OVM GOLDEN REFERENCE GUIDE

A concise guide to OVM – the Open Verification Methodology

Open Verification Methodology

DOULOS

OVM Golden Reference Guide

Version 2.0, September 2008

First published by Doulos 2008

ISBN 0-9547345-6-4

Doulos
Church Hatch
22 Market Place
Ringwood
Hampshire
BH24 1AW
UK

Tel +44 (0) 1425 471223
Fax +44 (0) 1425 471573

Email: info@doulos.com
Web: http://www.doulos.com

Contents

Page

Preface

The OVM Golden Reference Guide is a compact reference guide to the Open Verification Methodology for SystemVerilog.

The intention of the guide is to provide a handy reference. It does not offer a complete, formal description of all OVM classes and class members. Instead it offers answers to the questions most often asked during the practical application of OVM in a convenient and concise reference format. It is hoped that this guide will help you understand and use OVM more effectively.

This guide is not intended as a substitute for a full training course and will probably be of most use to those who have received some training. Also it is not a replacement for the official OVM Class Reference, which forms part of the OVM and is available from www.ovmworld.org.

The OVM Golden Reference Guide was developed to add value to the Doulos range of training courses and to embody the knowledge gained through Doulos methodology and consulting activities.

For more information about these, please visit the web-site www.doulos.com. You will find a set of OVM tutorials at www.doulos.com/knowhow. For those needing full scope training in OVM, see the OVM Adopter Class from Doulos.

Using This Guide

The OVM Golden Reference Guide comprises a Brief Introduction to OVM, information on Finding What You Need in This Guide, the Alphabetical Reference section and an Index.

This guide assumes a knowledge of SystemVerilog and testbench automation. It is not necessary to know the full SystemVerilog language to understand the OVM classes, but you do need to understand object-oriented programming in SystemVerilog. You will find some tutorials at http://www.doulos.com/knowhow.

Organization

The main body of this guide is organized alphabetically into sections and each section is indexed by a key term, which appears prominently at the top of each page. Often you can find the information you want by flicking through the guide looking for the appropriate key term. If that fails, there is a full index at the back.

Except in the index, the alphabetical ordering ignores the prefix `ovm_`. So you will find Field Macros between the articles `ovm_factory` and `ovm_in_order_*_comparator`.

Finding What You Need in This Guide on page 8 contains a thematic index to the sections in the alphabetical reference.

The Index

Bold index entries have corresponding pages in the main body of the guide. The remaining index entries are followed by a list of appropriate page references in the alphabetical reference sections.

Methods and Members

Most sections document the methods and members of OVM classes. Not all public members and methods are included; we have tried to concentrate on those that you may want to use when using the OVM. Also, deprecated features are not usually included. For details on all the members and methods, please refer to the official OVM Class Reference and the actual OVM source code.

A Brief Introduction To OVM

Background

Various verification methodologies have emerged in recent years. One of the first notable ones was the *e* Reuse Methodology for verification IP using the *e* language. This defines an architecture for verification components together with a set of naming and coding recommendations to support reuse across multiple projects. The architecture of eRM and some of its concepts (e.g. sequences) were used to create the Cadence Universal Reuse Methodology (URM), for SystemVerilog.

The SystemC TLM 1.0 library defines a transport layer for transaction level models. Mentor Graphics' Advanced Verification Methodology (AVM) uses SystemVerilog equivalents of the TLM ports, channels and interfaces to communicate between verification components (there is also a SystemC version of AVM that uses the TLM classes directly).

URM and AVM have been joined together to form the Open Verification Methodology (OVM). This still uses the TLM 1.0 transport layer for communicating between components, even though TLM 2.0 has subsequently been released.

The Open Verification Methodology combines the classes from AVM and URM. It is backwards compatible with both.

Transaction-level Modeling

Transaction-level modeling involves communication using function calls, with a transaction being the data structure passed to or from a function as an argument or a return value.

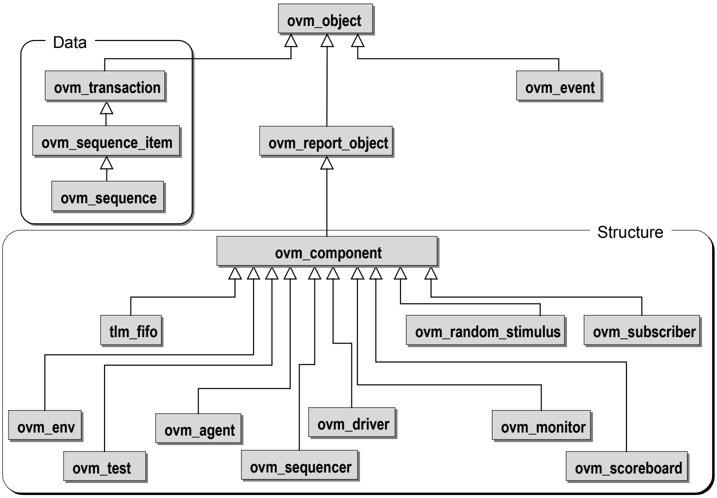
Transaction level modeling is a means of accelerating simulation by abstracting the way communication is modeled. Whereas an HDL simulator models communication by having a separate event for each pin wiggle, a transaction level model works by replacing a bunch of related pin wiggles by a single transaction. Obvious examples would be a bus read or bus write.

OVM

OVM is implemented entirely in SystemVerilog so it should work on any simulator that supports the full IEEE 1800 standard. Note that at the time of writing, none of the simulators available implements the complete 1800 standard. However, both Mentor Graphics QuestaSim and Cadence Incisive simulators support all of the language features required by OVM.

The OVM source code can be downloaded from the OVM web site. There is also an active user forum on this site that any registered user can freely participate in.

OVM Class Hierarchy



The main OVM classes form a hierarchy as shown here. The ovm_object class is the base class for all other OVM classes.

User defined transaction classes should be derived from ovm_transaction or one of its children.

TLM channels such as tlm_fifo are derived from ovm_report_object so include the ability to print their state. There are also implementations of the SystemC TLM 1.0 interface classes (not shown here) that are inherited by TLM channels.

The ovm_component class is for user-defined verification components. It has a run task that is automatically invoked at the start of a simulation.

Base classes for common verification components such as environments, drivers and monitors are also provided.

Finding What You Need in This Guide

This section highlights the major areas of concern when creating or modifying an OVM verification environment, and indicates the most important classes that you will need to use for each of these areas. Classes and OVM features highlighted in **bold** have their own articles in the Alphabetical Reference section of this Guide.

Designing Transaction Data

OVM verification environments are built using *transaction level modeling*. Stimulus, responses and other information flowing around the testbench are, as far as possible, stored as transactions – objects carrying a high-level, fairly abstract representation of the data. Because these objects are designed as SystemVerilog classes, they can all be derived from a common base class **ovm_object**, **ovm_transaction** or **ovm_sequence_item**. When designing classes derived from these, you not only add data members and methods to model the data itself, but also overload various base class methods so that each data object knows how to do a standard set of operations such as copying or printing itself.

Creating Problem-Specific Testbench Components

The core of a testbench is the set of *testbench components* that will manipulate the transaction data. Some of these components need a direct connection to signals in HDL modules representing the device-under-test (DUT) and its supporting structures. Other components operate at a higher level of abstraction and work only with transaction data. Notwithstanding this distinction, all the components you create should be represented by classes derived from `ovm_component`. This base class is described in detail in the article on **Component**.

Components need to pass data to and from other components. In OVM this is achieved by providing the components with suitable **ports and exports** through which they can communicate with one another. Components never need to know about neighboring components to which they are connected; instead, components send and receive data by means of calls to methods in their ports. The set of methods implemented by ports is known as the **TLM Interfaces**. This is the fundamental principle of transaction level modeling: one component calls a TLM interface method in its port and, thanks to the connection mechanism, the corresponding method is automatically called in a different component's export.

When a component makes data available for optional inspection by other parts of the testbench, it does so through a special kind of port known as a **tlm_analysis_port**, which connects to a **tlm_analysis_export** on each component that wishes to monitor the data.

Almost every block of testbench functionality should be coded as a component. However, some kinds of functional block are sufficiently common and sufficiently well-defined that special versions of the component base classes are provided for them, including **ovm_driver**, **ovm_monitor** and **ovm_scoreboard**. In some situations it is useful to build groupings of components, with the connections between them already defined; **ovm_agent** is such a predefined grouping, and you can also use **ovm_env** to create such blocks.

Choosing and Customizing Built-in OVM Components

Some components have standard functionality that can be provided in a base class and rather easily tailored to work on any kind of transaction data. OVM provides **ovm_in_order_*_comparator** and **ovm_algorithmic_comparator**.

Communication between components is often made simpler by using FIFO channels rather than direct connection. Built-in components **tlm_fifo** and **tlm_analysis_fifo** provide a complete implementation of such FIFOs.

All these built-in components can be tailored to work with any type of transaction data because they are defined as parameterized classes. It is merely necessary to instantiate them with appropriate type parameters.

Constructing the Testbench: Phases and the Factory

The structure of your testbench should be described as components that are members of a top-level environment derived from **ovm_env**. The top level test automatically calls a series of virtual methods in each object of any class derived from **ovm_component** (which in turn includes objects derived from **ovm_env**). The automatically-executed steps of activity that call these virtual methods are known as **Phases**.

Construction of the sub-components of any component or environment object is accomplished in the `build` phase; connections among sibling sub-components is performed in the `connect` phase; execution of the components' run-time activity is performed in the `run` phase. Each of these phases, and others not mentioned here, is customized by overriding the corresponding phase method in your derived component or environment class.

Construction of sub-components in a component's `build` phase can be achieved by directly calling the sub-components' constructors. However, it is more flexible to use the **ovm_factory** to construct components, because the factory's operation can be flexibly configured so that an environment can be modified, *without changing its source code*, to behave differently by constructing derived-class versions of some of its components.

The factory can also be used to create data objects in a flexible way. This makes it possible to adjust the behavior of stimulus generators without modifying their source code.

Structured Random Stimulus

A predefined component **ovm_random_stimulus** can be used to generate a stream of randomized transaction data items with minimal coding effort. However, although the randomization of each data item can be controlled using constraints, this component cannot create structured stimulus: there is no way to define relationships between successive stimulus data items in the random stream. To meet this important requirement, OVM provides the **Sequences** mechanism, described in more detail in articles on **ovm_sequence**, **ovm_sequence_item**, **ovm_sequencer**, **Special Sequences**, **Sequencer Interface and Port** and **Sequence Action Macros**.

Writing and Executing a Test

Having designed a test environment it is necessary to run it. OVM provides the **ovm_test** class as a base class for user-defined top-level tests, defining the specific usage of a test environment for a single test run or family of runs. In the 1.1 release of OVM, a new mechanism for encapsulating the whole of your OVM testbench and top-level test was introduced; it is described in the article on **ovm_root**.

Configuration

When a test begins, there is an opportunity for user code in the test class to provide configuration information that will control operation of various parts of the testbench. The mechanisms that allow a test to set this configuration information and, later, allow each part of the testbench to retrieve relevant configuration data, is known as **Configuration**. For configuration to work correctly it is necessary for user-written components to respect a set of conventions concerning the definition of data members in each class. Data members that can be configured are known as *fields*, and must be set up using **Field Macros** provided as part of OVM.

One of the most commonly used and important aspects of configuration is to choose which HDL instances and signals are to be accessed from each part of the testbench. Although OVM does not provide any specific mechanisms for this, the conventional approach is outlined in the article **Virtual Interface Wrapper**.

Reporting and Text Output

As it runs, a testbench will generate large amounts of textual information. To allow for flexible control and redirection of this text output, a comprehensive set of reporting facilities is provided. These reporting features are most easily accessed through methods of the base class **ovm_report_object**. Detailed control over text output formatting is achieved using **ovm_printer** and **ovm_printer_knobs**.

OVM GOLDEN REFERENCE GUIDE

OVM Alphabetical Reference

DOULOS

ovm_agent

The `ovm_agent` class is derived from `ovm_component`. Each user-defined agent should be created as a class derived from `ovm_agent`. There is no formal definition of an *agent* in OVM, but an agent should be used to encapsulate everything that's needed to stimulate and monitor one logical connection to a device-under-test.

A typical agent contains instances of a driver, monitor and sequencer (*described in more detail in later sections*). It represents a self-contained verification component designed to work with a specific, well-defined interface – for example, a standard bus such as AHB or Wishbone. An agent should be configurable to be *either* a purely passive monitor *or* an active verification component that can both monitor and stimulate its physical interface. This choice should be controlled by the value of a `bit` data member, conventionally named `is_active`; the driver and sequencer sub-components of the agent should be constructed only if this data member has been configured to be true (1) before the agent's `build` method is invoked.

Agents generally have rather little functionality of their own. An agent is primarily intended as a wrapper for its monitor, driver and sequencer.

Declaration

```
class ovm_agent extends ovm_component;

typedef enum bit { OVM_PASSIVE=0, OVM_ACTIVE=1 }
        ovm_active_passive_enum;
```

Methods

`function` **`new`**`( string name, ovm_component parent = null);`	Constructor; mirrors the superclass constructor in `ovm_component`

Members

Note: These fields are not defined in `ovm_agent`, but should almost always be provided as part of any user extension.

`ovm_active_passive_enum` **`is_active`**`;`	Controls whether this instance has an active driver.
`ovm_analysis_port #(`*`transaction_class`*`)` **`monitor_ap`**`;`	Exposes the monitor sub-component's analysis output to users of the agent.

Example

```
class example_agent extends ovm_agent;
  example_sequencer #(example_transaction) m_sequencer;
  example_driver m_driver;
  example_monitor m_monitor;
  ovm_active_passive_enum is_active;
  ovm_analysis_port #(example_transaction) monitor_ap;
  example_virtual_if_wrapper vi_wrapper;

  virtual function void build();
    super.build();
    $cast(m_monitor, create_component(
          "example_monitor", "m_monitor") );
    monitor_ap = new("monitor_ap", this);
    if (is_active) begin
      $cast(m_sequencer, create_component(
            "example_sequencer", "m_sequencer") );
      $cast(m_driver, create_component(
            "example_driver", "m_driver") );
    end
  endfunction: build

  virtual function void connect;
    m_monitor.monitor_ap.connect(monitor_ap);
    ... // code to pass physical connection information
        // (in vi_wrapper) to monitor omitted for clarity
    if (is_active) begin
        m_driver.seq_item_prod_if.connect_if(
          m_sequencer.seq_item_cons_if);
    end
  endfunction: connect
  ...
  `ovm_component_utils_begin(example_agent)
    `ovm_field_object(vi_wrapper,OVM_DEFAULT)
    `ovm_field_enum(ovm_active_passive_enum,
                    is_active,OVM_DEFAULT)
  `ovm_component_utils_end

endclass: example_agent
```

Tips

- An agent should represent a block of "verification IP", a re-usable component that can be connected to a given DUT interface and then used as part of the OVM test environment. It should be possible to specify a single virtual-interface connection for the agent, and then have the agent's `build` or `connect` method automatically provide that connection (or appropriate modports of it) to any relevant sub-components such as its driver and monitor.
- Every active agent should have a sequencer sub-component capable of generating randomized stimulus for its driver sub-component.
- Provide your agent with an analysis port that is directly connected to the analysis port of its monitor sub-component. In this way, users of the agent can get analysis output from it without needing to know about its internal structure.
- You will need to create appropriate sequences that the sequencer can use to generate useful stimulus.
- Use `ovm_active_passive_enum` rather than a bit to set the agent mode.

Gotchas

`ovm_agent` has no methods or data members of its own, apart from its constructor and what it inherits from `ovm_component`. However, to build a properly-formed agent requires you to follow various methodology guidelines, including the recommendations in this article. In particular, you should always provide an `is_active` flag, a configurable means to connect the agent to its target physical interface, and the three standard sub-components (driver, monitor and sequencer). An agent is in effect a piece of verification IP that can be deployed on many different projects; as such it should be designed to be completely self-contained and portable.

See also

ovm_component, ovm_driver, ovm_monitor, ovm_sequencer

A suitably parameterized and configured `ovm_algorithmic_comparator` can be used to check the end-to-end behavior of a DUT that manipulates (transforms) its input data before sending it to an output port. Its behavior is generally similar to `ovm_in_order_class_comparator`, but it requires a reference model of the DUT's data manipulation behavior, in the form of a special "transformer" object, to be passed to the comparator's constructor.

The comparator provides two analysis exports, `before_export` for input transactions and `after_export` for output transactions (after processing by the DUT). Unlike the other OVM comparator classes, these two exports are not required to support the same type of transaction.

Transactions received on `before_export` are first processed by the `transform` method of a user-provided "transformer" object – in effect, a reference model. The result of the `transform` method is the predicted DUT output, represented as a transaction of the same type as will be received from the DUT output on `after_export`.

Internally, there is an `in_order_class_comparator`. Its `before_export` is fed with the transformed (predicted) transactions; its `after_export` sees the real DUT output. In this way, the DUT's output is compared with the expected values computed from the observed DUT input.

Declarations

```
class ovm_algorithmic_comparator
  #( type BEFORE = int, type AFTER = int,
     type TRANSFORMER = int )
    extends ovm_component;
```

Methods

`function new( TRANSFORMER transformer, string name , ovm_component parent ) ;`	Constructor – "transformer" is the reference-model object whose `transform` method will be used to predict DUT output values

Members

`ovm_analysis_export #( BEFORE )` **`before_export`**`;`	Connect to first transaction stream analysis port, typically monitored from a DUT's input
`ovm_analysis_imp #(AFTER, ...)` **`after_export`**`;`	Connect to second transaction stream analysis port, typically monitored from a DUT's output

Code pattern for any user-defined transformer class:

```
class example_transformer;

   ... // member variables to represent internal
       // state of the reference model

   function new(any appropriate arguments);
      ... // initialize the state of the reference
          // model based on the constructor arguments
   endfunction

   function after_class_type transform (
       before_class_type b );
      ... // maintain the state of the reference model
          // based on the new input transaction, and
          // compute and return the expected result
   endfunction

endclass
```

Example

Using `ovm_algorithmic_comparator` within a scoreboard component

```
class cpu_scoreboard extends ovm_scoreboard;

 // Fetched instructions are observed here:
 ovm_analysis_export #(fetch_xact) af_fetch_export;
 // Execution results are observed here:
 ovm_analysis_export #(exec_xact) af_exec_export;
 ovm_algorithmic_comparator
   #(.BEFORE(fetch_xact), .AFTER(exec_xact),
     .TRANSFORMER(Instr_Set_Simulator) ) m_comp;

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

 virtual function void build();
  super.build();
  // Create the transformer object
  Instr_Set_Simulator m_iss = new(...);
  // Create analysis exports
  af_fetch_export = new("af_fetch_export", this);
  af_exec_export  = new("af_exec_export",  this);
  // Supply the transformer object to the comparator
  m_comp          = new(m_iss, "comp", this);
 endfunction: build
```

```
 virtual function void connect;
  af_fetch_export.connect( m_comp.before_export );
  af_cpu_export.connect( m_comp.after_export  );
 endfunction: connect

 integer m_log_file;
 virtual function void start_of_simulation;
  m_log_file = $fopen("cpu_comparator_log.txt");
  set_report_id_action_hier("Comparator Match",LOG);
  set_report_id_file_hier  ("Comparator Match",m_log_file);
  set_report_id_action_hier("Comparator Mismatch",LOG);
  set_report_id_file_hier  ("Comparator Mismatch",
                             m_log_file);
 endfunction: start_of_simulation

 virtual function void report;
  string txt;
  $sformat(txt, "#matches = %d, #mismatches = %d",
           m_comp.m_matches, m_comp.m_mismatches);
  ovm_report_info("", txt);
 endfunction: report

 `ovm_component_utils(cpu_scoreboard)
endclass: cpu_scoreboard
```

Tips

Although there is no ready-to-use OVM class for the transformer (reference model) object, it is probably a good idea to derive it from `ovm_component` so that its behavior and instantiation can be controlled by the configuration and factory mechanisms.

Gotchas

In current releases of OVM the transformer and comparator objects in an `ovm_algorithmic_comparator` have `local` access. Consequently they cannot easily be controlled from code written in a derived class In particular, there is no way to flush the comparator. Users may wish to write their own version of `ovm_algorithmic_comparator`, using the OVM code as a model, to provide better control over the comparison process. This problem is not so important for the transformer object, because it must be supplied in the algorithmic comparator's constructor and therefore the environment can easily keep a reference to it.

See also

ovm_analysis_port, ovm_analysis_export, ovm_in_order_*_comparator

ovm_analysis_export

Components that work with transactions typically make those transactions available to other parts of the testbench through analysis ports. A monitoring or analysis component that wishes to observe these transactions must subscribe to the producing component's analysis port. This is achieved by connecting an analysis export on each subscriber to the analysis port on the producer. The analysis export provides the write method required (called) by the analysis port. There is no limit to the number of analysis exports that can be connected to a given analysis port. An analysis export may be connected to one or more analysis exports on child components or implementations.

Declaration

```
class ovm_analysis_export #(type T = int)
  extends ovm_port_base #(tlm_if_base #(T,T));
```

Methods

`function new( string name, ovm_component parent );`	constructor
`function void write( input T t );`	called implicitly by connected analysis port's `write` method, forwards call to connected exports or implementation(s)
`virtual function string get_type_name();`	Returns `"ovm_analysis_export"`
`function void connect†( port_type provider);`	connects the analysis export to another analysis export, or to an analysis imp, that implements a subscriber's `write` functionality

†Inherited from ovm_port_base

Example

This example shows the implementation of a specialized analysis component that contains two different subscribers, both observing the same stream of transactions. The analysis component has just one analysis export that is connected to both subscribers.

```
class custom_subscr_1 extends
  ovm_subscriber #(example_transaction);
  ...  // code for first custom subscriber

class custom_subscr_2 extends
```

```
  ovm_subscriber #(example_transaction);
  ... // code for second custom subscriber

class example_double_subscriber extends ovm_component;
  custom_subscr_1 subscr1;
  custom_subscr_2 subscr2;
  ovm_analysis_export #(example_transaction)
    analysis_export;

  function void build();
    $cast(subscr1, create_component (
      "custom_subscr_1", "subscr1" );
    $cast(subscr2, create_component (
      "custom_subscr_2", "subscr2" );
    analysis_export = new ( "analysis_export", this );
  endfunction

  function void connect();
    // Connect the analysis export to both internal components
    analysis_export.connect(subscr1.analysis_export);
    analysis_export.connect(subscr2.analysis_export);
  endfunction

endclass
```

Tips

- Every analysis export must ultimately be connected to an `ovm_analysis_imp` implementation that provides a `write` method. It is possible to connect an analysis port directly to an `ovm_analysis_imp`, but user-written components more commonly have an `ovm_analysis_export` that in its turn is connected either to one or more `ovm_analysis_imp`, or to an `ovm_analysis_export` on a sub-component that is a member of the component.
- An especially useful and common idiom is for a subscriber component to have a `tlm_analysis_fifo`. The component's `ovm_analysis_export` is then connected to the analysis FIFO's `analysis_export`. In this way, the user has no need to code an `ovm_analysis_imp` explicitly. Transactions from a producer's analysis port are written into the analysis FIFO without blocking. A thread in the user-written component can take transactions from the analysis FIFO's `get` port at its leisure.
- `ovm_subscriber` provides a convenient base class for user-written subscriber components that observe transactions from exactly one analysis

port. In `ovm_subscriber` the necessary arrangement of analysis export and implementation has already been coded, and it is only necessary for the user to override the base class's `write` method in their class derived from `ovm_subscriber`.

- The overall pattern of connection of analysis ports, exports and imps is:
    - A producer of analysis data should write that data to an analysis port.
    - An analysis port can be connected to any number of subscribers (including zero). Each subscriber can be another analysis port on a parent component, or an analysis export or analysis imp on a sibling component.
    - An analysis export can be connected to any number of subscribers. Each subscriber can be an analysis export or an analysis imp on a child component.

Gotchas

- You must call `new()` to create an instance of an analysis export in a component's build method.
- Analysis ports and exports must be parameterized for the type of transaction they carry. The transaction parameters for connected analysis ports and exports must match exactly.
- Conventionally, a producer calls the non-blocking write method for its analysis port and assumes that the transaction object will not be required once write has returned. A subscriber should therefore never store a reference to a written transaction: if it needs to reference the transaction at some future time step, its write method should create a *copy* and store that instead.
- Analysis components should never write to objects they are given for analysis. If your analysis component needs to modify an object it is given, it should make a copy and work on that. Other analysis components might also have stored the same reference, and should be able to assume that the object will not change.

See also

ovm_subscriber, ovm_analysis_port, tlm_analysis_fifo, ports and exports

It is often necessary for some parts of a testbench – for example, end-to-end checkers and coverage collectors – to observe the activity of other testbench components. *Analysis ports* provide a consistent mechanism for such observation.

Declaration

```
class ovm_analysis_port #(type T = int)
  extends ovm_port_base #(tlm_if_base #(T,T));
```

Methods

`function new( string name, ovm_component parent = null);`	Constructor
`function void write( transaction_type t);`	Publishes transaction t to any connected subscribers
`virtual function string get_type_name();`	Returns `"ovm_analysis_port"`
`function void connect( port_type provider);`	Connects the analysis port to another analysis port, or to an analysis export that implements a subscriber's `write` functionality

Example

See the article on **ovm_monitor** for an example of using an analysis port.

