                   ovm_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.

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

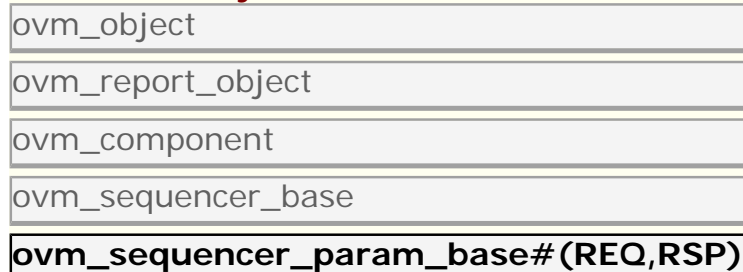
Provides base functionality used by the ovm\_sequencer and ovm\_push\_sequencer. The implementation is dependent on REQ and RSP parameters.

## Summary

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

Provides base functionality used by the ovm\_sequencer and ovm\_push\_sequencer.

#### Class Hierarchy



#### Class Declaration

```

class ovm_sequencer_param_base #(
    type REQ = ovm_sequence_item,
    type RSP = REQ
) extends ovm_sequencer_base
  
```

#### Ports

**rsp\_export** This is the analysis export used by drivers or monitors to send responses to the sequencer.

#### Methods

<b>new</b>	Creates and initializes an instance of this class using the normal constructor arguments for ovm_component: name is the name of the instance, and parent is the handle to the hierarchical parent, if any.
<b>send_request</b>	The send_request function may only be called after a wait_for_grant call.
<b>get_current_item</b>	Returns the request_item currently being executed by the sequencer.
<b>start_default_sequence</b>	Called when the run phase begins, this method starts the default sequence, as specified by the default_sequence member variable.
<b>get_num_reqs_sent</b>	Returns the number of requests that have been sent by this sequencer.
<b>get_num_rsps_received</b>	Returns the number of responses received thus far by this sequencer.
<b>set_num_last_reqs</b>	Sets the size of the last_requests buffer.
<b>get_num_last_reqs</b>	Returns the size of the last requests buffer, as set by set_num_last_reqs.
<b>last_req</b>	Returns the last request item by default.
<b>set_num_last_rsps</b>	Sets the size of the last_responses buffer.
<b>get_num_last_rsps</b>	Returns the max size of the last responses buffer, as set by set_num_last_rsps.
<b>last_rsp</b>	Returns the last response item by default.
<b>execute_item</b>	This task allows the user to supply an item or sequence to the sequencer and have it be executed procedurally.

## Ports

---

### rsp\_export

---

This is the analysis export used by drivers or monitors to send responses to the sequencer. When a driver wishes to send a response, it may do so through exactly one of three methods:

```
seq_item_port.item_done(response)
seq_item_done.put(response)
rsp_port.write(response)
```

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.

## Methods

---

### new

---

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

Creates and initializes an instance of this class using the normal constructor arguments for `ovm_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(ovm_sequence_base sequence_ptr,
                                   ovm_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.

## start\_default\_sequence

---

```
task start_default_sequence()
```

Called when the run phase begins, this method starts the default sequence, as specified by the default\_sequence member variable.

## get\_num\_reqs\_sent

---

```
function int get_num_reqs_sent()
```

Returns the number of requests that have been sent by this sequencer.

## get\_num\_rsps\_received

---

```
function int get_num_rsps_received()
```

Returns the number of responses received thus far 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<sub>th</sub> before last request item. If n is greater than the last request buffer size, the function will return null.

## set\_num\_last\_rsps

---

```
function void set_num_last_rsps(int unsigned max)
```

Sets the size of the last\_responses buffer. The maximum buffer size is 1024. If max is greater than 1024, a warning is issued, and the buffer is set to 1024. The default value is 1.

## get\_num\_last\_rsps

---

```
function int unsigned get_num_last_rsps()
```

Returns the max size of the last responses buffer, as set by set\_num\_last\_rsps.

## last\_rsp

---

```
function RSP last_rsp(int unsigned n = 0)
```

Returns the last response item by default. If n is not 0, then it will get the nth-before-last response item. If n is greater than the last response buffer size, the function will return null.

## execute\_item

---

```
virtual task execute_item(ovm_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.

# ovm\_sequencer #(REQ,RSP)

## Summary

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

#### Class Hierarchy

ovm_object
ovm_report_object
ovm_component
ovm_sequencer_base
ovm_sequencer_param_base #(REQ,RSP)
<b>ovm_sequencer #(REQ,RSP)</b>

#### Class Declaration

```
class ovm_sequencer #(
    type    REQ    =    ovm_sequence_item,
    type    RSP    =    REQ
) extends ovm_sequencer_param_base #(REQ, RSP)
```

#### Variables

[seq\\_item\\_export](#) This export provides access to this sequencer's implementation of the sequencer interface, [sqr\\_if\\_base #\(REQ,RSP\)](#), which defines the following methods:

#### Methods

[new](#) Creates and initializes an instance of this class using the normal constructor arguments for ovm\_component: name is the name of the instance, and parent is the handle to the hierarchical 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

```
ovm_seq_item_pull_imp #(REQ,
                        RSP,
                        this_type) seq_item_export
```

This export provides access to this sequencer's implementation of the sequencer interface, [sqr\\_if\\_base #\(REQ,RSP\)](#), which defines the following methods:

```

virtual task      get_next_item      (output REQ request);
virtual task      try_next_item      (output REQ request);
virtual function void item_done      (input RSP response=null);
virtual task      wait_for_sequences ();
virtual function bit has_do_available ();
virtual task      get                 (output REQ request);
virtual task      peek                (output REQ request);
virtual task      put                 (input RSP response);

```

See [sqr\\_if\\_base #\(REQ,RSP\)](#) for information about this interface.

## Methods

---

### new

---

```

function new (string      name,
               ovm_component parent)

```

Creates and initializes an instance of this class using the normal constructor arguments for ovm\_component: name is the name of the instance, and parent is the handle to the hierarchical 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.



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

## Summary

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

#### Class Hierarchy

ovm_object
ovm_report_object
ovm_component
ovm_sequencer_base
ovm_sequencer_param_base #(REQ,RSP)
<b>ovm_push_sequencer #(REQ,RSP)</b>

#### Class Declaration

```
class ovm_push_sequencer #(
    type    REQ    =    ovm_sequence_item,
    type    RSP    =    REQ
) extends ovm_sequencer_param_base #(REQ, RSP)
```

#### Ports

**req\_port** The push sequencer requires access to a blocking put interface.

#### Methods

- new** Creates and initializes an instance of this class using the normal constructor arguments for **ovm\_component**: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.
- run** 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. Continual sequence items, based on the list of available sequences loaded into this sequencer, are sent out this port.

## Methods

## new

---

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

Creates and initializes an instance of this class using the normal constructor arguments for [ovm\\_component](#): *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

## run

---

```
task run()
```

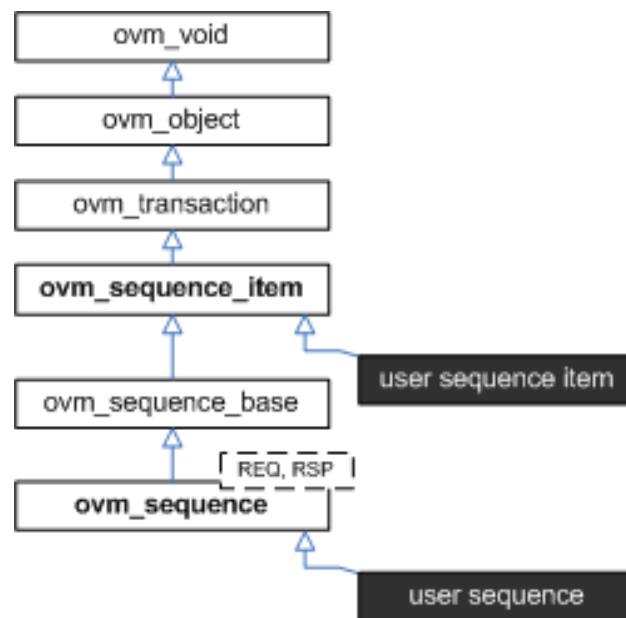
The push sequencer continuously selects from its list of available sequences and sends the next item from the selected sequence out its [req\\_port](#) using `req_port.put(item)`. Typically, the `req_port` would be connected to the `req_export` on an instance of an [ovm\\_push\\_driver #\(REQ,RSP\)](#),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

# Sequence Classes

Sequences encapsulate user-defined procedures that generate multiple `ovm_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 `ovm_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 OVM sequence item and sequence class hierarchy is shown below.



- `ovm_sequence_item` - The `ovm_sequence_item` is the base class for user-defined transactions that leverage the stimulus generation and control capabilities of the sequence-sequencer mechanism.
- `ovm_sequence #(REQ,RSP)` - The `ovm_sequence` extends `ovm_sequence_item` to add the ability to generate streams of `ovm_sequence_items`, either directly or by recursively executing other `ovm_sequences`.

# ovm\_sequence\_item

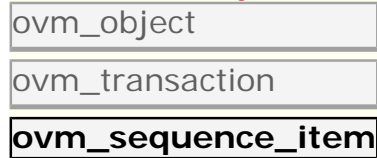
The base class for user-defined sequence items and also the base class for the ovm\_sequence class. The ovm\_sequence\_item class provides the basic functionality for objects, both sequence items and sequences, to operate in the sequence mechanism.

## Summary

### ovm\_sequence\_item

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

#### Class Hierarchy



#### Class Declaration

```
class ovm_sequence_item extends ovm_transaction
```

#### Methods

<a href="#">new</a>	The constructor method for ovm_sequence_item.
<a href="#">get_sequence_id</a>	private
<a href="#">set_use_sequence_info</a>	
<a href="#">get_use_sequence_info</a>	These methods are used to set and get the status of the use_sequence_info bit.
<a href="#">set_id_info</a>	Copies the sequence_id and transaction_id from the referenced item into the calling item.
<a href="#">set_sequencer</a>	
<a href="#">get_sequencer</a>	These routines set and get the reference to the sequencer to which this sequence_item communicates.
<a href="#">set_parent_sequence</a>	Sets the parent sequence of this sequence_item.
<a href="#">get_parent_sequence</a>	Returns a reference to the parent sequence of any sequence on which this method was called.
<a href="#">set_depth</a>	The depth of any sequence is calculated automatically.
<a href="#">get_depth</a>	Returns the depth of a sequence from it's parent.
<a href="#">is_item</a>	This function may be called on any sequence_item or sequence.
<a href="#">start_item</a>	start_item and finish_item together will initiate operation of either a sequence_item or sequence object.
<a href="#">finish_item</a>	finish_item, together with start_item together will initiate operation of either a sequence_item or sequence object.
<a href="#">get_root_sequence_name</a>	Provides the name of the root sequence (the top-most parent sequence).
<a href="#">get_root_sequence</a>	Provides a reference to the root sequence (the top-most parent sequence).
<a href="#">get_sequence_path</a>	Provides a string of names of each sequence in the full hierarchical path.

## Methods

## new

---

```
function new (string          name          = "ovm_sequence_item",
             ovm_sequencer_base sequencer   = null,
             ovm_sequence_base parent_sequence = null) )
```

The constructor method for ovm\_sequence\_item. The sequencer and parent\_sequence may be specified in the constructor, or directly using ovm\_sequence\_item methods.

## 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 ovm\_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. `ovm\_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(ovm_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

---

```
function void set_sequencer(ovm_sequencer_base sequencer)
```

### get\_sequencer

---

```
function ovm_sequencer_base get_sequencer()
```

These routines set and get the reference to the sequencer to which this sequence\_item communicates.

### set\_parent\_sequence

---

```
function void set_parent_sequence(ovm_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 ovm_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(ovm_sequence_item item,
                        int set_priority = -1)
```

`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(ovm_sequence_item item,
                        int set_priority = -1)
```

`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.

## **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 ovm_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.



# ovm\_sequence\_base

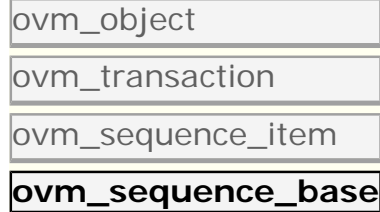
The ovm\_sequence\_base class provides the interfaces needed to create streams of sequence items and/or other sequences.

## Summary

### ovm\_sequence\_base

The ovm\_sequence\_base class provides the interfaces needed to create streams of sequence items and/or other sequences.

#### Class Hierarchy



#### Class Declaration

```
class ovm_sequence_base extends ovm_sequence_item
```

#### Variables

**seq\_kind** Used as an identifier in constraints for a specific sequence type.

#### Methods

<b>new</b>	The constructor for ovm_sequence_base.
<b>get_sequence_state</b>	Returns the sequence state as an enumerated value.
<b>wait_for_sequence_state</b>	Waits until the sequence reaches the given <i>state</i> .
<b>start</b>	The start task is called to begin execution of a sequence
<b>pre_body</b>	This task is a user-definable callback task that is called before the execution of the body, unless the sequence is started with call_pre_post=0.
<b>post_body</b>	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.
<b>pre_do</b>	This task is a user-definable callback task that is called after the sequence has issued a wait_for_grant() call and after the sequencer has selected this sequence, and before the item is randomized.
<b>body</b>	This is the user-defined task where the main sequence code resides.
<b>is_item</b>	This function may be called on any sequence_item or sequence object.
<b>mid_do</b>	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.
<b>post_do</b>	This function is a user-definable callback function that is called after the driver has indicated that it has completed the item, using either this item_done or put methods.
<b>num_sequences</b>	Returns the number of sequences in the sequencer's sequence library.
<b>get_seq_kind</b>	This function returns an int representing the sequence kind that has been registered with the sequencer.
<b>get_sequence</b>	This function returns a reference to a sequence specified by req_kind, which can be obtained using the get_seq_kind method.
<b>get_sequence_by_name</b>	Internal method.
<b>do_sequence_kind</b>	This task will start a sequence of kind specified by req_kind, which can be obtained using the get_seq_kind method.
<b>set_priority</b>	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>wait_for_relevant</code>	This method is called by the sequencer when all available sequences are not relevant.
<code>is_relevant</code>	The default <code>is_relevant</code> implementation returns 1, indicating that the sequence is always relevant.
<code>is_blocked</code>	Returns a bit indicating whether this sequence is currently prevented from running due to another lock or grab.
<code>has_lock</code>	Returns 1 if this sequence has a lock, 0 otherwise.
<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.
<code>ungrab</code>	Removes any locks or grabs obtained by this sequence on the specified sequencer.
<code>wait_for_grant</code>	This task issues a request to the current sequencer.
<code>send_request</code>	The <code>send_request</code> function may only be called after a <code>wait_for_grant</code> call.
<code>wait_for_item_done</code>	A sequence may optionally call <code>wait_for_item_done</code> .
<code>set_sequencer</code>	Sets the default sequencer for the sequence to run on.
<code>get_sequencer</code>	Returns a reference to the current default sequencer of the sequence.
<code>kill</code>	This function will kill the sequence, and cause all current locks and requests in the sequence's default sequencer to be removed.
<code>use_response_handler</code>	When called with <code>enable</code> set to 1, responses will be sent to the response handler.
<code>get_use_response_handler</code>	Returns the state of the <code>use_response_handler</code> bit.
<code>response_handler</code>	When the <code>use_response_handler</code> bit is set to 1, this virtual task is called by the sequencer for each response that arrives for this sequence.
<code>create_item</code>	<code>Create_item</code> will create and initialize a <code>sequence_item</code> or <code>sequence</code> using the factory.
<code>start_item</code>	<code>start_item</code> and <code>finish_item</code> together will initiate operation of either a <code>sequence_item</code> or <code>sequence</code> object.
<code>finish_item</code>	<code>finish_item</code> , together with <code>start_item</code> together will initiate operation of either a <code>sequence_item</code> or <code>sequence</code> object.

## Variables

---

### `seq_kind`

---

```
rand int unsigned seq_kind
```

Used as an identifier in constraints for a specific sequence type.

## Methods

---

## new

---

```
function new (string          name          = "ovm_sequence",
             ovm_sequencer_base sequencer_ptr = null,
             ovm_sequence_base parent_seq    = null          )
```

The constructor for ovm\_sequence\_base.

The sequencer\_ptr and parent\_seq arguments are deprecated in favor of their being set in the start method.

## get\_sequence\_state

---

```
function ovm_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(ovm_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 );

## start

---

```
virtual task start (ovm_sequencer_base sequencer,
                   ovm_sequence_base parent_sequence = null,
                   integer              this_priority  = 100,
                   bit                  call_pre_post  = 1          )
```

The start task is called to begin execution of a sequence

If parent\_sequence is null, then the sequence is a parent, otherwise it is a child of the specified parent.

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, 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 task that is called before 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.

## post\_body

---

```
virtual task post_body()
```

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.

## pre\_do

---

```
virtual task pre_do(bit is_item)
```

This task is a user-definable callback task that is called after the sequence has issued a `wait_for_grant()` call and after the sequencer has selected this sequence, and before the item is randomized. This method should not be called directly by the user.

Although `pre_do` is a task, consuming simulation cycles may result in unexpected behavior on the driver.

## 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.

## is\_item

---

```
virtual function bit is_item()
```

This function may be called on any sequence\_item or sequence object. It will return 1 on items and 0 on sequences.

## mid\_do

---

```
virtual function void mid_do(ovm_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.

## post\_do

---

```
virtual function void post_do(ovm_sequence_item this_item)
```

This function is a user-definable callback function that is called after the driver has indicated that it has completed the item, using either this item\_done or put methods. This method should not be called directly by the user.

## num\_sequences

---

```
function int num_sequences()
```

Returns the number of sequences in the sequencer's sequence library.

## get\_seq\_kind

---

```
function int get_seq_kind(string type_name)
```

This function returns an int representing the sequence kind that has been registered with the sequencer. The seq\_kind int may be used with the get\_sequence or do\_sequence\_kind methods.

## get\_sequence

---

```
function ovm_sequence_base get_sequence(int unsigned req_kind)
```

This function returns a reference to a sequence specified by req\_kind, which can be obtained using the get\_seq\_kind method.

## get\_sequence\_by\_name

---

```
function ovm_sequence_base get_sequence_by_name(string seq_name)
```

Internal method.

## do\_sequence\_kind

---

```
task do_sequence_kind(int unsigned req_kind)
```

This task will start a sequence of kind specified by req\_kind, which can be obtained using the get\_seq\_kind method.

## 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.

## 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.

## 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.

## 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.

## lock

---

```
task lock(ovm_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(ovm_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(ovm_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(ovm_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.

## 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(ovm_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.

## set\_sequencer

---

```
virtual function void set_sequencer(ovm_sequencer_base sequencer)
```

Sets the default sequencer for the sequence to run on. It will take effect immediately, so it should not be called while the sequence is actively communicating with the sequencer.

## get\_sequencer

---

```
virtual function ovm_sequencer_base get_sequencer()
```

Returns a reference to the current default sequencer of the sequence.

## 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.

## 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(ovm_sequence_item response)
```

When the `use_reponse_handler` bit is set to 1, this virtual task is called by the sequencer for each response that arrives for this sequence.

## create\_item

---

```
protected function ovm_sequence_item create_item(
    ovm_object_wrapper    type_var,
    ovm_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 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

```
virtual task start_item(ovm_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(ovm_sequence_item item, int set_priority = -1);
```

## ovm\_sequence #(REQ,RSP)

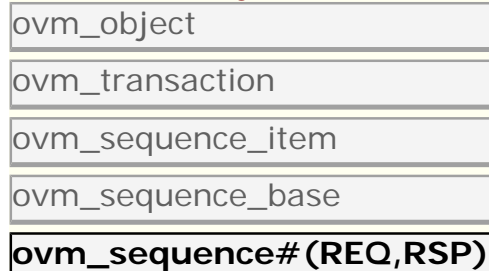
The ovm\_sequence class provides the interfaces necessary in order to create streams of sequence items and/or other sequences.

### Summary

#### ovm\_sequence #(REQ,RSP)

The ovm\_sequence class provides the interfaces necessary in order to create streams of sequence items and/or other sequences.

#### Class Hierarchy



#### Class Declaration

```

virtual class ovm_sequence #(
    type REQ = ovm_sequence_item,
    type RSP = REQ
) extends ovm_sequence_base
  
```

#### Methods

<a href="#">new</a>	Creates and initializes a new sequence object.
<a href="#">start</a>	The start task is called to begin execution of a sequence.
<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 <code>get_response</code> .
<a href="#">set_sequencer</a>	Sets the default sequencer for the sequence to <code>sequencer</code> .
<a href="#">set_response_queue_error_report_disabled</a>	By default, if the <code>response_queue</code> overflows, an error is reported.
<a href="#">get_response_queue_error_report_disabled</a>	When this bit is 0 (default value), error reports are generated when the response queue overflows.
<a href="#">set_response_queue_depth</a>	The default maximum depth of the response queue is 8.
<a href="#">get_response_queue_depth</a>	Returns the current depth setting for the response queue.

## Methods

## new

---

```
function new (string          name          = "ovm_sequence",
             ovm_sequencer_base sequencer_ptr = null,
             ovm_sequence_base parent_seq    = null          )
```

Creates and initializes a new sequence object.

The *sequencer\_ptr* and *parent\_seq* arguments are deprecated in favor of their being set in the start method.

## start

---

```
virtual task start (ovm_sequencer_base sequencer,
                   ovm_sequence_base parent_sequence = null,
                   integer              this_priority  = 100,
                   bit                  call_pre_post  = 1          )
```

The start task is called to begin execution of a sequence.

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 the sequence is a parent, otherwise it is a child of the specified parent.

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, then the *pre\_body* and *post\_body* tasks will be called before and after the sequence body is called.

## send\_request

---

```
function void send_request(ovm_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 [ovm\\_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

---

```
task get_response(output RSP response,
                 input int transaction_id = -1
                )
```

By default, sequences must retrieve responses by calling `get_response`. If no `transaction_id` is specified, this task will return the next response sent to this sequence. If no response is available in the response queue, the method will block until a response is recieved.