Tips

- When designing any component, use an analysis port to make data available to other parts of the testbench. The analysis port's `write` method does not block, and therefore cannot interfere with the procedural flow of your component. If there are no subscribers to the analysis port, calling its `write` method has very little overhead.
- Any component that wishes to make transaction data visible to other parts of the testbench should have a member of type `ovm_analysis_port`, parameterized for the transaction's data type. This analysis port should be constructed during execution of the component's `build` method.
- Whenever the component has a transaction that it wishes to publish, it should call the analysis port's `write` method with the transaction variable as its argument. This method is a function and so is guaranteed not to block. It has the effect of calling the `write` method in every connected subscriber. If there is no subscriber connected, the method has no effect.

- The member variable name for an analysis port conventionally has the suffix `_ap`. There is no limit to the number of analysis ports on a component.
- Monitor components designed to observe transactions on a physical interface (see **ovm_monitor**) are sure to have an analysis port through which they can deliver observed transactions. Other components may optionally have analysis ports to expose transaction data that they manipulate or generate, so that other parts of the testbench can observe those data. Note, in particular, that every `tlm_fifo` has two analysis ports named `put_ap` and `get_ap`; these ports expose, respectively, transactions pushed to and popped from the FIFO.

Gotchas

- You must call `new` to create an instance of an analysis port in a component's build method.
- The `write` method of an analysis port takes a reference (handle) to the transaction as an input argument. Consequently, it is possible (although not recommended) for the target of the `write()` to modify the transaction object. To avoid difficulties related to this issue, consider writing a *copy* of the transaction to the analysis port using the transaction's own `copy` method (although in a well-behaved system, it is usually the responsibility of the subscriber to make the copy):

```
my_ap.write(tr.copy());
```

- Other parts of the OVM library, including the built-in comparator components, assume that transactions received from an analysis port are "safe" and have already been cloned if necessary.

See also

ovm_monitor, ovm_subscriber, ovm_analysis_export, tlm_fifo, ports and exports

Components are used as the structural elements and functional models in an OVM testbench. Class `ovm_component` is the virtual base class for all components. It contains methods to configure and test the components within the hierarchy, placeholders for the phase callback methods, convenience functions for calling the OVM factory, functions to configure the OVM reporting mechanism and functions to support transaction recording. It inherits other methods from its `ovm_report_component` and `ovm_object` base classes.

Prior to OVM 2.0, `ovm_component` was only used as the base class for components that did not consume simulation time: components with independently-executing, time-consuming activity (known as "threads" or "processes").used an alternative `ovm_threaded_component` base class that extended `ovm_component` by adding a `run` method and a few methods for process control. From OVM 2.0 onwards, `ovm_threaded_component` is a typedef for `ovm_component`, provided for backwards compatibility (it should not be used in new code).

Declaration

```
virtual class ovm_component extends ovm_report_object;
```

Constructor and interaction with hierarchy

Functions are provided to access the child components (by name or by handle). The order in which these are returned is set by an underlying associative array that uses the child component names as its key. The lookup function searches for a named component (the name must be an exact match – wildcards are not supported). If the name starts with a ".", the search looks for a matching hierarchical name in ovm_top, otherwise it looks in the current component.

Methods

`function **new**( string name, ovm_component parent);`	Constructor
`function string **get_name**†();`	Returns the name
`virtual function string **get_full_name**();`	Returns the full hierarchical path name
`virtual function void **set_name**( string name);`	Renames the component and updates children's hierarchical names
`virtual function string **get_type_name**†();`	Returns type name
`virtual function ovm_component **get_parent**();`	Returns handle to parent component

function ovm_component **get_child**(string name);	Returns handle to named child component
function int **get_first_child**(ref string name);	Get the name of the first child
function int **get_next_child**(ref string name);	Get the name of the next child
function int **get_num_children**();	Return the number of child components
function int **has_child**(string name);	True if child exists
function ovm_component **lookup**(string name);	Search for named component (no wildcards)
function void **print**(ovm_printer printer=null);	Prints the component†

†Inherited from ovm_object

OVM phases and control

Components provide virtual callback methods for each OVM phase. These methods should be overridden in derived component classes to implement the required functionality. Additional methods are provided as hooks for operations that might be required within particular phases.

Phase Callback Methods

For further details of these, see **Phase**.

virtual function void **build**();	Build phase callback
virtual function void **connect**();	Connect phase callback
virtual function void **end_of_elaboration**();	End_of_elaboration phase callback
virtual function void **start_of_simulation**();	Start_of_simulation phase callback
virtual task **run**();	Run phase callback
virtual function void **extract**();	Extract phase callback
virtual function void **check**();	Check phase callback
virtual function void **report**();	Report phase callback

Phase Support Methods

The connections associated with a particular component may be checked by overriding the `resolve_bindings` function. This is called automatically immediately before the `end_of_elaboration` phase or may be called explicitly by calling `do_resolve_bindings`.

The `flush` function may be overridden for operations such as flushing queues and general clean up. It is not called automatically by any of the phases but is called for all children recursively by `do_flush`.

Process control for the currently executing task in an `ovm_component` is provided by the `suspend`, `resume`, `kill` and `status` methods. They are convenience methods for the standard SystemVerilog `std::process` class. The `do_kill` function recursively calls `kill` for a component and all of its children.

There are two methods of stopping the currently executing phase task: `kill` causes it to terminate immediately; calling `ovm_top.stop_request` by default also causes it to terminate immediately. If the `enable_stop_interrupt` bit is set, a stop request calls the `stop` task and does not terminate the phase until the `stop` task completes. If the `stop` task has been overridden and includes a time delay, this will give the phase task time to complete its current activity before it terminates.

The phases are usually executed automatically in the order defined by OVM. In special cases where the OVM scheduler is not used (e.g. a custom simulator/emulator), it is possible to launch the phases explicitly. This should not be attempted for typical testbenches.

`virtual function void` **`resolve_bindings`**`();`	Called immediately before end_of_elaboration phase – override to check connections
`function void` **`do_resolve_bindings`**`();`	Calls resolve_bindings for current component and recursively for its children
`virtual function void` **`flush`**`();`	Callback intended for clearing queues
`function void` **`do_flush`**`();`	Recursively calls flush for all children
`virtual task` **`suspend`** `();`	Suspend current task
`virtual task` **`resume`** `();`	Resume current task
`virtual function void` **`kill`** `();`	Kills the current task-based phase (e.g. run)
`virtual function void` **`do_kill_all`**`();`	Recursively call kill for all children
`function string` **`status`** `();`	Return status of current task

`virtual task` **`stop`**`(string ph_name);`	Called after stop request if enable_stop_interrupt bit is set. Override to delay stopping
`virtual function void` **`do_func_phase`**`(ovm_phase phase);`	Explicitly start a function-based phase
`virtual task` **`do_task_phase`** `(ovm_phase phase);`	Explicitly start a task-based phase

Members

`protected int` **`enable_stop_interrupt`** `= 0;`	Set to 1 to enable stop task

Component configuration

Components work with the OVM configuration mechanism to set the value of members using a string-based interface.

See **Configuration** for full details.

Methods

`virtual function void` **`set_config_int`**`(string inst_name, string field_name, ovm_bitstream_t value);`	Sets an integral-valued configuration item.
`virtual function void` **`set_config_string`**`(string inst_name, string field_name, string value);`	Sets a string-valued configuration item.
`virtual function void` **`set_config_object`**`(string inst_name, string field_name, ovm_object value, bit clone=1);`	Sets a configuration item as an `ovm_object` (or `null`). By default, the object is cloned.
`virtual function bit` **`get_config_int`**`(string field_name, inout ovm_bitstream_t value);`	Gets an integral-valued configuration item. Updates member and returns `1'b1` if field name found.
`virtual function bit` **`get_config_string`**`(string field_name, inout string value);`	Gets a string-valued configuration item. Updates member and returns `1'b1` if field name found.

`virtual function bit`
`get_config_object``(`
`  string field_name,`
`  inout ovm_object value,`
`  input bit clone=1);`
	Gets a configuration item as an `ovm_object` (or `null`). Updates member and returns `1'b1` if field name found. By default, the object is cloned.

`virtual function void`
`apply_config_settings``(`
`  bit verbose=0);`
	Searches for configuration items and updates members

`function void`
`print_config_settings``(`
`  string field="",`
`  ovm_component comp=null,`
`  bit recurse=0);`
	Prints configuration information.

Members

`static bit` **`print_config_matches`**` = 0;`	For debugging. If set, configuration matches are printed.

The Factory

Components work with the OVM factory. They provide a set of convenience functions that call the `ovm_factory` member functions with a simplified interface.

From OVM 2.0 onwards, the factory supports both parameterized and non-parameterized components using a proxy class for each component type that is derived from class `ovm_object_wrapper`. The component utility macros register a component with the factory. They also define a nested proxy class named `type_id` and a static function `get_type` that returns the singleton instance of the proxy class for a particular component type.

The `create` and `clone` methods inherited from `ovm_object` are disabled for components.

See **ovm_factory, ovm_component_registry**

Methods

`function ovm_component`
`create_component``(`
`  string requested_type_name,`
`  string name);`
	Creates component as a child of current component (parent set to “this”)

`function ovm_object`
`create_object``(`
`  string requested_type_name,`
`  string name="");`
	Creates object as a child of current component

`static function void`
`set_type_override``(`
`  string original_type_name,`
`  string override_type_name,`
`  bit replace=1);`
	Overrides the type used by the factory for specified type

`static function void`
`set_type_override_by_type``(`
`  ovm_object_wrapper original_type,`
`  ovm_object_wrapper override_type,`
`  bit replace=1);`
	Overrides the type used by the factory for specified type

`function void`
`set_inst_override``(`
`  string relative_inst_path,`
`  string original_type_name,`
`  string override_type_name);`
	Overrides the type used by the factory for the specified instance only

`function void`
`set_inst_override_by_type``(`
`  string relative_inst_path,`
`  ovm_object_wrapper original_type,`
`  ovm_object_wrapper override_type);`
	Overrides the type used by the factory for the specified instance only

`function void`
`print_override_info``(`
`  string requested_type_name,`
`  string name="");`
	Prints details about the type of object that would be created for the given arguments

`static function type_id` **`get_type`**‡`();`	Returns proxy (wrapper) for class type required by factory methods

‡*Created by utility macros*

Hierarchical configuration of component report handler

Components provide methods to configure the OVM report handler for a particular component and recursively for all of its children. The methods can apply to all reports of a particular severity, all reports with a matching id or all reports whose severity and id both match those specified. Where there are overlapping conditions, matching both severity and id takes precedence over matching only id which takes precedence over matching only severity.

The reports can be written to a file that has previously been opened (using `$fopen`) if the action is specified as `OVM_LOG`. The file descriptor used for writing can be selected according to the severity or id of the message.

See **ovm_report_object**.

Methods

`function void` **`set_report_severity_action_hier`**`( ovm_severity s, ovm_action a);`	Set the action for reports with severity s
`function void` **`set_report_id_action_hier`**`( string id, ovm_action a);`	Set the action for reports with matching id
`function void` **`set_report_severity_id_action_hier`**`( ovm_severity s, string id, ovm_action a);`	Set the action for reports with both severity s AND matching id
`function void` **`set_report_default_file_hier`**`( OVM_FILE f);`	Set the default file written by action `OVM_LOG`
`function void` **`set_report_severity_file_hier`**`( ovm_severity s, OVM_FILE f);`	Set the file written by action `OVM_LOG` for reports of severity s
`function void` **`set_report_id_file_hier`**`( string id, OVM_FILE f);`	Set the file written by action `OVM_LOG` for reports with matching id
`function void` **`set_report_severity_id_file_hier`**`( ovm_severity s, string id, OVM_FILE f);`	Set the file written by action `OVM_LOG` for reports with both severity s AND matching id
`function void` **`set_report_verbosity_level_hier`**`( int v);`	Set verbosity threshold – only messages with lower verbosity written

Types

`typedef int` **`OVM_FILE`**`;`	File descriptor

Recording component transactions

Components provide methods to record their transactions to streams that can be displayed in a waveform viewer. The stream format is vendor-specific – only the API is defined by OVM. Each component has an event pool containing `accept_tr`, `begin_tr` and `end_tr` events that are triggered when transactions are accepted, when they begin and when they end, respectively.

As of OVM 2.0, the API is not fully defined and is subject to change.

See **ovm_transaction**

Methods

`function void` **`accept_tr`**`(`
`  ovm_transaction tr,`
`  time accept_time=0);`
	Call transaction's accept_tr function and trigger accept_tr event

`function integer` **`begin_tr`**`(`
`  ovm_transaction tr,`
`  string stream_name="main",`
`  string label="",`
`  string desc="",`
`  time begin_time=0);`
	Call transaction's begin_tr function, trigger begin_tr event and write transaction details to stream. Return transaction handle.

`function integer` **`begin_child_tr`**
`( ovm_transaction tr,`
`  integer parent_handle=0,`
`  string stream_name="main",`
`  string label="",`
`  string desc="",`
`  time begin_time=0);`
	Call transaction's begin_child_tr function, trigger begin_tr event and write transaction details to stream. Return transaction handle.

`function void` **`end_tr`**`(`
`  ovm_transaction tr,`
`  time end_time=0,`
`  bit free_handle=1);`
	Call transaction's end_tr function, trigger end_tr event and write transaction details to stream.

`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);`
	Records error in transaction stream.

`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);`
	Records "event" in transaction stream.

`virtual protected function void`
`do_accept_tr``(`
`  ovm_transaction tr);`
	Callback from accept_tr (by default does nothing)

`virtual protected function void`
`do_begin_tr``(`
`  ovm_transaction tr,`
`  string stream_name,`
`  integer tr_handle);`
	Callback from begin_tr (by default does nothing)

virtual protected function void **do_end_tr**(ovm_transaction tr, integer tr_handle);	Callback from end_tr (by default does nothing)

Members

protected ovm_event_pool **event_pool**;	Events for transaction accept, begin and end

General

Macros

Utility macros generate factory methods and the `get_type_name` function for a component. (See **Utility Macros** for details.)

```
`ovm_component_utils(TYPE)
```

or

```
`ovm_component_utils_begin(TYPE)
  `ovm_field_*(ARG,FLAG)
  ...
`ovm_component_utils_end
```

Fields specified in field automation macros will automatically be handled correctly in `copy`, `compare`, `pack`, `unpack`, `record`, `print` and `sprint`.

Parameterized components should use the

```
`ovm_component_param_utils(TYPE#(T))
```

or

```
`ovm_component_param_utils_begin(TYPE#(T))
`ovm_field_*(ARG,FLAG)
  ...
`ovm_component_utils_end
```

macros instead. Note that these do not generate a `get_type_name` function and register the component with the factory with the type name `"<unknown>"`.

The following field utility macros enable field automation macros to be used without generating the factory methods or `get_type_name` function. This can be useful for abstract base classes that will never get built by the factory.

```
`ovm_field_utils_begin(TYPE)
  `ovm_field_*(ARG,FLAG)
  ...
`ovm_field_utils_end
```

Rules

- Components may only be created and their ports (if any) bound before the `end_of_elaboration` phase: the hierarchy must be fixed by the start of this phase.
- Components cannot be cloned: the `clone` and `create` methods inherited from `ovm_object` are disabled.
- The `ovm_component` class is abstract and cannot be used to create objects directly. Components are instances of classes derived from `ovm_component`.

Example

Using `ovm_component` for a simple parameterized testbench class

```
class lookup_table #(WIDTH=10) extends ovm_component;
  ovm_blocking_get_imp #(int,lookup_table) get_export;

  int lut [WIDTH];
  int index = 0;

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

  function void build();
    super.build();
    foreach (lut[i]) lut[i] = i * 10;
    get_export = new("get_export",this);
  endfunction: build

  task get (output int val);
    #10 val = lut[index++];
    if (index > WIDTH-1) index = 0;
  endtask: get

  `ovm_component_param_utils_begin(lookup_table#(WIDTH))
    `ovm_field_sarray_int(lut,OVM_ALL_ON + OVM_DEC)
  `ovm_component_utils_end
endclass: lookup_table
```

Finding a component, changing its name and configuring its report handler to write reports to a file

```
module top2;
...
OVM_FILE fd = $fopen("drv2.log");
initial begin
  ovm_component c;
  ovm_phase build_ph;
  build_ph = ovm_top.get_phase_by_name("build")
  wait(build_ph.is_done());
  c = ovm_top.find("env2.m_driver");
  c.set_name("m_drv2");
  c.set_report_default_file_hier(fd);
  c.set_report_severity_action_hier(OVM_INFO,OVM_LOG);
end
```

Delaying return from run task after stop request:

```
class cov_col extends ovm_component;
  virtual chip_if if1;
  ...
  function new(string name, ovm_component parent);
    super.new(name,parent);
    enable_stop_interrupt = 1;
  endfunction : new

  task run();   ...   endtask

  task stop(string ph_name);
    // wait until bus transaction completed
    wait(chip_if.bus.done == 1);
    ovm_report_info("DRV","Stopping now");
  endtask: stop

endclass: cov_col
```

Tips

- OVM defines virtual base classes for various common testbench components (e.g. `ovm_monitor`) that are themselves derived from `ovm_component`. These should be used as base classes for testbench components in preference to `ovm_component` where appropriate.
- Use *`class_name`*`::type_id::create` or `create_component` to create new component instances in the build phase rather than `new` or `ovm_factory::create_component`.

- Use the field automation macros for any fields that need to be configured automatically. These also enable the fields of one component instance to be copied or compared to those of another.
- Set the required reporting options by calling the hierarchical functions (`set_report_*_hier`) for a top-level component since these settings are applied recursively to all child components.
- Stop the simulation by calling `ovm_top.stop_request` rather than `kill`. It gives components the opportunity to complete their current actions before halting.

Gotchas

- Component names must be unique at each level of the hierarchy.
- `new` and `build` should call the base class `new` (`super.new`) and `build` (`super.build`) methods respectively.
- Do not forget to register components with the factory, using `` `ovm_component_utils `` or `` `ovm_component_param_utils ``.
- Reports are only written to a file if a file descriptor has been specified and the action has been set to `OVM_LOG` for the particular category of report generated.

See also

configuration, ovm_factory, ovm_driver, ovm_monitor, ovm_scoreboard, ovm_agent, ovm_env, ovm_test, ovm_root

The `ovm_component_registry` class is used to register components with the factory. It acts as a "proxy" which allows a component to be registered with the factory before any instance of the component has actually been created. It enables the factory to support parameterized components since each "specialization" has a unique corresponding proxy that is registered with the factory.

The proxy instance is a specialization of the registry class created automatically by the component utility macros as a singleton instance of a nested class named `type_id`.

Calling the static `create` member function of a component's `type_id` nested class is the simplest way to instantiate components with the factory. It returns a handle of the correct type so no type casts are necessary.

Declaration

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

Methods

`function ovm_component` `**create_component**( string name, ovm_component parent);`	Used by factory to create instance
`static function this_type **get**();`	Returns proxy instance
`function string **get_type_name**();`	Returns type name
`static function T **create**( string name, ovm_component parent, string contxt="");`	Called by user to create component instance with the factory
`static function void **set_type_override**( ovm_object_wrapper override_type, bit replace=1);`	Overrides the type used by the factory for specified type
`static function void **set_inst_override**( ovm_object_wrapper override_type, string inst_path, ovm_component parent=null);`	Overrides the type used by the factory for the specified instance (path is relative if parent specified)

Members

`const static string **type_name** = Tname;`	Type name string
`typedef ovm_component_registry #(T,Tname) **this_type**;`	Type of proxy (for internal use)

Example

```
class compA extends ovm_component;
  `ovm_component_utils(compA)    // creates nested registry class
  ...
endclass: compA

class compAA extends compA;
  `ovm_component_utils(compAA)   // creates nested registry class
  ...
endclass: compAA

class compB extends ovm_compnent;
  compA A1;
  `ovm_component_utils(compB)    // creates nested registry class
  ...
  function void build();
    ...
    if(useAA) begin
     // override compA to use compAA before creating A1
     compA::type_id::set_type_override(compAA::get_type());
     A1 = compA::type_id::create("A1",this);
    end
  endfunction: build
endclass: compB
```

Tips

- Use the utility macros to create the registry class rather than declaring a typdef for it yourself. This ensures interoperability across simulators which may use different internal type names for the registry specialization.
- Use the `ovm_component_param_utils` macro for parameterized classes.

See also

ovm_component, ovm_object_wrapper, ovm_factory

Configuration

Configuration is a mechanism that OVM provides to modify the default state of components, either when they are built or when a simulation is run. It provides an alternative to factory configuration for modifying the way components are created. Configuration acts on components, but not transactions.

Configuration can be used to specify which components should be instantiated and settings for run-time behavior. It may also be used to change run-time behavior dynamically.

(Prior to OVM 1.1 there was a "configure" simulation phase. This was not related to the configuration mechanism discussed here: new designs should use the `end_of_elaboration`.phase instead).

Configuration Settings Table

Each component has a configuration settings table. There is also a global configuration settings table. These are accessed using the `set_config_*/get_config_*` methods and global functions respectively. Components will use the configuration settings tables during the build phase.

Typically, configurations are used in tests to configure the environment (`set_config_*`) without having to modify any code in the environment. This relies on components in the environment being responsible for getting (`get_config_*`) their own configuration information; however, field automation has the side-effect of making fields available for configuration, in which case configuration is automatic.

Configuration works "top-down" – global configuration settings have precedence over local ones, and a parent's configuration takes precedence over those of its children.

Whilst configuration usually occurs at build time, the configuration tables can also be queried at run-time, if appropriate.

Wildcard Matching

The `inst_name` and `field_name` arguments of the `set_config_*` functions and methods may include wildcards: "*" matches zero or more characters, "?" matches exactly one character. Note that "*" matches ".", the hierarchy separator, so matching may be through the whole hierarchy. (Wildcard matching uses the global function `ovm_is_match`.)

Printing Configuration Information

The `print_config_settings` method of `ovm_component` may be used to print configuration information about the component. Called without arguments, it prints all the component's configuration information. If the `field` argument is given (wildcards may not be used), configuration for the matching field is printed. `print_config_settings` may also recursively print configuration for the component's children (`recurse=1`) .

Global Functions

These functions use the global configuration table. See below for a description of the `inst_name` and `field_name` arguments.

`function void set_config_int( string inst_name, string field_name, ovm_bitstream_t value);`	Sets an integral-valued configuration item. (See below for `ovm_bitstream_t`)
`function void set_config_string( string inst_name, string field_name, string value);`	Sets a string-valued configuration item.
`function void set_config_object( string inst_name, string field_name, ovm_object value, bit clone=1);`	Sets a configuration item as an `ovm_object` (or `null`). By default, the object is cloned.

The bitstream type `ovm_bitstream_t` is a global type for passing integral values:

```
parameter OVM_STREAMBITS = 4096;
typedef logic signed [OVM_STREAMBITS-1:0] ovm_bitstream_t;
```

Methods of `ovm_component`

These functions are members of `ovm_component` (or an extension) and use the component's local configuration table. They have been reproduced here for convenience.

`virtual function void` **`set_config_int`**`( string inst_name, string field_name, ovm_bitstream_t value);`	Sets an integral-valued configuration item.
`virtual function void` **`set_config_string`**`( string inst_name, string field_name, string value);`	Sets a string-valued configuration item.
`virtual function void` **`set_config_object`**`( string inst_name, string field_name, ovm_object value, bit clone=1);`	Sets a configuration item as an `ovm_object` (or `null`). By default, the object is cloned.

`virtual function bit` **`get_config_int`**`(`
`  string field_name,`
`  inout ovm_bitstream_t value);`
	Gets an integral-valued configuration item. Updates member and returns `1'b1` if field name found.

`virtual function bit` **`get_config_string`**`(`
`  string field_name,`
`  inout string value);`
	Gets a string-valued configuration item. Updates member and returns `1'b1` if field name found.

`virtual function bit` **`get_config_object`**`(`
`  string field_name,`
`  inout ovm_object value,`
`  input bit clone=1);`
	Gets a configuration item as an `ovm_object` (or `null`). Updates member and returns `1'b1` if field name found. By default, the object is cloned.

`virtual function void` **`apply_config_settings`**`(` `  bit verbose=0);`	Searches for configuration items and updates members

`function void` **`print_config_settings`**`(`
`  string field="",`
`  ovm_component comp=null,`
`  bit recurse=0);`
	Prints configuration information.

Members of ovm_component

`static bit` **`print_config_matches`** `= 0;`	For debugging. If set, configuration matches are printed.

Examples

Automatic configuration using field automation:

```
class verif_env extends ovm_env;
  int m_n_cycles;
  string m_lookup;
  instruction m_template;
  typedef enum {IDLE,FETCH,WRITE,READ} bus_state_t;
  bus_state_t m_bus_state;
  ...
  `ovm_component_utils_begin(verif_env)
    `ovm_field_string(m_lookup,OVM_DEFAULT)
    `ovm_field_object(m_template,OVM_DEFAULT)
    `ovm_field_enum(bus_state_t,m_bus_state,OVM_DEFAULT)
  `ovm_component_utils_end
endclass: verif_env
```

```
class test2 extends ovm_test;
  register_instruction inst = new();
  string str_lookup;
  ...
  function void build();
    ...
    set_config_int("env1.*","m_bus_state",verif_env::IDLE);
    set_config_string("*","m_lookup",str_lookup);
    set_config_object("*","m_template",inst);
    ...
  endfunction : build
  ...
endclass : test2
```

Manual configuration

```
// In a test, create an entry “count” in the global configuration settings table …
set_config_int("*","count",1000);

// ... and retrieve the value of “count”
if (!get_config_int("count",m_n_cycles) )
  m_n_cycles = 1500; // use default value
```

Tips

Standard (Verilog) command-line plusargs may be used to modify the configuration. This provides a simple, yet flexible way of configuring a test.