If a `transaction_id` 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`.

## set\_sequencer

---

```
virtual function void set_sequencer(ovm_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.

## 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.

## ovm\_random\_sequence

This sequence randomly selects and executes a sequence from the sequencer's sequence library, excluding ovm\_random\_sequence itself, and ovm\_exhaustive\_sequence.

The ovm\_random\_sequence class is a built-in sequence that is preloaded into every sequencer's sequence library with the name "ovm\_random\_sequence".

The number of selections and executions is determined by the count property of the sequencer (or virtual sequencer) on which ovm\_random\_sequence is operating. See [ovm\\_sequencer\\_base](#) for more information.

### Summary

#### ovm\_random\_sequence

This sequence randomly selects and executes a sequence from the sequencer's sequence library, excluding ovm\_random\_sequence itself, and ovm\_exhaustive\_sequence.

##### Class Hierarchy

```
ovm_sequence#(ovm_sequence_item)
```

```
ovm_random_sequence
```

##### Class Declaration

```
class ovm_random_sequence extends ovm_sequence #(
    ovm_sequence_item
)
```

##### Methods

[get\\_count](#) Returns the count of the number of sub-sequences which are randomly generated.

## Methods

#### [get\\_count](#)

```
function int unsigned get_count()
```

Returns the count of the number of sub-sequences which are randomly generated. By default, count is equal to the value from the sequencer's count variable. However, if the sequencer's count variable is -1, then a random value between 0 and sequencer.max\_random\_count (exclusive) is chosen. The sequencer's count variable is subsequently reset to the random value that was used. If get\_count() is call before the sequence has started, the return value will be sequencer.count, which may be -1.



## ovm\_exhaustive\_sequence

This sequence randomly selects and executes each sequence from the sequencer's sequence library once, excluding itself and ovm\_random\_sequence.

The ovm\_exhaustive\_sequence class is a built-in sequence that is preloaded into every sequencer's sequence library with the name "ovm\_exhaustive\_sequence".

### Summary

#### ovm\_exhaustive\_sequence

This sequence randomly selects and executes each sequence from the sequencer's sequence library once, excluding itself and ovm\_random\_sequence.

#### Class Hierarchy

```
ovm_sequence#(ovm_sequence_item)
```

```
ovm_exhaustive_sequence
```

#### Class Declaration

```
class ovm_exhaustive_sequence extends ovm_sequence #(
    ovm_sequence_item
)
```

## ovm\_simple\_sequence

This sequence simply executes a single sequence item.

The item parameterization of the sequencer on which the ovm\_simple\_sequence is executed defines the actual type of the item executed.

The ovm\_simple\_sequence class is a built-in sequence that is preloaded into every sequencer's sequence library with the name "ovm\_simple\_sequence".

See [ovm\\_sequencer #\(REQ,RSP\)](#) for more information on running sequences.

### Summary

#### ovm\_simple\_sequence

This sequence simply executes a single sequence item.

#### Class Hierarchy

```
ovm_sequence#(ovm_sequence_item)
```

```
ovm_simple_sequence
```

### **Class Declaration**

```
class ovm_simple_sequence extends ovm_sequence #(
    ovm_sequence_item
)
```

**end**

---

# Report Macros

This set of macros provides wrappers around the `ovm_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 ``ovm_info` reports.
- The ``__FILE__` and ``__LINE__` information is automatically provided to the underlying `ovm_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 `OVM_DISABLE_REPORT_FILE_LINE` on the command line.

The macros also enforce a verbosity setting of `OVM_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 `ovm_report_*` with the corresponding macro.

```
//Previous calls to ovm_report_*
ovm_report_info("MYINFO1", $sformatf("val: %0d", val), OVM_LOW);
ovm_report_warning("MYWARN1", "This is a warning");
ovm_report_error("MYERR", "This is an error");
ovm_report_fatal("MYFATAL", "A fatal error has occurred");
```

The above code is replaced by

```
//New calls to `ovm_*
`ovm_info("MYINFO1", $sformatf("val: %0d", val), OVM_LOW)
`ovm_warning("MYWARN1", "This is a warning")
`ovm_error("MYERR", "This is an error")
`ovm_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 `ovm_report_*` [Reporting](#) functions.

#### Macros

- ``ovm_info` Calls `ovm_report_info` if *VERBOSITY* is lower than the configured verbosity of the associated reporter.
- ``ovm_warning` Calls `ovm_report_warning` with a verbosity of `OVM_NONE`.
- ``ovm_error` Calls `ovm_report_error` with a verbosity of `OVM_NONE`.

``ovm_fatal` Calls `ovm_report_fatal` with a verbosity of `OVM_NONE`.

## Macros

---

### ``ovm_info`

---

Calls `ovm_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 `ovm_report_info` call.

### ``ovm_warning`

---

Calls `ovm_report_warning` with a verbosity of `OVM_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 `ovm_report_warning` call.

### ``ovm_error`

---

Calls `ovm_report_error` with a verbosity of `OVM_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 `ovm_report_error` call.

### ``ovm_fatal`

---

Calls `ovm_report_fatal` with a verbosity of `OVM_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 `ovm_report_fatal` call.

# 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 `ovm_object::create` method, which is needed for cloning, and the `ovm_object::get_type_name` method, which is needed for a number of debugging features.

``ovm_field_utils_begin`

``ovm_field_utils_end`

These macros form a block in which ``ovm_field_*` macros can be placed.

``ovm_object_utils`

``ovm_object_param_utils`

``ovm_object_utils_begin`

``ovm_object_param_utils_begin`

``ovm_object_utils_end`

`ovm_object`-based class declarations may contain one of the above forms of utility macros.

``ovm_component_utils`

``ovm_component_param_utils`

``ovm_component_utils_begin`

``ovm_component_param_utils_begin`

``ovm_component_end`

`ovm_component`-based class declarations may contain one of the above forms of utility macros.

#### Field Macros

The ``ovm_field_*` macros are invoked inside of the ``ovm_*_utils_begin` and ``ovm_*_utils_end` macro blocks to form “automatic” implementations of the core data methods: copy, compare, pack, unpack, record, print, and sprint.

#### ``ovm_field_* macros`

Macros that implement data operations for scalar properties.

``ovm_field_int`

Implements the data operations for any packed integral property.

``ovm_field_object`

Implements the data operations for an `ovm_object`-based property.

``ovm_field_string`

Implements the data operations for a string property.

``ovm_field_enum`

Implements the data operations for an enumerated property.

``ovm_field_real`

Implements the data operations for any real property.

``ovm_field_event`

Implements the data operations for an event property.

#### ``ovm_field_sarray_* macros`

Macros that implement data operations for one-dimensional static array properties.

``ovm_field_sarray_int`

Implements the data operations for a one-dimensional static array of integrals.

``ovm_field_sarray_object`

Implements the data operations for a one-dimensional static array of `ovm_object`-based objects.

``ovm_field_sarray_string`

Implements the data operations for a one-dimensional static array of strings.

``ovm_field_sarray_enum`

Implements the data operations for a one-dimensional static array of enums.

#### ``ovm_field_array_* macros`

Macros that implement data operations for one-dimensional dynamic array properties.

``ovm_field_array_int`

Implements the data operations for a one-dimensional dynamic array of integrals.

``ovm_field_array_object`

Implements the data operations for a one-dimensional dynamic array of `ovm_object`-based objects.

<code>`ovm_field_array_string</code>	Implements the data operations for a one-dimensional dynamic array of strings.
<code>`ovm_field_array_enum</code>	Implements the data operations for a one-dimensional dynamic array of enums.
<code>`ovm_field_queue_* macros</code>	Macros that implement data operations for dynamic queues.
<code>`ovm_field_queue_int</code>	Implements the data operations for a queue of integrals.
<code>`ovm_field_queue_object</code>	Implements the data operations for a queue of <code>ovm_object</code> -based objects.
<code>`ovm_field_queue_string</code>	Implements the data operations for a queue of strings.
<code>`ovm_field_queue_enum</code>	Implements the data operations for a one-dimensional queue of enums.
<code>`ovm_field_aa_*_string macros</code>	Macros that implement data operations for associative arrays indexed by <i>string</i> .
<code>`ovm_field_aa_int_string</code>	Implements the data operations for an associative array of integrals indexed by <i>string</i> .
<code>`ovm_field_aa_object_string</code>	Implements the data operations for an associative array of <code>ovm_object</code> -based objects indexed by <i>string</i> .
<code>`ovm_field_aa_string_string</code>	Implements the data operations for an associative array of strings indexed by <i>string</i> .
<code>`ovm_field_aa_*_int macros</code>	Macros that implement data operations for associative arrays indexed by an integral type.
<code>`ovm_field_aa_object_int</code>	Implements the data operations for an associative array of <code>ovm_object</code> -based objects indexed by the <i>int</i> data type.
<code>`ovm_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>`ovm_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>`ovm_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>`ovm_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>`ovm_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>`ovm_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>`ovm_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>`ovm_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>`ovm_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>`ovm_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>`ovm_field_aa_int_key</code>	Implements the data operations for an associative array of integral types indexed by any integral key data type.
<code>`ovm_field_aa_int_enumkey</code>	Implements the data operations for an associative array of integral types indexed by any enumeration key data type.

## Utility Macros

The utility macros provide implementations of the `ovm_object::create` method, which is

needed for cloning, and the `ovm_object::get_type_name` method, which is needed for a number of debugging features. They also register the type with the `ovm_factory`, and they implement a `get_type` method, which is used when configuring the factory. And they implement the virtual `ovm_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 `ovm_object`.

```
class mydata extends ovm_object;

    string str;
    mydata subdata;
    int field;
    myenum e1;
    int queue[$];

    `ovm_object_utils_begin(mydata) //requires ctor with default args
        `ovm_field_string(str, OVM_DEFAULT)
        `ovm_field_object(subdata, OVM_DEFAULT)
        `ovm_field_int(field, OVM_DEC) //use decimal radix
        `ovm_field_enum(myenum, e1, OVM_DEFAULT)
        `ovm_field_queue_int(queue, OVM_DEFAULT)
    `ovm_object_utils_end

endclass
```

## ``ovm_field_utils_begin`

---

## ``ovm_field_utils_end`

---

These macros form a block in which ``ovm_field_*` macros can be placed. Used as

```
`ovm_field_utils_begin(TYPE)
    `ovm_field_* macros here
`ovm_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).

## ``ovm_object_utils`

---

## ``ovm_object_param_utils`

---

## ``ovm_object_utils_begin`

---

## ``ovm_object_param_utils_begin`

---

## ``ovm_object_utils_end`

---

`ovm_object`-based class declarations may contain one of the above forms of utility macros.

For simple objects with no field macros, use

```
`ovm_object_utils(TYPE)
```

For simple objects with field macros, use

```
`ovm_object_utils_begin(TYPE)
  `ovm_field_* macro invocations here
`ovm_object_utils_end
```

For parameterized objects with no field macros, use

```
`ovm_object_param_utils(TYPE)
```

For parameterized objects, with field macros, use

```
`ovm_object_param_utils_begin(TYPE)
  `ovm_field_* macro invocations here
`ovm_object_utils_end
```

Simple (non-parameterized) objects use the `ovm_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 `ovm_object_param_utils*` versions. They differ from ``ovm_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 ``ovm_field_*` macros can be placed. The block must be terminated by ``ovm_object_utils_end`.

Objects deriving from `ovm_sequence` must use the ``ovm_sequence_*` macros instead of these macros. See ``ovm_sequence_utils` for details.

## ``ovm_component_utils`

---

## ``ovm_component_param_utils`

---

## ``ovm_component_utils_begin`

---

## ``ovm_component_param_utils_begin`

---

## ``ovm_component_end`

---

`ovm_component`-based class declarations may contain one of the above forms of utility macros.

For simple components with no field macros, use

```
`ovm_component_utils(TYPE)
```

For simple components with field macros, use

```
`ovm_component_utils_begin(TYPE)
`ovm_field_* macro invocations here
`ovm_component_utils_end
```

For parameterized components with no field macros, use

```
`ovm_component_param_utils(TYPE)
```

For parameterized components with field macros, use

```
`ovm_component_param_utils_begin(TYPE)
`ovm_field_* macro invocations here
`ovm_component_utils_end
```

Simple (non-parameterized) components must use the `ovm_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 `ovm_object_param_utils*` versions. They differ from ``ovm_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 ``ovm_field_*` macros can be placed. The block must be terminated

by ``ovm_component_utils_end`.

Components deriving from `ovm_sequencer` must use the ``ovm_sequencer_*` macros instead of these macros. See ``ovm_sequencer_utils` for details.

## Field Macros

The ``ovm_field_*` macros are invoked inside of the ``ovm_*_utils_begin` and ``ovm_*_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 ovm_transaction;
  string my_string;
  `ovm_object_utils_begin(my_trans)
    `ovm_field_string(my_string, OVM_ALL_ON)
  `ovm_object_utils_end
endclass
```

Each ``ovm_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 ``ovm_field_string` macro.

If *FLAG* is set to `OVM_ALL_ON`, as in the example, the *ARG* variable will be included in all data methods. The *FLAG*, if set to something other than `OVM_ALL_ON` or `OVM_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>OVM_ALL_ON</code>	Set all operations on (default).
<code>OVM_DEFAULT</code>	Use the default flag settings.
<code>OVM_NOCOPY</code>	Do not copy this field.
<code>OVM_NOCOMPARE</code>	Do not compare this field.
<code>OVM_NOPRINT</code>	Do not print this field.
<code>OVM_NODEFPRI</code>	Do not print the field if it is the same as its
<code>OVM_NOPACK</code>	Do not pack or unpack this field.
<code>OVM_PHYSICAL</code>	Treat as a physical field. Use physical setting in policy class for this field.
<code>OVM_ABSTRACT</code>	Treat as an abstract field. Use the abstract setting in the policy class for this field.
<code>OVM_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>OVM_BIN</i>	Print / record the field in binary (base-2).
<i>OVM_DEC</i>	Print / record the field in decimal (base-10).
<i>OVM_UNSIGNED</i>	Print / record the field in unsigned decimal (base-10).
<i>OVM_OCT</i>	Print / record the field in octal (base-8).
<i>OVM_HEX</i>	Print / record the field in hexadecimal (base-16).
<i>OVM_STRING</i>	Print / record the field in string format.
<i>OVM_TIME</i>	Print / record the field in time format.

Radix settings for integral types. Hex is the default radix if none is specified.

## ``ovm_field_*` macros

---

Macros that implement data operations for scalar properties.

### ``ovm_field_int`

---

Implements the data operations for any packed integral property.

```
`ovm_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.

### ``ovm_field_object`

---

Implements the data operations for an [ovm\\_object](#)-based property.

```
`ovm_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.

### ``ovm_field_string`

---

Implements the data operations for a string property.

```
`ovm_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.

## [`ovm\\_field\\_enum](#)

---

Implements the data operations for an enumerated property.

```
`ovm_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.

## [`ovm\\_field\\_real](#)

---

Implements the data operations for any real property.

```
`ovm_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.

## [`ovm\\_field\\_event](#)

---

Implements the data operations for an event property.

```
`ovm_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.

## ``ovm_field_sarray_*` macros`

---

Macros that implement data operations for one-dimensional static array properties.

### ``ovm_field_sarray_int``

---

Implements the data operations for a one-dimensional static array of integrals.

```
`ovm_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.

### ``ovm_field_sarray_object``

---

Implements the data operations for a one-dimensional static array of [ovm\\_object](#)-based objects.

```
`ovm_field_sarray_object( ARG, FLAG )
```

*ARG* is a one-dimensional static array of [ovm\\_object](#)-based objects, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

### ``ovm_field_sarray_string``

---

Implements the data operations for a one-dimensional static array of strings.

```
`ovm_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.

## ``ovm_field_sarray_enum`

---

Implements the data operations for a one-dimensional static array of enums.

```
`ovm_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.

## ``ovm_field_array_* macros`

---

Macros that implement data operations for one-dimensional dynamic array properties.

### ``ovm_field_array_int`

---

Implements the data operations for a one-dimensional dynamic array of integrals.

```
`ovm_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.

### ``ovm_field_array_object`

---

Implements the data operations for a one-dimensional dynamic array of [ovm\\_object](#)-based objects.

```
`ovm_field_array_object(ARG,FLAG)
```

*ARG* is a one-dimensional dynamic array of [ovm\\_object](#)-based objects, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

### ``ovm_field_array_string`

---

Implements the data operations for a one-dimensional dynamic array of strings.

```
`ovm_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.

## ``ovm_field_array_enum`

---

Implements the data operations for a one-dimensional dynamic array of enums.

```
`ovm_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.

## ``ovm_field_queue_* macros`

---

Macros that implement data operations for dynamic queues.

### ``ovm_field_queue_int`

---

Implements the data operations for a queue of integrals.

```
`ovm_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.

### ``ovm_field_queue_object`

---

Implements the data operations for a queue of [ovm\\_object](#)-based objects.



```
`ovm_field_queue_object(ARG,FLAG)
```

*ARG* is a one-dimensional queue of [ovm\\_object](#)-based objects, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

## [`ovm\\_field\\_queue\\_string](#)

---

Implements the data operations for a queue of strings.

```
`ovm_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.

## [`ovm\\_field\\_queue\\_enum](#)

---

Implements the data operations for a one-dimensional queue of enums.

```
`ovm_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.

## [`ovm\\_field\\_aa\\_\\*\\_string macros](#)

---

Macros that implement data operations for associative arrays indexed by *string*.

### [`ovm\\_field\\_aa\\_int\\_string](#)

---

Implements the data operations for an associative array of integrals indexed by *string*.

```
`ovm_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.

### ``ovm_field_aa_object_string`

---

Implements the data operations for an associative array of [ovm\\_object](#)-based objects indexed by *string*.

```
`ovm_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.

### ``ovm_field_aa_string_string`

---

Implements the data operations for an associative array of strings indexed by *string*.

```
`ovm_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.

## ``ovm_field_aa_*_int macros`

---

Macros that implement data operations for associative arrays indexed by an integral type.

### ``ovm_field_aa_object_int`

---

Implements the data operations for an associative array of [ovm\\_object](#)-based objects indexed by the *int* data type.

```
`ovm_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.

### ``ovm_field_aa_int_int`

---

Implements the data operations for an associative array of integral types indexed by the *int* data type.

```
`ovm_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.

### ``ovm_field_aa_int_int_unsigned`

---

Implements the data operations for an associative array of integral types indexed by the *int unsigned* data type.

```
`ovm_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.

### ``ovm_field_aa_int_integer`

---

Implements the data operations for an associative array of integral types indexed by the *integer* data type.

```
`ovm_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.

## ``ovm_field_aa_int_integer_unsigned`

---

Implements the data operations for an associative array of integral types indexed by the *integer unsigned* data type.

```
`ovm_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.

## ``ovm_field_aa_int_byte`

---

Implements the data operations for an associative array of integral types indexed by the *byte* data type.

```
`ovm_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.

## ``ovm_field_aa_int_byte_unsigned`

---

Implements the data operations for an associative array of integral types indexed by the *byte unsigned* data type.

```
`ovm_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.

## ``ovm_field_aa_int_shortint`

---

Implements the data operations for an associative array of integral types indexed by the *shortint* data type.

```
`ovm_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.

## [`ovm\\_field\\_aa\\_int\\_shortint\\_unsigned](#)

---

Implements the data operations for an associative array of integral types indexed by the *shortint unsigned* data type.

```
`ovm_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.

## [`ovm\\_field\\_aa\\_int\\_longint](#)

---

Implements the data operations for an associative array of integral types indexed by the *longint* data type.

```
`ovm_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.

## [`ovm\\_field\\_aa\\_int\\_longint\\_unsigned](#)

---

Implements the data operations for an associative array of integral types indexed by the *longint unsigned* data type.

```
`ovm_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.

## ``ovm_field_aa_int_key`

---

Implements the data operations for an associative array of integral types indexed by any integral key data type.

```
`ovm_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.

## ``ovm_field_aa_int_enumkey`

---

Implements the data operations for an associative array of integral types indexed by any enumeration key data type.

```
`ovm_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.

# Sequence and Do Action Macros

## Summary

### Sequence and Do Action Macros

#### Sequence Registration Macros

``ovm_declare_p_sequencer`

The sequence-specific macros perform the same function as the set of ``ovm_object_*_utils` macros, except they also set the default sequencer type the sequence will run on.

This macro is used to set up a specific sequencer type with the sequence type the macro is placed in.

``ovm_sequence_utils_begin`

``ovm_sequence_utils_end`

``ovm_sequence_utils`

The sequence macros can be used in non-parameterized `<ovm_sequence>` extensions to pre-register the sequence with a given `<ovm_sequencer>` type.

#### Sequencer Registration Macros

The sequencer-specific macros perform the same function as the set of ``ovm_component_*_utils` macros except that they also declare the plumbing necessary for creating the sequencer's sequence library.

``ovm_update_sequence_lib`

This macro populates the instance-specific sequence library for a sequencer.

``ovm_update_sequence_lib_and_item`

This macro populates the instance specific sequence library for a sequencer, and it registers the given *USER\_ITEM* as an instance override for the simple sequence's item variable.

``ovm_sequencer_utils`

``ovm_sequencer_utils_begin`

``ovm_sequencer_param_utils`

``ovm_sequencer_param_utils_begin`

``ovm_sequencer_utils_end`

The sequencer macros are used in `ovm_sequencer`-based class declarations in one of four ways.

#### Sequence Action Macros

These macros are used to start sequences and sequence items that were either registered with a `<`ovm-sequence_utils>` macro or whose associated sequencer was already set using the `<set_sequencer>` method.

``ovm_create`

This action creates the item or sequence using the factory.

``ovm_do`

This macro takes as an argument a `ovm_sequence_item` variable or object.

``ovm_do_pri`

This is the same as ``ovm_do` except that the sequene item or sequence is executed with the priority specified in the argument

``ovm_do_with`

This is the same as ``ovm_do` except that the constraint block in the 2nd argument is applied to the item or sequence in a randomize with statement before execution.

``ovm_do_pri_with`

This is the same as ``ovm_do_pri` except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.

``ovm_send`

This macro processes the item or sequence that has been created using ``ovm_create`.

``ovm_send_pri`

This is the same as ``ovm_send` except that the sequene item or sequence is executed with the priority specified in the argument.

``ovm_rand_send`

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

<code>`ovm_rand_send_pri</code>	This is the same as <code>`ovm_rand_send</code> except that the sequence item or sequence is executed with the priority specified in the argument.
<code>`ovm_rand_send_with</code>	This is the same as <code>`ovm_rand_send</code> except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.
<code>`ovm_rand_send_pri_with</code>	This is the same as <code>`ovm_rand_send_pri</code> except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.
<b>Sequence on Sequencer Action Macros</b> These macros are used to start sequences and sequence items on a specific sequencer, given in a macro argument.	
<code>`ovm_create_on</code>	This is the same as <code>`ovm_create</code> 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 <i>SEQUENCER_REF</i> argument.
<code>`ovm_do_on</code>	This is the same as <code>`ovm_do</code> 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 <i>SEQUENCER_REF</i> argument.
<code>`ovm_do_on_pri</code>	This is the same as <code>`ovm_do_pri</code> 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 <i>SEQUENCER_REF</i> argument.
<code>`ovm_do_on_with</code>	This is the same as <code>`ovm_do_with</code> 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 <i>SEQUENCER_REF</i> argument.
<code>`ovm_do_on_pri_with</code>	This is the same as <code>`ovm_do_pri_with</code> 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 <i>SEQUENCER_REF</i> argument.

## Sequence Registration Macros

The sequence-specific macros perform the same function as the set of ``ovm_object_*_utils` macros, except they also set the default sequencer type the sequence will run on.

### ``ovm_declare_p_sequencer`

This macro is used to set up a specific sequencer type with the sequence type the macro is placed in. This macro is implicit in the `<ovm_sequence_utils>` macro, but may be used directly in cases when the sequence is not to be registered in the sequencer's library.

The example below shows using the `ovm_declare_p_sequencer` macro along with the `ovm_object_utils` macros to set up the sequence but not register the sequence in the sequencer's library.



```

class mysequence extends ovm_sequence#(mydata);
  `ovm_object_utils(mysequence)
  `ovm_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

```

## ``ovm_sequence_utils_begin`

---

## ``ovm_sequence_utils_end`

---

## ``ovm_sequence_utils`

---

The sequence macros can be used in non-parameterized `<ovm_sequence>` extensions to pre-register the sequence with a given `<ovm_sequencer>` type.

### For sequences that do not use any ``ovm_field` macros

```
`ovm_sequence_utils(TYPE_NAME,SQR_TYPE_NAME)
```

### For sequences employing with field macros

```

`ovm_sequence_utils_begin(TYPE_NAME,SQR_TYPE_NAME)
  `ovm_field_* macro invocations here
`ovm_sequence_utils_end

```

The sequence-specific macros perform the same function as the set of ``ovm_object_*_utils` macros except that they also register the sequence's type, `TYPE_NAME`, with the given sequencer type, `SQR_TYPE_NAME`, and define the `p_sequencer` variable and `m_set_p_sequencer` method.

Use ``ovm_sequence_utils[_begin]` for non-parameterized classes and ``ovm_sequence_param_utils[_begin]` for parameterized classes.

## Sequencer Registration Macros

---

The sequencer-specific macros perform the same function as the set of ``ovm_component_*utils` macros except that they also declare the plumbing necessary for creating the sequencer's sequence library.

### ``ovm_update_sequence_lib`

---

This macro populates the instance-specific sequence library for a sequencer. It should be invoked inside the sequencer's constructor.

### ``ovm_update_sequence_lib_and_item`

---

This macro populates the instance specific sequence library for a sequencer, and it registers the given *USER\_ITEM* as an instance override for the simple sequence's item variable.

The macro should be invoked inside the sequencer's constructor.

### ``ovm_sequencer_utils`

---

### ``ovm_sequencer_utils_begin`

---

### ``ovm_sequencer_param_utils`

---

### ``ovm_sequencer_param_utils_begin`

---

### ``ovm_sequencer_utils_end`

---

The sequencer macros are used in `ovm_sequencer`-based class declarations in one of four ways.

For simple sequencers, no field macros

```
`ovm_sequencer_utils(SQR_TYPE_NAME)
```

For simple sequencers, with field macros

```
`ovm_sequencer_utils_begin(SQR_TYPE_NAME) `ovm_field_* macros here
`ovm_sequencer_utils_end
```

For parameterized sequencers, no field macros

```
`ovm_sequencer_param_utils(SQR_TYPE_NAME)
```

For parameterized sequencers, with field macros

```
`ovm_sequencer_param_utils_begin(SQR_TYPE_NAME) `ovm_field_* macros here
`ovm_sequencer_utils_end
```

The sequencer-specific macros perform the same function as the set of ``ovm_component_*utils` macros except that they also declare the plumbing necessary for creating the sequencer's sequence library. This includes:

1. Declaring the type-based static queue of strings registered on the sequencer type.
2. Declaring the static function to add strings to item #1 above.
3. Declaring the static function to remove strings to item #1 above.
4. Declaring the function to populate the instance specific sequence library for a sequencer.

Use ``ovm_sequencer_utils[_begin]` for non-parameterized classes and ``ovm_sequencer_param_utils[_begin]` for parameterized classes.

## Sequence Action Macros

---

These macros are used to start sequences and sequence items that were either registered with a `<`ovm-sequence_utils>` macro or whose associated sequencer was already set using the `<set_sequencer>` method.

### ``ovm_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.

## ``ovm_do`

---

This macro takes as an argument a `ovm_sequence_item` variable or object. `ovm_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.

## ``ovm_do_pri`

---

This is the same as ``ovm_do` except that the sequene item or sequence is executed with the priority specified in the argument

## ``ovm_do_with`

---

This is the same as ``ovm_do` except that the constraint block in the 2nd argument is applied to the item or sequence in a `randomize with` statement before execution.

## ``ovm_do_pri_with`

---

This is the same as ``ovm_do_pri` except that the given constraint block is applied to the item or sequence in a `randomize with` statement before execution.

## ``ovm_send`

---

This macro processes the item or sequence that has been created using ``ovm_create`. The processing is done without randomization. Essentially, an ``ovm_do` without the `create` or `randomization`.

## ``ovm_send_pri`

---

This is the same as ``ovm_send` except that the sequene item or sequence is executed with the priority specified in the argument.

## ``ovm_rand_send`

---

This macro processes the item or sequence that has been already been allocated (possibly with ``ovm_create`). The processing is done with randomization. Essentially, an ``ovm_do` without the create.

### ``ovm_rand_send_pri`

---

This is the same as ``ovm_rand_send` except that the sequene item or sequence is executed with the priority specified in the argument.

### ``ovm_rand_send_with`

---

This is the same as ``ovm_rand_send` except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.

### ``ovm_rand_send_pri_with`

---

This is the same as ``ovm_rand_send_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, given in a macro argument.

### ``ovm_create_on`

---

This is the same as ``ovm_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.

### ``ovm_do_on`

---

This is the same as ``ovm_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.

## ``ovm_do_on_pri`

---

This is the same as ``ovm_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.

## ``ovm_do_on_with`

---

This is the same as ``ovm_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.

## ``ovm_do_on_pri_with`

---

This is the same as ``ovm_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.

# 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 ovm_component;
  ovm_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, let's 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 ovm_put_ports
//and ovm_put_exports.

`ovm_put_imp_decl(_1)
`ovm_put_imp_decl(_2)

class my_put_imp#(type T=int) extends ovm_component;
  ovm_put_imp_1#(T) put_imp1;
  ovm_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 ``ovm_<interface>_imp_decl` creates a new class of type `ovm_<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

<code>`ovm_blocking_put_imp_decl</code>	Define the class <code>ovm_blocking_put_impSFX</code> for providing blocking put implementations.
<code>`ovm_nonblocking_put_imp_decl</code>	Define the class <code>ovm_nonblocking_put_impSFX</code> for providing non-blocking put implementations.
<code>`ovm_put_imp_decl</code>	Define the class <code>ovm_put_impSFX</code> for providing both blocking and non-blocking put implementations.
<code>`ovm_blocking_get_imp_decl</code>	Define the class <code>ovm_blocking_get_impSFX</code> for providing blocking get implementations.
<code>`ovm_nonblocking_get_imp_decl</code>	Define the class <code>ovm_nonblocking_get_impSFX</code> for providing non-blocking get implementations.
<code>`ovm_get_imp_decl</code>	Define the class <code>ovm_get_impSFX</code> for providing both blocking and non-blocking get implementations.
<code>`ovm_blocking_peek_imp_decl</code>	Define the class <code>ovm_blocking_peek_impSFX</code> for providing blocking peek implementations.
<code>`ovm_nonblocking_peek_imp_decl</code>	Define the class <code>ovm_nonblocking_peek_impSFX</code> for providing non-blocking peek implementations.
<code>`ovm_peek_imp_decl</code>	Define the class <code>ovm_peek_impSFX</code> for providing both blocking and non-blocking peek implementations.
<code>`ovm_blocking_get_peek_imp_decl</code>	Define the class <code>ovm_blocking_get_peek_impSFX</code> for providing the blocking <code>get_peek</code> implementation.
<code>`ovm_nonblocking_get_peek_imp_decl</code>	Define the class <code>ovm_nonblocking_get_peek_impSFX</code> for providing non-blocking <code>get_peek</code> implementation.
<code>`ovm_get_peek_imp_decl</code>	Define the class <code>ovm_get_peek_impSFX</code> for providing both blocking and non-blocking <code>get_peek</code> implementations.
<code>`ovm_blocking_master_imp_decl</code>	Define the class <code>ovm_blocking_master_impSFX</code> for providing the blocking master implementation.
<code>`ovm_nonblocking_master_imp_decl</code>	Define the class <code>ovm_nonblocking_master_impSFX</code> for providing the non-blocking master implementation.
<code>`ovm_master_imp_decl</code>	Define the class <code>ovm_master_impSFX</code> for providing both blocking and non-blocking master implementations.
<code>`ovm_blocking_slave_imp_decl</code>	Define the class <code>ovm_blocking_slave_impSFX</code> for providing the blocking slave implementation.
<code>`ovm_nonblocking_slave_imp_decl</code>	Define the class <code>ovm_nonblocking_slave_impSFX</code> for providing the non-blocking slave implementation.
<code>`ovm_slave_imp_decl</code>	Define the class <code>ovm_slave_impSFX</code> for providing both blocking and non-blocking slave implementations.
<code>`ovm_blocking_transport_imp_decl</code>	Define the class <code>ovm_blocking_transport_impSFX</code> for providing the blocking transport implementation.
<code>`ovm_nonblocking_transport_imp_decl</code>	Define the class <code>ovm_nonblocking_transport_impSFX</code> for providing the non-blocking transport implementation.
<code>`ovm_transport_imp_decl</code>	Define the class <code>ovm_transport_impSFX</code> for providing both blocking and non-blocking transport implementations.
<code>`ovm_analysis_imp_decl</code>	Define the class <code>ovm_analysis_impSFX</code> for providing an analysis implementation.

## Macros



### ``ovm_blocking_put_imp_decl`

---

Define the class `ovm_blocking_put_impSFX` for providing blocking put implementations. *SFX* is the suffix for the new class type.

### ``ovm_nonblocking_put_imp_decl`

---

Define the class `ovm_nonblocking_put_impSFX` for providing non-blocking put implementations. *SFX* is the suffix for the new class type.

### ``ovm_put_imp_decl`

---

Define the class `ovm_put_impSFX` for providing both blocking and non-blocking put implementations. *SFX* is the suffix for the new class type.

### ``ovm_blocking_get_imp_decl`

---

Define the class `ovm_blocking_get_impSFX` for providing blocking get implementations. *SFX* is the suffix for the new class type.

### ``ovm_nonblocking_get_imp_decl`

---

Define the class `ovm_nonblocking_get_impSFX` for providing non-blocking get implementations. *SFX* is the suffix for the new class type.

### ``ovm_get_imp_decl`

---

Define the class `ovm_get_impSFX` for providing both blocking and non-blocking get implementations. *SFX* is the suffix for the new class type.

### ``ovm_blocking_peek_imp_decl`

---

Define the class `ovm_blocking_peek_impSFX` for providing blocking peek implementations. *SFX* is the suffix for the new class type.

### ``ovm_nonblocking_peek_imp_decl`

---

Define the class `ovm_nonblocking_peek_impSFX` for providing non-blocking peek implementations. *SFX* is the suffix for the new class type.

### ``ovm_peek_imp_decl`

---

Define the class `ovm_peek_impSFX` for providing both blocking and non-blocking peek implementations. *SFX* is the suffix for the new class type.

### ``ovm_blocking_get_peek_imp_decl`

---

Define the class `ovm_blocking_get_peek_impSFX` for providing the blocking `get_peek` implementation.

### ``ovm_nonblocking_get_peek_imp_decl`

---

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

### ``ovm_get_peek_imp_decl`

---

Define the class `ovm_get_peek_impSFX` for providing both blocking and non-blocking `get_peek` implementations. *SFX* is the suffix for the new class type.

### ``ovm_blocking_master_imp_decl`

---

Define the class `ovm_blocking_master_impSFX` for providing the blocking master implementation.

### ``ovm_nonblocking_master_imp_decl`

---

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

### ``ovm_master_imp_decl`

---

Define the class `ovm_master_impSFX` for providing both blocking and non-blocking master implementations. *SFX* is the suffix for the new class type.

### ``ovm_blocking_slave_imp_decl`

---

Define the class `ovm_blocking_slave_impSFX` for providing the blocking slave implementation.

### ``ovm_nonblocking_slave_imp_decl`

---

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

### ``ovm_slave_imp_decl`

---

Define the class `ovm_slave_impSFX` for providing both blocking and non-blocking slave implementations. *SFX* is the suffix for the new class type.

### ``ovm_blocking_transport_imp_decl`

---

Define the class `ovm_blocking_transport_impSFX` for providing the blocking transport implementation.

### ``ovm_nonblocking_transport_imp_decl`

---

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

### ``ovm_transport_imp_decl`

---

Define the class `ovm_transport_impSFX` for providing both blocking and non-blocking transport implementations. *SFX* is the suffix for the new class type.

## ``ovm_analysis_imp_decl`

---

Define the class `ovm_analysis_impSFX` for providing an analysis implementation. *SFX* is the suffix for the new class type. The analysis implementation is the write function. The ``ovm_analysis_imp_decl` allows for a scoreboard (or other analysis component) to support input from many places. For example:

```
`ovm_analysis_imp_decl(_ingress)
`ovm_analysis_imp_port(_egress)

class myscoreboard extends ovm_component;
  ovm_analysis_imp_ingress#(mydata, myscoreboard) ingress;
  ovm_analysis_imp_egress#(mydata, myscoreboard) egress;
  mydata ingress_list[$];
  ...

  function new(string name, ovm_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
```

# Callback Macros

## Summary

### ovm\_callback\_defines.svh

#### Callback Macros

<code>`ovm_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>`ovm_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>`ovm_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>`ovm_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> .
<code>`ovm_do_task_callbacks</code>	Calls the given <i>METHOD</i> of all callbacks of type <i>CB</i> registered with the calling object (i.e.
<code>`ovm_do_ext_task_callbacks</code>	This macro is identical to <code>&lt;ovm_do_task_callbacks&gt;</code> macro except there is an additional <i>OBJ</i> argument that allows the user to execute callbacks associated with an external object instance <i>OBJ</i> instead of the calling ( <i>this</i> ) object.

## Callback Macros

### ``ovm_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 *FNC* 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 ovm_cb;
  pure function void my_function (mycomp comp, int addr, int data);
endclass
```

A component would invoke the macro as

```
task mycomp::run();
  int curr_addr, curr_data;
  ...
  `ovm_do_callbacks(mycb, mycomp, my_function(this, curr_addr, curr_data)
  ...
endtask
```

## **`ovm\_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 `<ovm_do_callbacks (CB,T,METHOD)>` 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.

## **`ovm\_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 *FNC* argument.
- *T* is the type associated with the callback. Typically, an instance of type *T* is passed as one the arguments in the *METHOD* call.
- *METHOD* is the method call to invoke, with all required arguments as if they were invoked directly.
- *VAL*, if 1, says return upon the first callback invocation that returns 1. If 0, says return upon the first callback invocation that returns 0.

**For example, given the following callback class definition**

```
virtual class mycb extends ovm_cb;
  pure function bit drop_trans (mycomp comp, my_trans trans);
endclass
```

A component would invoke the macro as

```
task mycomp::run();
  my_trans trans;
  forever begin
    get_port.get(trans);
    if (`ovm_do_callbacks_exit_on(mycb, mycomp, extobj, drop_trans(this,trans), 1)
      ovm_report_info("DROPPED",{"trans dropped: %s",trans.convert2string()});
    // execute transaction
  end
endtask
```

## [`ovm\\_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*.

## [`ovm\\_do\\_task\\_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 is the same as the `<ovm_do_callbacks>` macro except that each callback is executed inside of its own thread. The threads are concurrent, but the execution order of the threads is simulator dependent. The macro does not return until all forked callbacks have completed.

```
virtual class mycb extends ovm_cb;
  pure task my_task(mycomp, int addr, int data);
endclass
```

```
task mycomp::run();
  int curr_addr, curr_data;
  ...
  `ovm_callback(mycb, mycomp, my_task(this, curr_addr, curr_data))
  ...
endtask
```

## ``ovm_do_ext_task_callbacks`

---

This macro is identical to `<ovm_do_task_callbacks>` macro except there is an additional *OBJ* argument that allows the user to execute callbacks associated with an external object instance *OBJ* instead of the calling (*this*) object.



# Types and Enumerations

## Summary

### Types and Enumerations

**ovm\_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`.

**ovm\_radix\_enum**

**ovm\_recursion\_policy\_enum**

### Reporting

**ovm\_severity** Defines all possible values for report severity.

**ovm\_action** Defines all possible values for report actions.

**ovm\_verbosity** Defines standard verbosity levels for reports.

### Port Type

**ovm\_port\_type\_e**

### Sequences

**ovm\_sequence\_state\_enum**

**Default Policy Classes** Policy classes for `ovm_object` basic functions, `ovm_object::copy`, `ovm_object::compare`, `ovm_object::pack`, `ovm_object::unpack`, and `ovm_object::record`.

**ovm\_default\_table\_printer** The table printer is a global object that can be used with `ovm_object::do_print` to get tabular style printing.