Gotchas

- `set_config_* / get_config_*` only work within the hierarchy of ovm_components, not with other object types such as transactions and sequences.
- A wildcard “*” in an instance name will match any expanded path at that point in the hierarchical name, not just a single level of hierarchy. A wildcard “*” in a call to get_config will only match a corresponding wildcard in a call to set_config: it will not match a more specific name.
- When `set_config` is called, there is no obligation for a corresponding field to have been registered, but if it has been then the type should match (that is, `int`, `string`, or `object`), and the value of the field will be overwritten.
- `print_config_settings` does not give any indication about whether the configuration has "successfully" set the value of a component member.

See also

ovm_component, ovm_factory, Field Macros, ovm_root

ovm_driver

The `ovm_driver` class is derived from `ovm_component`. User-defined drivers should be built using classes derived from `ovm_driver`. A driver is typically used as part of an agent (*see ovm_agent*) where it will pull transactions from a sequencer and implement the necessary BFM-like functionality to drive those transactions onto a physical interface.

A sequence item pull port that may be connected to a corresponding export on a sequencer was added in OVM 2.0. This is backwards-compatible with the deprecated sequence interface (`seq_item_prod_if`) used in earlier versions.

Declaration

```
class ovm_driver #(type REQ = ovm_sequence_item,
                   type RSP = REQ) extends ovm_component;
```

Methods

`function new( string name, ovm_component parent = null);`	Constructor, mirrors the superclass constructor in `ovm_component`

Members

`ovm_seq_item_pull_port #(REQ,RSP) seq_item_port;`	Port for connecting the driver to the sequence item export of a sequencer
`ovm_analysis_port #(RSP) rsp_port;`	Analysis port for responses
`REQ req;`	Handle for request
`RSP rsp;`	Handle for response

ovm_seq_item_pull_port

`function new( string name, ovm_component parent, int min_size=0, int max_size=1);`	Constructor. Default minimum size of 0 makes connection to sequencer optional
`task get_next_item( output REQ req_arg);`	Blocks until item is returned from sequencer. There must be a subsequent call to `item_done`

`task` **`try_next_item`**`( output REQ req_arg);`	Attempts to fetch item. If item is available, returns immediately and there must be a subsequent call to `item_done`. Otherwise `req_arg` set to `null`
`function void` **`item_done`**`( RSP rsp_arg = null);`	Indicates to the sequencer that the driver has processed the item and clears the item from the sequencer fifo. Optionally also sends response
`task` **`wait_for_sequences`**`();`	Calls connected sequencer's `wait_for_sequences` task (by default waits #100)
`function bit` **`has_do_available`**`();`	Returns 1 if item available, otherwise 0
`task` **`get`**`(output REQ req_arg);`	Blocks until item is returned from sequencer. Call `item_done` before returning.
`task` **`peek`**`(output REQ req_arg);`	Blocks until item is returned from sequencer. Does not remove item from sequencer fifo
`task` **`put`**`(RSP rsp_arg);`	Sends response back to sequencer

Example

```
class example_driver extends ovm_driver #(my_transaction);
  ...
  virtual task run();
    forever begin
      seq_item_port.get_next_item(req);
      // code to generate physical signal activity
      // as specified by transaction data in req
      seq_item_port.item_done();
    end
  endtask

  `ovm_component_utils_begin(example_driver)
  `ovm_component_utils_end

endclass: example_driver
```

Tips

- The driver's physical connection is usually specified by means of a virtual interface wrapper object. This object can be configured using the `set_config` mechanism, or can be passed into the driver by its enclosing agent.
- If a driver sends a response back to a sequencer, the sequence ID and transaction ID of the response must match those of the request. These can be set by calling `rsp.set_id_info(req)` before calling `item_done`.

Gotchas

- Do not forget to call the sequence item pull port's `item_done` method when your code has finished consuming the transaction item.

See also

ovm_agent, Virtual Interface Wrapper, Sequencer Interface and Ports, ovm_sequencer

A class derived from `ovm_env` should be used to model and control the test environment (testbench), but does not include the tests themselves. An environment may instantiate other environments to form a hierarchy. The leaf environments will include all the main methodology components: stimulus generator; driver; monitor and scoreboard. An environment is connected to the device under test (DUT) through a virtual interface. The top-level environment should be instantiated and configured in a (top-level) test.

Prior to OVM 1.1, simulation was started by calling the top-level test's `run_test` method and controlled by its `run` method. From OVM 1.1 onwards, the `run_test` method has been moved to `ovm_root`. The "`do_test`" mode is deprecated and its associated methods are not shown. (See OVM Class Reference)

Declaration

```
virtual class ovm_env extends ovm_component;
```

Methods

`function` **`new`**`(`
`  string name="env",`
`  ovm_component parent=null);`
	Constructor.

`static task` **`run_test`**`(` `  string test_name="");`	Runs all simulation phases for all components in the environment (deprecated).
Also, inherited methods, including *`build, connect, run`*	

Members

Only inherited members	

Example

This is a minimal environment:

```
class verif_env extends ovm_env;
  `ovm_component_utils(verif_env)

  // Testbench methodology components
  ...

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

  function void build();
    // Instantiate top-level components using "new" or
    // the factory, as appropriate
    ...
  endfunction: build

  virtual function void connect();
    // Connect ports-to-exports
    ...
  endfunction: connect

  virtual task run();
    // Control stimulus generation
    ...
    // Control simulation run length
    ...
    ovm_top.stop_request();
  endtask: run

endclass: verif_env
```

Tips

- The `new`, `build`, `connect` and `run` methods should be overridden.
- Control simulation using the `run` method. Call `ovm_top.stop_request` in the `run` method to stop simulation. Alternatively, set `ovm_top.phase_timeout` before the start of simulation, e.g. in the `start_of_simulation` method.

- You would not normally use `do_global_phase`. Instead, the phases are run by calling `run_test`.
- For maximum flexibility, use the command-line to set the test name, rather than passing it as an argument to `run_test`.
- Instantiate one top-level environment in a (top-level) test. This may in turn instantiate other (lower-level) environments.

Gotchas

- If no test is specified, no new components are created – so simulation proceeds with an empty environment, and nothing much happens!
- `new` and `build` should call the base class `new` (`super.new`) and `build` (`super.build`) methods respectively.
- Do not forget to register the environment with the factory, using `` `ovm_component_utils ``.
- Do not call the `set_inst_override` member function (inherited from `ovm_component`) for a top-level environment.

See also

ovm_test, Configuration. Virtual Interface Wrapper

ovm_event

Class `ovm_event` is an `ovm_object` that adds additional features to standard SystemVerilog events. These features include the ability to store named events in an `ovm_event_pool`, to store data when triggered and to register callbacks with particular events. When an event is triggered, it remains in that state until explicitly reset. `ovm_event` keeps track of the number of processes that are waiting for it.

Several built-in OVM classes make use of `ovm_event`. However, they should only be used in applications where their additional features are required due to the simulation overhead compared to plain SystemVerilog events.

Declaration

```
class ovm_event extends ovm_object;
```

Methods

`function new(string name="");`	Constructor
`virtual task wait_on( bit delta=0);`	Waits until event triggered. If already triggered, returns immediately (or after #0)
`virtual task wait_off( bit delta=0);`	Waits until event reset. If not triggered, returns immediately (or after #0)
`virtual task wait_trigger();`	Like Verilog `@event`
`virtual task wait_ptrigger();`	Like `wait_trigger` but returns immediately if triggered in current time-step
`virtual task wait_trigger_data( output ovm_object data);`	Calls `wait_trigger`. Returns event data
`virtual task wait_ptrigger_data( output ovm_object data);`	Calls `wait_ptrigger`. Returns event data
`virtual function void trigger( ovm_object data=null);`	Triggers event and sets event data
`virtual function ovm_object get_trigger_data();`	Returns event data
`virtual function time get_trigger_time();`	Time that event was triggered
`virtual function bit is_on();`	True if triggered
`virtual function bit is_off();`	True if not triggered

`virtual function void` **`reset`**`(` `  bit wakeup=0);`	Resets event and clears data. If wakeup bit is set, any process waiting for trigger resumes

`virtual function void`
`add_callback``(`
`  ovm_event_callback cb,`
`  bit append=1);`
	Add callback (class with `pre_trigger` and `post_trigger` function). Adds to end of list by default

`virtual function void`
`delete_callback``(`
`  ovm_event_callback cb);`
	Removes callback

`virtual function void` **`cancel`**`();`	Decrements count of waiting processes by 1
`virtual function int` **`get_num_waiters`**`();`	Number of waiting processes

Example

Using event to synchronize two tasks and send data

```
class C extends ovm_component;
  ovm_event e1;
  function new (string name="", ovm_component parent=null);
   super.new(name,parent);
  endfunction : new

  function void build();
    super.build();
    e1 = new ("e1");
  endfunction: build

  task run();
    basic_transaction tx,rx;
    tx = new();
    fork
      begin
        tx.data = 10;
        tx.addr = 1;
        #10 e1.trigger(tx);
      end
      begin
        e1.wait_ptrigger();
        $cast(rx,e1.get_trigger_data());
        rx.print();
      end
```

```
    join
  endtask: run

  `ovm_component_utils(C)
endclass: C
```

Creating a callback

```
class my_e_callback extends ovm_event_callback;
  function new (string name="");
    super.new(name);
  endfunction : new

  function void post_trigger(ovm_event e,
                             ovm_object data=null);
    basic_transaction rx;
    if (data) begin
      $cast(rx,data);
      ovm_report_info("CBACK",$psprintf("Received %s",
                                rx.convert2string()));
    end
  endfunction: post_trigger
endclass: my_e_callback
```

To use the callback, create an instance of the callback class and register it with the event

```
my_e_callback cb1;
...
cb1 = new ("cb1");  //in build
...
e1.add_callback(cb1);  //in run
```

Tips

Use `wait_ptrigger` rather than `wait_trigger` to avoid race conditions

Gotchas

- An `ovm_object` handle must be used to hold the data returned by `wait_trigger_data` and `wait_ptrigger_data`. This must be explicitly cast to the actual data type before the data can be accessed. Use `wait_(p)trigger` and `get_trigger_data` instead since the return value of `get_trigger_data` can be passed directly to `$cast`.

Class `ovm_event_pool` is an object that behaves like an associative array of named events. A global `ovm_event_pool` exists that may be used to synchronize processes running across multiple components or modules. It is also possible to create a local event pool if required.

Declaration

```
class ovm_event_pool extends ovm_object;
```

Methods

`function new(string name="");`	Constructor
`function ovm_object create( string name="");`	Convenience function to create an event pool
`static function ovm_event_pool get_global_pool();`	Returns a handle to the global even pool
`virtual function ovm_event get( string name);`	Returns the named event (if not found, creates it and adds to pool)
`virtual function int num();`	Number of events in pool
`virtual function void delete( string name);`	Deletes named event
`virtual function int exists( string name);`	Returns 1 if named event is in pool, otherwise 0
`virtual function int first( ref string name);`	Gets name of first event in pool
`virtual function int last( ref string name);`	Gets name of last event in pool
`virtual function int next( ref string name);`	Gets name of event in pool after specified name
`virtual function int prev( ref string name);`	Gets name of event in pool before specified name

Example

A stimulus task that waits for the end of each transaction using its event pool

```
virtual task generate_stimulus(basic_transaction t = null,
                               input int max_count = 30 );
  basic_transaction temp;
  ovm_event_pool tx_epool;
  ovm_event tx_end;
  if( t == null ) t = new("trans",this);
  for( int i = 0;
     (max_count == 0 || i < max_count-1);
     i++ ) begin
    assert( t.randomize() );
    $cast( temp , t.clone() );
    //get handle to transaction's event pool
    tx_epool = temp.get_event_pool();
    blocking_put_port.put( temp );
    //get transaction's "end" event
    tx_end = tx_epool.get("end");
    tx_end.wait_trigger();
  end
endtask: generate_stimulus
```

Tips

- The contents of an `ovm_event_pool` may be displayed by calling `print`

Gotchas

- You need to call `get` to add an event to the event pool.

See also

ovm_event

The OVM factory is provided as a fully configurable mechanism to create objects from classes derived from `ovm_object` (sequences and transactions) and `ovm_component` (testbench components).

The benefit of using the factory rather than constructors (`new`) is that the actual class types that are used to build the test environment are determined at run-time (during the build phase). This makes it possible to write tests that modify the test environment, without having to edit the test environment code directly.

Classes derived from `ovm_object` and `ovm_component` can be substituted with alternative types using the factory override methods. The substitution is made when the component or object is built. The substitution mechanism only makes sense if the substitute is an extended class of the original type. Both the original type and its replacement must have been registered with the factory using one of the utility macros `` `ovm_component_utils ``, `` `ovm_component_param_utils ``, `` `ovm_object_utils `` or `` `ovm_object_param_utils ``.

The factory keeps tables of overrides in component and object registries (`ovm_component_registry` and `ovm_object_registry`). To help with debugging, the factory provides methods that print the information in these registries.

Prior to OVM 2.0, the factory methods used strings to specify the type of component to create (or the overrides). OVM 2.0 introduced a new mechanism based around a proxy class to specify object or component types with a new user interface. The type-based methods enable the compiler to detect type name errors and also support parameterized objets and components. The original string-based methods have been replaced by new methods with more descriptive names and are not shown here (they have been deprecated).

A singleton instance of `ovm_factory` named `factory` is instantiated within the OVM package (`ovm_pkg`). The `factory` object can therefore be accessed from SystemVerilog modules and classes, as well as from within an OVM environment.

Declaration

```
class ovm_factory extends ovm_component;
```

Methods

`function ovm_object`
`create_object_by_type``(`
`  ovm_object_wrapper requested_type,`
`  string parent_inst_path="",`
`  string name="");`
	Creates and returns an object. Type is set by proxy. Name and parent specified by strings

`function ovm_component` **`create_component_by_type`**`(`
`  ovm_object_wrapper requested_type,`
`  string parent_inst_path="",`
`  string name,`
`  ovm_component parent);`
	Creates and returns a component. Type is set by proxy. Name and parent specified by strings

`function ovm_object` **`create_object_by_name`**`(`
`  string requested_type_name,`
`  string parent_inst_path="",`
`  string name="");`
	Creates and returns an object. Type, name and parent are specified by strings

`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. Type, name and parent are specified by strings

`function void` **`set_inst_override_by_type`**`(`
`  ovm_object_wrapper original_type,`
`  ovm_object_wrapper override_type,`
`  string full_inst_path);`
	Register an instance override with the factory based on proxies (see below)

`function void` **`set_inst_override_by_name`**`(`
`  string original_type_name,`
`  string override_type_name,`
`  string full_inst_path);`
	Register an instance override with the factory based on type names (see below)

`function void` **`set_type_override_by_type`**`(`
`  ovm_object_wrapper original_type,`
`  ovm_object_wrapper override_type,`
`  bit replace=1);`
	Register a type override with the factory based on proxies (see below)

`function void` **`set_type_override_by_name`**`(`
`  string original_type_name,`
`  string override_type_name,`
`  bit replace=1);`
	Register a type override with the factory based on type names (see below)

`function ovm_object_wrapper` **`find_override_by_type`**`(`
`  ovm_object_wrapper requested_type,`
`  string full_inst_path);`
	Return the proxy to the object that would be created for the given override

`function ovm_object_wrapper` **`find_override_by_name`**`(`
`  string requested_type_name,`
`  string full_inst_path);`
	Return the proxy to the object that would be created for the given override

`function void register( ovm_object_wrapper obj);`	Registers a proxy with the factory (called by utility macros)
`function void debug_create_by_type( ovm_object_wrapper requested_type, string parent_inst_path="", string name="");`	Prints information about the type of object that would be created with the given proxy, parent and name
`function void debug_create_by_name( string requested_type_name, string parent_inst_path="", string name="");`	Prints information about the type of object that would be created with the given type name, parent and name
`function void print( int all_types=1);`	`all_types` is 0: Prints the factory overrides `all_types` is 1: Prints the factory overrides + registered types `all_types` is 2: Prints the factory overrides + registered types (including OVM types)

Registration

Components and objects are generally registered with the factory using the macros `` `ovm_component_utils `` and `` `ovm_object_utils `` respectively. Parameterized components and objects should use `` `ovm_component_param_utils `` and `` `ovm_object_param_utils `` respectively. Registration using these macros creates a specialization of the `ovm_component_registry #(T,Tname)` (for components) or `ovm_object_registry #(T,Tname)` (for objects) and adds it as a nested class to the component or object named `type_id`. If you try to create a component or object using a type that has not been registered, nothing is created, and the value `null` is returned.

Overriding instances and types

You can configure the factory so that the type of component or object that it creates is not the type specified by the proxy or string argument. Instead, if a matching instance or type override is in place, the override type is used.

The `set_inst_override_by_*` function requires a string argument that specifies the path name of the object or component to be substituted (wildcards "*" and "?" can be used), together with the original type and the replacement type (strings or proxies). A warning is issued if the types have not been registered with the factory. The `ovm_component` class also has a set_inst_override member function that calls the factory method – this adds its hierarchical name to the search path so should NOT be used for components with no parent (e.g. top-level environments or tests).

Creating Components and Objects

The create_object_by_* and create_component_by_* member functions of ovm_factory can be called from modules or classes using the factory instance. The type of object/component to build is requested by passing a proxy or string argument. The instance name and path are specified by string arguments. The path argument is used when searching the configuration table for path-specific overrides. The create_component_by_* functions require a 4th argument: a handle to their parent component. This is not required when objects are created.

The ovm_component class provides create_object and create_component member functions that only require two string arguments – the type name and instance name of the object being created. These can only be called for (or more likely, within) an existing component. The path is taken from the component that the function is called for/within. The ovm_component::create component function always sets the parent of the created component to this.

The create_object_by_* and create_object functions always return a handle to the new object using the virtual ovm_object base class. The create_component_by_* and create_component functions always return a handle to the new component using the virtual ovm_component base class. A $cast of the returned handle is therefore usually necessary to assign it to a handle of a derived object or component class.

The easiest way of creating objects and components with the factory is to make use of the create function provided by the proxy. This function has three arguments: a name (string), a parent component handle and an optional path string. The name and parent handle are optional when creating objects.

When you create an object or component, the factory looks for an instance override, and if there is none, a type override. If an override is found, a component or object of that type is created. Otherwise, the requested type is used.

Examples

```
class verif_env extends ovm_env;
  // Register the environment with the factory
  `ovm_component_utils (verif_env)
  ...
  instruction m_template;
  ...
  function void build();
```

```
    super.build();
    ...
    // Use the factory to create the m_template object
    m_template = instruction::type_id::create("m_template",
                                              this);
  endfunction : build
endclass: verif_env

class test1 extends ovm_test;
  verif_env env1;
  ...
  function void build();
    ...
    // Change type of m_template from instruction  to register_instruction
    //  using factory method
    factory.set_inst_override_by_name("instruction",
           "register_instruction","*env?.m_template");

     // Type overrides have lower precedence than inst overrides
     factory.set_type_override_by_type(
                      instruction::get_type(),
                      same_regs_instruction::get_type() );

     // Print all factory overrides and registered classes
     factory.print();

     // Call factory method to create top-level environment (requires cast so
     // type_id::create is generally preferred)
     $cast(env1,factory.create_component_by_type(
           verif_env::get_type(),"","env1",null) );
  endfunction : build

  ...
endclass : test1
```

Tips

- Do not create an `ovm_factory` object or derive a class from `ovm_factory`. A factory is created automatically. It is a singleton – there is only one instance named `factory`.
- Use the factory to create components and objects whenever possible. This makes the test environment more flexible and reusable.

- Prefer the type-based (proxy) functions to the string-based (type name) functions
- For convenience, use the create function from an object or component proxy – it requires less arguments, is more likely to pick up errors in class names at compile time and returns a handle of the correct type. If this is not possible, then prefer a component's own `create_component` or `create_object` method to `ovm_factory::create_component_by_name` or `ovm_factory::create_object_by_name` respectively.
- With `create_component_by_name` and `create_object_by_name` use the same name for the instance that you used for the instance (handle) variable.

Gotchas

- Do not forget to register all components and objects with the factory, using `` `ovm_component_utils ``, `` `ovm_component_param_utils ``, `` `ovm_object_utils `` or `` `ovm_object_param_utils `` as appropriate.
- Errors in type names passed as strings to the factory create and override methods may not be detected until run time (or not at all)!
- If you use the `create_component_by_*` methods remember to use `$cast`, because the return type is `ovm_component`: you will probably want to assign the function to a class derived from `ovm_component`, which is a virtual class.

See also

Field Macros, Configuration, Sequences, ovm_component, ovm_component_registry, ovm_object, Utility macros

Fields are the data members or properties of OVM classes. The field macros automate the provision of a number of *data methods*:

- `copy`
- `compare`
- `pack`
- `unpack`
- `record`
- `print`
- `sprint`

There are field automation macros for integer types (any packed integral type), enum types, strings and objects (classes derived from `ovm_object`). These macros are placed inside of the `` `ovm_*_utils_begin `` and `'ovm_*_utils_end` macro blocks.

The field macros enable automatic initialization of fields during the *build* phase. The initial values may be set using OVM's configuration mechanism (`set_config_*`) from the top level of the testbench. (Fields that have not been automated may still be configured manually, using the `get_config_*` functions.)

Field Macros

Macro	Declares a field for this type:
`` `ovm_field_int `` `(ARG, FLAG)`	Any packed integral type
`` `ovm_field_enum `` `( TYPE, ARG, FLAG)`	Enum of `TYPE`
`` `ovm_field_object `` `(ARG, FLAG)`	`ovm_object`
`` `ovm_field_object `` `(ARG, FLAG)`	event
`` `ovm_field_string `` `(ARG, FLAG)`	string
`` `ovm_field_sarray_int `` `( ARG, FLAG)`	(Fixed-size) array of packed integral type
`` `ovm_field_array_int `` `( ARG, FLAG)`	Dynamic array of packed integral type
`` `ovm_field_sarray_object `` `( ARG, FLAG)`	(Fixed-size) array of `ovm_object`
`` `ovm_field_array_object `` `( ARG, FLAG)`	Dynamic array of `ovm_object`
`` `ovm_field_sarray_string `` `( ARG, FLAG)`	(Fixed-size) array of string

`ovm_field_array_string (ARG, FLAG)	Dynamic array of string
`ovm_field_queue_int (ARG, FLAG)	Queue of packed integral type
`ovm_field_queue_object (ARG, FLAG)	Queue of `ovm_object`
`ovm_field_queue_string (ARG, FLAG)	Queue of string
`ovm_field_aa_int_string (ARG, FLAG)	Associative array of integral type with string keys
`ovm_field_aa_object_string (ARG, FLAG)	Associative array of `ovm_object` with string keys
`ovm_field_aa_string_string (ARG, FLAG)	Associative array of string with string keys
`ovm_field_aa_int_<key_type> (ARG, FLAG)	Associative array of *key_type* (an integer type: int, integer, byte, …) with integral keys
`ovm_field_aa_string_int (ARG, FLAG)	Associative array of string with int keys
`ovm_field_aa_object_int (ARG, FLAG)	Associative array of objects with int keys

Flags

The FLAG argument is specified to indicate which, if any, of the data methods (`copy`, `compare`, `pack`, `unpack`, `record`, `print`, `sprint`) NOT to implement. Flags can be combined using bitwise-or or addition operators.

OVM_DEFAULT	Use the default settings.
OVM_ALL_ON	All flags are on (default)
OVM_COPY, OVM_NOCOPY	Do/Do not do a copy
OVM_COMPARE, OVM_NOCOMPARE	Do/Do not do a compare
OVM_PRINT, OVM_NOPRINT	Do/Do not print
OVM_NODEFPRINT	Do not print if the field is the same as its default value.
OVM_PACK, OVM_NOPACK	Do/Do not pack/unpack.
OVM_PHYSICAL	Treat as a physical field.
OVM_ABSTRACT	Treat as an abstract field.

OVM_READONLY	Do not allow this field to be set using set_config_*
OVM_BIN, OVM_DEC, OVM_UNSIGNED, OVM_OCT, OVM_HEX, OVM_STRING, OVM_TIME, OVM_NORADIX	Radix settings (integral types only). The default is *OVM_HEX*.

Examples

```
class basic_transaction extends ovm_sequence_item;
  rand bit[7:0] addr, data;
  ...
  `ovm_object_utils_begin(basic_transaction)
    `ovm_field_int(addr,OVM_ALL_ON)
    `ovm_field_int(data,OVM_ALL_ON | OVM_BIN)
  `ovm_object_utils_end
endclass : basic_transaction
```

Tips

- Call field macro for every member of a transaction class or sequence.
- Declare as fields any data members that require configuration using set_config_*, for example an instance of a virtual interface wrapper class.
- Mark as “readonly” (`OVM_READONLY`) fields that you do not want to be affected by configuration.

Gotchas

- If you use + instead of bitwise-or to combine flags, make sure that the same bit is not added more than once.
- The macro FLAG argument is required (macro arguments cannot have defaults). Typically, use *OVM_ALL_ON*.

See also

Configuration

ovm_in_order_*_comparator

The `ovm_in_order_*_comparator` family of components can be used to compare two streams of transactions in an OVM environment. They each provide a pair of analysis exports that act as a subscribers to the transaction streams (the streams typically originate from analysis ports on OVM monitors and drivers). The transactions may be built-in types (e.g. int, enumerations, structs) or classes: you should use `ovm_in_order_class_comparator` to compare class objects, and `ovm_in_order_built_in_comparator` to compare objects of built-in type. In each case the type is set by a parameter. Both versions are derived from a parent class `ovm_in_order_comparator`; this underlying class is not normally appropriate in user code, and is not described here.