**ovm\_default\_tree\_printer** The tree printer is a global object that can be used with `ovm_object::do_print` to get multi-line tree style printing.

**ovm\_default\_line\_printer** The line printer is a global object that can be used with `ovm_object::do_print` to get single-line style printing.

**ovm\_default\_printer** The default printer is a global object that is used by `ovm_object::print` or `ovm_object::sprint` when no specific printer is set.

**ovm\_default\_packer** The default packer policy.

**ovm\_default\_comparer** The default compare policy.

**ovm\_default\_recorder** The default recording policy.

## ovm\_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`.

## ovm\_radix\_enum

**OVM\_BIN** Selects binary (%b) format

**OVM\_DEC** Selects decimal (%d) format

**OVM\_UNSIGNED** Selects unsigned decimal (%u) format

<i>OVM_OCT</i>	Selects octal (%o) format
<i>OVM_HEX</i>	Selects hexadecimal (%h) format
<i>OVM_STRING</i>	Selects string (%s) format
<i>OVM_TIME</i>	Selects time (%t) format
<i>OVM_ENUM</i>	Selects enumeration value (name) format

## **ovm\_recursion\_policy\_enum**

---

<i>OVM_DEEP</i>	Objects are deep copied (object must implement copy method)
<i>OVM_SHALLOW</i>	Objects are shallow copied using default SV copy.
<i>OVM_REFERENCE</i>	Only object handles are copied.

# **Reporting**

---

## **ovm\_severity**

---

Defines all possible values for report severity.

<i>OVM_INFO</i>	Informative message.
<i>OVM_WARNING</i>	Indicates a potential problem.
<i>OVM_ERROR</i>	Indicates a real problem. Simulation continues subject to the configured message action.
<i>OVM_FATAL</i>	Indicates a problem from which simulation can not recover. Simulation exits via \$finish after a #0 delay.

## **ovm\_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>OVM_NO_ACTION</i>	No action is taken
<i>OVM_DISPLAY</i>	Sends the report to the standard output
<i>OVM_LOG</i>	Sends the report to the file(s) for this (severity,id) pair
<i>OVM_COUNT</i>	Counts the number of reports with the COUNT attribute. When this value reaches max_quit_count, the simulation terminates
<i>OVM_EXIT</i>	Terminates the simulation immediately.
<i>OVM_CALL_HOOK</i>	Callback the report hook methods

## **ovm\_verbosity**

---

Defines standard verbosity levels for reports.

*OVM\_NONE* Report is always printed. Verbosity level setting can not disable it.

*OVM\_LOW* Report is issued if configured verbosity is set to *OVM\_LOW* or above.

*OVM\_MEDIUM* Report is issued if configured verbosity is set to *OVM\_MEDIUM* or above.

*OVM\_HIGH* Report is issued if configured verbosity is set to *OVM\_HIGH* or above.

*OVM\_FULL* Report is issued if configured verbosity is set to *OVM\_FULL* or above.

## Port Type

---

### [ovm\\_port\\_type\\_e](#)

---

*OVM\_PORT* The port requires the interface that is its type parameter.

*OVM\_EXPORT* The port provides the interface that is its type parameter via a connection to some other export or implementation.

*OVM\_IMPLEMENTATION* The port provides the interface that is its type parameter, and it is bound to the component that implements the interface.

## Sequences

---

### [ovm\\_sequence\\_state\\_enum](#)

---

*CREATED* The sequence has been allocated.

*PRE\_BODY* The sequence is started and the pre\_body task is being executed.

*BODY* The sequence is started and the body task is being executed.

*POST\_BODY* The sequence is started and the post\_body task is being executed.

*ENDED* The sequence has ended by the completion of the body task.

*STOPPED* The sequence has been forcibly ended by issuing a kill() on the sequence.

*FINISHED* The sequence is completely finished executing.

## Default Policy Classes

---

Policy classes for [ovm\\_object](#) basic functions, [ovm\\_object::copy](#), [ovm\\_object::compare](#), [ovm\\_object::pack](#), [ovm\\_object::unpack](#), and [ovm\\_object::record](#).

### [ovm\\_default\\_table\\_printer](#)

---

```
ovm_table_printer ovm_default_table_printer = new()
```

The table printer is a global object that can be used with `ovm_object::do_print` to get tabular style printing.

### `ovm_default_tree_printer`

---

```
ovm_tree_printer ovm_default_tree_printer = new()
```

The tree printer is a global object that can be used with `ovm_object::do_print` to get multi-line tree style printing.

### `ovm_default_line_printer`

---

```
ovm_line_printer ovm_default_line_printer = new()
```

The line printer is a global object that can be used with `ovm_object::do_print` to get single-line style printing.

### `ovm_default_printer`

---

```
ovm_printer ovm_default_printer = ovm_default_table_printer
```

The default printer is a global object that is used by `ovm_object::print` or `ovm_object::sprint` when no specific printer is set.

The default printer may be set to any legal `ovm_printer` derived type, including the global line, tree, and table printers described above.

### `ovm_default_packer`

---

```
ovm_packer ovm_default_packer = new()
```

The default packer policy. If a specific packer instance is not supplied in calls to `ovm_object::pack` and `ovm_object::unpack`, this instance is selected.

### `ovm_default_comparer`

---

```
ovm_comparer ovm_default_comparer = new()
```

The default compare policy. If a specific comparer instance is not supplied in calls to `ovm_object::compare`, this instance is selected.

## `ovm_default_recorder`

---

```
ovm_recorder ovm_default_recorder = new()
```

The default recording policy. If a specific recorder instance is not supplied in calls to `ovm_object::record`.

# Globals

## Summary

### Globals

#### Simulation Control

`run_test`

Convenience function for `ovm_top.run_test()`.

`ovm_test_done`

An instance of the `ovm_test_done_objection` class, this object is used by components to coordinate when to end the currently running task-based phase.

`global_stop_request`

Convenience function for `ovm_top.stop_request()`.

`set_global_timeout`

Convenience function for `ovm_top.phase_timeout = timeout`.

`set_global_stop_timeout`

Convenience function for `ovm_top.stop_timeout = timeout`.

#### Reporting

`ovm_report_enabled`

Returns 1 if the configured verbosity in `<ovm_top>` is greater than *verbosity* and the action associated with the given *severity* and *id* is not `OVM_NO_ACTION`, else returns 0.

`ovm_report_info`

`ovm_report_warning`

`ovm_report_error`

`ovm_report_fatal`

These methods, defined in package scope, are convenience functions that delegate to the corresponding component methods in *ovm\_top*. do not inadvertently filter them out.

Verbosity is ignored for warnings, errors, and fatals to ensure users

#### Configuration

`set_config_int`

This is the global version of `set_config_int` in `ovm_component`.

`set_config_object`

This is the global version of `set_config_object` in `ovm_component`.

`set_config_string`

This is the global version of `set_config_string` in `ovm_component`.

#### Miscellaneous

`ovm_is_match`

Returns 1 if the two strings match, 0 otherwise.

`ovm_string_to_bits`

Converts an input string to its bit-vector equivalent.

`ovm_bits_to_string`

Converts an input bit-vector to its string equivalent.

`ovm_wait_for_nba_region`

Call this task to wait for a delta cycle.

## Simulation Control

## run\_test

---

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

Convenience function for `ovm_top.run_test()`. See `ovm_root` for more information.

## ovm\_test\_done

---

```
ovm_test_done_objection ovm_test_done = ovm_test_done_objection::get()
```

An instance of the [ovm\\_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 `ovm_top.stop_request()`. See `ovm_root` for more information.

## set\_global\_timeout

---

```
function void set_global_timeout(time timeout)
```

Convenience function for `ovm_top.phase_timeout = timeout`. See `ovm_root` for more information.

## set\_global\_stop\_timeout

---

```
function void set_global_stop_timeout(time timeout)
```

Convenience function for `ovm_top.stop_timeout = timeout`. See `ovm_root` for more information.

# Reporting

---

## ovm\_report\_enabled

---

```
function bit ovm_report_enabled (int      verbosity,
                                ovm_severity severity = OVM_INFO,
                                string    id         = " ")
```

Returns 1 if the configured verbosity in <ovm\_top> is greater than *verbosity* and the action associated with the given *severity* and *id* is not OVM\_NO\_ACTION, else returns 0.

See also [ovm\\_report\\_object::ovm\\_report\\_enabled](#).

Static methods of an extension of `ovm_report_object`, e.g. `ovm_compoent`-based objects, can not call `ovm_report_enabled` because the call will resolve to the [ovm\\_report\\_object::ovm\\_report\\_enabled](#), which is non-static. Static methods can not call non-static methods of the same class.

## ovm\_report\_info

---

```
function void ovm_report_info(string id,
                              string message,
                              int    verbosity = OVM_MEDIUM,
                              string filename = "",
                              int    line     = 0)
```

## ovm\_report\_warning

---

```
function void ovm_report_warning(string id,
                                 string message,
                                 int    verbosity = OVM_MEDIUM,
                                 string filename = "",
                                 int    line     = 0)
```

## ovm\_report\_error

---

```
function void ovm_report_error(string id,
                               string message,
                               int    verbosity = OVM_LOW,
                               string filename = "",
                               int    line     = 0)
```



## ovm\_report\_fatal

---

These methods, defined in package scope, are convenience functions that delegate to the corresponding component methods in *ovm\_top*. They can be used in module-based code to use the same reporting mechanism as class-based components. See [ovm\\_report\\_object](#) for details on the reporting mechanism.

## 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,
                             ovm_bitstream_t value      )
```

This is the global version of `set_config_int` in [ovm\\_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 [ovm\\_component::set\\_config\\_int](#) for details on setting configuration.

### set\_config\_object

---

```
function void set_config_object (string      inst_name,
                                string      field_name,
                                ovm_object value,
                                bit         clone      = 1      )
```

This is the global version of `set_config_object` in [ovm\\_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 [ovm\\_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 [ovm\\_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 [ovm\\_component::set\\_config\\_string](#) for details on setting configuration.

## Miscellaneous

---

### [ovm\\_is\\_match](#)

---

```
`ifdef OVM_DPI import "DPI" function bit ovm_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.

### [ovm\\_string\\_to\\_bits](#)

---

```
function logic[OVM_LARGE_STRING:0] ovm_string_to_bits(string str)
```

Converts an input string to its bit-vector equivalent. Max bit-vector length is approximately 14000 characters.

### [ovm\\_bits\\_to\\_string](#)

---

```
function string ovm_bits_to_string(logic [OVM_LARGE_STRING:0] str)
```

Converts an input bit-vector to its string equivalent. Max bit-vector length is approximately 14000 characters.

### [ovm\\_wait\\_for\\_nba\\_region](#)

---

```
task ovm_wait_for_nba_region
```

Call this task to wait for a delta cycle. Program blocks don't have an nba so just delay for a #0 in a program block.

# 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](#)

## \$#!

- [ovm\\_analysis\\_imp\\_decl](#)
- [ovm\\_blocking\\_get\\_imp\\_decl](#)
- [ovm\\_blocking\\_get\\_peek\\_imp\\_decl](#)
- [ovm\\_blocking\\_master\\_imp\\_decl](#)
- [ovm\\_blocking\\_peek\\_imp\\_decl](#)
- [ovm\\_blocking\\_put\\_imp\\_decl](#)
- [ovm\\_blocking\\_slave\\_imp\\_decl](#)
- [ovm\\_blocking\\_transport\\_imp\\_decl](#)
- [ovm\\_component\\_end](#)
- [ovm\\_component\\_param\\_utils](#)
- [ovm\\_component\\_param\\_utils\\_begin](#)
- [ovm\\_component\\_utils](#)
- [ovm\\_component\\_utils\\_begin](#)
- [ovm\\_create](#)
- [ovm\\_create\\_on](#)
- [ovm\\_declare\\_p\\_sequencer](#)
- [ovm\\_do](#)
- [ovm\\_do\\_callbacks](#)
- [ovm\\_do\\_callbacks\\_exit\\_on](#)
- [ovm\\_do\\_ext\\_task\\_callbacks](#)
- [ovm\\_do\\_obj\\_callbacks](#)
- [ovm\\_do\\_obj\\_callbacks\\_exit\\_on](#)
- [ovm\\_do\\_on](#)
- [ovm\\_do\\_on\\_pri](#)
- [ovm\\_do\\_on\\_pri\\_with](#)
- [ovm\\_do\\_on\\_with](#)
- [ovm\\_do\\_pri](#)
- [ovm\\_do\\_pri\\_with](#)
- [ovm\\_do\\_task\\_callbacks](#)
- [ovm\\_do\\_with](#)
- [ovm\\_error](#)
- [ovm\\_fatal](#)
- [ovm\\_field\\_\\*macros](#)
- [ovm\\_field\\_aa\\_\\*\\_int macros](#)
- [ovm\\_field\\_aa\\_\\*\\_string macros](#)
- [ovm\\_field\\_aa\\_int\\_byte](#)
- [ovm\\_field\\_aa\\_int\\_byte\\_unsigned](#)
- [ovm\\_field\\_aa\\_int\\_enumkey](#)
- [ovm\\_field\\_aa\\_int\\_int](#)
- [ovm\\_field\\_aa\\_int\\_int\\_unsigned](#)
- [ovm\\_field\\_aa\\_int\\_integer](#)
- [ovm\\_field\\_aa\\_int\\_integer\\_unsigned](#)
- [ovm\\_field\\_aa\\_int\\_key](#)
- [ovm\\_field\\_aa\\_int\\_longint](#)
- [ovm\\_field\\_aa\\_int\\_longint\\_unsigned](#)
- [ovm\\_field\\_aa\\_int\\_shortint](#)
- [ovm\\_field\\_aa\\_int\\_shortint\\_unsigned](#)

- ` ovm\_field\_aa\_int\_string
- ` ovm\_field\_aa\_object\_int
- ` ovm\_field\_aa\_object\_string
- ` ovm\_field\_aa\_string\_string
- ` ovm\_field\_array\_\*macros
- ` ovm\_field\_array\_enum
- ` ovm\_field\_array\_int
- ` ovm\_field\_array\_object
- ` ovm\_field\_array\_string
- ` ovm\_field\_enum
- ` ovm\_field\_event
- ` ovm\_field\_int
- ` ovm\_field\_object
- ` ovm\_field\_queue\_\*macros
- ` ovm\_field\_queue\_enum
- ` ovm\_field\_queue\_int
- ` ovm\_field\_queue\_object
- ` ovm\_field\_queue\_string
- ` ovm\_field\_real
- ` ovm\_field\_sarray\_\*macros
- ` ovm\_field\_sarray\_enum
- ` ovm\_field\_sarray\_int
- ` ovm\_field\_sarray\_object
- ` ovm\_field\_sarray\_string
- ` ovm\_field\_string
- ` ovm\_field\_utils\_begin
- ` ovm\_field\_utils\_end
- ` ovm\_get\_imp\_decl
- ` ovm\_get\_peek\_imp\_decl
- ` ovm\_info
- ` ovm\_master\_imp\_decl
- ` ovm\_nonblocking\_get\_imp\_decl
- ` ovm\_nonblocking\_get\_peek\_imp\_decl
- ` ovm\_nonblocking\_master\_imp\_decl
- ` ovm\_nonblocking\_peek\_imp\_decl
- ` ovm\_nonblocking\_put\_imp\_decl
- ` ovm\_nonblocking\_slave\_imp\_decl
- ` ovm\_nonblocking\_transport\_imp\_decl
- ` ovm\_object\_param\_utils
- ` ovm\_object\_param\_utils\_begin
- ` ovm\_object\_utils
- ` ovm\_object\_utils\_begin
- ` ovm\_object\_utils\_end
- ` ovm\_peek\_imp\_decl
- ` ovm\_phase\_func\_bottomup\_decl
- ` ovm\_phase\_func\_decl
- ` ovm\_phase\_func\_topdown\_decl
- ` ovm\_phase\_task\_bottomup\_decl
- ` ovm\_phase\_task\_decl
- ` ovm\_phase\_task\_topdown\_decl
- ` ovm\_put\_imp\_decl
- ` ovm\_rand\_send
- ` ovm\_rand\_send\_pri
- ` ovm\_rand\_send\_pri\_with
- ` ovm\_rand\_send\_with

- ` ovm\_send
- ` ovm\_send\_pri
- ` ovm\_sequence\_utils
- ` ovm\_sequence\_utils\_begin
- ` ovm\_sequence\_utils\_end
- ` ovm\_sequencer\_param\_utils
- ` ovm\_sequencer\_param\_utils\_begin
- ` ovm\_sequencer\_utils
- ` ovm\_sequencer\_utils\_begin
- ` ovm\_sequencer\_utils\_end
- ` ovm\_slave\_imp\_decl
- ` ovm\_transport\_imp\_decl
- ` ovm\_update\_sequence\_lib
- ` ovm\_update\_sequence\_lib\_and\_item
- ` ovm\_warning

## A

### **abstract**

- ovm\_comparer
- ovm\_packer
- ovm\_recorder

### **accept\_tr**

- ovm\_component
- ovm\_transaction

### **add**

- ovm\_pool#(T)

### **add\_callback**

- ovm\_event

### **add\_cb**

- ovm\_callbacks#(T,CB)

### **add\_sequence**

- ovm\_sequencer\_base

### **after\_export**

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

### **all\_dropped**

- ovm\_component
- ovm\_objection
- ovm\_root
- ovm\_test\_done\_objection

### **Analysis**

- Global
- tIm\_if\_base#(T1,T2)

### **analysis\_export**

- ovm\_subscriber

### **analysis\_port#(T)**

- tIm\_analysis\_fifo#(T)

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

- ovm\_component

## B

**before\_export**

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

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

**begin\_child\_tr**

ovm\_component

ovm\_transaction

**begin\_elements**

ovm\_printer\_knobs

**begin\_tr**

ovm\_component

ovm\_transaction

**Bidirectional Interfaces&Ports****big\_endian**

ovm\_packer

**bin\_radix**

ovm\_printer\_knobs

**Blocking get**

tlm\_if\_base#(T1,T2)

**Blocking peek**

tlm\_if\_base#(T1,T2)

**Blocking put**

tlm\_if\_base#(T1,T2)

**Blocking transport**

tlm\_if\_base#(T1,T2)

**blocking\_put\_port**

ovm\_random\_stimulus#(T)

**body**

ovm\_sequence\_base

**BODY****build**

ovm\_component

**C****call\_func**

ovm\_phase

**call\_task**

ovm\_phase

**Callback Hooks**

ovm\_objection

**Callback Macros****callback\_mode**

ovm\_callback

**Callbacks**

ovm\_report\_object

**can\_get**

tlm\_if\_base#(T1,T2)

**can\_peek**

tlm\_if\_base#(T1,T2)

**can\_put**[tlm\\_if\\_base#\(T1,T2\)](#)**cancel**[ovm\\_barrier](#)[ovm\\_event](#)**CB**[ovm\\_callbacks#\(T,CB\)](#)**check**[ovm\\_component](#)**check\_config\_usage**[ovm\\_component](#)**check\_type**[ovm\\_comparer](#)**clone**[ovm\\_object](#)**Comparators**[comparators.txt](#)[methodology/ovm\\_algorithmic\\_comparator.svh](#)**compare**[ovm\\_object](#)**compare\_field**[ovm\\_comparer](#)**compare\_field\_int**[ovm\\_comparer](#)**compare\_field\_real**[ovm\\_comparer](#)**compare\_object**[ovm\\_comparer](#)**compare\_string**[ovm\\_comparer](#)**Comparing**[ovm\\_object](#)**compose\_message**[ovm\\_report\\_server](#)**Configuration**[Global](#)[ovm\\_object](#)[ovm\\_report\\_object](#)**Configuration Interface**[ovm\\_component](#)**connect**[ovm\\_component](#)[ovm\\_port\\_base#\(IF\)](#)**convert2string**[ovm\\_object](#)**copy**[ovm\\_object](#)



**Copying**

ovm\_object

**Core Base Classes****count**

ovm\_sequencer\_base

**create**

ovm\_component\_registry#(T,Tname)

ovm\_object

ovm\_object\_registry#(T,Tname)

**create\_component**

ovm\_component

ovm\_component\_registry#(T,Tname)

ovm\_object\_wrapper

**create\_component\_by\_name**

ovm\_factory

**create\_component\_by\_type**

ovm\_factory

**create\_item**

ovm\_sequence\_base

**create\_object**

ovm\_component

ovm\_object\_registry#(T,Tname)

ovm\_object\_wrapper

**create\_object\_by\_name**

ovm\_factory

**create\_object\_by\_type**

ovm\_factory

**CREATED****Creation**

ovm\_factory

ovm\_object

**current\_grabber**

ovm\_sequencer\_base

**D****Debug**

ovm\_factory

**debug\_connected\_to**

ovm\_port\_base#(IF)

**debug\_create\_by\_name**

ovm\_factory

**debug\_create\_by\_type**

ovm\_factory

**debug\_provided\_to**

ovm\_port\_base#(IF)

**dec\_radix**

ovm\_printer\_knobs

**Default Policy Classes**

**default\_radix**

ovm\_printer\_knobs

ovm\_recorder

**default\_sequence**

ovm\_sequencer\_base

**delete**

ovm\_barrier\_pool

ovm\_event\_pool

ovm\_object\_string\_pool#(T)

ovm\_pool#(T)

ovm\_queue#(T)

**delete\_callback**

ovm\_event

**delete\_cb**

ovm\_callbacks#(T,CB)

**depth**

ovm\_printer\_knobs

**die**

ovm\_report\_object

**disable\_recording**

ovm\_transaction

**display\_cbs**

ovm\_callbacks#(T,CB)

**display\_objections**

ovm\_objection

**do\_accept\_tr**

ovm\_component

ovm\_transaction

**do\_begin\_tr**

ovm\_component

ovm\_transaction

**do\_compare**

ovm\_object

**do\_copy**

ovm\_object

**do\_end\_tr**

ovm\_component

ovm\_transaction

**do\_kill\_all**

ovm\_component

**do\_pack**

ovm\_object

**do\_print**

ovm\_object

**do\_record**

ovm\_object

**do\_sequence\_kind**

ovm\_sequence\_base

**do\_unpack**

ovm\_object

**drop**

ovm\_test\_done\_objection

**drop\_objection**

ovm\_objection

**dropped**

ovm\_component

ovm\_objection

**dump\_report\_state**

ovm\_report\_object

**dump\_server\_state**

ovm\_report\_server

**E****enable\_print\_topology**

ovm\_root

**enable\_recording**

ovm\_transaction

**enable\_stop\_interrupt**

ovm\_component

**end**

methodology/sequences/ovm\_sequence\_builtin.svh

tlm/sqr\_connections.svh

**end\_elements**

ovm\_printer\_knobs

**end\_of\_elaboration**

ovm\_component

**end\_tr**

ovm\_component

ovm\_transaction

**ENDED****execute\_item**

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

**exists**

ovm\_barrier\_pool

ovm\_event\_pool

ovm\_pool#(T)

**extract**

ovm\_component

**F****Factory Classes****Factory Interface**

ovm\_component

**Field Macros****Fields declared in**`<`ovm_field_*>`**macros,if used,will not**

ovm\_object

**find**

ovm\_root

**find\_all**

ovm\_root

**find\_override\_by\_name**

ovm\_factory

**find\_override\_by\_type**

ovm\_factory

**finish\_item**

ovm\_sequence\_base

ovm\_sequence\_item

**finish\_on\_completion**

ovm\_root

**FINISHED****first**

ovm\_barrier\_pool

ovm\_event\_pool

ovm\_pool#(T)

**flush**

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

tlm\_fifo#(T)

**footer**

ovm\_printer\_knobs

**force\_stop**

ovm\_test\_done\_objection

**format\_action**

ovm\_report\_handler

**full\_name**

ovm\_printer\_knobs

# 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](#)

## G

### **generate\_stimulus**

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

### **get**

[ovm\\_barrier\\_pool](#)

[ovm\\_component\\_registry#\(T,Tname\)](#)

[ovm\\_event\\_pool](#)

[ovm\\_object\\_registry#\(T,Tname\)](#)

[ovm\\_object\\_string\\_pool#\(T\)](#)

[ovm\\_pool#\(T\)](#)

[ovm\\_queue#\(T\)](#)

[sqr\\_if\\_base#\(REQ,RSP\)](#)

[tlm\\_if\\_base#\(T1,T2\)](#)

### **Get and Peek**

### **get\_accept\_time**

[ovm\\_transaction](#)

### **get\_action**

[ovm\\_report\\_handler](#)

### **get\_ap**

[tlm\\_fifo\\_base#\(T\)](#)

### **get\_begin\_time**

[ovm\\_transaction](#)

### **get\_child**

[ovm\\_component](#)

### **get\_comp**

[ovm\\_port\\_base#\(IF\)](#)

### **get\_config\_int**

[ovm\\_component](#)

### **get\_config\_object**

[ovm\\_component](#)

### **get\_config\_string**

[ovm\\_component](#)

**get\_count**

ovm\_random\_sequence

**get\_current\_item**

ovm\_sequence#(REQ,RSP)

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

**get\_current\_phase**

ovm\_root

**get\_depth**

ovm\_sequence\_item

**get\_drain\_time**

ovm\_objection

**get\_end\_time**

ovm\_transaction

**get\_event\_pool**

ovm\_transaction

**get\_file\_handle**

ovm\_report\_handler

**get\_first\_child**

ovm\_component

**get\_full\_name**

ovm\_component

ovm\_object

ovm\_port\_base#(IF)

**get\_global**

ovm\_pool#(T)

ovm\_queue#(T)

**get\_global\_cbs**

ovm\_callbacks#(T,CB)

**get\_global\_pool**

ovm\_barrier\_pool

ovm\_event\_pool

ovm\_object\_string\_pool#(T)

ovm\_pool#(T)

**get\_global\_queue**

ovm\_queue#(T)

**get\_id\_count**

ovm\_report\_server

**get\_if**

ovm\_port\_base#(IF)

**get\_initiator**

ovm\_transaction

**get\_inst\_count**

ovm\_object

**get\_inst\_id**

ovm\_object

**get\_max\_quit\_count**

ovm\_report\_server

**get\_name**

ovm\_object

ovm\_phase

ovm\_port\_base#(IF)

**get\_next\_child**

ovm\_component

**get\_next\_item**

sqr\_if\_base#(REQ,RSP)

**get\_num\_children**

ovm\_component

**get\_num\_last\_reqs**

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

**get\_num\_last\_rsps**

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

**get\_num\_reqs\_sent**

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

**get\_num\_rsps\_received**

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

**get\_num\_waiters**

ovm\_barrier

ovm\_event

**get\_object\_type**

ovm\_object

**get\_objection\_count**

ovm\_objection

**get\_objection\_total**

ovm\_objection

**get\_packed\_size**

ovm\_packer

**get\_parent**

ovm\_component  
ovm\_port\_base#(IF)

**get\_parent\_sequence**

ovm\_sequence\_item

**get\_peek\_export**

tlm\_fifo\_base#(T)

**get\_peek\_request\_export**

tlm\_req\_rsp\_channel#(REQ,RSP)

**get\_peek\_response\_export**

tlm\_req\_rsp\_channel#(REQ,RSP)

**get\_phase\_by\_name**

ovm\_root

**get\_priority**

ovm\_sequence\_base

**get\_quit\_count**

ovm\_report\_server

**get\_radix\_str**

ovm\_printer\_knobs

**get\_report\_action**

ovm\_report\_object

**get\_report\_file\_handle**

ovm\_report\_object

**get\_report\_handler**

ovm\_report\_object

**get\_report\_server**

ovm\_report\_object

**get\_report\_verbosity\_level**

ovm\_report\_object

**get\_response**

ovm\_sequence#(REQ,RSP)

**get\_response\_queue\_depth**

ovm\_sequence#(REQ,RSP)

**get\_response\_queue\_error\_report\_disabled**

ovm\_sequence#(REQ,RSP)

**get\_root\_sequence**

ovm\_sequence\_item



**get\_root\_sequence\_name**

ovm\_sequence\_item

**get\_seq\_kind**

ovm\_sequence\_base

ovm\_sequencer\_base

**get\_sequence**

ovm\_sequence\_base

ovm\_sequencer\_base

**get\_sequence\_by\_name**

ovm\_sequence\_base

**get\_sequence\_id**

ovm\_sequence\_item

**get\_sequence\_path**

ovm\_sequence\_item

**get\_sequence\_state**

ovm\_sequence\_base

**get\_sequencer**

ovm\_sequence\_base

ovm\_sequence\_item

**get\_server**

ovm\_report\_server

**get\_severity\_count**

ovm\_report\_server

**get\_threshold**

ovm\_barrier

**get\_tr\_handle**

ovm\_transaction

**get\_transaction\_id**

ovm\_transaction

**get\_trigger\_data**

ovm\_event

**get\_trigger\_time**

ovm\_event

**get\_type**

ovm\_object

**get\_type\_name**

ovm\_callback

ovm\_component\_registry#(T,Tname)

ovm\_object

ovm\_object\_registry#(T,Tname)

ovm\_object\_string\_pool#(T)

ovm\_object\_wrapper

ovm\_phase

ovm\_port\_base#(IF)

**get\_use\_response\_handler**

ovm\_sequence\_base

**get\_use\_sequence\_info**

ovm\_sequence\_item

**get\_verbosity\_level**

ovm\_report\_handler

**global\_indent**

ovm\_printer\_knobs

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

ovm\_sequence\_base

ovm\_sequencer\_base

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

ovm\_component

**has\_do\_available**

ovm\_sequencer\_base

sqr\_if\_base#(REQ,RSP)

**has\_lock**

ovm\_sequence\_base

ovm\_sequencer\_base

**header**

ovm\_printer\_knobs

**hex\_radix**

ovm\_printer\_knobs

**Hierarchical Reporting Interface**

ovm\_component

**Hierarchy Interface**

ovm\_component

**id\_count**[ovm\\_report\\_server](#)**Identification**[ovm\\_object](#)**identifier**[ovm\\_printer\\_knobs](#)[ovm\\_recorder](#)**in\_order\_built\_in\_comparator#(T)****in\_order\_class\_comparator#(T)****in\_stop\_request**[ovm\\_root](#)**incr\_id\_count**[ovm\\_report\\_server](#)**incr\_quit\_count**[ovm\\_report\\_server](#)**incr\_severity\_count**[ovm\\_report\\_server](#)**indent\_str**[ovm\\_hier\\_printer\\_knobs](#)**insert**[ovm\\_queue#\(T\)](#)**insert\_phase**[ovm\\_root](#)**is\_active**[ovm\\_transaction](#)**is\_blocked**[ovm\\_sequence\\_base](#)[ovm\\_sequencer\\_base](#)**is\_child**[ovm\\_sequencer\\_base](#)**is\_done**[ovm\\_phase](#)**is\_empty**[tlm\\_fifo#\(T\)](#)**is\_enabled**[ovm\\_callback](#)

**is\_export**

ovm\_port\_base#(IF)

**is\_full**

tlm\_fifo#(T)

**is\_grabbed**

ovm\_sequencer\_base

**is\_imp**

ovm\_port\_base#(IF)

**is\_in\_progress**

ovm\_phase

**is\_item**

ovm\_sequence\_base

ovm\_sequence\_item

**is\_null**

ovm\_packer

**is\_off**

ovm\_event

**is\_on**

ovm\_event

**is\_port**

ovm\_port\_base#(IF)

**is\_quit\_count\_reached**

ovm\_report\_server

**is\_recording\_enabled**

ovm\_transaction

**is\_relevant**

ovm\_sequence\_base

**is\_task**

ovm\_phase

**is\_top\_down**

ovm\_phase

**is\_unbounded**

ovm\_port\_base#(IF)

**item\_done**

sqr\_if\_base#(REQ,RSP)

**K**

**kill**

ovm\_component  
ovm\_sequence\_base

**knobs**

ovm\_printer  
ovm\_table\_printer  
ovm\_tree\_printer

**L****last**

ovm\_barrier\_pool  
ovm\_event\_pool  
ovm\_pool#(T)

**last\_req**

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

**last\_rsp**

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

**lock**

ovm\_sequence\_base  
ovm\_sequencer\_base

**lookup**

ovm\_component

**M****Macros**

base/ovm\_phases.sv  
macros/ovm\_message\_defines.svh  
macros/tlm\_defines.svh

**Master and Slave****master\_export**

tlm\_req\_rsp\_channel#(REQ,RSP)

**max\_random\_count**

ovm\_sequencer\_base

**max\_random\_depth**

ovm\_sequencer\_base

**max\_size**

ovm\_port\_base#(IF)

**max\_width**

ovm\_printer\_knobs

**mcd**

ovm\_printer\_knobs

**Methods**

ovm\_\*\_export#(REQ,RSP)  
ovm\_\*\_export#(T)  
ovm\_\*\_imp#(REQ,RSP,IMP,REQ\_IMP,RSP\_IMP)  
ovm\_\*\_imp#(T,IMP)  
ovm\_\*\_port#(REQ,RSP)  
ovm\_\*\_port#(T)  
ovm\_agent  
ovm\_algorithmic\_comparator#(BEFORE,AFTER,TRANSFORMER)  
ovm\_barrier  
ovm\_barrier\_pool  
ovm\_built\_in\_pair#(T1,T2)  
ovm\_callback  
ovm\_callbacks#(T,CB)  
ovm\_comparer  
ovm\_component\_registry#(T,Tname)  
ovm\_driver#(REQ,RSP)  
ovm\_env  
ovm\_event  
ovm\_event\_callback  
ovm\_event\_pool  
ovm\_in\_order\_comparator#(T,comp\_type,convert,pair\_type)  
ovm\_line\_printer  
ovm\_monitor  
ovm\_object\_string\_pool#(T)  
ovm\_object\_wrapper  
ovm\_pair#(T1,T2)  
ovm\_phase  
ovm\_pool#(T)  
ovm\_port\_base#(IF)  
ovm\_printer\_knobs  
ovm\_push\_driver#(REQ,RSP)  
ovm\_push\_sequencer#(REQ,RSP)  
ovm\_queue#(T)  
ovm\_random\_sequence  
ovm\_random\_stimulus#(T)  
ovm\_recorder  
ovm\_report\_handler  
ovm\_report\_server  
ovm\_root  
ovm\_scoreboard  
ovm\_sequence#(REQ,RSP)  
ovm\_sequence\_base  
ovm\_sequence\_item  
ovm\_sequencer#(REQ,RSP)

- ovm\_sequencer\_base
- ovm\_sequencer\_param\_base#(REQ,RSP)
- ovm\_subscriber
- ovm\_test
- ovm\_test\_done\_objection
- ovm\_transaction
- sqr\_if\_base#(REQ,RSP)
- tlm\_analysis\_fifo#(T)
- tlm\_fifo#(T)
- tlm\_fifo\_base#(T)
- tlm\_req\_rsp\_channel#(REQ,RSP)
- tlm\_transport\_channel#(REQ,RSP)

### Methods for printer subtyping

- ovm\_printer

### Methods for printer usage

- ovm\_printer

### mid\_do

- ovm\_sequence\_base

### min\_size

- ovm\_port\_base#(IF)

### Miscellaneous

#### miscompares

- ovm\_comparer

## N

### name\_width

- ovm\_table\_printer\_knobs

### nb\_transport

- tlm\_if\_base#(T1,T2)

### new

- ovm\_\*\_export#(REQ,RSP)
- ovm\_\*\_export#(T)
- ovm\_\*\_imp#(REQ,RSP,IMP,REQ\_IMP,RSP\_IMP)
- ovm\_\*\_imp#(T,IMP)
- ovm\_\*\_port#(REQ,RSP)
- ovm\_\*\_port#(T)
- ovm\_agent
- ovm\_algorithmic\_comparator#(BEFORE,AFTER,TRANSFORMER)
- ovm\_barrier
- ovm\_barrier\_pool
- ovm\_built\_in\_pair#(T1,T2)
- ovm\_callback

ovm\_callbacks#(T,CB)  
ovm\_component  
ovm\_driver#(REQ,RSP)  
ovm\_env  
ovm\_event  
ovm\_event\_callback  
ovm\_event\_pool  
ovm\_line\_printer  
ovm\_monitor  
ovm\_object  
ovm\_object\_string\_pool#(T)  
ovm\_objection  
ovm\_pair#(T1,T2)  
ovm\_phase  
ovm\_pool#(T)  
ovm\_port\_base#(IF)  
ovm\_push\_driver#(REQ,RSP)  
ovm\_push\_sequencer#(REQ,RSP)  
ovm\_queue#(T)  
ovm\_random\_stimulus#(T)  
ovm\_report\_handler  
ovm\_report\_object  
ovm\_report\_server  
ovm\_scoreboard  
ovm\_sequence#(REQ,RSP)  
ovm\_sequence\_base  
ovm\_sequence\_item  
ovm\_sequencer#(REQ,RSP)  
ovm\_sequencer\_base  
ovm\_sequencer\_param\_base#(REQ,RSP)  
ovm\_subscriber  
ovm\_table\_printer  
ovm\_test  
ovm\_transaction  
ovm\_tree\_printer  
tlm\_analysis\_fifo#(T)  
tlm\_fifo#(T)  
tlm\_fifo\_base#(T)  
tlm\_req\_rsp\_channel#(REQ,RSP)  
tlm\_transport\_channel#(REQ,RSP)

**next**

ovm\_barrier\_pool  
ovm\_event\_pool  
ovm\_pool#(T)



**Non-blocking get**

tlm\_if\_base#(T1,T2)

**Non-blocking peek**

tlm\_if\_base#(T1,T2)

**Non-blocking put**

tlm\_if\_base#(T1,T2)

**Non-blocking transport**

tlm\_if\_base#(T1,T2)

**num**

ovm\_barrier\_pool

ovm\_event\_pool

ovm\_pool#(T)

**num\_sequences**

ovm\_sequence\_base

ovm\_sequencer\_base

# 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](#)

## O

### Objection Control

[ovm\\_objection](#)

### Objection Interface

[ovm\\_component](#)

### Objection Status

[ovm\\_objection](#)

### oct\_radix

[ovm\\_printer\\_knobs](#)

### OVM Class Reference

#### OVM Factory

[ovm\\_\\*\\_export#\(REQ,RSP\)](#)

[ovm\\_\\*\\_export#\(T\)](#)

[ovm\\_\\*\\_imp\\_ports](#)

[ovm\\_\\*\\_imp#\(REQ,RSP,IMP,REQ\\_IMP,RSP\\_IMP\)](#)

[ovm\\_\\*\\_imp#\(T,IMP\)](#)

[ovm\\_\\*\\_port#\(REQ,RSP\)](#)

[ovm\\_\\*\\_port#\(T\)](#)

[ovm\\_action](#)

[ovm\\_agent](#)

[ovm\\_algorithmic\\_comparator#\(BEFORE,AFTER,TRANSFORMER\)](#)

[ovm\\_algorithmic\\_comparator.svh](#)

[ovm\\_barrier](#)

[ovm\\_barrier\\_pool](#)

[OVM\\_BIN](#)

[ovm\\_bits\\_to\\_string](#)

[ovm\\_bitstream\\_t](#)

[ovm\\_built\\_in\\_clone#\(T\)](#)

[ovm\\_built\\_in\\_comp#\(T\)](#)

[ovm\\_built\\_in\\_converter#\(T\)](#)

[ovm\\_built\\_in\\_pair#\(T1,T2\)](#)

[OVM\\_CALL\\_HOOK](#)

[ovm\\_callback](#)

[ovm\\_callback\\_defines.svh](#)

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

[ovm\\_class\\_clone#\(T\)](#)

[ovm\\_class\\_comp#\(T\)](#)

[ovm\\_class\\_converter#\(T\)](#)

[ovm\\_comparer](#)

[ovm\\_component](#)

[ovm\\_component\\_registry#\(T,Tname\)](#)

[OVM\\_COUNT](#)

OVM\_DEC  
OVM\_DEEP  
ovm\_default\_comparer  
ovm\_default\_line\_printer  
ovm\_default\_packer  
ovm\_default\_printer  
ovm\_default\_recorder  
ovm\_default\_table\_printer  
ovm\_default\_tree\_printer  
OVM\_DISPLAY  
ovm\_driver#(REQ,RSP)  
OVM\_ENUM  
ovm\_env  
OVM\_ERROR  
ovm\_event  
ovm\_event\_callback  
ovm\_event\_pool  
ovm\_exhaustive\_sequence  
OVM\_EXIT  
OVM\_EXPORT  
ovm\_factory  
OVM\_FATAL  
OVM\_FULL  
OVM\_HEX  
ovm\_hier\_printer\_knobs  
OVM\_HIGH  
OVM\_IMPLEMENTATION  
ovm\_in\_order\_comparator#(T,comp\_type,convert,pair\_type)  
OVM\_INFO  
ovm\_is\_match  
ovm\_line\_printer  
OVM\_LOG  
OVM\_LOW  
OVM\_MEDIUM  
ovm\_monitor  
OVM\_NO\_ACTION  
OVM\_NONE  
ovm\_object  
ovm\_object\_registry#(T,Tname)  
ovm\_object\_string\_pool#(T)  
ovm\_object\_wrapper  
ovm\_objection  
OVM\_OCT  
ovm\_packer  
ovm\_pair#(T1,T2)  
ovm\_phase  
ovm\_policies.svh  
ovm\_pool#(T)  
OVM\_PORT  
ovm\_port\_base#(IF)