The incoming transactions are held in FIFO buffers and compared in order of arrival. Individual transactions may therefore arrive at different times and still be matched successfully. A count of matches and mismatches is maintained by the comparator. Each pair of transactions that has been compared is written to an analysis port in the form of a pair object – a pair is a simple class that contains just the two transactions as its data members.

Declarations

```
class ovm_in_order_class_comparator #( type T = int )
  extends ovm_in_order_comparator #(T, ...);

class ovm_in_order_built_in_comparator #( type T = int )
  extends ovm_in_order_comparator #(T, ...);
```

Methods

`function` **`new`**`( string name , ovm_component parent ) ;`	Constructor
`function void` **`flush`**`();`	Clears (mis)matches counts

Members

`ovm_analysis_export #( T )` **`before_export`**`;`	Connect to first transaction stream analysis port, typically monitored from a DUT's input
`ovm_analysis_export #( T )` **`after_export`**`;`	Connect to second transaction stream analysis port, typically monitored from a DUT's output
`ovm_analysis_port #(pair_type)` **`pair_ap`**`;`	Pair of matched transactions that has been compared
`int` **`m_matches`**`;`	Number of matches
`int` **`m_mismatches`**`;`	Number of mismatches

Example

Using `ovm_in_order_class_comparator` within a scoreboard component

```
class cpu_scoreboard extends ovm_scoreboard;

  ovm_analysis_export #(exec_xact) af_iss_export;
  ovm_analysis_export #(exec_xact) af_cpu_export;
  ovm_in_order_class_comparator #(exec_xact) m_comp;

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

  virtual function void build();
    super.build();
    af_iss_export = new("af_iss_export", this);
    af_cpu_export = new("af_cpu_export", this);
    m_comp        = new("comp",          this);
  endfunction: build

  virtual function void connect();
    af_iss_export.connect( m_comp.before_export );
    af_cpu_export.connect( m_comp.after_export  );
  endfunction: connect

  integer m_log_file;
  virtual function void start_of_simulation();
    m_log_file = $fopen("cpu_comparator_log.txt");
    set_report_id_action_hier("Comparator Match",LOG);
    set_report_id_file_hier  ("Comparator Match",
                               m_log_file);
    set_report_id_action_hier("Comparator Mismatch",LOG);
    set_report_id_file_hier  ("Comparator Mismatch",
                               m_log_file);
  endfunction: start_of_simulation

  virtual function void report();
    string txt;
    $sformat(txt, "#matches = %d, #mismatches = %d",
             m_comp.m_matches, m_comp.m_mismatches);
    ovm_report_info("", txt);
  endfunction: report

  `ovm_component_utils(cpu_scoreboard)
```

```
endclass: cpu_scoreboard
```

Tips

- The comparator writes a message for each match and mismatch that it finds. These messages are of class “Comparator Match” and “Comparator Mismatch” respectively. You may wish to disable these messages or redirect them to a log file as shown in the example.
- If you need your comparator also to model the transformation that a DUT applies to its data, you may find `ovm_algorithmic_comparator` more appropriate – it allows you to incorporate a reference model of the DUT's data-transformation behavior in a convenient way.
- Consider using `ovm_in_order_class_comparator` as a base class for a type-specific comparator that can be built by the factory, for example

```
class exec_comp extends
       ovm_in_order_class_comparator #(exec_xact);
  `ovm_component_utils(exec_comp)
  function new(string name, ovm_component parent);
    super.new(name,parent);
  endfunction: new
endclass: exec_comp
```

Gotchas

- `ovm_in_order_class_comparator` requires transaction classes to have a `comp` member function (this is not defined by the field automation macros). Its signature is:

```
bit comp( input exec_xact t );
```

  This can be easily implemented by calling `compare` (which is created for you by the field automation macros).
- `ovm_in_order_class_comparator` does not have a `type_id` proxy so cannot be used directly with the factory (see above for a work-around).

See also

ovm_analysis_port, ovm_analysis_export, ovm_transaction, ovm_algorithmic_comparator

ovm_monitor

The `ovm_monitor` class is derived from `ovm_component`. User-defined monitors should be built using classes derived from `ovm_monitor`. A monitor is typically used to detect transactions on a physical interface, and to make those transactions available to other parts of the testbench through an analysis port.

Declaration

```
class ovm_monitor extends ovm_component;
```

Methods

`function` **`new`** `( string name, ovm_component parent = null);`	Constructor, mirrors the superclass constructor in `ovm_component`

Members

`ovm_analysis_port #(`*`transaction_class_type`*`)` **`monitor_ap`**`;`	Analysis port through which monitored transactions are delivered to other parts of the testbench. ***Note***: this field is not defined in `ovm_monitor`, but should always be provided as part of any user extensions.

Example

```
class example_monitor extends ovm_monitor;
  ovm_analysis_port #(example_transaction) monitor_ap;
  example_virtual_if_wrapper vi_wrapper;

  virtual function void build();
    super.build();
  endfunction: build
  ..
  virtual task run();
    example_transaction tr;
    forever begin
      // Start with a new, clean transaction so that
      // already-monitored transactions are unaffected
      tr = new;
      // code to observe physical signal activity
      // and assemble transaction data in tr
      monitor_ap.write(tr);
```

```
    end
  endtask

  `ovm_component_utils_begin(example_monitor)
    `ovm_field_utils(vi_wrapper)
  `ovm_component_utils_end

endclass: example_monitor
```

Tips

- A monitor can be useful "stand-alone", observing activity on a set of signals so that the rest of the testbench can see that activity in the form of complete transaction objects. Alternatively it can form part of an *agent*.
- By using an analysis port to pass its output to the rest of the testbench, a monitor can guarantee that it can deliver this output data without consuming time. Consequently, the monitor's `run` method can immediately begin work on receiving the next transaction on its physical interface.
- The monitor's physical connection is specified by means of a virtual interface wrapper object. This object can be configured using the `set_config` mechanism, or can be passed into the monitor by its enclosing agent.

Gotchas

`ovm_monitor` has no methods or data members of its own, apart from its constructor and what it inherits from `ovm_component`. However, building a properly-formed monitor usually requires additional methodology guidelines, including the recommendations in this article.

See also

ovm_agent, Virtual Interface Wrapper

`ovm_object` is the virtual base class for all components and transactions in an OVM environment. It has a minimal memory footprint with only one dynamic member variable – a string that is used to name instances of derived classes and which is usually left uninitialized for data objects.

Declaration

```
virtual class ovm_object extends ovm_void;
```

Methods

`function new(string name="");`	Constructor
`virtual function string get_name();`	Returns the name
`virtual function string get_full_name();`	Returns the full hierarchical path name
`virtual function void set_name( string name);`	Sets the name
`static function int get_inst_count();`	Returns running total count of number of `ovm_object`-based objects created
`virtual function int get_inst_id();`	Returns unique ID for object (count value when object created)
`static function ovm_object_wrapper get_type();`	Returns the type proxy for this class (overriden by utils macro)
`virtual function string get_type_name();`	Returns type name. Override unless utility macros called
`virtual function ovm_object create(string name="");`	Creates a new object. Override unless utility macros called
`virtual function ovm_object clone();`	Creates a copy of the object
`function void copy( ovm_object rhs);`	Copies rhs to this
`function bit compare( ovm_object rhs, ovm_comparer comparer=null);`	Comparison against rhs
`function void print( ovm_printer printer=null);`	Prints the object

`function string` **`sprint`**`(` `  ovm_printer printer=null);`	Prints the object to a string
`function void` **`record`**`(` `  ovm_recorder recorder=null);`	Used for transaction recording

`function int` **`pack`**`(`
`  ref bit bitstream[],`
`  input ovm_packer packer=null);`
	Packs object to array of bits. Returns number of bits packed.

`function int` **`pack_ints`**`(`
`  ref int unsigned intstream[],`
`  input ovm_packer packer=null);`
	Packs object to array of ints. Returns number of ints packed.

`function int` **`pack_bytes`**`(`
`  ref byte unsigned bytestream[],`
`  input ovm_packer packer=null);`
	Packs object to array of bytes. Returns number of bytes packed.

`function int unpack (`
`  ref bit bitstream[],`
`  input ovm_packer packer=null);`
	Unpacks array of bits to object

`function int` **`unpack_ints`**`(`
`  ref int unsigned intstream[],`
`  input ovm_packer packer=null);`
	Unpacks array of ints to object

`function int` **`unpack_bytes`**`(`
`  ref byte unsigned bytestream[],`
`  input ovm_packer packer=null);`
	Unpacks array of bytes to object

`function void` **`reseed`**`();`	Set seed based on object type and name if `use_ovm_seeding` = 1
`virtual function void` **`do_print`**`(` `  ovm_printer printer);`	Override for custom printing (called by print)
`virtual function string` **`do_sprint`**`(ovm_printer printer);`	Override for custom printing (called by sprint)
`virtual function void` **`do_record`**`(` `  ovm_recorder recorder);`	Override for custom reporting (called by report)
`virtual function void` **`do_copy`**`(` `  ovm_object  rhs);`	Override for custom copying (called by copy)

`virtual function bit` **`do_compare`**`(`
`  ovm_object rhs,`
`  ovm_comparer comparer);`
	Override for custom compare (called by compare)

`virtual function void` **`do_pack`**`(` `  ovm_packer packer);`	Override for custom packing (called by pack)
`virtual function void` **`do_unpack`**`(` `  ovm_packer packer);`	Override for custom unpacking (called by unpack)

Members

`static bit use_ovm_seeding = 1;`	Enables the OVM seeding mechanism (based on type and hierarchical name)

Macros

The utility macros generate overridden `get_type_name()` and `create()` functions for derived object classes.

```
`ovm_object_utils(TYPE)
```

or

```
`ovm_object_utils_begin(TYPE)
  `ovm_field_*(ARG,FLAG)
  ...
`ovm_object_utils_end
```

Use `` `ovm_object_param_utils(TYPE#(T)) `` or

`` `ovm_object_param_utils_begin(TYPE#(T)) `` for parameterized objects.

Fields specified in field automation macros will automatically be handled correctly in `copy()`, `compare()`, `pack()`, `unpack()`, `record()`, `print()` and `sprint()` functions.

Example

Using `ovm_object` to create a wrapper around a virtual interface

```
class if_wrapper extends ovm_object;
  virtual chip_if if1;
  function new(string name,virtual chip_if if_);
    super.new(name);
    if1 = if_;
  endfunction : new

`ovm_object_utils(if_wrapper)
endclass : if_wrapper
```

Tips

- Objects that need to be configured automatically at run-time using OVM configurations should use `ovm_component` as their base class instead.

- Call the utility macros in derived classes to ensure the `get_type_name` and `create` functions are automatically generated. This will also enable these classes to be used with the OVM factory.

See also

ovm_factory, ovm_printer

The `ovm_object_registry` class is used to register objects with the factory. It acts as a "proxy" which allows an object to be registered with the factory before any instance of the object has actually been created. It enables the factory to support parameterized objects since each "specialization" has a unique corresponding proxy that is registered with the factory.

The proxy instance is a specialization of the registry class created automatically by the object utility macros as a singleton instance of a nested class named `type_id`.

Calling the static `create` member function of an object's `type_id` nested class is the simplest way to instantiate objects with the factory. It returns a handle of the correct type so no type casts are necessary.

Declaration

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

Methods

`function ovm_object` `create_object(string name);`	Used by factory to create instance
`static function this_type get();`	Returns proxy instance
`function string get_type_name();`	Returns type name

`static function T create(`
`  string name="",`
`  ovm_component parent=null,`
`  string contxt="");`
	Called by user to create object instance with the factory

`static function void set_type_override(`
`  ovm_object_wrapper override_type,`
`  bit replace=1);`
	Overrides the type used by the factory for specified type

`static function void set_inst_override(`
`  ovm_object_wrapper override_type,`
`  string inst_path,`
`  ovm_component parent=null);`
	Overrides the type used by the factory for the specified instance (path is relative if parent specified)

Members

`const static string type_name = Tname;`	Type name string
`typedef ovm_object_registry #(T,Tname) this_type;`	Type of proxy (for internal use)

Example

```
class pktA extends ovm_transaction;
  rand  bit signed[7:0] A;
  function new(string name="", ovm_component parent=null);
    super.new(name,parent);
  endfunction: new
 `ovm_object_utils_begin(pktA)   // Creates nested registry class
   `ovm_field_int(A,OVM_DEC)
 `ovm_object_utils_end
endclass: pktA

class pktA_cons extends pktA;
  constraint lowA { A != 0; A > -10; A < 10; }
  function new(string name="", ovm_component parent=null);
    super.new(name,parent);
  endfunction: new
 `ovm_object_utils(pktA_cons)   // Creates nested registry class
endclass: pktA_cons

class pktgen extends ovm_component;
  ...
  task run();
    pktA p1;
    // Override pktA (pktA_cons::get_type() calls pktA_cons::type_id::get() )
    pktA::type_id::set_type_override(
                                 pktA_cons::get_type());
    // Create p1 using the factory
    p1 = pktA::type_id::create({get_full_name(),"_p1"});
    assert(p1.randomize());
    p1.print();
    ...
   endtask: run
endclass: pktgen
```

Tips

- Use the utility macro to create the registry class rather than declaring a typdef for it yourself. This ensures interoperability across simulators which may use different internal type names for the registry specialization.
- Use the `ovm_object_param_utils` macro for parameterized classes.
- Use the `get_type` function of objects and components to get the proxy instance for objects rather than calling the `get` function of `ovm_object_registry`

See also

ovm_object, ovm_object_wrapper, ovm_factor

ovm_object_wrapper

The proxy classes used by the factory to create objects and components of a particular type are derived from the virtual class `ovm_object_wrapper`. This class is used internally by the factory and various factory methods require arguments of type `ovm_object_wrapper`. The proxy classes override its virtual methods which by default do nothing. It is shown here for completeness: users do not usually need to derived classes from `ovm_object_wrapper` or call its methods explicitly.

Declaration

```
virtual class ovm_object_wrapper;
```

Methods

`virtual function ovm_object` **`create_object`**`(string name="");`	Called by the factory to create an object
`virtual function ovm_component` **`create_component`**`(string name, ovm_component parent);`	Called by the factory to create a component
`pure virtual function string` **`get_type_name`**`();`	Must be overriden in derived classes to return type name as a string

See also

ovm_component_registry, ovm_object_registry, ovm_factory

When a test is started by calling `run_test`, the simulation proceeds by calling a predefined sequence of functions and tasks in every component. Each step in this sequence is known as a *phase*. Phases provide a synchronization mechanism between activities in multiple components. During each phase, a corresponding callback function or task in each component in the hierarchy is invoked in either top-down or bottom-up order (depending on the phase). It is also possible for users to create custom phases which can be inserted into the standard OVM phase sequence.

All components implement a common set of phases.

Standard OVM Phases

Phase Name (in order of execution)	Callback Type	Order	Main Activity
build	function	top-down	Call factory to create child components
connect	function	bottom-up	Connect ports, exports and channels
end_of_elaboration	function	bottom-up	Check connections (hierarchy fixed)
start_of_simulation	function	bottom-up	Prepare for simulation (e.g. open files, load memories)
run	task	bottom-up	Run simulation until explicitly stopped or maximum time step reached
extract	function	bottom-up	Collect results
check	function	bottom-up	Check results
report	function	bottom-up	Issue reports

Additional phases for backwards compatibility with AVM and URM are deprecated and not listed here (see OVM Class Reference for details).

ovm_phase

Each phase is an object derived from class `ovm_phase`. Users do not usually need to know about this class. Macros exist to automate the definition of derived classes for user-defined phases and apply them to particular component classes.

Declaration

```
virtual class ovm_phase;
```

Methods

function **new**(string name, bit is_top_down,bit is_task);	Constructor
function string **get_name**();	Returns phase name
function bit **is_task**();	True if phase is a task
function bit **is_top_down**();	True if top-level component callback is executed first
virtual function string **get_type_name**();	Returns phase type (name appended with "_phase")
task **wait_start**();	Wait until phase begins
task **wait_done**();	Wait until phase completed
function bit **is_in_progress**();	True while phase running
function bit **is_done**();	True once phase completed
function void **reset**();	Reset status flags
virtual task **call_task**(ovm_component parent);	Override to invoke a task phase callback
virtual function void **call_func**(ovm_component parent);	Override to invoke a function phase callback

Macros

The macros listed below define a new function-based or task-based phase class: the `NAME` argument sets the phase name; `TOP_DOWN` is 1 or 0 and sets the order that the phase callbacks execute within the component hierarchy (1 = top downwards). The resulting class requires a single parameter that specifies the name of the component that includes the new phase. Its constructor does not take any arguments.

`ovm_phase_func_decl(NAME,TOP_DOWN)
`ovm_phase_task_decl(NAME,TOP_DOWN)

\`ovm_phase_func_topdown_decl(NAME)
\`ovm_phase_func_bottomup_decl(NAME)
\`ovm_phase_task_topdown_decl(NAME)
\`ovm_phase_task_bottomup_decl(NAME)

Example

Creating a function-based `my_post_run` phase for class `my_verif_env`

1. Add the `my_post_run` callback to the class

```
class my_verif_env extends ovm_env;
  ...
  virtual function void my_post_run();
    ovm_report_info("ENV","my_post_run");
  endfunction: my_post_run
endclass: my_verif_env
```

2. Use a macro to define the phase class and create a global instance of it

```
`ovm_phase_func_decl(my_post_run,1)
typedef class my_verif_env;

my_post_run_phase #(my_verif_env) my_post_run_ph = new();
```

3. Insert the phase into the sequence for `ovm_top` (after the `run` phase) in the environment's constructor or top module initial block

```
ovm_top.insert_phase(my_post_run_ph,
                     ovm_top.get_phase_by_name("run"));
```

Waiting until the end of the `build` phase in a module's initial block before finding and modifying a component

```
initial begin
  ovm_component c;
  ovm_phase build_ph;
  build_ph = ovm_top.get_phase_by_name("build");
  build_ph.wait_done();
  c = ovm_top.find("env2.m_driver");
  c.set_name("m_drv2");
```

Tips

- Do not create additional phases unless you really need them!
- Use the macros if you want to define your own phases

- You will need a forward class declaration (for example `typedef class myclass`) if you create a phase object before the class it applies to has been declared.
- If a phase needs to be added to multiple classes, create a common base class with a virtual callback function/task and use it as the parameter for the phase object

Gotchas

- Task-based phases can only be added to threaded components.
- Phases cannot be inserted once the first phase has started.
- Prior to OVM 1.1, ovm_root::insert_phase was a member of ovm_component and had different semantics.

See also

ovm_component, ovm_root

OVM *ports* and *exports* are classes that are associated with one of the OVM interfaces. Prior to OVM 2.0, ports and exports were only provided for implementations of the OSCI TLM 1.0 standard interfaces. OVM 2.0 added a port and an export for the `sqr_if_base` interface used by OVM 2.0 sequencers and sequence drivers.

Ports and exports are used as members of components and channels derived from `ovm_component`. They provide a mechanism to decouple the initiator and target of a transaction, providing encapsulation and improving the reusability. Interface methods can be called as member functions and tasks of a port. The implementations of the interface methods are not defined within the port class – the port passes on the function and task calls to another port or an export instead. Ports therefore "require" a connection to a remote implementation of the interface methods. Exports are classes that (directly or indirectly) "provide" the implementation to a remote port. An OVM *imp* is an export that contains the functional implementation of the interface methods. It is used to terminate a chain of connected ports and exports.

In addition to the methods required by a particular interface, all ports and exports have a common set of methods that are inherited from their `ovm_port_base` base class (see **ovm_port_base**).

The type of transaction carried by a port or export is set by a type parameter.

A port of a child component may be connected to one or more exports of other child components (usually at the same level in the hierarchy), or to one or more ports of its parent component. Within a component, each export that is not an imp may be connected to one or more exports of child components. The minimum and maximum number of interfaces that can be provided/required by a port/export is set by a constructor argument.

Port and export binding is achieved by calling the port's/export's `connect` function. The `connect` function takes a single argument: the port or export that provides the required interface: *`p_or_e_requires`*`.connect(`*`p_or_e_provides`*`)`. The order in which `connect` functions are called does not matter – bindings are resolved at the end of the connect phase.

A unidirectional TLM port can be connected to any unidirectional TLM export that has an identical transaction type parameter (since all TLM interfaces share a common `tlm_if_base` base class). However, a run-time error will be reported when an interface method required by the port is called if that method is not provided by the connected export.

The class name of a TLM port, export or imp is based on the name of the TLM interface it is related to, e.g. the `ovm_blocking_put_port` can call any of the methods that are members of the `tlm_blocking_put_if` interface. Every TLM interface has a corresponding port, export and imp. The complete set is listed below.

Blocking unidirectional interfaces

```
ovm_blocking_put_port           ovm_blocking_get_port
ovm_blocking_peek_port          ovm_blocking_get_peek_port

ovm_blocking_put_export         ovm_blocking_get_export
ovm_blocking_peek_export        ovm_blocking_get_peek_export

ovm_blocking_put_imp            ovm_blocking_get_imp
ovm_blocking_peek_imp           ovm_blocking_get_peek_imp
```

Non-blocking unidirectional interfaces

```
ovm_nonblocking_put_port        ovm_nonblocking_get_port
ovm_nonblocking_peek_port       ovm_nonblocking_get_peek_port

ovm_nonblocking_put_export      ovm_nonblocking_get_export
ovm_nonblocking_peek_export
ovm_nonblocking_get_peek_export

ovm_nonblocking_put_imp         ovm_nonblocking_get_imp
ovm_nonblocking_peek_imp        ovm_nonblocking_get_peek_imp
```

Combined Interfaces

```
ovm_put_port                    ovm_get_port
ovm_peek_port                   ovm_get_peek_port

ovm_put_export                  ovm_get_export
ovm_peek_export                 ovm_get_peek_export

ovm_put_imp                     ovm_get_imp
ovm_peek_imp                    ovm_get_peek_imp
```

Bidirectional Interfaces

```
ovm_blocking_master_port        ovm_nonblocking_master_port
ovm_master_port

ovm_blocking_master_export      ovm_nonblocking_master_export
ovm_master_export

ovm_blocking_master_imp         ovm_nonblocking_master_imp
ovm_master_imp

ovm_blocking_slave_port         ovm_nonblocking_slave_port
ovm_slave_port

ovm_blocking_slave_export       ovm_nonblocking_slave_export
ovm_slave_export

ovm_blocking_slave_imp          ovm_nonblocking_slave_imp
ovm_slave_imp

ovm_blocking_transport_port
ovm_nonblocking_transport_port
ovm_transport_port

ovm_blocking_transport_export
ovm_nonblocking_transport_export
ovm_transport_export

ovm_blocking_transport_imp
ovm_nonblocking_transport_imp
ovm_transport_imp
```

Sequencer interface

`ovm_seq_item_pull_port` `ovm_seq_item_pull_export`

`ovm_seq_item_pull_imp`

Declarations

```
class ovm_put_port #( type T = int )
extends ovm_port_base #( tlm_if_base #(T,T) );

class ovm_blocking_put_export #( type T = int )
extends ovm_port_base #( tlm_if_base #(T,T) );

class ovm_master_port #( type REQ = int , type RSP = int )
extends ovm_port_base #( tlm_if_base #(REQ, RSP) );

class ovm_put_imp #( type T = int , type IMP = int )
  extends ovm_port_base #( tlm_if_base #(T,T) );

class ovm_master_imp
  #( type REQ = int , type RSP = int ,type IMP = int ,
     type REQ_IMP = IMP , type RSP_IMP = IMP )
  extends ovm_port_base #( tlm_if_base #(REQ, RSP) );

class ovm_seq_item_pull_port #(type REQ=int, type RSP=REQ)

extends ovm_port_base #(sqr_if_base #(REQ, RSP));
```

(The other port and export classes are similarly derived from `ovm_port_base`)

Common methods for TLM ports and exports

`function new(string name ,`
`  ovm_component parent ,`
`  int min_size = 1 ,`
`  int max_size = 1 );`
	Constructor. min_size and max_size set minimum and maximum number of required/provided interfaces respectively (unlimited = -1)

plus methods inherited from ovm_port_base

Common methods for unidirectional imps

`function new(string name , IMP imp );`	Constructor. imp is handle to object that implements interface methods

Plus methods inherited from ovm_port_base

Common methods for bidirectional imps

`function new(string name , IMP imp , REQ_IMP req_imp = null , RSP_IMP rsp_imp = null );`	Constructor. imp is handle to object that implements interface methods

Plus methods inherited from ovm_port_base

Example

A component with a multi-port that can drive an unlimited number of interfaces

```
class compA extends ovm_component;
  ovm_blocking_put_port #(int) p0;
  function new(string name, ovm_component parent);
    super.new(name,parent);
  endfunction: new
  virtual function void build();
    super.build();
    p0 = new("p0",this,1,-1);
  endfunction: build
  task run();
    p0.debug_connected_to();
    for (int i=1; i<= p0.size(); i++) begin
     p0.put(i);
     p0.set_if(i);
    end
  endtask: run
  `ovm_component_utils(compA)
endclass: compA
```

A component that provides the implementation of a single interface

```
class compB extends ovm_component;
  ovm_blocking_put_imp #(int,compB) put_export;
  function new(string name, ovm_component parent);
    super.new(name,parent);
  endfunction: new
  virtual function void build();
```

```
    super.build();
    put_export = new("put_export",this);
  endfunction: build
  task put(int val);  //interface method
    ovm_report_info("compB",$psprintf("Received %0d",val));
  endtask: put
  `ovm_component_utils(compB)
endclass: compB
```

A component that connects port elements to exports locally while passing the remaining port elements to a higher level port

```
class compC extends ovm_component;
  compA A;
  compB B00,B01;
  ovm_blocking_put_port #(int) put_port;
  function new(string name, ovm_component parent);
    super.new(name,parent);
  endfunction: new
  virtual function void build();
    super.build();
    put_port = new("put_port",this,1,-1);
    $cast(A,create_component("compA","A"));
    $cast(B00,create_component("compB","B00"));
    $cast(B01,create_component("compB","B01"));
  endfunction: build
  function void connect();
    // the order here does not matter
    A.p0.connect(B00.put_export);
    A.p0.connect(B01.put_export);
    A.p0.connect(put_port);
  endfunction: connect
  `ovm_component_utils(compC)
endclass: compC
```

An environment that instantiates compC and further compB components to provide the required number of interfaces to compA

```
class sve extends ovm_env;
  compC C;
  compB ZB1,B2,B03;
  ...
  function void connect();
    C.put_port.connect(ZB1.put_export);
    C.put_port.connect(B03.put_export);
    C.put_port.connect(B2.put_export);
  endfunction: connect
```

```
  `ovm_component_utils(sve)
endclass: sve
```

Simulation output (note order of outputs)

```
OVM_INFO @ 0: sve1.C.A.p0 [Connections Debug] has 5
interfaces from 3 places
OVM_INFO @ 0: sve1.C.A.p0 [Connections Debug]  has 1
interface  provided by sve1.C.B00.put_export
OVM_INFO @ 0: sve1.C.A.p0 [Connections Debug]  has 1
interface  provided by sve1.C.B01.put_export
OVM_INFO @ 0: sve1.C.A.p0 [Connections Debug]  has 3
interfaces provided by sve1.C.put_port
OVM_INFO @ 0: sve1.C.put_port [Connections Debug] has 3
interfaces from 3 places
OVM_INFO @ 0: sve1.C.put_port [Connections Debug]  has 1
interface  provided by sve1.B03.put_export
OVM_INFO @ 0: sve1.C.put_port [Connections Debug]  has 1
interface  provided by sve1.B2.put_export
OVM_INFO @ 0: sve1.C.put_port [Connections Debug]  has 1
interface  provided by sve1.ZB1.put_export
OVM_INFO @ 0: sve1.B03 [compB] Received 1
OVM_INFO @ 0: sve1.B2 [compB] Received 2
OVM_INFO @ 0: sve1.C.B00 [compB] Received 3
OVM_INFO @ 0: sve1.C.B01 [compB] Received 4
OVM_INFO @ 0: sve1.ZB1 [compB] Received 5
```

Tips

- If appropriate, give "producer" and "consumer" components ports and connect them using a `tlm_fifo` channel. This usually requires less coding effort than giving the consumer an imp that can be directly connected to the producer.
- It is often easier to use the combined exports when creating a channel since these can be connected to blocking, non-blocking or combined ports..