**ovm\_port\_type\_e**  
**ovm\_printer**  
**ovm\_printer\_knobs**  
**ovm\_push\_driver#(REQ,RSP)**  
**ovm\_push\_sequencer#(REQ,RSP)**  
**ovm\_queue#(T)**  
**ovm\_radix\_enum**  
**ovm\_random\_sequence**  
**ovm\_random\_stimulus#(T)**  
**ovm\_recorder**  
**ovm\_recursion\_policy\_enum**  
**OVM\_REFERENCE**  
**ovm\_report\_enabled**  
[Global](#)  
[ovm\\_report\\_object](#)  
  
**ovm\_report\_error**  
[Global](#)  
[ovm\\_report\\_object](#)  
  
**ovm\_report\_fatal**  
[Global](#)  
[ovm\\_report\\_object](#)  
  
**ovm\_report\_handler**  
**ovm\_report\_info**  
[Global](#)  
[ovm\\_report\\_object](#)  
  
**ovm\_report\_object**  
**ovm\_report\_server**  
**ovm\_report\_warning**  
[Global](#)  
[ovm\\_report\\_object](#)  
  
**ovm\_root**  
**ovm\_scoreboard**  
**ovm\_seq\_item\_pull\_export#(REQ,RSP)**  
**ovm\_seq\_item\_pull\_imp#(REQ,RSP,IMP)**  
**ovm\_seq\_item\_pull\_port#(REQ,RSP)**  
**ovm\_sequence#(REQ,RSP)**  
**ovm\_sequence\_base**  
**ovm\_sequence\_item**  
**ovm\_sequence\_state\_enum**  
**ovm\_sequencer#(REQ,RSP)**  
**ovm\_sequencer\_base**  
**ovm\_sequencer\_param\_base#(REQ,RSP)**  
**ovm\_severity**  
**OVM\_SHALLOW**  
**ovm\_simple\_sequence**  
**OVM\_STRING**  
**ovm\_string\_to\_bits**  
**ovm\_subscriber**

**ovm\_table\_printer**  
**ovm\_table\_printer\_knobs**  
**ovm\_test**  
**ovm\_test\_done**  
**ovm\_test\_done\_objection**  
**OVM\_TIME**  
**ovm\_top**  
 ovm\_root  
  
**ovm\_transaction**  
**ovm\_tree\_printer**  
**ovm\_tree\_printer\_knobs**  
**OVM\_UNSIGNED**  
**ovm\_verbosity**  
**ovm\_void**  
**ovm\_wait\_for\_nba\_region**  
**OVM\_WARNING**

## P

**pack**  
 ovm\_object  
  
**pack\_bytes**  
 ovm\_object  
  
**pack\_field**  
 ovm\_packer  
  
**pack\_field\_int**  
 ovm\_packer  
  
**pack\_ints**  
 ovm\_object  
  
**pack\_object**  
 ovm\_packer  
  
**pack\_real**  
 ovm\_packer  
  
**pack\_string**  
 ovm\_packer  
  
**pack\_time**  
 ovm\_packer  
  
**Packing**  
 ovm\_object  
 ovm\_packer  
  
**pair\_ap**  
 ovm\_in\_order\_comparator#(T,comp\_type,convert,pair\_type)  
  
**Parameters**  
 ovm\_callbacks#(T,CB)  
  
**peek**  
 sqr\_if\_base#(REQ,RSP)  
 tlm\_if\_base#(T1,T2)

**phase\_timeout**

ovm\_root

**Phasing Interface**

ovm\_component

**physical**

ovm\_comparer

ovm\_packer

ovm\_recorder

**policy**

ovm\_comparer

**Policy Classes**

policies.txt

methodology/ovm\_policies.svh

**pop\_back**

ovm\_queue#(T)

**pop\_front**

ovm\_queue#(T)

**Port Type****Ports**

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

ovm\_driver#(REQ,RSP)

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

ovm\_push\_driver#(REQ,RSP)

ovm\_push\_sequencer#(REQ,RSP)

ovm\_random\_stimulus#(T)

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

ovm\_subscriber

tlm\_analysis\_fifo#(T)

tlm\_fifo\_base#(T)

tlm\_req\_rsp\_channel#(REQ,RSP)

tlm\_transport\_channel#(REQ,RSP)

**Ports,Exports,and Imps****post\_body**

ovm\_sequence\_base

**POST\_BODY****post\_do**

ovm\_sequence\_base

**post\_trigger**

ovm\_event\_callback

**pound\_zero\_count**

ovm\_sequencer\_base

**pre\_body**

ovm\_sequence\_base

**PRE\_BODY**

**pre\_do**[ovm\\_sequence\\_base](#)**pre\_trigger**[ovm\\_event\\_callback](#)**Predefined Component Classes****prefix**[ovm\\_printer\\_knobs](#)**prev**[ovm\\_barrier\\_pool](#)[ovm\\_event\\_pool](#)[ovm\\_pool#\(T\)](#)**print**[ovm\\_factory](#)[ovm\\_object](#)**print\_array\_footer**[ovm\\_printer](#)**print\_array\_header**[ovm\\_printer](#)**print\_array\_range**[ovm\\_printer](#)**print\_config\_matches**[ovm\\_component](#)**print\_config\_settings**[ovm\\_component](#)**print\_enabled**[ovm\\_component](#)**print\_field**[ovm\\_printer](#)**print\_footer**[ovm\\_printer](#)**print\_header**[ovm\\_printer](#)**print\_id**[ovm\\_printer](#)**print\_msg**[ovm\\_comparer](#)**print\_newline**[ovm\\_line\\_printer](#)[ovm\\_printer](#)**print\_object**[ovm\\_printer](#)**print\_object\_header**[ovm\\_printer](#)

**print\_override\_info**

ovm\_component

**print\_size**

ovm\_printer

**print\_string**

ovm\_printer

**print\_time**

ovm\_printer

**print\_type\_name**

ovm\_printer

**print\_value**

ovm\_printer

**print\_value\_array**

ovm\_printer

**print\_value\_object**

ovm\_printer

**print\_value\_string**

ovm\_printer

**Printing**

ovm\_object

**process\_report**

ovm\_report\_server

**push\_back**

ovm\_queue#(T)

**push\_front**

ovm\_queue#(T)

**put**

sqr\_if\_base#(REQ,RSP)

tlm\_if\_base#(T1,T2)

**Put****put\_ap**

tlm\_fifo\_base#(T)

**put\_export**

tlm\_fifo\_base#(T)

**put\_request\_export**

tlm\_req\_rsp\_channel#(REQ,RSP)

**put\_response\_export**

tlm\_req\_rsp\_channel#(REQ,RSP)

**Q****qualify**

ovm\_test\_done\_objection

**R**



**raise\_objection**[ovm\\_objection](#)[ovm\\_test\\_done\\_objection](#)**raised**[ovm\\_component](#)[ovm\\_objection](#)[ovm\\_root](#)**record**[ovm\\_object](#)**record\_error\_tr**[ovm\\_component](#)**record\_event\_tr**[ovm\\_component](#)**record\_field**[ovm\\_recorder](#)**record\_field\_real**[ovm\\_recorder](#)**record\_generic**[ovm\\_recorder](#)**record\_object**[ovm\\_recorder](#)**record\_string**[ovm\\_recorder](#)**record\_time**[ovm\\_recorder](#)**Recording**[ovm\\_object](#)**Recording Interface**[ovm\\_component](#)**recursion\_policy**[ovm\\_recorder](#)**reference**[ovm\\_printer\\_knobs](#)**register**[ovm\\_factory](#)**Registering Types**[ovm\\_factory](#)**report**[ovm\\_component](#)[ovm\\_report\\_handler](#)**Report Macros****report\_error\_hook**[ovm\\_report\\_object](#)

**report\_fatal\_hook**[ovm\\_report\\_object](#)**report\_header**[ovm\\_report\\_object](#)**report\_hook**[ovm\\_report\\_object](#)**report\_info\_hook**[ovm\\_report\\_object](#)**report\_summarize**[ovm\\_report\\_object](#)**report\_warning\_hook**[ovm\\_report\\_object](#)**Reporting**

Global

[base/ovm\\_globals.svh](#)[base/ovm\\_object\\_globals.svh](#)[ovm\\_report\\_object](#)**Reporting Classes****req\_export**[ovm\\_push\\_driver#\(REQ,RSP\)](#)**req\_port**[ovm\\_push\\_sequencer#\(REQ,RSP\)](#)**request\_ap**[tlm\\_req\\_rsp\\_channel#\(REQ,RSP\)](#)**reseed**[ovm\\_object](#)**reset**[ovm\\_barrier](#)[ovm\\_event](#)[ovm\\_phase](#)**reset\_quit\_count**[ovm\\_report\\_server](#)**reset\_report\_handler**[ovm\\_report\\_object](#)**reset\_severity\_counts**[ovm\\_report\\_server](#)**resolve\_bindings**[ovm\\_component](#)[ovm\\_port\\_base#\(IF\)](#)**response\_ap**[tlm\\_req\\_rsp\\_channel#\(REQ,RSP\)](#)**response\_handler**[ovm\\_sequence\\_base](#)

**result**[ovm\\_comparer](#)**resume**[ovm\\_component](#)**rsp\_export**[ovm\\_sequencer\\_param\\_base#\(REQ,RSP\)](#)**rsp\_port**[ovm\\_driver#\(REQ,RSP\)](#)[ovm\\_push\\_driver#\(REQ,RSP\)](#)**run**[ovm\\_component](#)[ovm\\_push\\_sequencer#\(REQ,RSP\)](#)**run\_hooks**[ovm\\_report\\_handler](#)**run\_test**[Global](#)[ovm\\_root](#)

# 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](#)

## S

### Seeding

[ovm\\_object](#)

### send\_request

[ovm\\_sequence#\(REQ,RSP\)](#)

[ovm\\_sequence\\_base](#)

[ovm\\_sequencer\\_base](#)

[ovm\\_sequencer\\_param\\_base#\(REQ,RSP\)](#)

### separator

[ovm\\_tree\\_printer\\_knobs](#)

### seq\_item\_export

[ovm\\_sequencer#\(REQ,RSP\)](#)

### seq\_item\_port

[ovm\\_driver#\(REQ,RSP\)](#)

### seq\_kind

[ovm\\_sequence\\_base](#)

### Sequence Action Macros

### Sequence and Do Action Macros

### Sequence Classes

### Sequence on Sequencer Action Macros

### Sequence Registration Macros

### Sequencer Classes

### Sequencer Registration Macros

### Sequences

### set\_arbitration

[ovm\\_sequencer\\_base](#)

### set\_auto\_reset

[ovm\\_barrier](#)

### set\_config\_int

[Global](#)

[ovm\\_component](#)

### set\_config\_object

[Global](#)

[ovm\\_component](#)

### set\_config\_string

[Global](#)

[ovm\\_component](#)

**set\_default\_index**  
ovm\_port\_base#(IF)

**set\_depth**  
ovm\_sequence\_item

**set\_drain\_time**  
ovm\_objection

**set\_global\_stop\_timeout**

**set\_global\_timeout**

**set\_id\_count**  
ovm\_report\_server

**set\_id\_info**  
ovm\_sequence\_item

**set\_initiator**  
ovm\_transaction

**set\_inst\_override**  
ovm\_component  
ovm\_component\_registry#(T,Tname)  
ovm\_object\_registry#(T,Tname)

**set\_inst\_override\_by\_name**  
ovm\_factory

**set\_inst\_override\_by\_type**  
ovm\_component  
ovm\_factory

**set\_int\_local**  
ovm\_object

**set\_max\_quit\_count**  
ovm\_report\_server

**set\_name**  
ovm\_component  
ovm\_object

**set\_num\_last\_reqs**  
ovm\_sequencer\_param\_base#(REQ,RSP)

**set\_num\_last\_rsps**  
ovm\_sequencer\_param\_base#(REQ,RSP)

**set\_object\_local**  
ovm\_object

**set\_parent\_sequence**  
ovm\_sequence\_item

**set\_priority**  
ovm\_sequence\_base

**set\_quit\_count**  
ovm\_report\_server

**set\_report\_default\_file**  
ovm\_report\_object

**set\_report\_default\_file\_hier**  
ovm\_component

**set\_report\_handler**  
ovm\_report\_object

**set\_report\_id\_action**  
ovm\_report\_object

**set\_report\_id\_action\_hier**  
ovm\_component

**set\_report\_id\_file**  
ovm\_report\_object

**set\_report\_id\_file\_hier**  
ovm\_component

**set\_report\_max\_quit\_count**  
ovm\_report\_object

**set\_report\_severity\_action**  
ovm\_report\_object

**set\_report\_severity\_action\_hier**  
ovm\_component

**set\_report\_severity\_file**  
ovm\_report\_object

**set\_report\_severity\_file\_hier**  
ovm\_component

**set\_report\_severity\_id\_action**  
ovm\_report\_object

**set\_report\_severity\_id\_action\_hier**  
ovm\_component

**set\_report\_severity\_id\_file**  
ovm\_report\_object

**set\_report\_severity\_id\_file\_hier**  
ovm\_component

**set\_report\_verbosity\_level**  
ovm\_report\_object

**set\_report\_verbosity\_level\_hier**  
ovm\_component

**set\_response\_queue\_depth**  
ovm\_sequence# (REQ,RSP)

**set\_response\_queue\_error\_report\_disabled**  
ovm\_sequence# (REQ,RSP)

**set\_sequencer**

ovm\_sequence#(REQ,RSP)

ovm\_sequence\_base

ovm\_sequence\_item

**set\_severity\_count**

ovm\_report\_server

**set\_string\_local**

ovm\_object

**set\_threshold**

ovm\_barrier

**set\_transaction\_id**

ovm\_transaction

**set\_type\_override**

ovm\_component

ovm\_component\_registry#(T,Tname)

ovm\_object\_registry#(T,Tname)

**set\_type\_override\_by\_name**

ovm\_factory

**set\_type\_override\_by\_type**

ovm\_component

ovm\_factory

**set\_use\_sequence\_info**

ovm\_sequence\_item

**Setup**

ovm\_report\_object

**sev**

ovm\_comparer

**show\_max**

ovm\_comparer

**show\_radix**

ovm\_printer\_knobs

**show\_root**

ovm\_hier\_printer\_knobs

**Simulation Control****size**

ovm\_port\_base#(IF)

ovm\_printer\_knobs

ovm\_queue#(T)

tlm\_fifo#(T)

**size\_width**

ovm\_table\_printer\_knobs

**slave\_export**

tlm\_req\_rsp\_channel#(REQ,RSP)

**sprint**

ovm\_object

**sqr\_if\_base#(REQ,RSP)****start**

ovm\_sequence#(REQ,RSP)

ovm\_sequence\_base

**start\_default\_sequence**

ovm\_sequencer\_base

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

**start\_item**

ovm\_sequence\_base

ovm\_sequence\_item

**start\_of\_simulation**

ovm\_component

**status**

ovm\_component

**stop**

ovm\_component

**stop\_request**

ovm\_root

**stop\_sequences**

ovm\_sequencer#(REQ,RSP)

ovm\_sequencer\_base

**stop\_stimulus\_generation**

ovm\_random\_stimulus#(T)

**stop\_timeout**

ovm\_root

**STOPPED****summarize**

ovm\_report\_server

**suspend**

ovm\_component

**Synchronization Classes****T****T**

ovm\_callbacks#(T,CB)

**TLM Implementation Port Declaration Macros****TLM Interfaces,Ports,and Exports**

tlm\_analysis\_fifo#(T)

tlm\_fifo#(T)

tlm\_fifo\_base#(T)

tlm\_if\_base#(T1,T2)

tlm\_req\_rsp\_channel#(REQ,RSP)



**tlm\_transport\_channel#(REQ,RSP)**

**tr\_handle**

ovm\_recorder

**trace\_mode**

ovm\_callbacks#(T,CB)

**transport**

tlm\_if\_base#(T1,T2)

**Transport**

**transport\_export**

tlm\_transport\_channel#(REQ,RSP)

**trigger**

ovm\_event

**truncation**

ovm\_printer\_knobs

**try\_get**

tlm\_if\_base#(T1,T2)

**try\_next\_item**

sqr\_if\_base#(REQ,RSP)

**try\_peek**

tlm\_if\_base#(T1,T2)

**try\_put**

tlm\_if\_base#(T1,T2)

**Type&Instance Overrides**

ovm\_factory

**type\_name**

ovm\_printer\_knobs

**type\_width**

ovm\_table\_printer\_knobs

**Types and Enumerations**

## U

**ungrab**

ovm\_sequence\_base

ovm\_sequencer\_base

**Unidirectional Interfaces&Ports**

**unlock**

ovm\_sequence\_base

ovm\_sequencer\_base

**unpack**

ovm\_object

**unpack\_bytes**

ovm\_object

**unpack\_field**

ovm\_packer

**unpack\_field\_int**

ovm\_packer

**unpack\_ints**

ovm\_object

**unpack\_object**

ovm\_packer

**unpack\_real**

ovm\_packer

**unpack\_string**

ovm\_packer

**unpack\_time**

ovm\_packer

**Unpacking**

ovm\_object

ovm\_packer

**unsigned\_radix**

ovm\_printer\_knobs

**Usage**

Global

tlm\_ifs\_and\_ports.txt

base/ovm\_phases.sv

ovm\_factory

ovm\_object\_registry#(T,Tname)

**use\_metadata**

ovm\_packer

**use\_ovm\_seeding**

ovm\_object

**use\_response\_handler**

ovm\_sequence\_base

**used**

tlm\_fifo#(T)

**user\_priority\_arbitration**

ovm\_sequencer\_base

**Utility and Field Macros for Components and Objects****Utility Macros****V****value\_width**

ovm\_table\_printer\_knobs

**Variables**

[ovm\\_comparer](#)  
[ovm\\_hier\\_printer\\_knobs](#)  
[ovm\\_line\\_printer](#)  
[ovm\\_packer](#)  
[ovm\\_printer\\_knobs](#)  
[ovm\\_recorder](#)  
[ovm\\_report\\_server](#)  
[ovm\\_root](#)  
[ovm\\_sequence\\_base](#)  
[ovm\\_sequencer#\(REQ,RSP\)](#)  
[ovm\\_sequencer\\_base](#)  
[ovm\\_table\\_printer](#)  
[ovm\\_table\\_printer\\_knobs](#)  
[ovm\\_tree\\_printer](#)  
[ovm\\_tree\\_printer\\_knobs](#)

**verbosity**

[ovm\\_comparer](#)

**Verbosity is ignored for warnings,errors,and fatals to ensure users**

**W****wait\_done**

[ovm\\_phase](#)

**wait\_for**

[ovm\\_barrier](#)

**wait\_for\_grant**

[ovm\\_sequence\\_base](#)  
[ovm\\_sequencer\\_base](#)

**wait\_for\_item\_done**

[ovm\\_sequence\\_base](#)  
[ovm\\_sequencer\\_base](#)

**wait\_for\_relevant**

[ovm\\_sequence\\_base](#)

**wait\_for\_sequence\_state**

[ovm\\_sequence\\_base](#)

**wait\_for\_sequences**

[ovm\\_sequencer\\_base](#)  
[sqr\\_if\\_base#\(REQ,RSP\)](#)

**wait\_off**

[ovm\\_event](#)

**wait\_on**

[ovm\\_event](#)

**wait\_ptrigger**

[ovm\\_event](#)

**wait\_pttrigger\_data**

ovm\_event

**wait\_start**

ovm\_phase

**wait\_trigger**

ovm\_event

**wait\_trigger\_data**

ovm\_event

**write**

ovm\_subscriber

tlm\_if\_base#(T1,T2)

# 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

## I

**in\_order\_built\_in\_comparator#(T)**  
**in\_order\_class\_comparator#(T)**

## O

**ovm\_\*\_export#(REQ,RSP)**  
**ovm\_\*\_export#(T)**  
**ovm\_\*\_imp#(REQ,RSP,IMP,REQ\_IMP,RSP\_IMP)**  
**ovm\_\*\_imp#(T,IMP)**  
**ovm\_\*\_port#(REQ,RSP)**  
**ovm\_\*\_port#(T)**  
**ovm\_agent**  
**ovm\_algorithmic\_comparator#(BEFORE,AFTER,TRANSFORMER)**  
**ovm\_barrier**  
**ovm\_barrier\_pool**  
**ovm\_built\_in\_clone#(T)**  
**ovm\_built\_in\_comp#(T)**  
**ovm\_built\_in\_converter#(T)**  
**ovm\_built\_in\_pair#(T1,T2)**  
**ovm\_callback**  
**ovm\_callbacks#(T,CB)**  
**ovm\_class\_clone#(T)**  
**ovm\_class\_comp#(T)**  
**ovm\_class\_converter#(T)**  
**ovm\_comparer**  
**ovm\_component**  
**ovm\_component\_registry#(T,Tname)**  
**ovm\_driver#(REQ,RSP)**  
**ovm\_env**  
**ovm\_event**  
**ovm\_event\_callback**  
**ovm\_event\_pool**  
**ovm\_exhaustive\_sequence**  
**ovm\_factory**  
**ovm\_hier\_printer\_knobs**  
**ovm\_in\_order\_comparator#(T,comp\_type,convert,pair\_type)**  
**ovm\_line\_printer**  
**ovm\_monitor**  
**ovm\_object**  
**ovm\_object\_registry#(T,Tname)**  
**ovm\_object\_string\_pool#(T)**  
**ovm\_object\_wrapper**  
**ovm\_objection**

ovm\_packer  
ovm\_pair#(T1,T2)  
ovm\_phase  
ovm\_pool#(T)  
ovm\_port\_base#(IF)  
ovm\_printer  
ovm\_printer\_knobs  
ovm\_push\_driver#(REQ,RSP)  
ovm\_push\_sequencer#(REQ,RSP)  
ovm\_queue#(T)  
ovm\_random\_sequence  
ovm\_random\_stimulus#(T)  
ovm\_recorder  
ovm\_report\_handler  
ovm\_report\_object  
ovm\_report\_server  
ovm\_root  
ovm\_scoreboard  
ovm\_seq\_item\_pull\_export#(REQ,RSP)  
ovm\_seq\_item\_pull\_imp#(REQ,RSP,IMP)  
ovm\_seq\_item\_pull\_port#(REQ,RSP)  
ovm\_sequence#(REQ,RSP)  
ovm\_sequence\_base  
ovm\_sequence\_item  
ovm\_sequencer#(REQ,RSP)  
ovm\_sequencer\_base  
ovm\_sequencer\_param\_base#(REQ,RSP)  
ovm\_simple\_sequence  
ovm\_subscriber  
ovm\_table\_printer  
ovm\_table\_printer\_knobs  
ovm\_test  
ovm\_test\_done\_objection  
ovm\_transaction  
ovm\_tree\_printer  
ovm\_tree\_printer\_knobs

**S**

sqr\_if\_base#(REQ,RSP)

**T**

tlm\_analysis\_fifo#(T)  
tlm\_fifo#(T)  
tlm\_fifo\_base#(T)  
tlm\_if\_base#(T1,T2)  
tlm\_req\_rsp\_channel#(REQ,RSP)  
tlm\_transport\_channel#(REQ,RSP)

# 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

## O

**ovm\_algorithmic\_comparator.svh**

**ovm\_callback\_defines.svh**

**ovm\_policies.svh**

# 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](#)

## \$#!

- ovm\_analysis\_imp\_decl
- ovm\_blocking\_get\_imp\_decl
- ovm\_blocking\_get\_peek\_imp\_decl
- ovm\_blocking\_master\_imp\_decl
- ovm\_blocking\_peek\_imp\_decl
- ovm\_blocking\_put\_imp\_decl
- ovm\_blocking\_slave\_imp\_decl
- ovm\_blocking\_transport\_imp\_decl
- ovm\_component\_end
- ovm\_component\_param\_utils
- ovm\_component\_param\_utils\_begin
- ovm\_component\_utils
- ovm\_component\_utils\_begin
- ovm\_create
- ovm\_create\_on
- ovm\_declare\_p\_sequencer
- ovm\_do
- ovm\_do\_callbacks
- ovm\_do\_callbacks\_exit\_on
- ovm\_do\_ext\_task\_callbacks
- ovm\_do\_obj\_callbacks
- ovm\_do\_obj\_callbacks\_exit\_on
- ovm\_do\_on
- ovm\_do\_on\_pri
- ovm\_do\_on\_pri\_with
- ovm\_do\_on\_with
- ovm\_do\_pri
- ovm\_do\_pri\_with
- ovm\_do\_task\_callbacks
- ovm\_do\_with
- ovm\_error
- ovm\_fatal
- ovm\_field\_aa\_int\_byte
- ovm\_field\_aa\_int\_byte\_unsigned
- ovm\_field\_aa\_int\_enumkey



- ` ovm\_field\_aa\_int\_int
- ` ovm\_field\_aa\_int\_int\_unsigned
- ` ovm\_field\_aa\_int\_integer
- ` ovm\_field\_aa\_int\_integer\_unsigned
- ` ovm\_field\_aa\_int\_key
- ` ovm\_field\_aa\_int\_longint
- ` ovm\_field\_aa\_int\_longint\_unsigned
- ` ovm\_field\_aa\_int\_shortint
- ` ovm\_field\_aa\_int\_shortint\_unsigned
- ` ovm\_field\_aa\_int\_string
- ` ovm\_field\_aa\_object\_int
- ` ovm\_field\_aa\_object\_string
- ` ovm\_field\_aa\_string\_string
- ` ovm\_field\_array\_enum
- ` ovm\_field\_array\_int
- ` ovm\_field\_array\_object
- ` ovm\_field\_array\_string
- ` ovm\_field\_enum
- ` ovm\_field\_event
- ` ovm\_field\_int
- ` ovm\_field\_object
- ` ovm\_field\_queue\_enum
- ` ovm\_field\_queue\_int
- ` ovm\_field\_queue\_object
- ` ovm\_field\_queue\_string
- ` ovm\_field\_real
- ` ovm\_field\_sarray\_enum
- ` ovm\_field\_sarray\_int
- ` ovm\_field\_sarray\_object
- ` ovm\_field\_sarray\_string
- ` ovm\_field\_string
- ` ovm\_field\_utils\_begin
- ` ovm\_field\_utils\_end
- ` ovm\_get\_imp\_decl
- ` ovm\_get\_peek\_imp\_decl
- ` ovm\_info
- ` ovm\_master\_imp\_decl
- ` ovm\_nonblocking\_get\_imp\_decl
- ` ovm\_nonblocking\_get\_peek\_imp\_decl
- ` ovm\_nonblocking\_master\_imp\_decl
- ` ovm\_nonblocking\_peek\_imp\_decl
- ` ovm\_nonblocking\_put\_imp\_decl
- ` ovm\_nonblocking\_slave\_imp\_decl
- ` ovm\_nonblocking\_transport\_imp\_decl

- ` ovm\_object\_param\_utils
- ` ovm\_object\_param\_utils\_begin
- ` ovm\_object\_utils
- ` ovm\_object\_utils\_begin
- ` ovm\_object\_utils\_end
- ` ovm\_peek\_imp\_decl
- ` ovm\_phase\_func\_bottomup\_decl
- ` ovm\_phase\_func\_decl
- ` ovm\_phase\_func\_topdown\_decl
- ` ovm\_phase\_task\_bottomup\_decl
- ` ovm\_phase\_task\_decl
- ` ovm\_phase\_task\_topdown\_decl
- ` ovm\_put\_imp\_decl
- ` ovm\_rand\_send
- ` ovm\_rand\_send\_pri
- ` ovm\_rand\_send\_pri\_with
- ` ovm\_rand\_send\_with
- ` ovm\_send
- ` ovm\_send\_pri
- ` ovm\_sequence\_utils
- ` ovm\_sequence\_utils\_begin
- ` ovm\_sequence\_utils\_end
- ` ovm\_sequencer\_param\_utils
- ` ovm\_sequencer\_param\_utils\_begin
- ` ovm\_sequencer\_utils
- ` ovm\_sequencer\_utils\_begin
- ` ovm\_sequencer\_utils\_end
- ` ovm\_slave\_imp\_decl
- ` ovm\_transport\_imp\_decl
- ` ovm\_update\_sequence\_lib
- ` ovm\_update\_sequence\_lib\_and\_item
- ` ovm\_warning

# 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**

[ovm\\_component](#)

[ovm\\_transaction](#)

### **add**

[ovm\\_pool#\(T\)](#)

### **add\_callback**

[ovm\\_event](#)

### **add\_cb**

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

### **add\_sequence**

[ovm\\_sequencer\\_base](#)

### **all\_dropped**

[ovm\\_component](#)

[ovm\\_objection](#)

[ovm\\_root](#)

[ovm\\_test\\_done\\_objection](#)

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

[ovm\\_component](#)

## B

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

[ovm\\_component](#)

[ovm\\_transaction](#)

### **begin\_tr**

[ovm\\_component](#)

[ovm\\_transaction](#)

### **body**

[ovm\\_sequence\\_base](#)

### **build**

[ovm\\_component](#)

## C

### **call\_func**

[ovm\\_phase](#)

### **call\_task**

[ovm\\_phase](#)

### **callback\_mode**

[ovm\\_callback](#)

### **can\_get**

[tlm\\_if\\_base#\(T1,T2\)](#)

### **can\_peek**

[tlm\\_if\\_base#\(T1,T2\)](#)

**can\_put**

tlm\_if\_base#(T1,T2)

**cancel**

ovm\_barrier

ovm\_event

**check**

ovm\_component

**check\_config\_usage**

ovm\_component

**clone**

ovm\_object

**compare**

ovm\_object

**compare\_field**

ovm\_comparer

**compare\_field\_int**

ovm\_comparer

**compare\_field\_real**

ovm\_comparer

**compare\_object**

ovm\_comparer

**compare\_string**

ovm\_comparer

**compose\_message**

ovm\_report\_server

**connect**

ovm\_component

ovm\_port\_base#(IF)

**convert2string**

ovm\_object

**copy**

ovm\_object

**create**

ovm\_component\_registry#(T,Tname)

ovm\_object

ovm\_object\_registry#(T,Tname)

**create\_component**

ovm\_component

ovm\_component\_registry#(T,Tname)

ovm\_object\_wrapper

**create\_component\_by\_name**

ovm\_factory

**create\_component\_by\_type**

ovm\_factory

**create\_item**

ovm\_sequence\_base

**create\_object**

ovm\_component

ovm\_object\_registry#(T,Tname)

ovm\_object\_wrapper

**create\_object\_by\_name**

ovm\_factory

**create\_object\_by\_type**

ovm\_factory

**current\_grabber**

ovm\_sequencer\_base

**D****debug\_connected\_to**

ovm\_port\_base#(IF)

**debug\_create\_by\_name**

ovm\_factory

**debug\_create\_by\_type**

ovm\_factory

**debug\_provided\_to**

ovm\_port\_base#(IF)

**delete**

ovm\_barrier\_pool

ovm\_event\_pool

ovm\_object\_string\_pool#(T)

ovm\_pool#(T)

ovm\_queue#(T)

**delete\_callback**

ovm\_event

**delete\_cb**

ovm\_callbacks#(T,CB)

**die**

ovm\_report\_object

**disable\_recording**

ovm\_transaction

**display\_cbs**

ovm\_callbacks#(T,CB)

**display\_objections**

ovm\_objection

**do\_accept\_tr**

ovm\_component

ovm\_transaction

**do\_begin\_tr**

ovm\_component

ovm\_transaction

**do\_compare**

ovm\_object

**do\_copy**

ovm\_object

**do\_end\_tr**

ovm\_component

ovm\_transaction

**do\_kill\_all**

ovm\_component

**do\_pack**

ovm\_object

**do\_print**

ovm\_object

**do\_record**

ovm\_object

**do\_sequence\_kind**

ovm\_sequence\_base

**do\_unpack**

ovm\_object

**drop**

ovm\_test\_done\_objection

**drop\_objection**

ovm\_objection

**dropped**

ovm\_component

ovm\_objection

**dump\_report\_state**

ovm\_report\_object

**dump\_server\_state**

ovm\_report\_server

**E****enable\_recording**

ovm\_transaction

**end\_of\_elaboration**

ovm\_component

**end\_tr**

ovm\_component

ovm\_transaction

**execute\_item**

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

**exists**

ovm\_barrier\_pool

ovm\_event\_pool

ovm\_pool#(T)

**extract**

ovm\_component

**F****find**

ovm\_root

**find\_all**

ovm\_root

**find\_override\_by\_name**

ovm\_factory

**find\_override\_by\_type**

ovm\_factory

**finish\_item**

ovm\_sequence\_base

ovm\_sequence\_item

**first**

ovm\_barrier\_pool

ovm\_event\_pool

ovm\_pool#(T)

**flush**

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

tlm\_fifo#(T)

**force\_stop**

ovm\_test\_done\_objection

**format\_action**

ovm\_report\_handler

**G****generate\_stimulus**

ovm\_random\_stimulus#(T)

**get**

ovm\_barrier\_pool

ovm\_component\_registry#(T,Tname)

ovm\_event\_pool

ovm\_object\_registry#(T,Tname)

ovm\_object\_string\_pool#(T)

ovm\_pool#(T)

ovm\_queue#(T)

sqr\_if\_base#(REQ,RSP)

tlm\_if\_base#(T1,T2)

**get\_accept\_time**

ovm\_transaction

**get\_action**

ovm\_report\_handler

**get\_begin\_time**

ovm\_transaction

**get\_child**

ovm\_component

**get\_comp**

ovm\_port\_base#(IF)

**get\_config\_int**

ovm\_component

**get\_config\_object**

ovm\_component

**get\_config\_string**

ovm\_component

**get\_count**[ovm\\_random\\_sequence](#)**get\_current\_item**[ovm\\_sequence#\(REQ,RSP\)](#)[ovm\\_sequencer\\_param\\_base#\(REQ,RSP\)](#)**get\_current\_phase**[ovm\\_root](#)**get\_depth**[ovm\\_sequence\\_item](#)**get\_drain\_time**[ovm\\_objection](#)**get\_end\_time**[ovm\\_transaction](#)**get\_event\_pool**[ovm\\_transaction](#)**get\_file\_handle**[ovm\\_report\\_handler](#)**get\_first\_child**[ovm\\_component](#)**get\_full\_name**[ovm\\_component](#)[ovm\\_object](#)[ovm\\_port\\_base#\(IF\)](#)**get\_global**[ovm\\_pool#\(T\)](#)[ovm\\_queue#\(T\)](#)**get\_global\_cbs**[ovm\\_callbacks#\(T,CB\)](#)**get\_global\_pool**[ovm\\_barrier\\_pool](#)[ovm\\_event\\_pool](#)[ovm\\_object\\_string\\_pool#\(T\)](#)[ovm\\_pool#\(T\)](#)**get\_global\_queue**[ovm\\_queue#\(T\)](#)**get\_id\_count**[ovm\\_report\\_server](#)**get\_if**[ovm\\_port\\_base#\(IF\)](#)**get\_initiator**[ovm\\_transaction](#)**get\_inst\_count**[ovm\\_object](#)**get\_inst\_id**[ovm\\_object](#)**get\_max\_quit\_count**[ovm\\_report\\_server](#)



**get\_name**

ovm\_object

ovm\_phase

ovm\_port\_base#(IF)

**get\_next\_child**

ovm\_component

**get\_next\_item**

sqr\_if\_base#(REQ,RSP)

**get\_num\_children**

ovm\_component

**get\_num\_last\_reqs**

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

**get\_num\_last\_rsps**

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

**get\_num\_reqs\_sent**

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

**get\_num\_rsps\_received**

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

**get\_num\_waiters**

ovm\_barrier

ovm\_event

**get\_object\_type**

ovm\_object

**get\_objection\_count**

ovm\_objection

**get\_objection\_total**

ovm\_objection

**get\_packed\_size**

ovm\_packer

**get\_parent**

ovm\_component

ovm\_port\_base#(IF)

**get\_parent\_sequence**

ovm\_sequence\_item

**get\_phase\_by\_name**

ovm\_root

**get\_priority**

ovm\_sequence\_base

**get\_quit\_count**

ovm\_report\_server

**get\_radix\_str**

ovm\_printer\_knobs

**get\_report\_action**

ovm\_report\_object

**get\_report\_file\_handle**

ovm\_report\_object

**get\_report\_handler**

ovm\_report\_object

**get\_report\_server**

ovm\_report\_object

**get\_report\_verbosity\_level**

ovm\_report\_object

**get\_response**

ovm\_sequence#(REQ,RSP)

**get\_response\_queue\_depth**

ovm\_sequence#(REQ,RSP)

**get\_response\_queue\_error\_report\_disabled**

ovm\_sequence#(REQ,RSP)

**get\_root\_sequence**

ovm\_sequence\_item

**get\_root\_sequence\_name**

ovm\_sequence\_item

**get\_seq\_kind**

ovm\_sequence\_base

ovm\_sequencer\_base

**get\_sequence**

ovm\_sequence\_base

ovm\_sequencer\_base

**get\_sequence\_by\_name**

ovm\_sequence\_base

**get\_sequence\_id**

ovm\_sequence\_item

**get\_sequence\_path**

ovm\_sequence\_item

**get\_sequence\_state**

ovm\_sequence\_base

**get\_sequencer**

ovm\_sequence\_base

ovm\_sequence\_item

**get\_server**

ovm\_report\_server

**get\_severity\_count**

ovm\_report\_server

**get\_threshold**

ovm\_barrier

**get\_tr\_handle**

ovm\_transaction

**get\_transaction\_id**

ovm\_transaction

**get\_trigger\_data**

ovm\_event

**get\_trigger\_time**

ovm\_event

**get\_type**

ovm\_object

**get\_type\_name**

ovm\_callback

ovm\_component\_registry#(T,Tname)

ovm\_object

ovm\_object\_registry#(T,Tname)

ovm\_object\_string\_pool#(T)

ovm\_object\_wrapper

ovm\_phase

ovm\_port\_base#(IF)

**get\_use\_response\_handler**

ovm\_sequence\_base

**get\_use\_sequence\_info**

ovm\_sequence\_item

**get\_verbosity\_level**

ovm\_report\_handler

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

ovm\_sequence\_base

ovm\_sequencer\_base

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

ovm\_component

**has\_do\_available**

ovm\_sequencer\_base

sqr\_if\_base#(REQ,RSP)

**has\_lock**

ovm\_sequence\_base

ovm\_sequencer\_base

**I****in\_stop\_request**

ovm\_root

**incr\_id\_count**

ovm\_report\_server

**incr\_quit\_count**

ovm\_report\_server

**incr\_severity\_count**

ovm\_report\_server

**insert**

ovm\_queue#(T)

**insert\_phase**

ovm\_root

**is\_active**

ovm\_transaction

**is\_blocked**

ovm\_sequence\_base

ovm\_sequencer\_base

**is\_child**[ovm\\_sequencer\\_base](#)**is\_done**[ovm\\_phase](#)**is\_empty**[tlm\\_fifo#\(T\)](#)**is\_enabled**[ovm\\_callback](#)**is\_export**[ovm\\_port\\_base#\(IF\)](#)**is\_full**[tlm\\_fifo#\(T\)](#)**is\_grabbed**[ovm\\_sequencer\\_base](#)**is\_imp**[ovm\\_port\\_base#\(IF\)](#)**is\_in\_progress**[ovm\\_phase](#)**is\_item**[ovm\\_sequence\\_base](#)[ovm\\_sequence\\_item](#)**is\_null**[ovm\\_packer](#)**is\_off**[ovm\\_event](#)**is\_on**[ovm\\_event](#)**is\_port**[ovm\\_port\\_base#\(IF\)](#)**is\_quit\_count\_reached**[ovm\\_report\\_server](#)**is\_recording\_enabled**[ovm\\_transaction](#)**is\_relevant**[ovm\\_sequence\\_base](#)**is\_task**[ovm\\_phase](#)**is\_top\_down**[ovm\\_phase](#)**is\_unbounded**[ovm\\_port\\_base#\(IF\)](#)**item\_done**[sqr\\_if\\_base#\(REQ,RSP\)](#)**K****kill**[ovm\\_component](#)[ovm\\_sequence\\_base](#)

**L****last**[ovm\\_barrier\\_pool](#)[ovm\\_event\\_pool](#)[ovm\\_pool#\(T\)](#)**last\_req**[ovm\\_sequencer\\_param\\_base#\(REQ,RSP\)](#)**last\_rsp**[ovm\\_sequencer\\_param\\_base#\(REQ,RSP\)](#)**lock**[ovm\\_sequence\\_base](#)[ovm\\_sequencer\\_base](#)**lookup**[ovm\\_component](#)**M****max\_size**[ovm\\_port\\_base#\(IF\)](#)**mid\_do**[ovm\\_sequence\\_base](#)**min\_size**[ovm\\_port\\_base#\(IF\)](#)

# 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](#)

## N

### **nb\_transport**

[tlm\\_if\\_base#\(T1,T2\)](#)

### **new**

[ovm\\_\\*\\_export#\(REQ,RSP\)](#)

[ovm\\_\\*\\_export#\(T\)](#)

[ovm\\_\\*\\_imp#\(REQ,RSP,IMP,REQ\\_IMP,RSP\\_IMP\)](#)

[ovm\\_\\*\\_imp#\(T,IMP\)](#)

[ovm\\_\\*\\_port#\(REQ,RSP\)](#)

[ovm\\_\\*\\_port#\(T\)](#)

[ovm\\_agent](#)

[ovm\\_algorithmic\\_comparator#\(BEFORE,AFTER,TRANSFORMER\)](#)

[ovm\\_barrier](#)

[ovm\\_barrier\\_pool](#)

[ovm\\_built\\_in\\_pair#\(T1,T2\)](#)