Gotchas

- Remember that the export that provides the actual implementation of the interface methods should use `ovm_*_imp` rather than `ovm_*export`.
- An `ovm_*_imp` instance requires a type parameter that gives the type of the class that defines its interface methods (this is often its parent class). This object should also be passed as an argument to its constructor
- The order that interfaces are stored in a multi-port depends on their hierarchical names, not the order in which `connect` is called.

- The interface elements in a multi-port are accessed using index 1 to size(). Index 0 and index 1 return the same interface!

See also

ovm_port_base, tlm_fifo, TLM Interfaces

Class `ovm_port_base` is the base class for all OVM ports and exports. It provides a set of common functions for connecting and interrogating ports and exports.

A port or export may be connected to multiple interfaces (it is then known as a "multi-port"). Constructor arguments set the minimum and maximum number of interfaces that can be connected to a multi-port.

Declaration

```
virtual class ovm_port_base #(type IF=ovm_void) extends IF;

typedef enum {
  OVM_PORT ,
  OVM_EXPORT ,
  OVM_IMPLEMENTATION
} ovm_port_type_e;

typedef ovm_port_component_base ovm_port_list[string];
```

Methods

`function new(string name, ovm_component parent, ovm_port_type_e port_type, int min_size=0, int max_size=1);`	Constructor
`function string get_name();`	Returns port name
`virtual function string get_full_name();`	Returns hierarchical path name
`virtual function ovm_component get_parent();`	Returns a handle to parent component
`virtual function string get_type_name();`	Returns type as string
`function int max_size();`	Returns maximum number of connected interfaces
`function int min_size();`	Returns minimum number of connected interfaces
`function bit is_unbounded();`	True if no limit on connected interfaces (`max_size` = -1)
`function bit is_port();`	True if port
`function bit is_export();`	True if export
`function bit is_imp();`	True if imp

`function int size();`	Number of connected interfaces for “multi-port”
`function void set_if( int i = 0);`	Select indexed interface of multi-port
`function void set_default_index( int index);`	set default interface of multi-port
`function void connect( this_type provider);`	Connect to port/export†
`function void debug_connected_to( int level = 0 , int max_level = -1 );`	Print locations of interfaces connected to port. Recurse through multi-ports as necessary†
`function void debug_provided_to( int level = 0 , int max_level = -1 );`	Print locations of ports connected to export. Recurse through multi-ports as necessary†
`function void get_connected_to( ref ovm_port_list list);`	Returns list of ports/exports connected to port
`function void get_provided_to( ref ovm_port_list list);`	Returns list of ports connected to export
`function void resolve_bindings();`	Resolve port connections (called automatically)
`function ovm_port_base #(IF) get_if(int index=0);`	Returns the selected interface of multi-port

Definitions

`typedef ovm_port_base #( IF ) this_type;`	Base type of port/export with interface `IF`

See also

Ports, Exports and Imps

SystemVerilog does not provide data introspection of class objects for automatically printing objects, their members, or their contents. OVM, however, provides the machinery for automatically displaying an object by calling its `print` function. This can recursively, print its state and the state of its subcomponents in several pre- or user-defined formats, to the standard output or a file.

The field automation macros (`` `ovm_field_int ``, `` `ovm_field_enum ``, etc.) can be used to specify the *fields* (members) that are shown whenever an object or component is printed and the radix to use. For example,

```
`ovm_field_int ( data, OVM_ALL_ON | OVM_HEX )
```

tells the OVM machinery to provide all automation functions, including printing. Automatic printing of fields can be enabled and disabled using the `OVM_PRINT` and `OVM_NOPRINT` options respectively:

```
`ovm_field_int ( data, OVM_PRINT | OVM_DEC )
```

or

```
`ovm_field_int ( data, OVM_NOPRINT )
```

The field data type must correspond to the macro name suffix (`_int`, `_object`, etc). The radix (`OVM_HEX`, `OVM_DEC`, etc.) can be optionally OR-ed together with the macro flag. See **Field Macros** for more details.

Users can define their own custom printing for an object or component by overriding its `do_print` function. Whenever an object's `print` function is called, it first prints any automatic printing from the field macros (unless `OVM_NOPRINT` is specified), and then calls its `do_print` function (that for `ovm_object` and `ovm_component` does nothing by default). Overriding `do_print` in a derived class enables custom or addition information to be displayed. Note that some OVM classes (`ovm_transaction`, `ovm_sequence_item` and `ovm_sequencer` in particular) already override `do_print`to provide customized printing.

The `do_print` function receives an `ovm_printer` as an input argument. The `ovm_printer` class defines an OVM printing facility that can be extended and customized to create custom formatting. Since all printing can occur through the same printer, changes made in the printer class are immediately reflected throughout all test bench code. Printer classes also contain variable controls called *knobs*. Knob classes called `ovm_printer_knobs` allow the addition of new printer controls and can be swapped out dynamically to change the printer's configuration.

The `ovm_printer` can also be used to print other things besides OVM objects. While this can be done easily enough with one of the global printer instances, OVM also provides a series of macros to handle this automatically. For example,

```
`ovm_print_string( filename )
```

These macros print variables using the same formatting as an `ovm_object`, providing a consistent look-and-feel to the printing interface.

Printer Types

OVM defines a basic printer type called `ovm_printer`. The `ovm_printer` prints a raw dump of an object. This printer type is extended into 3 variations:

- a tabular printer for printing in columnar format (`ovm_table_printer`),
- a tree printer for printing objects in a tree format (`ovm_tree_printer`),
- a line printer that prints objects out on a single line (`ovm_line_printer`).

The default `ovm_printer` type prints objects in the following raw format:

```
               members     type       size  value
packet_obj  (packet_object)
packet_obj.data  (da(integral)) (6)
packet_obj.data[0] (integral) (32) 'd281
packet_obj.data[1] (integral) (32) 'd428
packet_obj.data[2] (integral) (32) 'd62
packet_obj.data[3] (integral) (32) 'd892
packet_obj.data[4] (integral) (32) 'd503
packet_obj.data[5] (integral) (32) 'd74
packet_obj.addr  integral) (32) 'h95e
packet_obj.size (size_t) (32) tiny
packet_obj.tag (string) (4) good
```

The `ovm_table_printer` prints objects in the following tabular format:

```
-----------------------------------------------------
Name    Type            Size    Value                   header
-----------------------------------------------------
packet_obj  packet_object     -      @{packet_obj} tiny+ object
 data   da(integral)    6                        -
  [0]   integral        32      'd281
  [1]   integral        32      'd428
  [2]   integral        32      'd62
  [3]   integral        32      'd892                   object
  [4]   integral        32      'd503                   fields
  [5]   integral        32      'd74
 addr   integral        32      'h95e
 size   size_t  32      tiny
 tag    string  4       good
-----------------------------------------------------
```

Here is the same object printed using the `ovm_tree_printer`:

```
packet_obj: (packet_object)  {
 data:  {
```

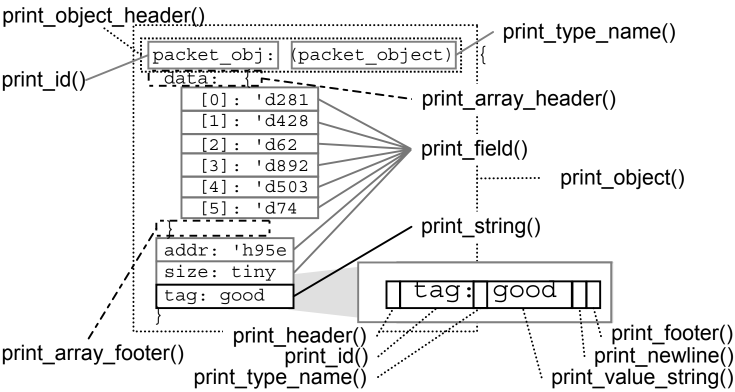
```
  [0]: 'd281
  [1]: 'd428
  [2]: 'd62
  [3]: 'd892
  [4]: 'd503
  [5]: 'd74
 }
 addr: 'h95e
 size: tiny
 tag: good
}
```

The `ovm_line_printer` prints an object's contents all on one line:

```
packet_obj: (packet_object)  { data:  { [0]: 'd281 [1]:
'd428 [2]: 'd62 [3]: 'd892 [4]: 'd503 [5]: 'd74 } addr:
'h95e size: tiny tag: good }
```

Printer Functions

The `ovm_printer` class defines many functions that can be overridden to make a custom printer and custom formatting. These functions define the OVM printing API, which the printing and field automation macros use. The table, tree, and line printers override several of these functions to create their own custom formatting. For example, the following diagram illustrates some of the printer function calls used to render the printing of an object:



Any of these methods can be overridden to specify different ways to print out the OVM object members. See `ovm_printer` for more details and examples.

Printer Knobs

Each printer is controllable through a set of printing knobs which control the format of the printer's output. A class called `ovm_printer_knobs` contains the knobs as variables, and it can be extended to add additional controls for a custom printer. In fact, the table, tree, and line printers use different knob classes to control their outputs. Knobs can control things like the column widths for the table printer, or what radix prefix to use when printing integral types.

Every derived `ovm_printer` class has a variable called knobs, which is used to point to an `ovm_knobs_class`. Printer functions can access these knobs through the knobs class reference. See `ovm_printer_knobs` for more details and examples.

Print Macros

In addition to printing OVM objects and components, OVM provides macros for printing many kinds of variables using the standard printer facility. These macros often include a printer argument for specifying the printer, which controls the format of the output. By using the print macros instead of `$display`, all printing remains consistent and controlled through a common interface. The following table lists the available print macros in `base/ovm_printer_defines.svh`. The macro parameters are:

`field`	Name of variable
`radix`	Radix to use
`printer`	Printer to use
`arraytype`	Name of array to print (without quotes)

Note, macros that do not include a printer parameter will print to the global default printer.

`` `ovm_print_int```(field, radix)`	Prints an integral type
`` `ovm_print_int3```( field, radix, printer)`	Prints an integral type to a specific printer
`` `ovm_print_object```(field)`	Prints an ovm_object
`` `ovm_print_object2```( field, printer)`	Prints an ovm_object to a specific printer
`` `ovm_print_string```(field)`	Prints a string
`` `ovm_print_string2```( field, printer)`	Prints a string to a specific printer
`` `ovm_print_array_int```( field, radix)`	Prints an array of integers

`ovm_print_array_int3(field, radix, printer)	Prints an array of integers to a specific printer
`ovm_print_qda_int4(field, radix, printer, arraytype)	Prints an integral array type to a specific printer
`ovm_print_queue_int(field, radix)	Prints a queue of integers
`ovm_print_queue_int3(field, radix, printer)	Prints a queue of integers to a specific printer
`ovm_print_array_string(field)	Prints an array of strings
`ovm_print_array_string2(field, printer)	Prints an array of strings to a specific printer
`ovm_print_string_qda3(field, printer, arraytype)	Prints an array of strings
`ovm_print_string_queue(field)	Prints a queue of strings
`ovm_print_string_queue2(field, printer)	Prints a queue of strings to a specific printer
`ovm_print_aa_string_int(field)	Prints an associative array of integral types with a string key
`ovm_print_aa_string_int3(field, radix, printer)	Prints an associative array of integral types with a string key to a specific printer
`ovm_print_aa_string_string(field)	Prints an associative array of string types with a string key
`ovm_print_aa_string_string(field, printer)	Prints an associative array of string types with a string key to a specific printer
`ovm_printer_aa_int_key4(key, field, radix, printer)	Prints an associative array of integral types with an arbitrary key type to a specific printer

Example

Here are examples of using the print macros:

```
initial
begin
  int  mem[255:0];
  string  msg = "Test passed successfully";
  string  sarray[2:0] = '{"string1", "string2", "string3"};
```

```
  foreach (mem[i])
    mem[i] = i * 3;     // Initialize the memory

  // Print out the memory (do not use quotes with the arraytype!)
  `ovm_print_qda_int4(mem,OVM_HEX,ovm_default_printer,mem)

  // Print out the string
  `ovm_print_string ( msg )

  // Print out the array of strings
  `ovm_print_array_string ( sarray )
end
```

This produces the following results:

```
mem           mem(integral)   256            -
 [0]          integral        32             'h0
 [1]          integral        32             'h3
 [2]          integral        32             'h6
 [3]          integral        32             'h9
 [4]          integral        32             'hc
 ...          ...             ...            ...
 [251]        integral        32             'h2f1
 [252]        integral        32             'h2f4
 [253]        integral        32             'h2f7
 [254]        integral        32             'h2fa
 [255]        integral        32             'h2fd
------------------------------------------------------------
------------------------------------------------------------
Name             Type          Size          Value
------------------------------------------------------------
msg              string        24            Test passed
success+
------------------------------------------------------------
------------------------------------------------------------
Name             Type          Size          Value
------------------------------------------------------------
sarray           da(string)    4                -
 [0]             string        7             string4
 [1]             string        7             string3
 [2]             string        7             string2
 [3]             string        7             string1
------------------------------------------------------------
```

Globals

In every OVM environment, four global printers are available:

`ovm_default_table_printer`

`ovm_default_tree_printer`

`ovm_default_line_printer`

`ovm_default_printer`

It is also possible to create other instances of the standard OVM printers or derived printer classes.

The printer to use can be specified by the argument to an object's `print` function. If `print` is called for an object and no printer argument is provided, then by default the `ovm_default_printer` is used. Initially, this is set to point to the `ovm_default_table_printer` so everything is printed in tabular form.

The print macros that do not take a printer argument also use the `ovm_default_printer`.

To globally change the default format of the print messages, assign a different printer to `ovm_default_printer`. For example,

```
initial
  ovm_default_printer = ovm_default_tree_printer;
```

Example

See `ovm_printer` and `ovm_printer_knobs`.

Tips

- Do not use quotes for the `arraytype` with the print macros or it will cause a compiler error.
- To globally change the format of all print messages, assign the `ovm_default_printer` a specific printer type or change the `ovm_default_printer.knobs`.

Gotchas

- Redefining the printer functions to create a new printer type requires a bit of work and finesse. They are not as straightforward as they may appear! It is often easier to copy and modify the functions from one of the standard printers than to create them from scratch.
- Do not include a semicolon at the end of the line when calling the macros.

See also

ovm_printer, ovm_printer_knobs

The ovm_printer class provides a facility for printing an ovm_object in various formats when the object's print function is called. The field automation macros specify the fields that are passed to the printer and their required format. Alternatively, an object's virtual do_print function may be overridden to print its fields explicitly by calling member functions of ovm_printer (do_printis called implicitly by print).

Several built-in printer classes are available, which are all derived from ovm_printer:

ovm_printer	Raw, unformatted dump of object
ovm_table_printer	Prints object in tabular format
ovm_tree_printer	Prints multi-line tree format
ovm_line_printer	Prints all object information on a single line

The ovm_printer class can also be extended to create a user defined printer format. Both the derived and user defined printer classes do not extend the printer's API, but simply add new *knobs*. Printer knobs provide control over the format of the printed output. Separate ovm_printer_knobs classes contain the knobs for each kind of printer.

Four default printers are globally instantiated in every OVM environment:

ovm_default_printer	Default table printer used by ovm_object::print() or ovm_object::sprint() when no printer is specified
ovm_default_line_printer	Line printer that can be used with ovm_object::do_print()
ovm_default_tree_printer	Tree printer that can be used with ovm_object::do_print()
ovm_default_table_printer	Table printer that can be used with ovm_object::do_print()

When an object's print function is called, if no optional printer argument is specified, then the ovm_default_printer is used. The ovm_default_printer variable can be assigned to any printer derived from ovm_printer.

Declaration

```
class ovm_printer;
```

Methods

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="");	Called from `do_print` to print an integral field
virtual function void **print_generic**(string name, string type_name, int size, string value, byte scope_separator=".");	Called from `do_print` to print a generic value
virtual function void **print_object**(string name, ovm_object value, byte scope_separator=".");	Called from `do_print` to print an object, recursively depending on the `depth` knob
virtual function void **print_object_header**(string name, ovm_object value, byte scope_separator=".");	Called from `do_print` to print the header of an object
virtual function void **print_string**(string name, string value, byte scope_separator=".");	Called from `do_print` to print a string field
virtual function void **print_time**(string name, time value, byte scope_separator=".");	Called from `do_print` to print a time value
virtual function void **print_array_footer**[†](int size=0);	Prints footer information for arrays and marks the completion of array printing
virtual function void **print_array_header**[†](string name, int size, string arraytype="array", byte scope_separator=".");	Prints header information for arrays
virtual function void **print_array_range**[†](int min,int max);	Prints a range using ellipses for values
virtual function void **print_footer**[†]();	Prints footer information
virtual function void **print_header**[†]();	Prints header information

`virtual protected function void` **`print_id`**†`(string id, byte scope_separator=".");`	Prints a field's name
`virtual protected function void` **`print_newline`**†`( bit do_global_indent=1);`	Prints a newline
`virtual protected function void` **`print_size`**†`(int size=-1);`	Prints a field's size
`virtual protected function void` **`print_type_name`**†`(string name, bit is_object=0);`	Prints a field's type
`virtual protected function void` **`print_value`**†`(ovm_bitstream_t value, int size, radix_enum radix=OVM_NORADIX);`	Prints an integral field's value
`virtual protected function void` **`print_value_array`**†`(string value="", int size=0);`	Prints an array's value
`virtual protected function void` **`print_value_object`**†`( ovm_object value);`	Prints a unique identifier associated with an object
`virtual protected function void` **`print_value_string`**†`(string value);`	Prints a string field (unless it is "-")
`virtual protected function void` **`indent`**†`(int depth, string indent_str="  ");`	Prints an indentation (`depth` copies of `indent_str`)
`protected function void` **`write_stream`**†`(string str);`	Prints a string

†Only use in derived printer classes

Members

`ovm_printer_knobs` **`knobs`**`;`	Knob object providing access to printer knobs
`string` **`m_string`**	Printer output is written to this string when `sprint` knob is set to 1 (only use in derived printer classes)

Deprecated

`function void` **`ovm_print_topology`**`();`	Replaced by ovm_top.print_topology

`function void print_unit_list( ovm_component comp=null);`
 	
`function void print_unit( string name, ovm_printer printer=null);`
 	
`function void print_units( ovm_printer printer=null);`
 	
`function void print_topology( ovm_printer printer=null);`
	Moved to ovm_root and deprecated

Example

```
class my_object extends ovm_object;
  int addr = 198;
  int data = 89291;
  string name = "This is my test string";
  `ovm_object_utils_begin( my_object )
    `ovm_field_int( addr, OVM_ALL_ON )
    `ovm_field_int( data, OVM_ALL_ON )
    `ovm_field_string( name, OVM_ALL_ON )
  `ovm_object_utils_end
  ...
endclass : my_object

module top;
  my_object my_obj = new("my_obj");
  initial begin
    // Print using the table printer
    ovm_default_printer = ovm_default_table_printer;
    $display("# This is from the table printer\n");
    my_obj.print();

    // Print using the tree printer
    ovm_default_printer = ovm_default_tree_printer;
    $display("# This is from the tree printer\n");
    my_obj.print();

    // Print using the line printer
    $display("# This is from the line printer\n");
    my_obj.print(ovm_default_line_printer);
  end
endmodule : top
```

Produces the following simulation results:

```
# This is from the table printer
-----------------------------------------------------------
Name          Type            Size          Value
-----------------------------------------------------------
my_obj        my_object       -       @{my_obj} 198 8929+
 addr         integral        32              'hc6
 data         integral        32              'h15ccb
 name         string          22      This is my test str+
-----------------------------------------------------------

# This is from the tree printer
my_obj: (my_object)  {
  addr: 'hc6
  data: 'h15ccb
  name: This is my test string
}

# This is from the line printer
my_obj: (my_object)  { addr: 'hc6 data: 'h15ccb name: This
is my test string }
```

A custom printer can also be created from `ovm_printer` or its derivatives. Here is an example:

```
class my_printer extends ovm_table_printer;

  // Print out the time and name before printing an object
  function void print_object( string name,
    ovm_object value, byte scope_separator=".");

    // Header information to print out (use write_steam())
    write_stream( $psprintf(
      "Printing object %s at time %0t:\n",name, $time) );

    // Call the parent function to print out object
    super.print_object(name, value, scope_separator );
  endfunction : print_object
endclass : my_printer

my_printer my_special_printer = new();

module top;
  my_object my_obj = new( "my_obj" );
  initial begin
```

```
    #100;
    // Print using my_printer
    my_obj.print( my_special_printer );
  end
endmodule : top
```

Produces the following simulation results:

```
Printing object my_obj at time 100:
-----------------------------------------------------------
Name          Type            Size          Value
-----------------------------------------------------------
my_obj        my_object       -      @{my_obj} 198 8929+
 addr         integral        32             'hc6
 data         integral        32             'h15ccb
 name         string          22     This is my test str+
-----------------------------------------------------------
```

Tips

- Set `ovm_default_printer` to `ovm_default_line_printer`, `ovm_default_tree_printer`, or `ovm_default_table_printer` to control the default format of the object printing.
- The `print_time` function is subject to the formatting set by the `$timeformat` system task.
- The `print_object` tasks prints an object recursively, based on the depth knob of the default printer knobs (see `ovm_printer_knobs`). By default, components are printed, but this can be disabled by setting the `ovm_component::print_enabled` bit to 0 for specific components that should not be automatically printed.
- The `do_global_indent` argument to the `print_newline` function determines if it should honor the `indent` knob.

Gotchas

- The printing facility is limited to printing values up to 4096 bits.
- The OVM printing facility is separate from the reporting facility and is not affected by the severity or verbosity level settings. It is possible to create a customized printer that takes account of the reporting verbosity settings (by overriding its `print_object` function for example). Another alternative is to redirect the printer's output from the standard output (or file) to a string. This can be achieved by setting the `sprint` knob. The formatted string can then be printed using the OVM reporting facility from within an object's `do_print` function. This is shown in the following example.


```
class my_object extends ovm_object;
 ...
`ovm_object_utils_begin( my_object )
  `ovm_field_int( addr, OVM_ALL_ON )
  `ovm_field_int( data, OVM_ALL_ON )
  `ovm_field_string( name, OVM_ALL_ON )
`ovm_object_utils_end

  function void do_print( ovm_printer printer );
    // The printer prints to m_string when sprint set
    ovm_report_info(get_name(),{":\n",printer.m_string});
  endfunction : do_print
endclass : my_object

module top;
  my_object my_obj = new( "my_obj" );
  initial begin
    #100;
    ovm_default_printer = ovm_default_table_printer;
    // Set printer to print to a string instead of STDOUT
    ovm_default_printer.knobs.sprint = 1;
    // Now, print the object, which calls the object's
    // do_print() function and uses the report mechanism
    my_obj.print();
  end
endmodule : top
```

This produces the following simulation result:

```
OVM_INFO @ 100: reporter [my_obj]:
-----------------------------------------------------------
Name           Type             Size           Value
-----------------------------------------------------------
my_obj         my_object        -       @{my_obj} 198 8929+
 addr          integral         32              'hc6
 data          integral         32              'h15ccb
 name          string           22      This is my test str+
-----------------------------------------------------------
```

See also

Print, ovm_printer_knobs

ovm_printer_knobs

Printer knobs provide control over the formatting of an `ovm_printer`'s output. The `ovm_printer_knobs` class contains a set of variables that are common to all printers. The knobs class can be extended to include additional controls for other derived printers. OVM defines 3 derived knob classes: `ovm_hier_printer_knobs`, `ovm_table_printer_knobs`, and `ovm_tree_printer_knobs`. By default, the `ovm_printer` uses the `ovm_printer_knobs` class. The `ovm_hier_printer_knobs` is not used directly by any printer class, but provides additional controls common to any hierarchical printing. The table and tree printer knob classes are derived from the hierarchical knob class.