[ovm\\_callback](#)

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

[ovm\\_component](#)

[ovm\\_driver#\(REQ,RSP\)](#)

[ovm\\_env](#)

[ovm\\_event](#)

[ovm\\_event\\_callback](#)

[ovm\\_event\\_pool](#)

[ovm\\_monitor](#)

[ovm\\_object](#)

[ovm\\_object\\_string\\_pool#\(T\)](#)

[ovm\\_objection](#)

[ovm\\_pair#\(T1,T2\)](#)

[ovm\\_phase](#)

[ovm\\_pool#\(T\)](#)

[ovm\\_port\\_base#\(IF\)](#)

[ovm\\_push\\_driver#\(REQ,RSP\)](#)

[ovm\\_push\\_sequencer#\(REQ,RSP\)](#)

[ovm\\_queue#\(T\)](#)

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

[ovm\\_report\\_handler](#)

[ovm\\_report\\_object](#)

[ovm\\_report\\_server](#)

[ovm\\_scoreboard](#)

[ovm\\_sequence#\(REQ,RSP\)](#)

[ovm\\_sequence\\_base](#)

[ovm\\_sequence\\_item](#)

[ovm\\_sequencer#\(REQ,RSP\)](#)

[ovm\\_sequencer\\_base](#)  
[ovm\\_sequencer\\_param\\_base#\(REQ,RSP\)](#)  
[ovm\\_subscriber](#)  
[ovm\\_test](#)  
[ovm\\_transaction](#)  
[tlm\\_analysis\\_fifo#\(T\)](#)  
[tlm\\_fifo#\(T\)](#)  
[tlm\\_fifo\\_base#\(T\)](#)  
[tlm\\_req\\_rsp\\_channel#\(REQ,RSP\)](#)  
[tlm\\_transport\\_channel#\(REQ,RSP\)](#)

**next**

[ovm\\_barrier\\_pool](#)  
[ovm\\_event\\_pool](#)  
[ovm\\_pool#\(T\)](#)

**num**

[ovm\\_barrier\\_pool](#)  
[ovm\\_event\\_pool](#)  
[ovm\\_pool#\(T\)](#)

**num\_sequences**

[ovm\\_sequence\\_base](#)  
[ovm\\_sequencer\\_base](#)

**O**

**[ovm\\_bits\\_to\\_string](#)**  
**[ovm\\_is\\_match](#)**  
**[ovm\\_report\\_enabled](#)**  
[Global](#)  
[ovm\\_report\\_object](#)  
  
**[ovm\\_report\\_error](#)**  
[Global](#)  
[ovm\\_report\\_object](#)  
  
**[ovm\\_report\\_fatal](#)**  
[Global](#)  
[ovm\\_report\\_object](#)  
  
**[ovm\\_report\\_info](#)**  
[Global](#)  
[ovm\\_report\\_object](#)  
  
**[ovm\\_report\\_warning](#)**  
[Global](#)  
[ovm\\_report\\_object](#)  
  
**[ovm\\_string\\_to\\_bits](#)**  
**[ovm\\_wait\\_for\\_nba\\_region](#)**

**P****pack**

[ovm\\_object](#)

**pack\_bytes**

ovm\_object

**pack\_field**

ovm\_packer

**pack\_field\_int**

ovm\_packer

**pack\_ints**

ovm\_object

**pack\_object**

ovm\_packer

**pack\_real**

ovm\_packer

**pack\_string**

ovm\_packer

**pack\_time**

ovm\_packer

**peek**

sqr\_if\_base#(REQ,RSP)

t1m\_if\_base#(T1,T2)

**pop\_back**

ovm\_queue#(T)

**pop\_front**

ovm\_queue#(T)

**post\_body**

ovm\_sequence\_base

**post\_do**

ovm\_sequence\_base

**post\_trigger**

ovm\_event\_callback

**pre\_body**

ovm\_sequence\_base

**pre\_do**

ovm\_sequence\_base

**pre\_trigger**

ovm\_event\_callback

**prev**

ovm\_barrier\_pool

ovm\_event\_pool

ovm\_pool#(T)

**print**

ovm\_factory

ovm\_object



**print\_array\_footer**

ovm\_printer

**print\_array\_header**

ovm\_printer

**print\_array\_range**

ovm\_printer

**print\_config\_settings**

ovm\_component

**print\_field**

ovm\_printer

**print\_footer**

ovm\_printer

**print\_header**

ovm\_printer

**print\_id**

ovm\_printer

**print\_msg**

ovm\_comparer

**print\_newline**

ovm\_line\_printer

ovm\_printer

**print\_object**

ovm\_printer

**print\_object\_header**

ovm\_printer

**print\_override\_info**

ovm\_component

**print\_size**

ovm\_printer

**print\_string**

ovm\_printer

**print\_time**

ovm\_printer

**print\_type\_name**

ovm\_printer

**print\_value**

ovm\_printer

**print\_value\_array**

ovm\_printer

**print\_value\_object**

ovm\_printer

**print\_value\_string**

ovm\_printer

**process\_report**

ovm\_report\_server

**push\_back**

ovm\_queue#(T)

**push\_front**

ovm\_queue#(T)

**put**

sqr\_if\_base#(REQ,RSP)

tlm\_if\_base#(T1,T2)

**Q****qualify**

ovm\_test\_done\_objection

**R****raise\_objection**

ovm\_objection

ovm\_test\_done\_objection

**raised**

ovm\_component

ovm\_objection

ovm\_root

**record**

ovm\_object

**record\_error\_tr**

ovm\_component

**record\_event\_tr**

ovm\_component

**record\_field**

ovm\_recorder

**record\_field\_real**

ovm\_recorder

**record\_generic**

ovm\_recorder

**record\_object**

ovm\_recorder

**record\_string**

ovm\_recorder

**record\_time**

ovm\_recorder

**register**

ovm\_factory

**report**

ovm\_component

ovm\_report\_handler

**report\_error\_hook**

ovm\_report\_object

**report\_fatal\_hook**

ovm\_report\_object

**report\_header**

ovm\_report\_object

**report\_hook**

ovm\_report\_object

**report\_info\_hook**

ovm\_report\_object

**report\_summarize**

ovm\_report\_object

**report\_warning\_hook**

ovm\_report\_object

**reseed**

ovm\_object

**reset**

ovm\_barrier

ovm\_event

ovm\_phase

**reset\_quit\_count**

ovm\_report\_server

**reset\_report\_handler**

ovm\_report\_object

**reset\_severity\_counts**

ovm\_report\_server

**resolve\_bindings**

ovm\_component

ovm\_port\_base#(IF)

**response\_handler**

ovm\_sequence\_base

**resume**

ovm\_component

**run**

ovm\_component

ovm\_push\_sequencer#(REQ,RSP)

**run\_hooks**

ovm\_report\_handler

**run\_test**

Global

ovm\_root

**S**

**send\_request**

ovm\_sequence#(REQ,RSP)  
ovm\_sequence\_base  
ovm\_sequencer\_base  
ovm\_sequencer\_param\_base#(REQ,RSP)

**set\_arbitration**

ovm\_sequencer\_base

**set\_auto\_reset**

ovm\_barrier

**set\_config\_int**

Global  
ovm\_component

**set\_config\_object**

Global  
ovm\_component

**set\_config\_string**

Global  
ovm\_component

**set\_default\_index**

ovm\_port\_base#(IF)

**set\_depth**

ovm\_sequence\_item

**set\_drain\_time**

ovm\_objection

**set\_global\_stop\_timeout****set\_global\_timeout****set\_id\_count**

ovm\_report\_server

**set\_id\_info**

ovm\_sequence\_item

**set\_initiator**

ovm\_transaction

**set\_inst\_override**

ovm\_component  
ovm\_component\_registry#(T,Tname)  
ovm\_object\_registry#(T,Tname)

**set\_inst\_override\_by\_name**

ovm\_factory

**set\_inst\_override\_by\_type**

ovm\_component  
ovm\_factory

**set\_int\_local**

ovm\_object

**set\_max\_quit\_count**[ovm\\_report\\_server](#)**set\_name**[ovm\\_component](#)[ovm\\_object](#)**set\_num\_last\_reqs**[ovm\\_sequencer\\_param\\_base#\(REQ,RSP\)](#)**set\_num\_last\_rsps**[ovm\\_sequencer\\_param\\_base#\(REQ,RSP\)](#)**set\_object\_local**[ovm\\_object](#)**set\_parent\_sequence**[ovm\\_sequence\\_item](#)**set\_priority**[ovm\\_sequence\\_base](#)**set\_quit\_count**[ovm\\_report\\_server](#)**set\_report\_default\_file**[ovm\\_report\\_object](#)**set\_report\_default\_file\_hier**[ovm\\_component](#)**set\_report\_handler**[ovm\\_report\\_object](#)**set\_report\_id\_action**[ovm\\_report\\_object](#)**set\_report\_id\_action\_hier**[ovm\\_component](#)**set\_report\_id\_file**[ovm\\_report\\_object](#)**set\_report\_id\_file\_hier**[ovm\\_component](#)**set\_report\_max\_quit\_count**[ovm\\_report\\_object](#)**set\_report\_severity\_action**[ovm\\_report\\_object](#)**set\_report\_severity\_action\_hier**[ovm\\_component](#)**set\_report\_severity\_file**[ovm\\_report\\_object](#)**set\_report\_severity\_file\_hier**[ovm\\_component](#)**set\_report\_severity\_id\_action**[ovm\\_report\\_object](#)

**set\_report\_severity\_id\_action\_hier**  
ovm\_component

**set\_report\_severity\_id\_file**  
ovm\_report\_object

**set\_report\_severity\_id\_file\_hier**  
ovm\_component

**set\_report\_verbosity\_level**  
ovm\_report\_object

**set\_report\_verbosity\_level\_hier**  
ovm\_component

**set\_response\_queue\_depth**  
ovm\_sequence#(REQ,RSP)

**set\_response\_queue\_error\_report\_disabled**  
ovm\_sequence#(REQ,RSP)

**set\_sequencer**  
ovm\_sequence#(REQ,RSP)  
ovm\_sequence\_base  
ovm\_sequence\_item

**set\_severity\_count**  
ovm\_report\_server

**set\_string\_local**  
ovm\_object

**set\_threshold**  
ovm\_barrier

**set\_transaction\_id**  
ovm\_transaction

**set\_type\_override**  
ovm\_component  
ovm\_component\_registry#(T,Tname)  
ovm\_object\_registry#(T,Tname)

**set\_type\_override\_by\_name**  
ovm\_factory

**set\_type\_override\_by\_type**  
ovm\_component  
ovm\_factory

**set\_use\_sequence\_info**  
ovm\_sequence\_item

**size**  
ovm\_port\_base#(IF)  
ovm\_queue#(T)  
tlm\_fifo#(T)

**sprint**  
ovm\_object

**start**[ovm\\_sequence#\(REQ,RSP\)](#)[ovm\\_sequence\\_base](#)**start\_default\_sequence**[ovm\\_sequencer\\_base](#)[ovm\\_sequencer\\_param\\_base#\(REQ,RSP\)](#)**start\_item**[ovm\\_sequence\\_base](#)[ovm\\_sequence\\_item](#)**start\_of\_simulation**[ovm\\_component](#)**status**[ovm\\_component](#)**stop**[ovm\\_component](#)**stop\_request**[ovm\\_root](#)**stop\_sequences**[ovm\\_sequencer#\(REQ,RSP\)](#)[ovm\\_sequencer\\_base](#)**stop\_stimulus\_generation**[ovm\\_random\\_stimulus#\(T\)](#)**summarize**[ovm\\_report\\_server](#)**suspend**[ovm\\_component](#)**T****trace\_mode**[ovm\\_callbacks#\(T,CB\)](#)**transport**[tlm\\_if\\_base#\(T1,T2\)](#)**trigger**[ovm\\_event](#)**try\_get**[tlm\\_if\\_base#\(T1,T2\)](#)**try\_next\_item**[sqr\\_if\\_base#\(REQ,RSP\)](#)**try\_peek**[tlm\\_if\\_base#\(T1,T2\)](#)**try\_put**[tlm\\_if\\_base#\(T1,T2\)](#)

# 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](#)

## U

### **ungrab**

[ovm\\_sequence\\_base](#)

[ovm\\_sequencer\\_base](#)

### **unlock**

[ovm\\_sequence\\_base](#)

[ovm\\_sequencer\\_base](#)

### **unpack**

[ovm\\_object](#)

### **unpack\_bytes**

[ovm\\_object](#)

### **unpack\_field**

[ovm\\_packer](#)

### **unpack\_field\_int**

[ovm\\_packer](#)

### **unpack\_ints**

[ovm\\_object](#)

### **unpack\_object**

[ovm\\_packer](#)

### **unpack\_real**

[ovm\\_packer](#)

### **unpack\_string**

[ovm\\_packer](#)

### **unpack\_time**

[ovm\\_packer](#)

### **use\_response\_handler**

[ovm\\_sequence\\_base](#)

### **used**

[tlm\\_fifo#\(T\)](#)

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

[ovm\\_sequencer\\_base](#)

## W

### **wait\_done**

[ovm\\_phase](#)



**wait\_for**[ovm\\_barrier](#)**wait\_for\_grant**[ovm\\_sequence\\_base](#)[ovm\\_sequencer\\_base](#)**wait\_for\_item\_done**[ovm\\_sequence\\_base](#)[ovm\\_sequencer\\_base](#)**wait\_for\_relevant**[ovm\\_sequence\\_base](#)**wait\_for\_sequence\_state**[ovm\\_sequence\\_base](#)**wait\_for\_sequences**[ovm\\_sequencer\\_base](#)[sqr\\_if\\_base#\(REQ,RSP\)](#)**wait\_off**[ovm\\_event](#)**wait\_on**[ovm\\_event](#)**wait\_ptrigger**[ovm\\_event](#)**wait\_ptrigger\_data**[ovm\\_event](#)**wait\_start**[ovm\\_phase](#)**wait\_trigger**[ovm\\_event](#)**wait\_trigger\_data**[ovm\\_event](#)**write**[ovm\\_subscriber](#)[tlm\\_if\\_base#\(T1,T2\)](#)

# 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

## O

- ovm\_action
- ovm\_bitstream\_t
- ovm\_port\_type\_e
- ovm\_radix\_enum
- ovm\_recursion\_policy\_enum
- ovm\_sequence\_state\_enum
- ovm\_severity
- ovm\_verbosity

# 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](#)

## A

**abstract**

ovm\_comparer

ovm\_packer

ovm\_recorder

## B

**begin\_elements**

ovm\_printer\_knobs

**big\_endian**

ovm\_packer

**bin\_radix**

ovm\_printer\_knobs

## C

**check\_type**

ovm\_comparer

**count**

ovm\_sequencer\_base

## D

**dec\_radix**

ovm\_printer\_knobs

**default\_radix**

ovm\_printer\_knobs

ovm\_recorder

**default\_sequence**

ovm\_sequencer\_base

**depth**

ovm\_printer\_knobs

## E

**enable\_print\_topology**

ovm\_root

**enable\_stop\_interrupt**

ovm\_component

**end\_elements**

ovm\_printer\_knobs

**F****finish\_on\_completion**

ovm\_root

**footer**

ovm\_printer\_knobs

**full\_name**

ovm\_printer\_knobs

**G****global\_indent**

ovm\_printer\_knobs

**H****header**

ovm\_printer\_knobs

**hex\_radix**

ovm\_printer\_knobs

**I****id\_count**

ovm\_report\_server

**identifier**

ovm\_printer\_knobs

ovm\_recorder

**indent\_str**

ovm\_hier\_printer\_knobs

**K****knobs**

ovm\_printer

ovm\_table\_printer

ovm\_tree\_printer

**M****max\_random\_count**

ovm\_sequencer\_base

**max\_random\_depth**

ovm\_sequencer\_base

**max\_width**

ovm\_printer\_knobs

**mcd**

ovm\_printer\_knobs

**miscompares**

ovm\_comparer

## N

**name\_width**

ovm\_table\_printer\_knobs

**new**

ovm\_line\_printer

ovm\_table\_printer

ovm\_tree\_printer

## O

**oct\_radix**

ovm\_printer\_knobs

**ovm\_default\_comparer****ovm\_default\_line\_printer****ovm\_default\_packer****ovm\_default\_printer****ovm\_default\_recorder****ovm\_default\_table\_printer****ovm\_default\_tree\_printer****ovm\_test\_done****ovm\_top**

ovm\_root

## P

**phase\_timeout**

ovm\_root

**physical**

ovm\_comparer

ovm\_packer

ovm\_recorder

**policy**

ovm\_comparer

**pound\_zero\_count**  
ovm\_sequencer\_base

**prefix**  
ovm\_printer\_knobs

**print\_config\_matches**  
ovm\_component

**print\_enabled**  
ovm\_component

## R

**recursion\_policy**  
ovm\_recorder

**reference**  
ovm\_printer\_knobs

**result**  
ovm\_comparer

## S

**separator**  
ovm\_tree\_printer\_knobs

**seq\_item\_export**  
ovm\_sequencer#(REQ,RSP)

**seq\_kind**  
ovm\_sequence\_base

**sev**  
ovm\_comparer

**show\_max**  
ovm\_comparer

**show\_radix**  
ovm\_printer\_knobs

**show\_root**  
ovm\_hier\_printer\_knobs

**size**  
ovm\_printer\_knobs

**size\_width**  
ovm\_table\_printer\_knobs

**stop\_timeout**  
ovm\_root

## T

**tr\_handle**

ovm\_recorder

**truncation**

ovm\_printer\_knobs

**type\_name**

ovm\_printer\_knobs

**type\_width**

ovm\_table\_printer\_knobs

## U

**unsigned\_radix**

ovm\_printer\_knobs

**use\_metadata**

ovm\_packer

**use\_ovm\_seeding**

ovm\_object

## V

**value\_width**

ovm\_table\_printer\_knobs

**verbosity**

ovm\_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

☐ BODY

## C

☐ CREATED

## E

☐ ENDED

## F

☐ FINISHED

## O

☐ OVM\_BIN  
☐ OVM\_CALL\_HOOK  
☐ OVM\_COUNT  
☐ OVM\_DEC  
☐ OVM\_DEEP  
☐ OVM\_DISPLAY  
☐ OVM\_ENUM  
☐ OVM\_ERROR  
☐ OVM\_EXIT  
☐ OVM\_EXPORT  
☐ OVM\_FATAL  
☐ OVM\_FULL  
☐ OVM\_HEX  
☐ OVM\_HIGH  
☐ OVM\_IMPLEMENTATION  
☐ OVM\_INFO  
☐ OVM\_LOG  
☐ OVM\_LOW  
☐ OVM\_MEDIUM  
☐ OVM\_NO\_ACTION  
☐ OVM\_NONE  
☐ OVM\_OCT  
☐ OVM\_PORT  
☐ OVM\_REFERENCE  
☐ OVM\_SHALLOW



**OVM\_STRING**  
**OVM\_TIME**  
**OVM\_UNSIGNED**  
**OVM\_WARNING**

## **P**

**POST\_BODY**  
**PRE\_BODY**

## **S**

**STOPPED**

# 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**

[ovm\\_algorithmic\\_comparator#\(BEFORE,AFTER,TRANSFORMER\)](#)

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

### **analysis\_export**

[ovm\\_subscriber](#)

### **analysis\_port#(T)**

[tlm\\_analysis\\_fifo#\(T\)](#)

## B

### **before\_export**

[ovm\\_algorithmic\\_comparator#\(BEFORE,AFTER,TRANSFORMER\)](#)

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

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

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

## G

### **get\_ap**

[tlm\\_fifo\\_base#\(T\)](#)

### **get\_peek\_export**

[tlm\\_fifo\\_base#\(T\)](#)

### **get\_peek\_request\_export**

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

### **get\_peek\_response\_export**

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

## M

### **master\_export**

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

## P

### **pair\_ap**

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

**put\_ap**

tlm\_fifo\_base#(T)

**put\_export**

tlm\_fifo\_base#(T)

**put\_request\_export**

tlm\_req\_rsp\_channel#(REQ,RSP)

**put\_response\_export**

tlm\_req\_rsp\_channel#(REQ,RSP)

**R****req\_export**

ovm\_push\_driver#(REQ,RSP)

**req\_port**

ovm\_push\_sequencer#(REQ,RSP)

**request\_ap**

tlm\_req\_rsp\_channel#(REQ,RSP)

**response\_ap**

tlm\_req\_rsp\_channel#(REQ,RSP)

**rsp\_export**

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

**rsp\_port**

ovm\_driver#(REQ,RSP)

ovm\_push\_driver#(REQ,RSP)

**S****seq\_item\_port**

ovm\_driver#(REQ,RSP)

**slave\_export**

tlm\_req\_rsp\_channel#(REQ,RSP)

**T****transport\_export**

tlm\_transport\_channel#(REQ,RSP)