Declaration

```
class ovm_printer_knobs;

class ovm_hier_printer_knobs extends ovm_printer_knobs;

class ovm_table_printer_knobs extends
  ovm_hier_printer_knobs;

class ovm_tree_printer_knobs extends
   ovm_hier_printer_knobs;
```

Methods

`function string` **`get_radix_str`**`(radix_enum radix);`	Returns *radix* in a printable form

Members

From *ovm_printer_knobs:*

`int` **`begin_elements`** `= 5;`	Number of elements at the head of a list that should be printed
`string` **`bin_radix`** `= "'b";`	String prepended to any integral type when `OVM_BIN` used for a radix
`int` **`column`** `= 0;`	Current column that the printer is pointing to
`string` **`dec_radix`** `= "'d";`	String prepended to any integral type when `OVM_DEC` used for a radix

`radix_enum` **`default_enum`** `= OVM_HEX;`	Default radix to use for integral values when `OVM_NORADIX` is specified
`int` **`depth`** `= -1;`	Indicates how deep to recurse when printing objects, where depth of -1 prints everything
`int` **`end_elements`** `= 5;`	Number of elements at the end of a list that should be printed
`bit` **`footer`** `= 1;`	Specifies if the footer should be printed
`bit` **`full_name`** `= 1;`	Specifies if leaf name or full name is printed
`int` **`global_indent`** `= 0;`	Number of columns of indentation printed when newline is printed
`bit` **`header`** `= 1;`	Specifies if the header should be printed
`string` **`hex_radix`** `= "'h ";`	String prepended to any integral type when `OVM_HEX` used for a radix
`bit` **`identifier`** `= 1;`	Specifies if an identifier should be printed
`int` **`max_width`** `= 999;`	Maximum column width to print
`integer` **`mcd`** `= OVM_STDOUT;`	File descriptor or multi-channel descriptor where print output is directed
`string` **`oct_radix`** `= "'o";`	String prepended to any integral type when `OVM_OCT` used for a radix
`bit` **`reference`** `= 1;`	Specifies if a unique reference ID for an ovm_object should be printed
`bit` **`show_radix`** `= 1;`	Specifies if the radix should be printed for integral types
`bit` **`size`** `= 1;`	Specifies if the size of the field should be printed
`bit` **`sprint`** `= 0;`	If set to 1, prints to a string instead of `mcd`

`string truncation = "+";`	Specifies truncation character to print when a field is too large to print
`bit type_name = 1;`	Specifies if the type name of a field should be printed
`string unsigned_radix = "'d ";`	Default radix to use for integral values when `OVM_UNSIGNED` is specified

From *ovm_hier_printer_knobs:*

`string indent_str = "";`	Specifies string to use for indentation
`bit show_root = 0;`	Specifies if root object show have its full path name printed

From *ovm_table_printer_knobs:*

`int name_width = 25;`	Sets the width of the name column
`int size_width = 5;`	Sets the width of the size column
`int type_width = 20;`	Sets the width of the type column
`int value_width = 20;`	Sets the width of the value column

From *ovm_tree_printer_knobs:*

`string separator = "{}";`	Two character string, representing the first and last characters printed when printing an object's value

Example

Using default printer:

```
ovm_default_printer.knobs.global_indent = 5;   // Indent 5
ovm_default_printer.knobs.type_name = 0;   // No type values
ovm_default_printer.knobs.truncation = "***";
```

Output:

```
     ------------------------------------------------------
     Name        Type         Size                   Value
     ------------------------------------------------------
     my_obj                   -    @{my_obj} RX {2398***
     payload                  6                          -
     [0]                      32                     'h95e
     [1]                      32                    'h2668
     [2]                      32                    'h206a
     ...
     [5]                      32                      'he7
     crc                      32                       'h3
     kind                     32                        RX
     msg                      7                    send_tx
     ------------------------------------------------------
```

Using tree printer:

```
ovm_default_printer = ovm_default_tree_printer;
ovm_default_printer.knobs.hex_radix = "0x"; // Change radix
ovm_default_printer.knobs.separator = "@@";
```

Output:

```
my_obj: (my_object)  @
 payload:  @
  [0]: 0x95e
  [1]: 0x2668
  [2]: 0x206a
  [3]: 0x276d
  [4]: 0x33d
  [5]: 0xe7
 @
 crc: 0x3
 kind: RX
 msg: send_tx
@
```

Turning off identifiers (not very useful in practice):

```
ovm_default_printer = ovm_default_line_printer;
ovm_default_printer.knobs.identifier = 0;
```

Output:

```
: (my_object)  { :  { : 'h95e : 'h2668 : 'h206a : 'h276d :
'h33d : 'he7 } : 'h3 : RX : send_tx }
```

Tips

- To turn off all printing to STDOUT, set the `mcd` field to 0.

  ```
  ovm_default_printer.knobs.mcd = 0;
  ```

  This will stop all standard printing messages issued for transactions and sequences.

Gotchas

- As of OVM 2.0, the `show_radix` knob is only implemented in the `ovm_table_printer`, but has no affect because `ovm_table_printer::print_value` calls `ovm_printer::print_value`, which ignores the `show_radix` knob. To turn off printing the radix, set the `dec_radix`, `hex_radix`, or `bin_radix` knobs to empty strings or override the `ovm_printer_knobs::get_radix_str` function to return an empty string.
- The results of the `reference` knob is simulator-dependent.
- When negative numbers are printed, the radix is not printed.
- If the maximum width of a column is reached, then nothing else is printed until a new line is printed.
- Line, table, and tree printers ignore the `full_name` knob and always print the leaf name.

See also

Print, ovm_printer

The `ovm_random_stimulus` class is a component that can be used to generate a sequence of randomized transactions. This class may be used directly or as the base class for a more specialized stimulus generator. The transactions are written to an `ovm_blocking_put_port` named `blocking_put_port`. This port may be connected to a `tlm_fifo` channel. If the fifo depth is set to 1, the stimulus generator will block after each write, until the component at the other end of the channel (usually a driver) has read the transaction. This provides a simple mechanism to synchronize the random stimulus with the actions of the driver.

The type of transaction generated is set by a type parameter.

The `generate_stimulus` task must be called to start the random stimulus sequence. It blocks until the sequence is complete. The length of the sequence can be specified as a task argument. By default, an infinite sequence is produced. Another optional argument allows a transaction "template" to be specified. This template is generally a class derived from the transaction parameter type that adds additional constraints.

Declaration

```
class ovm_random_stimulus
#(type trans_type=ovm_transaction) extends ovm_component;
```

Methods

`function` **`new`**`(string name , ovm_component parent);`	Constructor
`virtual task` **`generate_stimulus`**`( trans_type t = null, input int max_count = 0 );`	Starts stimulus of length `max_count` (0 = infinite). Optional transaction template t
`virtual function void` **`stop_stimulus_generation`**`();`	Ends `generate_stimulus` task

Members

`ovm_blocking_put_port #(trans_type)` **`blocking_put_port`**`;`	Port for generated stimulus

Example

Using ovm_random_stimulus in an environment

```
class verif_env extends ovm_env;
 ovm_random_stimulus #(basic_transaction) m_stimulus;
 dut_driver m_driver;
 tlm_fifo #(basic_transaction) m_fifo;
```

```
 int test_length;
 ...

 virtual function void build();
  super.build();
  get_config_int("run_test_length",test_length));
  m_stimulus = new ("m_stimulus",this);
  m_fifo = new("m_fifo",this);
  $cast(m_driver,
        create_component("dut_driver","m_driver"));
 endfunction: build

 virtual function void connect();
  m_stimulus.blocking_put_port.connect(m_fifo.put_export);
  m_driver.tx_in_port.connect(m_fifo.get_export);
 endfunction: connect

 virtual task run();
  m_stimulus.generate_stimulus(null,test_length);
endtask: run

`ovm_component_utils(verif_env)
endclass: verif_env
```

Tips

The transaction type must be a SystemVerilog class with a constructor that does not require arguments and `convert2string` and `clone` functions. Using a class derived from `ovm_transaction` with field automation macros for all of its fields satisfies this requirement.

Gotchas

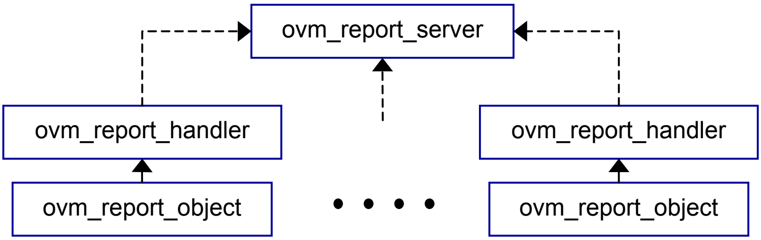
If you want to interrupt the `blocking generate_stimulus` task before its sequence is complete, you need to call it within a `fork-join` (or `fork-join_none`) block and call `stop_stimulus_generation` from a separate thread

See also

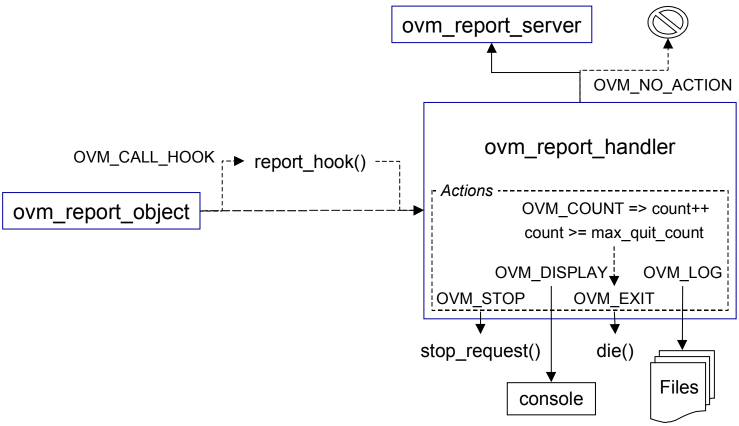
ovm_transaction

OVM offers a powerful reporting facility, providing the mechanisms to display messages in a uniformed format to different destinations, filtering of messages, and assigning actions to trigger when specific messages are issued. All reporting messages are issued through a global `ovm_report_server`; however, the report server is not intended to be directly accessed. Rather, an `ovm_report_object` is provided as the user interface into the reporting machinery.

Each `ovm_report_object` delegates its report issuing to an `ovm_report_handler`, which contains instance specific reporting controls and issues reporting actions if specified. The handler configures the server as necessary and then issues messages. Each reporting object has an associated handler, but all handlers use the same global report server unless configured otherwise. The following illustrates this association:



A report handler can be configured to perform actions that may trigger upon certain messages occurring. The handler can even be configured to not forward on report messages to the report server. The handler and its actions are shown in the following diagram:



Report Messages

There are four basic functions used for issuing messages. These methods are available both globally and by `ovm_report_object`:

```
function void ovm_report_info(string id,
                              string message,
                              int verbosity_level=100,
                              string filename="",
                              int line=0);
function void ovm_report_warning(string id,
                              string message,
                              int verbosity_level=100,
                              string filename="",
                              int line=0);
function void ovm_report_error(string id,
                               string message,
                               int verbosity_level=100,
                               string filename="",
                               int line=0);
function void ovm_report_fatal(string id,
                               string message,
                               int verbosity_level=100,
                               string filename="",
                               int line=0);
```

OVM provides four levels of severity: `INFO`, `WARNING`, `ERROR`, and `FATAL`. The severity levels print messages by default, but the `ERROR` and `FATAL` severity perform additional actions such as exiting simulation. The severity levels are defined as:

`OVM_INFO`	Informational message
`OVM_WARNING`	Warning message
`OVM_ERROR`	Error message
`OVM_FATAL`	Fatal message

The string `id` is an identifier used to group messages. For example, all the messages from a particular component could be grouped together using the same `id` to aid in debugging. This `id` then indicates where the messages are issued from so messages can quickly be traced to their origin.

The string `message` contains the text message to be issued.

The verbosity level represents an arbitrary value used for filtering messages. Messages with a verbosity level below the default verbosity level are issued; whereas, messages with a higher verbosity level are not issued but filtered out. The verbosity level provides a useful mechanism for controlling non-essential messages like debugging messages or for filtering out certain messages to reduce simulation log file size for simulation runs used in regression testing. To change the default verbosity level, call the `set_report_verbosity_level` function. OVM defines the default verbosity levels as:

`OVM_NONE`	0
`OVM_LOW`	10000
`OVM_MEDIUM`	20000
`OVM_HIGH`	30000
`OVM_FULL`	40000

The filename and line are optional arguments to indicate the filename and line number where a report message is being issued from. This extra information can help locate where the message occurred.

Report Actions

When a particular message occurs of a specific severity or id type, several possible report handling actions are possible. OVM defines 7 types of actions:

`OVM_NO_ACTION`	6'b000000	no action
`OVM_DISPLAY`	6'b000001	send report to STDOUT
`OVM_LOG`	6'b000010	send report to one or more files
`OVM_COUNT`	6'b000100	increment report counter
`OVM_EXIT`	6'b001000	calls `ovm_top.stop_request()` if called from within the `run` phase. Otherwise it forks a call to `$finish` to terminate simulation immediately
`OVM_CALL_HOOK`	6'b010000	call the report_hook() methods
`OVM_STOP`	6'b100000	call the stop_request() method and end the current phase

These action types can be OR-ed together to enable more than one action. By default, the severity levels are configured to perform the following actions:

`OVM_INFO`	`OVM_DISPLAY`
`OVM_WARNING`	`OVM_DISPLAY`
`OVM_ERROR`	`OVM_DISPLAY | OVM_COUNT`
`OVM_FATAL`	`OVM_DISPLAY | OVM_EXIT`

Actions can be assigned using the `set_report_*_action` functions (see `ovm_report_object`):

```
set_report_severity_action()
set_report_id_action()
set_report_severity_id_action()
```

For example, to disable warning messages on a particular reporting object, the `OVM_NO_ACTION` can be used:

```
set_report_severity_action( OVM_WARNING, OVM_NO_ACTION );
```

The `OVM_COUNT` action has a special behavior. If `OVM_COUNT` is set, a report issue counter is maintained in the report server. Once this count reaches the `max_quit_count`, then the `die` method is called (see `ovm_report_object`). Likewise, if the `OVM_EXIT` action is set, then the `die` method is also called and simulation ends. By default, `max_quit_count` is set to 0, meaning that no upper limit is set for `OVM_COUNT` reports. To set an upper limit, use `set_report_max_quit_count`.

The `OVM_LOG` action specifies that report messages should be issued to one or more files. To use this action, one or more files need to be opened and registered with the report handler. A multi-channel file id can be used, allowing duplication of messages to up to 31 open files. To associate a file with a handler, use the `set_report_*_file` functions:

```
set_report_default_file()
set_report_severity_file()
set_report_id_file()
set_report_severity_id_file()
```

For instance, the following example demonstrates how to log all `ERROR` and `FATAL` messages into a separate error log file:

```
// Open a file and associate it with a severity level
f = $fopen( "errors.log", "w" );
set_report_severity_file( OVM_ERROR, f );
set_report_severity_file( OVM_FATAL, f );
```

Report Hooks

In addition to assigning actions, OVM allows user-definable *report hooks*. Hooks are functions that determine whether or not a message should be issued. If the hook returns a boolean value of true, then the message is issued; otherwise, it is not sent to the report server. Customizable control like this can be quite useful to disable reporting messages during certain periods of simulation. For example, disabling all error messages while a reset signal is active by creating a custom report hook for error messages:

```
// Override the error report hook
function bit report_error_hook();
  if (reset)
    return 0;    // Turn off error messages if during reset
  else
    return 1;
endfunction : report_error_hook
```

Now, whenever an error message is issued, the report handler will first invoke the error report hook to see if it is acceptable to issue. Once the error report hook is called, then the `report_hook` is also called. The `report_hook` method acts as a catch-all function that can affect all messages, regardless of their id or severity.

Report State

While the report handler and server are not intended for direct use by testbench code, two functions are available that provide report statistics (collected by the server whenever messages are issued) and the state of the handler:

`function void` **`report_summarize`**`(FILE f=0);`	Generates statistical information on the reports issued by the server
`function void` **`dump_report_state`**`();`	Dumps the internal state of the report handler (max quit count, verbosity level, actions, and file handles)

For example, here is summary printed by `report_summarize()`:

```
#
# --- OVM Report Summary ---
#
# ** Report counts by severity
# OVM_INFO :   76
# OVM_WARNING :    0
# OVM_ERROR :   16
# OVM_FATAL :    0
# ** Report counts by id
# [ENV                ]      1
# [RNTST              ]      1
...
```

The report server has two virtual functions: `process_report` and `compose_ovm_info`. These functions control the construction of the reporting messages and the processing of the report actions. Both can be overridden, but are only intended to be changed by expert users.

Globals

All OVM components are derived from `ovm_report_object` so all the reporting functions and machinery are available from inside any component. There are global versions of the four reporting functions that can be called from SystemVerilog modules and from any OVM class that does not have an `ovm_report_object` base class (such as transactions and sequences). This provides them with the same reporting interface. Enumeration types for the report severity, verbosity, and reporting actions are also globally defined so they can be used anywhere.

A global report object `_global_reporter` is provided for the global report functions (this is actually an instance class derived from `ovm_report_object` whose name is set to `"reporter"`). Its methods can be called to set up the report handler and server for messages that are printed using the global report message functions.

Thresholds for the severity and verbosity levels may be set using command line arguments:

`+OVM_SEVERITY=`*`value`*	sets the global severity level threshold, where *value* equals one of the following:
	
	`INFO`
	`WARNING`
	`ERROR`
	`FATAL`

`+OVM_VERBOSITY=`*`value`*	sets the global verbosity level threshold, where *value* equals one of the following:
	
	any integer value
	`NONE │ OVM_NONE`
	`LOW │ OVM_LOW`
	`MEDIUM │ OVM_MEDIUM`
	`HIGH │ OVM_HIGH`
	`FULL │ OVM_FULL`
	`DEBUG │ OVM_DEBUG`

Example

See **ovm_report_object**.

Tips

- Use the `ovm_report_(info|warning|error|fatal)` functions for full control of messages rather than `$display` and `$fdisplay`. The system tasks do not provide configure and filtering capabilities in your environment.
- To pass in multiple variables and format them into a single string that can be passed to `ovm_report_(info|warning|error|fatal)` functions,

use a system function like `$psprintf`. (The OVM reporting functions do not accept variable length arguments nor a format string specifier as `$display` does).:

```
ovm_report_info( get_name(), $psprintf("Received
                 transaction.  Data = %d", data));
```

Gotchas

- The command line options `+OVM_SEVERITY` and `+OVM_VERBOSITY` cause the format of the leading information printed before the message string to be changed.
- Not all information printed out by OVM is controlled through the reporting mechanism. OVM also provides a printing facility for traversing objects hierarchies and printing their internal contents. To control printing of data objects (such as sequence transactions), see Printing.

See also

ovm_report_object

ovm_report_object

The OVM reporting facility provides three important features for reporting messages:

- tagging messages with specific id's
- assigning 4 levels of severity (INFO, WARNING, ERROR, and FATAL)
- controlling the verbosity of reported messages.

Each of these features provide users with different ways to filter or control the generation of messages and actions associated with them.

The user interface into the OVM reporting facility is provided through the ovm_report_object class. Report objects are delegated to report handlers, which control the issuing of report messages. Additional *hooks* are also provided so users can filter or control the issuing of messages. These hooks are provided as the user definable functions report_hook and report_*_hook that return 1'b1 if a message should be printed, otherwise they return 1'b0.. The hooks are executed if the OVM_CALL_HOOK action is associated with a message severity or id. (Note, the report_*_hook function is called first and then the catch-all report_hook function, providing two possible levels of filtering).

In addition to the call hook action, OVM defines several other report actions: OVM_NO_ACTION, OVM_DISPLAY, OVM_LOG, etc. (see Report for full details). These actions may be performed for messages of a specific type or severity. The OVM_LOG action enables reports to send messages to one or more files based on a message's type or severity using the set_report_*_file methods.

Since all ovm_components are derived from ovm_report_object, the report member functions are also members of every component. Typically, the ovm_report_info, ovm_report_warning, ovm_report_error, and ovm_report_fatal methods are called for issuing messages of a specified severity level. The ovm_component class extends several of the ovm_report_object functions to operate recursively on a component and all its subcomponents. These functions have the additional _hier suffix added to their name (see ovm_component).

Objects not derived from ovm_report_object can also use the reporting facility by calling the global ovm_report_info, ovm_report_warning, ovm_report_error, and ovm_report_fatal functions. A global ovm_report_object called _global_reporter is provided for these global methods.

Report objects provide a mechanism to increment a message count in the report server (by setting a message's action to OVM_COUNT) and to specify a maximum permitted count. If this maximum count is exceeded, the report server will call the die function. An OVM_EXIT action will also invoke die. If the die function is called within the run phase, ovm_top.stop_request is called which stops the simulation after a specified time delay (default = 0). The remaining phases (extract, check, and report) are then executed (see **Phase**). If die is called from other locations, the `report_summarize` function is called and simulation is terminated immediately with a forked call to `$finish`.

Declaration

```
virtual class ovm_report_object extends ovm_object;
```

Methods

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

`function void ovm_report_fatal(`
`  string id,`
`  string message,`
`  int verbosity_level=100,`
`  string filename="",`
`  int line=0);`
	Produces reports of severity `OVM_FATAL` (see Report)

`function void ovm_report_error(`
`  string id,`
`  string message,`
`  int verbosity_level=100,`
`  string filename="",`
`  int line=0);`
	Produces reports of severity `OVM_ERROR` (see Report)

`function void ovm_report_warning(`
`  string id,`
`  string message,`
`  int verbosity_level=100,`
`  string filename="",`
`  int line=0);`
	Produces reports of severity `OVM_WARNING` (see Report)

`function void ovm_report_info(`
`  string id,`
`  string message,`
`  int verbosity_level=100,`
`  string filename="",`
`  int line=0);`
	Produces reports of severity `OVM_INFO` (see Report)

`virtual function void die();`	Called by report server max quit count reached or `OVM_EXIT` action (fatal error)
`function void dump_report_state();`	Dumps the report handler's internal state
`function ovm_report_handler get_report_handler();`	Returns a reference to the report handler
`function ovm_report_server get_report_server();`	Returns the report server associated with this report
`virtual function void`	Prints copyright and

`report_header(FILE f=0);`	version numbers to command line if `f=0` or to a file if `f` is a file descriptor

`virtual function bit report_hook(`
`  string id,`
`  string message,`
`  int verbosity,`
`  string filename,`
`  int line);`
 	
`virtual function bit`
`report_info_hook(`
`  string id,`
`  string message,`
`  int verbosity,`
`  string filename,`
`  int line);`
 	
`virtual function bit`
`report_warning_hook(`
`  string id,`
`  string message,`
`  int verbosity,`
`  string filename,`
`  int line);`
 	
`virtual function bit`
`report_error_hook(`
`  string id,`
`  string message,`
`  int verbosity,`
`  string filename,`
`  int line);`
 	
`virtual function bit`
`report_fatal_hook(`
`  string id,`
`  string message,`
`  int verbosity,`
`  string filename,`
`  int line);`
	User-definable functions allowing additional actions to be performed when reports are issued if the report action is `OVM_CALL_HOOK`,
	
	Return value of 1 (default) allows reporting to proceed; otherwise, reporting is not processed
	
	The `report_*_hook` functions are only called for messages with a corresponding severity

`virtual function void` `report_summarize(FILE f=0);`	Prints report server statistical information to command line if `f=0` or to a file if `f` is a file descriptor
`function void` `reset_report_handler();`	Resets this object's report handler to default values
`function void set_report_handler(` `  ovm_report_handler hndlr);`	Sets the report hander

`function void`
`set_report_max_quit_count``(`
`  int m);`
	Sets maximum number of `OVM_COUNT` actions before `die` method is called; default value of 0 sets no maximum upper limit

`function void`
`set_report_default_file``(`
`  FILE file);`
 	
`function void`
`set_report_severity_file``(`
`  ovm_severity sev,`
`  FILE file);`
 	
`function void` **`set_report_id_file`**`(`
`  string id,`
`  FILE file);`
 	
`function void`
`set_report_severity_id_file``(`
`  ovm_severity sev,`
`  string id,`
`  FILE file);`
	Sets the output file for the `OVM_LOG` action. Specifying both severity and id takes precedence over id only which takes precedence over severity only which takes precedence over the default.

`function void`
`set_report_severity_action``(`
`  ovm_severity sev,`
`  ovm_action action);`
 	
`function void`
`set_report_id_action``(`
`  string id,`
`  ovm_action action);`
 	
`function void`
`set_report_severity_id_action``(`
`  ovm_severity sev,`
`  string id,`
`  ovm_action action);`
	Sets the report handler to perform a specific action for all reports matching the specified severity, id, or both, respectively, where action equals
	
	`OVM_NO_ACTION`
	
	or
	
	`OVM_DISPLAY | OVM_LOG | OVM_COUNT | OVM_EXIT | OVM_CALL_HOOK`

`function void`
`set_report_verbosity_level``(`
`  int verbosity_level);`
	Sets the maximum verbosity threshold (reports with a lower level are not processed)

Members

`protected ovm_report_handler` **`m_rh`**`;`	Handle to a report handler

Deprecated

function string **get_report_name**();	Removed, use `get_name` instead
function void set_report_name(string s);	Removed, use `set_name` instead

function void
ovm_report_message(
  string id, string message,
  int verbosity = 300,
  string filename = "",
  int line = 0);
 	
virtual function bit
report_message_hook(
  string id, string message,
  int verbosity,
  string filename, int line);
 	
function void
avm_report_message(
  string id, string message,
  int verbosity = 300,
  string filename = "",
  int line = 0);
	Replaced by `ovm_report_info` and `report_info_hook`

function void
avm_report_warning(
  string id, string message,
  int verbosity = 200,
  string filename = "",
  int line = 0);
 	
function void **avm_report_error**(
  string id, string message,
  int verbosity = 100,
  string filename = "",
  int line = 0);
 	
function void **avm_report_fatal**(
  string id, string message,
  int verbosity = 0,
  string filename = "",
  int line = 0);
	Replaced by ovm-prefixed functions

Example

```
class my_test extends ovm_test;
  ...
  // Turn off messages tagged with the id = “debug” for the my_env
  // component only
  my_env.set_report_id_action ("debug", OVM_NO_ACTION);

  // Turn all OVM_INFO messages off -
  // (Use set_report_*_hier version [from ovm_component]
  // to recursively traverse the hierarchy and set the action)
  set_report_severity_action_hier(OVM_INFO, OVM_NO_ACTION);

  // Turn all messages back on
  set_report_severity_action_hier( OVM_INFO,
                   OVM_DISPLAY | OVM_LOG );
  set_report_severity_action_hier( OVM_WARNING,
                   OVM_DISPLAY | OVM_LOG );
  set_report_severity_action_hier( OVM_ERROR,
                   OVM_DISPLAY | OVM_COUNT | OVM_LOG );
  set_report_severity_action_hier( OVM_FATAL,
                   OVM_DISPLAY | OVM_EXIT | OVM_LOG );

  // Setup the global reporting for messages that use the
  // global report handler (like the sequences machinery)
  _global_reporter.set_report_verbosity_level( OVM_ERROR );
  _global_reporter.dump_report_state();  // Print out state

  // Configure the environment to quit/die after one OVM_ERROR message
  set_report_max_quit_count( 1 );
endclass : my_test

// Example with user-definable report hooks
class my_env extends ovm_env;
  bit under_reset = 0;  // Indicates device under reset
  FILE f;
  ...
  // Override the report_hook function
  function bit report_hook(input id, string message,
                           int verbosity, string
                           filename, int line);
```



```
    // Turn off all reporting during the boot-up,
    // initialization, and reset period
    if (!under_reset && ( $time > 100ns ))
      return 1;
    else
      return 0;  // Either under_reset or time less than
                 // 100ns so do not issue report messages
  endfunction : report_hook

  function void start_of_simulation();
    // Duplicate report messages to a file
    f = $fopen( "sim.log", "w" );
    set_report_default_file_hier( f );

    // Setup the environment to not print INFO, WARNING,
    // and ERROR message during reset or initialization by
    // adding the OVM_CALL_HOOK reporting action
    set_report_severity_action_hier( OVM_INFO,
              OVM_DISPLAY | OVM_CALL_HOOK );
    set_report_severity_action_hier( OVM_WARNING,
              OVM_DISPLAY | OVM_CALL_HOOK );
    set_report_severity_action_hier( OVM_ERROR,
              OVM_DISPLAY | OVM_COUNT | OVM_CALL_HOOK );
  endfunction : start_of_simulation
endclass : my_env
```

Tips

- An `ovm_component` derives from `ovm_report_object` so all the reporting functions are available inside components. The `ovm_component` also extends the reporting functions so that they can hierarchically traverse a component and all of its subcomponents to set the reporting activity. The additional functions provided are:

  ```
  set_report_severity_action_hier
  set_report_id_action_hier
  set_report_severity_id_action_hier
  set_report_severity_file_hier
  set_report_default_file_hier
  set_report_id_file_hier
  set_report_severity_id_file_hier
  set_report_verbosity_level_hier
  ```

  See `ovm_component` for function details.

- Use set_report_max_quit_count to globally set the number of error messages before simulation is forced to quit. To include warning messages in the quit count, add the OVM_COUNT action to OVM_WARNING.
- The set_report_*_file functions can use multi-channel descriptors. Multi-channel file descriptors allow up to 31 files to be simultaneously opened so report messages can be sent to multiple log files by OR-in the file descriptors together. Verilog uses the MCD value 32'h1 for STDOUT.
- See **Report** for OVM severity, action, and verbosity definitions.
- Both the report_*_hook and report_hook functions are called when the OVM_CALL_HOOK action is set. This means that both a severity-specific action can be set as well as a general catch-all action. Note, however, if report_*_hook returns 1'b0, then report_hook is not called since the message reporting will have already been disabled.

Gotchas

- Many components in OVM use the global reporter. For example, sequences print general reporting messages using the global reporter. To turn these off, set the reporting actions on _global_reporter.
- By default, reporting messages are not sent to a file since the initial default file descriptor is set to 0 (even if OVM_LOG action is set). Use set_report_default_file() to set a different file descriptor.
- Actions set by set_report_id_action take precedence over actions set by set_report_severity_action.
- Actions set by set_report_severity_id_action take precedence over set_report_id_action.
- If the die function is called in a report object that is not an ovm_component or from an ovm_component instantiated outside of ovm_env, then report_summarize is called and the simulation ends by calling $finish.

See also

Report

ovm_root

Used from OVM 1.1 onwards as the top level object in an OVM testbench. Every OVM testbench contains a single instance of `ovm_root` named `ovm_top`. Users should not attempt to create any other instance of `ovm_root`. Any component that does not have a parent specified when it is created has its parent set automatically to `ovm_top`. This allows components created in multiple modules to share a common parent.

The `ovm_top` instance is used to search for named components within the testbench. The searching functions are passed a string argument containing a full hierarchical component name, which may also include wildcards: "?" matches any single character while "*" will match any sequence of characters, including ".". The component hierarchy is searched by following each branch from the top downwards: a match near the top of a branch takes precedence over a match near its bottom. The order in which the branches of the hierarchy are searched depends on the child component names: at each level, starting from `ovm_top`, these are searched in alphanumeric order. The `find` function returns a handle to the first component it comes across whose name matches the string (even if there are other matching components in subsequent branches that are closer to the top of the hierarchy). The `find_all` function returns a queue of matched component handles (by reference). An optional third argument specifies a component other than `ovm_top` to start the search.

The `ovm_top` instance may also be used to manage the OVM simulation phases. A new phase (derived from ovm_phase) may be inserted after any other phase (or at the start).

Declaration

```
virtual class ovm_root extends ovm_component;
```

Methods

`virtual task **run_test**( string test_name="");`	Runs all simulation phases for all components in the environment.
`virtual function string **get_type_name**();`	Returns "ovm_root"
`function void **stop_request**();`	Stops execution of the current phase
`function ovm_component **find**( string comp_match);`	Returns handle to component with name matching pattern
`function void **find_all**( string comp_match, ref ovm_component comps[$], input ovm_component comp=null);`	Returns queue of handles to components with matching names. `comp` specifies component to start search

`function void insert_phase( ovm_phase new_phase, ovm_phase exist_phase);`	Inserts a new phase after `exist_phase` (at start if `exist_phase = null`)
`function ovm_phase get_current_phase();`	Returns a handle to the currently executing phase
`function ovm_phase get_phase_by_name( string name);`	Returns a handle to the named phase

Members

`bit finish_on_completion = 1;`	When set run_test calls $finish on completion of the report phase.
`bit enable_print_topology = 0;`	When set the testbench hierarchy is printed when end_of_elaboration completes
``time phase_timeout = `OVM_DEFAULT_TIMEOUT;``	Sets the maximum simulated-time for task-based phases (e.g. `run`). Exits with error if reached.
``time stop_timeout = `OVM_DEFAULT_TIMEOUT;``	Sets the maximum time that any phase may remain active, after a call to `stop_request`

Global defines and methods

`` `define OVM_DEFAULT_TIMEOUT 64'h000000FFFFFFFFFF``	Default phase and stop timeout
`task run_test( string test_name="");`	Calls `ovm_top.run_test`
`function void global_stop_request();`	Calls `ovm_top.stop_request`
`function void set_global_timeout( time timeout);`	Sets `ovm_top.phase_timeout`
`function void set_global_stop_timeout( time timeout);`	Sets `ovm_top.stop_timeout`

Example

Configuring and running a test:

```
module top;
...
initial
  begin
    ovm_top.stop_timeout = 10000ns;
    ovm_top.enable_print_topology = 1;
    ovm_top.finish_on_completion = 0;
    run_test("test1");
  end
endmodule: top
```

Searching for components:

```
class test1 extends ovm_test;
  `ovm_component_utils(test1)
  verif_env env1;
  ovm_component c;
  ovm_component cq[$];
  function new (string name="", ovm_component parent=null);
    super.new(name,parent);
  endfunction : new
  virtual function void build();
    super.build();
    $cast(env1,ovm_factory::create_component(
                     "verif_env","","env1",null) );
  endfunction : build

  function void end_of_elaboration();
    c = ovm_top.find("env1.m_driver");
    ovm_top.find_all("*",cq,c);
    foreach(cq[i])
      ovm_report_info("FIND_ALL",
            $psprintf("Found %s of type %s",
            cq[i].get_full_name(),cq[i].get_type_name()));
  endfunction: end_of_elaboration
endclass: test1
```

Tips

- The global functions to stop the simulation and set the timeouts were deprecated in OVM 1.1.

- Call `ovm_top.stop_request` in the `run` method to stop simulation. Alternatively, assign `ovm_top.phase_timeout` before the start of simulation.
- For flexibility, use `+OVM_TESTNAME` command-line "plusarg" to set the test name, rather than passing it as an argument to `run_test`.

Gotchas

- Objects of class `ovm_root` should not be created explicitly.
- The name of `ovm_top` is set to "" so it does not appear in the hierarchical name of child components.
- A top-level environment instantiated by the test using the factory will have a parent named "ovm_test_top" that must be included in the search string for find (or use "*.")
- Phases can only be inserted before calling `run_test`.

See also

ovm_test

ovm_scoreboard

Class `ovm_scoreboard` is derived from `ovm_component`. User-defined scoreboards should be built using classes derived from `ovm_scoreboard`.

A scoreboard typically observes transactions on one or more inputs of a DUT, computes the expected effects of those transactions, and stores a representation of those effects in a form suitable for later checking when the corresponding transactions appear (or fail to appear) on the DUT's outputs.

Declaration

```
class ovm_scoreboard extends ovm_component;
```

Methods

`function new(string name, ovm_component parent = null);`	Constructor, mirrors the superclass constructor in `ovm_component`

Members

The `ovm_scoreboard` class has no members of its own. A scoreboard should provide an analysis export (commonly connected to an internal analysis FIFO) for each data stream that it observes, in addition to any internal structure that it needs to perform its checking.

Tips

- Use `ovm_scoreboard` as the base class for any user-defined scoreboard classes. In this way, scoreboards can be differentiated from other kinds of testbench component such as monitors, agents or stimulus generators.
- For DUTs whose input and output can each be merged into a single stream, and that deliver output in the same order as their input was received, scoreboard-like checking can be more easily accomplished with the built-in comparator classes `ovm_in_order_class_comparator` and `ovm_algorithmic_comparator`.

Gotchas

`ovm_scoreboard` has no methods or data members of its own, apart from its constructor and what it inherits from `ovm_component`.

See also

ovm_in_order_*_comparator

Sequences provide a structured approach to developing layered, random stimulus. A sequence represents a series of data or control transactions generated either at random or non-randomly, and executed either sequentially or in parallel. Sequences differ from the `ovm_random_stimulus` generation in that they provide *chaining* or *layering* of other sequences to produce complex data and control flows. Conceptually, a sequence can be thought of as a chain of function calls (to other sequences) resulting in the generation of sequence items (derived from the `ovm_sequence_item` class).

When sequences invoke other sequences, they are referred to as complex or *hierarchical sequences*. Hierarchical sequences allow for the creation of *sequence libraries*, which define basic sequence operations (such as reading and writing) that can be developed into more complex control or data operations.

OVM sequences are derived from the `ovm_sequence` class. Each sequence is associated with a sequencer (derived from the `ovm_sequencer` class). The sequencer is used to execute the sequence and place the generated sequence items on the sequencer's built-in sequence item export (`seq_item_export`). Generated sequence items are *pulled* from the sequencer by a driver (see **ovm_driver**). Each driver has a built-in sequencer port (called `seq_item_port`) that can be connected to a sequencer's sequence item export. The driver pulls sequence items from the sequencer by calling `get_next_item()` and sends an acknowledgement back to the sequencer by calling `item_done()` (both of these functions are members of `seq_item_port`).

A sequencer is registered with the OVM factory by calling the `` `ovm_sequencer_utils `` macro. A sequence is registered with the OVM factory and associated with a sequencer by calling the `` `ovm_sequence_utils `` macro.

The main functionality of a sequence is defined by declaring a `body()` task. In the body, nine steps are performed by the sequence:

1. Creation of an sequence item
2. Call `wait_for_grant`
3. Execution of the `pre_do` task
4. Optional randomization of the sequence item
5. Execution of the `mid_do` task
6. Call `send_request`
7. Call `wait_for_item_done`
8. Execution of the `post_do` function
9. Optionally call `get_response`

The following OVM macros perform most or all of these steps automatically:

```
`ovm_do
`ovm_do_pri
`ovm_do_with
`ovm_do_pri_with
`ovm_create
```

```
`ovm_send
`ovm_send_pri
`ovm_rand_send
`ovm_rand_send_pri
`ovm_rand_send_with
`ovm_rand_send_pri_with
`ovm_create_on
`ovm_do_on
`ovm_do_on_with
`ovm_do_on_pri
`ovm_do_on_pri_with
```

Variables must be declared for any `ovm_sequence` or `ovm_sequence_item` used by the OVM macros. Class members of the `ovm_sequence` or `ovm_sequence_item` can be declared `rand` so when the sequence is generated, the field values may be constrained using the `` `ovm_do_with ``, `` `ovm_do_pri_with ``, `` `ovm_rand_send_with ``, `` `ovm_rand_send_pri_with ``, `` `ovm_do_on_with `` or `` `ovm_do_on_pri_with `` macros. The `` `ovm_*_pri `` macros allow sequence items and sequences to be assigned a priority that is used when multiple sequence items are waiting to be pulled by a driver.

An OVM sequence has the following basic structure:

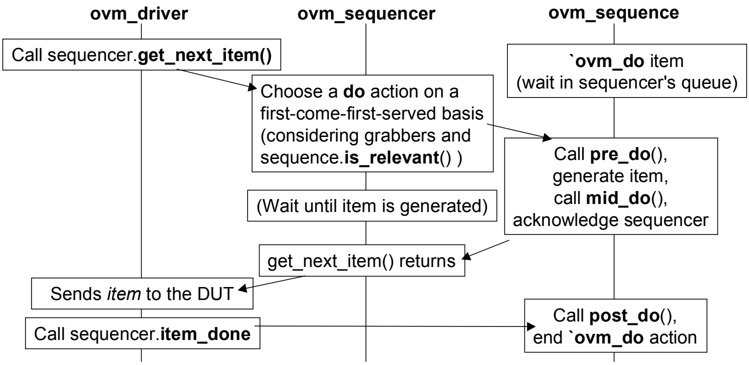
```
class my_sequence extends ovm_sequence #(my_sequence_item);
  my_sequence_item trans;

  virtual task pre_do(); endtask                // Optional

  virtual task body();
    // OVM sequence macro such as `ovm_do or `ovm_do_with
    `ovm_do ( trans )
  endtask : body

  virtual task post_do; endtask                 // Optional
endclass : my_sequence
```

The following diagram illustrates the flow of interactions between the driver, sequencer, and sequence objects:



Higher-level sequences can be created by managing sequences from multiple sequencers and are referred to as *virtual sequences*. Virtual sequences are "virtual" in that they do not generate their own sequence items; instead they control the spawning and execution of sequences associated with non-virtual sequencers. Virtual sequencers are also derived from the `ovm_sequencer` class. A virtual sequencer has one or more handles to the other sequencers that it controls (see Virtual Sequences for details).

Example

Declaring a sequence item

```
class my_seq_item extends ovm_sequence_item;
  rand int data;
  rand bit [4:0] addr;
  ...
  `ovm_object_utils_begin ( my_seq_item )
    `ovm_field_int ( data, OVM_ALL_ON + OVM_HEX )
    `ovm_field_int ( addr, OVM_ALL_ON + OVM_HEX )
    ...
  `ovm_object_utils_end
endclass : my_seq_item
```

Declaring a sequencer

```
class my_sequencer extends ovm_sequencer #(my_seq_item);
  `ovm_sequencer_utils ( my_sequencer )
  function new (string name = "",
                ovm_component parent = null);
    super.new( name, parent );
    `ovm_update_sequence_lib_and_item ( my_seq_item )
```

```
  endfunction : new
endclass : my_sequencer
```

Declaring a sequence

```
class my_sequence extends ovm_sequence #(my_seq_item);
  `ovm_sequence_utils ( my_sequence, my_sequencer )
  my_seq_item    m_seq_item;

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

  virtual task body();
      `ovm_do ( m_seq_item )
  endtask : body
endclass : my_sequence
```

Tips

- Register all sequence related components with the OVM factory using the registration macros for maximum flexibility and configurability.
- The OVM factory configuration functions (`set_config()`, `set_inst_override_by_*()`, and `set_type_override_by_*()`) can be used to override `ovm_sequence` and `ovm_sequence_item`s in order to change the sequence generation.
- Use the sequence action macros (like `` `ovm_do `` and `` `ovm_do_with ``) to automatically create and execute a sequence.

Gotchas

- As of OVM 2.0, the `ovm_sequencer` implements only PULL mode, meaning that the `ovm_driver` controls or pulls the `ovm_sequence_item`s from the `ovm_sequencer`. For the sequencer to control the interaction (i.e., PUSH mode), user modifications are required.
- No `ovm_virtual_sequence` exists. You must use `ovm_sequence` for both virtual and non-virtual sequences.
- Take care not to confuse `ovm_sequence` with `ovm_sequencer`. They differ by one letter only, but have quite different functionality.

See also

ovm_sequence_item, ovm_sequence, ovm_sequencer, Virtual Sequences, Special Sequences, Sequence Action Macros

OVM *sequences* are derived from the `ovm_sequence` class which itself is derived from the `ovm_sequence_base` and `ovm_sequence_item` classes. Multiple sequences are typically used to automatically generate the transactions required to drive the design under test. A sequence may generate either *sequence items* or invoke additional *subsequences*.

The main functionality of a sequence is placed in the `body` task. This task is either called directly by a *sequencer* (when it is a *root sequence*) or from the body of another sequence, (when it is run as a subsequence). If the sequence is a root sequence then its `pre_body` and `post_body` tasks are also called.

The purpose of the sequence body is to generate a sequence item that can be sent to a *driver* that controls the interaction with the design. A set of *do* actions provide an automated way to generate the sequence item, randomize it, send it to the driver, and wait for the response. These *do* actions are provided using the *sequence action macros* such as `` `ovm_do `` or `` `ovm_do_with ``. The *do* actions operate on both sequence items and subsequences.

An alternative mechanism in OVM 1.0 sent existing sequence items to the driver and waited for a response by calling an `apply` task. This mechanism is deprecated and was removed from OVM 2.0.

It is also possible to create sequence items, send them to a sequencer and wait for a response by explicitly calling the member tasks and functions of `ovm_sequence`. A sequence has a response queue that buffers the responses from a driver and permits multiple sequence items to be sent before the responses are processed. Member functions are provided to control the behavior of the response queue.

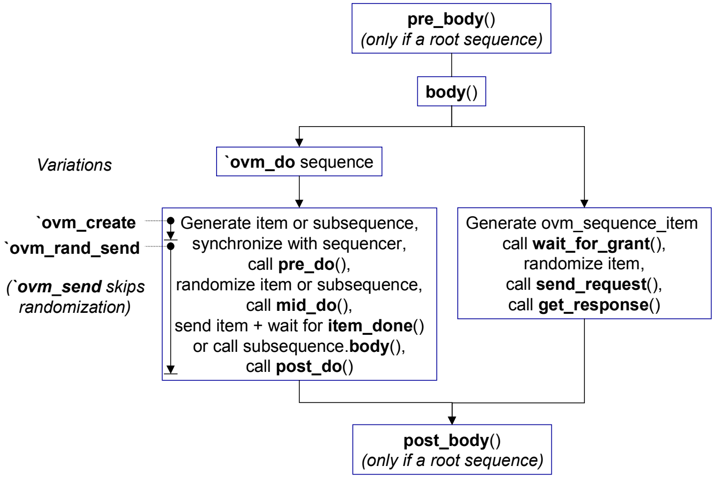
By default, responses from the driver are retrieved by calling `get_response`. Alternatively, the sequence behavior can be set to use an automatic report handler to fetch responses instead. The report handler is an overridden `response_handler` function (inherited from `ovm_sequence_base`).

Additional `pre_`, `mid_`, and `post_` virtual tasks can be defined for the sequence. These tasks provide additional control over the sequence's behavior.

A pre-defined request sequence item handle named `req` and response sequence item handle named `rsp` are provided as members of ovm_sequence. It is also possible to use your own sequence item handles in user-defined sequence classes.

Significant changes were made to the structure and functionality of the `ovm_sequence` class in OVM 2.0. These changes simplified the implementation of virtual sequences and provided a unified approach to replace OVM 1.0 sequences and scenarios (`ovm_scenario` is deprecated).

The following diagram illustrates the possible flows of execution for an OVM sequence:



Declaration

```
virtual class ovm_sequence #(type REQ = ovm_sequence_item,
                             type RSP = REQ)
  extends ovm_sequence_base;
```

Methods

`function new( string name = "ovm_sequence", ovm_sequencer_base sequencer_ptr = null, ovm_sequence_base parent_seq = null);`	Constructor Note: `sequencer_ptr` and `parent_seq` arguments not used from OVM 2.0 onwards
`virtual function void set_sequencer( ovm_sequencer_base sequencer);`	Sets the sequencer that sequence runs on (usually done by macro)
`virtual task start( ovm_sequencer_base sequencer, ovm_sequence_base parent_sequence = null, integer this_priority = 100, bit call_pre_post = 1);`	Starts execution of the sequence on the specified sequencer. If `call_pre_post` = 1 `pre_body` and `post_body` tasks are called

`function void` **`send_request`**`(`
`  ovm_sequence_item request,`
`  bit rerandomize = 0);`
	Sends sequence item to driver. Randomize item if `rerandomize` = 0

`function REQ` **`get_current_item`**`();`	Returns sequence item currently executing (on sequencer)

`task` **`get_response`**`(`
`  output RSP response,`
`  integer transaction_id = -1);`
	Retrieves response with matchingID (or next response if ID = -1) and removes it from queue. Blocks until response available

`virtual function void` **`put_response`**`(` `  ovm_sequence_item response_item);`	Puts a response back into the queue
`function void` **`set_response_queue_error_report_`** **`disabled`**`(bit value);`	If `value` = 1, turns off error reporting when response queue overflows
`function bit` **`get_response_queue_error_report_`** **`disabled`**`();`	Returns response queue error reporting state (1 = disabled)

`function void`
`set_response_queue_depth``(`
`  integer value);`
	Sets max depth of response queue (default = 8, unbounded = -1)

`function integer` **`get_response_queue_depth`**`();`	Get max allowed depth of response queue

Plus methods inherited from ovm_sequence_base

Members

`REQ` **`req`**`;`	Request sequence item
`RSP` **`rsp`**`;`	Response sequence item
`SEQUENCER` **`p_sequencer`**†`;`	Handle to sequencer executing this sequence

†Added by calling one of the sequence macros (see below)

Macros

Utility macros register sequences with a sequencer's sequence library. They also initialize the `p_sequencer` variable to a sequencer of type `SEQUENCER`.

The macros are:

```
(1) `ovm_sequence_utils(TYPE_NAME, SEQUENCER)
```

Used for sequences that do not require field macros to be called for their members, for example:

```
class my_sequence extends ovm_sequence #(my_seq_item);
    `ovm_sequence_utils( my_sequence, my_sequencer )
    ...
   endclass : my_sequence
```

```
(2)`ovm_sequence_utils_begin(TYPE_NAME, SEQUENCER)
   `ovm_sequence_utils_end
```

This allows the `` `ovm_field_* `` macros to be used. For example,

```
   `ovm_sequence_utils_begin(my_sequencer, my_sequencer)
     `ovm_field_int(data, OVM_ALL_ON)
     `ovm_field_int(size, OVM_ALL_ON)
     ...
   `ovm_sequence_utils_end
```

```
(3)`ovm_sequence_param_utils(TYPE_NAME, SEQUENCER)
   `ovm_sequence_param_utils_begin(TYPE_NAME, SEQUENCER)
   `ovm_sequence_utils_end
```

Like (1) and (2) but used for parameterized sequences, for example:

```
class my_sequence #(type T=my_seq_item)
  extends ovm_sequence #(T);
    `ovm_sequence_utils( my_sequence#(T), my_sequencer )
    ...
endclass : my_sequence
```

Example

```
// Define an ovm_sequence_item
typedef enum { read, write } dir_t;
class my_seq_item extends ovm_sequence_item;
  bit [31:0]  data;
  bit [9:0]  addr;
  dir_t     dir;

  // Register sequence item with the factory and add the
  // field automation macros
  `ovm_object_utils_begin( my_seq_item )
     `ovm_field_int( data, OVM_ALL_ON )
     `ovm_field_int( addr, OVM_ALL_ON )
     `ovm_field_enum( dir_t, dir, OVM_ALL_ON )
  `ovm_object_utils_end

endclass : my_seq_item
```

```
// Create a sequence that uses the sequence item
class my_seq extends ovm_sequence #(my_seq_item);
  `ovm_sequence_utils ( my_seq, my_sequencer )

  //my_seq_item req;    // built-in sequence item
  my_other_seq  subseq;    // A nested subsequence

  // Define a constructor
  function new ( string name = "my_seq);
      super.new (name);
  endfunction : new

  // Define the sequence functionality in the body()
  virtual task body();
    ovm_report_info ( get_name(),
                      "Starting the sequence ..." );

    // Use the do action macros on the sequence item
    `ovm_do_with( req,{ addr > 10'hfff; dir == read; } )

    // Invoke a nested subsequence
    `ovm_do_with( subseq, { ctrl_flag == `TRUE; } )
      ...
  endtask : body

endclass : my_seq
```

Tips

- Use the sequence action macros like `` `ovm_do `` and `` `ovm_do_with `` to automatically allocate and generate the `ovm_sequence_item`.
- Objects derived from `ovm_component` have a class member available named `m_name`, which is useful for printing out the component's name in reporting messages. Sequences are NOT derived from `ovm_component` and do not have a corresponding public member. Instead, use `get_name()` when writing reporting messages. For example,

```
ovm_report_info(get_name(), "Now executing sequence" );

ovm_report_error(get_name(), $psprintf(
  "Write to an invalid address!  Address = %s", addr)
);
```

- When building a sequence library, it is useful to create macros to simplify sequence definitions and avoid mistakes. The following provides an example of how such macros might look like:

```
`define SEQUENCE( seq, seqr ) \
   `ovm_sequence_utils ( seq, seqr ) \
   seq seq_item; \
   function new ( string name = `"seq); \
        super.new ( name ); \
   endfunction : new

`define ENDSEQUENCE endclass
```

Now use these macros to develop the sequence library:

```
// Create a basic write sequence
`SEQUENCE ( intf_write_seq, intf_sequencer )
   rand int reg_addr;
   rand int reg_data;
   virtual task body();
      `ovm_do_with ( seq_item,
                     { type == WRITE;
                       addr == reg_addr;
                       data == reg_data; })
   endtask : body
`ENDSEQUENCE

// Create a register initialization sequence
`SEQUENCE ( intf_init_regs_seq, intf_sequencer)
   intf_write_seq  write_seq;  // Nested subsequence
   virtual task body();
      for ( int i = 0; i < 256; i += 4 ) begin
         // Clear registers by passing values using constraints
         `ovm_do_with ( write_seq,
                        { type == WRITE;
                          reg_addr == i;
                          reg_data == 0; })
      end
   endtask : body
`ENDSEQUENCE
```

- Use the factory to override the sequence item types of sequences for greater flexibility and randomization, allowing the same sequence to be used with different configurations. For example,

```
// Modify the normal sequence to send error items
class error_seq extends normal_seq;
  // intf_seq_item seq_item; - defined in normal_seq
  ...
```

```
    factory.set_type_override_by_name(
      "intf_seq_item", "intf_error_seq_item" );

    `ovm_do ( seq_item )
    ...
  endclass : error_seq

  // Create a random sequence using randcase and factory type overrides
  class rand_seq extends ovm_sequence;
    intf_seq_item  seq_item;
    ...
    randcase
      // Send an error item 25% of the time
      1 : factory.set_type_override_by_name(
              "intf_seq_item", "error_seq_item" );
      // Send a complex item 25% of the time
      1 : factory.set_type_override_by_name(
              "intf_seq_item", "complex_seq_item" );
      // Send a normal item 50% of the time
      2 : factory.set_type_override_by_name(
              "intf_seq_item", "intf_seq_item" );
    endcase

    // Now send the randomly selected item
    `ovm_do ( seq_item )
    ...
  endclass : rand_seq
```

Gotchas

- Take care to use an `ovm_sequence_item` or `ovm_sequence` instead of an `ovm_transaction` with the *do* sequence action macros.
- No `ovm_virtual_sequence` exists so use `ovm_sequence` for both virtual and non-virtual sequences.

See also

Sequence, ovm_sequence_item, ovm_sequencer, Sequence Action Macros, Special Sequences

Sequence Action Macros

OVM defines a set of macros, known as the *sequence action macros*, which simplify the execution of sequences and sequence items. These macros are used inside of the `body` task of an `ovm_sequence` to perform one or more of the following steps on an item or sequence:

(1) *Create* – allocates item or sequence and initializes its sequencer and parent sequence

(2) *Synchronize with sequencer* – if an item, wait until the sequencer is ready

(3) *pre_do* – execute the user defined `pre_do` task of the executing sequence with an argument of 1 for an item and 0 for a sequence

(4) *Randomize* – randomize the item or sequence

(5) *mid_do* – execute the user defined `mid_do` task of the executing sequence with the specified item or sequence as an argument

(6) *Post-synchronization or body execution* – for an item, indicate to the sequencer that the item is ready to send to the consumer and wait for it to be consumed; for a sequence, execute the `body` task

(7) *post_do* – execute the user defined `post_do` task of the executing sequence with the specified item or sequence as an argument

These macros can be further divided into 2 groups: (1) macros that operate on *EITHER* a sequence item or sequence and invoked on sequencers, and (2) macros that operate *ONLY* on sequences such as used on a virtual sequencers. The latter group contain "`_on`" as part of their names, implying their use on sequences instead of items.

Macros

Macros used on regular sequences or sequence items:

`` `ovm_do ```(item_or_sequence)`	Performs all sequence actions on an item or sequence
`` `ovm_do_pri ```( item_or_sequence,priority)`	Same as `` `ovm_do `` but assigns a priority
`` `ovm_do_with ```( item_or_sequence, {constraint-block} )`	Performs all sequence actions on an item or sequence using the specified constraints to randomize the variable
`` `ovm_do_with ```( item_or_sequence,priority, {constraint-block} )`	Same as `` `ovm_do_with `` but assigns a priority
`` `ovm_create ```(item_or_sequence)`	Performs *ONLY* the create sequence action on an item or sequence using the factory

`ovm_send(item_or_sequence)	Similar to `` `ovm_do `` but skipping the create and randomization stages
`ovm_send_pri(item_or_sequence,priority)	Same as `ovm_send but assigns a priorty
`ovm_rand_send(item_or_sequence)	Similar to `` `ovm_do `` but skipping the create stage
`ovm_rand_send_pri(item_or_sequence,priority)	Same as `` `ovm_rand_send `` but assigns a priority
`ovm_rand_send_with(item_or_sequence, {constraint-block})	Similar to `` `ovm_do_with `` but skipping the create stage
`ovm_rand_send_pri_with(item_or_sequence, {constraint-block})	Same as `` `ovm_rand_send_with `` but assigns a priority

Macros used only with virtual sequences:

`ovm_do_on(item_or_sequence, sequencer)	Performs all sequence actions on a virtual sequence started on the specified sequencer. Its parent member is set to this sequence.
`ovm_do_on_with(item_or_sequence, sequencer, {constraints-block})	Same as `` `ovm_do_on `` but assigns a adds constraints
`ovm_do_on_pri(item_or_sequence, sequencer, priority)	Same as `` `ovm_do_on `` but assigns a priority
`ovm_do_on_pri_with(item_or_sequence, sequencer, priority, {constraints-block})	Same as `` `ovm_do_on `` but assigns a priority and adds constraints
`ovm_create_on(item_or_sequence, sequencer)	Performs *ONLY* the create sequence action on a virtual sequence

Example

```
// Examples of `ovm_do and `ovm_do_with
class example_sequence extends
ovm_sequence #(example_sequence_item);
  ...
  task body();
    // Send the sequence item to the driver
```

```
    `ovm_do( req )

    // Send item again, but add constraints
    `ovm_do_with( req,
                { addr > 0 && addr < 'hffff; })
  endtask : body
endclass : example_sequence

// Examples of `ovm_create and `ovm_rand_send
class fixed_size_sequence extends
    ovm_sequence #(example_sequence_item);
  ...
  task body();
    // Allocate the sequence item
    `ovm_create ( req )

    // Modify the sequence item before sending
    req.size = 128;
    req.size.rand_mode(0);  // No randomization

    // Now send the sequence item
    `ovm_rand_send ( req )
  endtask : body
endclass : fixed_size_sequence

// Example of `ovm_do_on in a virtual sequence
class virtual_sequence extends ovm_sequence;
    `ovm_sequence_utils ( virtual_sequence, virtual_seqr )
  write_sequence   wr_seq;
  read_sequence    rd_seq;
  ...
  task body();
    fork // Launch the virtual sequences in parallel
      `ovm_do_on_with ( wr_seq,
                        p_sequencer.seqr_a,
                        { parity == 1; addr > 48; })

      `ovm_do_on_with ( rd_seq,
                        p_sequencer.seqr_b,
                        { width == 32; type == LONG; })
    join
  endtask : body
endclass : virtual_sequence
```

Tips

- A variable used in a macro for an item or sequence only needs to be declared; there is no need to allocate it using `new()` or `create_component()`.

Gotchas

- Do not use a semicolon after a macro.
- Take care to use a semicolon after the last constraint in a macro's constraint block.

See also

Sequence, ovm_sequence, Virtual_Sequences, ovm_sequencer

ovm_sequence_base

Virtual class `ovm_sequence_base` is the base class for `ovm_sequence`. It provides a set of methods that are common to all sequences and do not depend on the sequence item type. It also defines a set of virtual functions and tasks that may be overridden in user-defined sequences and provide the standard interface required to create streams of sequence items and other sequences.

Declaration

```
class ovm_sequence_base extends ovm_sequence_item;

typedef enum {CREATED, PRE_BODY, BODY, POST_BODY, ENDED,
              STOPPED, FINISHED} ovm_sequence_state_enum;
```

Methods

`function` **`new`**`(`
`  string name = "ovm_sequence",`
`  ovm_sequencer_base sequencer_ptr`
`                     = null,`
`  ovm_sequence_base parent_seq`
`                     = null);`
	Constructor
	
	Note: `sequencer_ptr` and `parent_seq` arguments not used from OVM 2.0 onwards

`function ovm_sequence_state_enum` **`get_sequence_state`**`();`	
`task` **`wait_for_sequence_state`**`(` `  ovm_sequence_state_enum state);`	

`virtual task` **`start`**`(`
`  ovm_sequencer_base sequencer,`
`  ovm_sequence_base`
`    parent_sequence = null,`
`  integer this_priority = 100,`
`  bit call_pre_post = 1);`
	Starts execution of the sequence on the specified sequencer. If `call_pre_post` = 1 `pre_body` and `post_body` tasks are called

`function void` **`stop`**`();`	Disables the `body()` task of the sequence
`virtual task` **`pre_body`**`();`	
`virtual task` **`post_body`**`();`	User defined task called on root sequences immediately after the body() is executed
`virtual task` **`body`**`();`	Main method of the sequence
`virtual task` **`pre_do`**`(bit is_item);`	User defined task called at the start of a *do* action performed by a sequence

`virtual function void` **`mid_do`**`(` `  ovm_sequence_item this_item);`	User defined function called during a *do* action just after this_item is randomized and before either the item is sent to the consumer or subsequence body executed
`virtual function void` **`post_do`**`(` `  ovm_sequence_item this_item);`	User defined function called after either the consumer indicates the item is done or a subsequence body() completes
`virtual function bit` **`is_item`**`();`	Returns 1 for sequence items and 0 for sequences
`function integer` **`num_sequences`**`();`	Returns the number of sequences available on the attached sequencer's sequence library, 0 if none exist
`function integer` **`get_seq_kind`**`(` `  string type_name);`	Returns the integer location in the sequence kind map of the sequence type_name

`function ovm_sequence_base`
`get_sequence``(`
`  integer unsigned req_kind);`
	Creates a sequence of the type found at the specified integer location of sequence kind map

`function ovm_sequence_base`
`get_sequence_by_name``(`
`  string seq_name);`
	Creates a sequence of the specified name

`task` **`do_sequence_kind`**`(` `  integer unsigned req_kind);`	Invokes the sequence found at the specified integer location of sequence kind map
`function void` **`set_priority`**`(` `  integer value);`	Changes the priority of a sequence (default = 100)
`function integer` **`get_priority`**`();`	Returns the current priority of a sequence

`virtual task` **`wait_for_relevant`**`();`	User defined task that defines the trigger condition when the sequencer should re-evaluation if the sequence item is relevant (only called when is_relevant() returns 1'b0)
`virtual function bit` **`is_relevant`**`();`	User defined function that defines when sequence items are relevant for a *do* action; i.e., a *do* action on a sequence item is only selected if function returns 1'b1 (default implementation is 1'b1)
`function bit` **`is_blocked`**`();`	Returns 1 if sequence is blocked; otherwise, 0
`task` **`lock`**`(ovm_sequencer_base sequencer = null);`	Request to lock a sequencer (null = current default sequencer). Locks are arbitrated
`task` **`grab`**`(ovm_sequencer_base sequencer = null);`	Request to lock a sequencer (null = current default sequencer). Happens before arbitration
`function void` **`unlock`**`( ovm_sequencer_base sequencer = null);`	Removes any locks or grabs from this sequence
`function void` **`ungrab`**`( ovm_sequencer_base sequencer = null);`	Removes any locks or grabs from this sequence
`virtual task` **`wait_for_grant`**`( integer item_priority = -1, bit lock_request = 0);`	Blocks until sequencer granted. Must be followed by call to `send_request` in same time step
`virtual function void` **`send_request`**`( ovm_sequence_item request, bit rerandomize = 0);`	Sends item to sequencer. Must follow call to `wait_for_grant`. Item is randomized if `rerandomize` = 1
`virtual task` **`wait_for_item_done`**`( integer transaction_id = -1);`	Blocks until driver calls `item_done` or `put`. If specified, will wait until response with matching ID is seen

virtual function void **set_sequencer**(ovm_sequencer_base sequencer);	Sets the default sequencer to use
virtual function ovm_sequencer_base **get_sequencer**();	Returns current default sequencer
function void **kill**();	Kills the sequence
function void **use_response_handler**(bit enable);	Changes the behavior to use the response handler if `enable` = 1
function bit **get_use_response_handler**();	Returns 1 if behavior set to use response handler
virtual function void **response_handler**(ovm_sequence_item response);	The response handler – called automatically with response item if enabled
function int **get_id**();	Returns the sequence id

Members

event **started**;	Event indicating that the body() of the sequence has started
event **ended**;	Event indicating that the body() of the sequence has finished
constraint **pick_sequence**{}	Constraint used to select a valid value for seq_kind
rand integer unsigned **seq_kind**;	Sequence id

Tips

User-defined sequence classes should be derived from ovm_sequence rather than `ovm_sequence_base`.

Use a handle of type `ovm_sequence_base` if you want to reference sequences with different types (for example when passing arguments to functions).

See also

ovm_sequence, ovm_sequence_item

ovm_sequence_item

Class `ovm_sequence_item` is derived from `ovm_transaction` and is used to represent the most basic transaction item within a sequence. When sequences are executed, they generate one or more sequence items, which the sequencer passes through its consumer interface to the driver's producer interface. The driver calls the `get_next_item` and `item_done` functions provided by its `ovm_seq_item_port` to pull the sequence items from the sequencer.

When a sequencer creates a sequence, it gives it a unique integer identifier. `ovm_sequence_item` has member functions that enable this identifier to be set and returned. By default, the sequence identifier and other information about the sequence is not copied, printed or recorded. This behavior can be changed by setting a sequence info bit.

Declaration

```
class ovm_sequence_item extends ovm_transaction;
```

Methods

`function new( string name ="ovm_sequence_item", ovm_sequencer_base sequencer = null, ovm_sequence parent_base = null);`	Constructor
`function string get_type_name();`	Returns "ovm_sequence_item"
`function void set_sequence_id( integer id);`	Sets ID
`function integer get_sequence_id();`	Returns ID
`function void set_use_sequence_info(bit value);`	Sets sequence info bit (use info = 1)
`function bit get_use_sequence_info();`	Returns current value of sequence info bit
`function void set_id_info(ovm_sequence_item item);`	Copies sequence ID and transaction ID between sequence items. Should be called by drivers to ensure response matches request
`function void set_sequencer( ovm_sequencer_base sequencer);`	Sets the sequencer for the sequence item *(not needed when using sequence action macros)*

`function ovm_sequencer_base` **`get_sequencer`**`();`	Returns the sequencer for the sequence item
`function void` **`set_parent_sequence`**`( ovm_sequence_base parent);`	Sets the parent sequence for a sequence item *(not needed when using sequence action macros)*
`function ovm_sequence_base` **`get_parent_sequence`**`();`	Returns the parent sequence of the sequence item
`virtual function bit` **`is_item`**`();`	Returns 1 if object is a sequence item, 0 otherwise
`function void` **`set_depth`**`( integer value);`	Sets depth of sequence item (for example, 1 for a root sequence item, 2 for a child sequence item, etc.)
`function integer` **`get_depth`**`();`	Returns depth of sequence item
`function string` **`get_full_name`**`();`	Get the hierarchical sequence name (sequencer and parent sequences)
`function string` **`get_root_sequence_name`**`();`	
`function ovm_sequence_base` **`get_root_sequence`**`();`	
`function string` **`get_sequence_path`**`();`	Get the sequence name (including its parent sequences)

Members

`bit` **`print_sequence_info`** `= 0;`	If 1, sequence specific information is printed by item's print() function (set to 1 automatically by sequence macros)

Example

```
typedef enum { RX, TX } kind_t;
```

```
class my_seq_item extends ovm_sequence_item;

  rand  bit [4:0]    addr;
  rand  bit [31:0]   data;
  rand kind_t        kind;

  // Constructor
  function new (  string new = "my_seq_item",
              ovm_sequencer_base sequencer = null,
              ovm_sequence parent_seq = null );
    super.new ( name, sequencer, parent_seq );
  endfunction : new

  // Register with OVM factory and use field automation
  `ovm_object_utils_begin( my_seq_item )
    `ovm_field_int    ( addr,  OVM_ALL_ON + OVM_DEC)
    `ovm_field_int    ( data,  OVM_ALL_ON + OVM_DEC)
    `ovm_field_enum  ( kind_t, kind, OVM_ALL_ON )
  `ovm_object_utils_end
endclass : my_seq_item
```

Tips

- Use the sequence action macros like `` `ovm_do `` and `` `ovm_do_with `` to automatically allocate and generate the `ovm_sequence_item`.
- If a sequence item is created manually using `new()`, then set `print_sequence_info` = 1 to see sequence specific information (this is automatically set by the sequence action macros).
- The `get_sequence_path` function returns a string containing a trace of the sequence's hierarchy, which can be useful in debug and error messages.

Gotchas

- The member function `is_item()` is declared virtual, but is not intended to be overridden except by the OVM machinery.
- The `ovm_sequence_item` constructor does not take the same arguments as the `ovm_transaction` constructor. Be careful to note the difference and call `super.new()` with the correct arguments!

See also

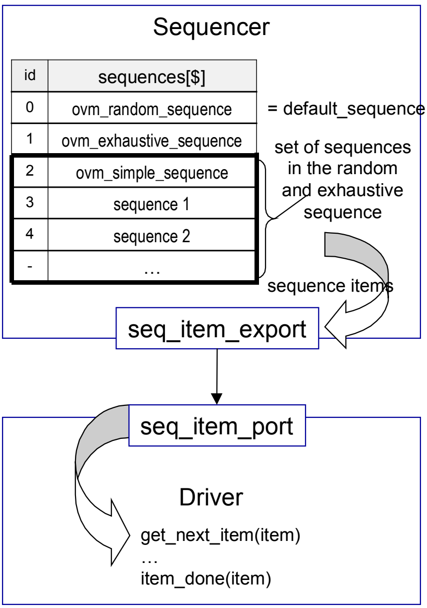
ovm_transaction, ovm_sequence, ovm_sequencer, Sequencer Interface and Ports, Sequence Action Macros

Class `ovm_sequencer` is used for both virtual and non-virtual sequences from OVM 2.0 onwards. Its behavior is defined in several base classes: its immediate base class is `ovm_sequencer_param_base`. This is derived from the abstract base class `ovm_sequencer_base`, which is itself derived from `ovm_component`.

Every Sequencer contains a library of added sequences. A sequencer provides a mechanism to start and manage the execution of the sequences in its library. A sequencer can also execute sequences that have not been added to its library. Sequencers provide a sequence item export, which drivers connect to in order to fetch the sequence items. A sequencer can also manage parallel execution of sequences, queuing up sequence items as they are generated. Other testbench components such as virtual sequencers can grab exclusive access to a sequencer in order to control the generation of transactions that it sends to its attached driver.

The `ovm_sequencer_base` class and the parameterized `ovm_sequencer_param_base` class provide the functionality and members common to all sequencers. In particular, they define the `default_sequence`, `count`, and `max_random_count` variables, which control the default random behavior of the sequencer. The starting and controlling of sequences on a sequencer is generally handled by the sequence action macros.

The following diagram illustrates the connection of a sequencer with a driver:



Declaration

```
class ovm_sequencer #(type REQ = ovm_sequence_item,
                      type RSP = REQ)
extends ovm_sequencer_param_base #(REQ, RSP);

typedef ovm_sequencer #(ovm_sequence_item)
ovm_virtual_sequencer;

typedef enum {SEQ_TYPE_REQ, SEQ_TYPE_LOCK, SEQ_TYPE_GRAB}
  SEQ_REQ_TYPE;

typedef enum {SEQ_ARB_FIFO, SEQ_ARB_WEIGHTED,
              SEQ_ARB_RANDOM, SEQ_ARB_STRICT_FIFO,
              SEQ_ARB_STRICT_RANDOM, SEQ_ARB_USER}
  SEQ_ARB_TYPE;
```

Methods

From *ovm_sequencer_base:*

`function bit` **`is_child`**`(`
`ovm_sequence_base parent,`
`ovm_sequence_base child);`
	Returns 1 if `child` is child of `parent`

`task` **`wait_for_grant`**`(`
`ovm_sequence_base sequence_ptr,`
`integer item_priority = -1,`
`bit lock_request = 0);`
	Issues request for sequence and waits until granted

`task` **`wait_for_item_done`**`(`
`ovm_sequence_base sequence_ptr,`
`integer transaction_id);`
	Waits until driver calls `item_done` or `put`

`function bit` **`is_blocked`**`(` `ovm_sequence_base sequence_ptr);`	Returns 1 if sequence is blocked
`function bit` **`is_locked`**`(` `ovm_sequence_base sequence_ptr);`	Returns 1 if sequence is locked
`task` **`lock`**`(` `ovm_sequence_base sequence_ptr);`	Requests a lock on sequence
`task` **`grab`**`(` `ovm_sequence_base sequence_ptr);`	Grants exclusive access to the sequencer's action queue, blocking all other requests
`function void` **`unlock`**`(` `ovm_sequence_base sequence_ptr);`	Removes locks from sequence

`function void` **`ungrab`**`(` `  ovm_sequence_base sequence_ptr);`	Releases exclusive access to the sequencer's action queue granted by grab()
`virtual function ovm_sequence_base` **`current_grabber`**`();`	Returns a reference to the current grabbing sequence, *null* if no grabbing sequence
`function void` **`stop_sequences`**`();`	Kills all sequences running on this sequencer
`virtual function bit` **`is_grabbed`**`();`	Returns 1 if a sequence is currently grabbing exclusive access, 0 if no sequence is grabbing
`function bit` **`has_do_available`**`();`	Returns 1 if the sequencer has an item available for immediate processing, 0 if no items are available
`function void` **`set_arbitration`**`(` `  SEQ_ARB_TYPE val);`	Change the arbitratrion mode

`virtual function integer`
`user_priority_arbitration``(`
`  integer avail_sequences[$]);`
	Called if arbitration mode = `SEQ_ARB_USER` to arbitrate between sequences. Default behavior is `SEQ_ARB_FIFO`

`virtual task` **`wait_for_sequences`**`();`	User overridable task used to introduce delta delay cycles (default 100) , allowing processes placing items in the consumer interface to complete before the producer interface retrieves the items
`function void` **`add_sequence`**`(` `  string type_name);`	Add a sequence to the sequence library
`function void` **`remove_sequence`**`(` `  string type_name);`	Remove a sequence from the sequence library

`function integer` **`get_seq_kind`**`(` `  string type_name);`	Returns the integer mapping of a sequence in the sequence library specified by type_name
`function ovm_sequence_base` **`get_sequence`**`(integer req_kind);`	Creates a sequence of the type located at the specified integer mapping in the sequence library
`function integer` **`num_sequences`**`();`	Returns number of sequences in sequence library

`virtual function void`
`send_request``(`
`  ovm_sequence_base sequence_ptr,`
`  ovm_sequence_item t,`
`  bit rerandomize = 0);`
	Sends a request to the sequencer. May only be called immediately after `wait_for_grant`

From *ovm_sequencer_param_base:*

`function REQ` **`get_current_item`**`();`	Returns the sequence item being executed
`function void` **`put_response`**`(RSP t);`	Put item in response queue
`task` **`start_default_sequence`**`();`	Called by sequencer's run task to start default sequence
`function int` **`get_num_reqs_sent`**`();`	Returns number of requests sent by sequencer
`function int` **`get_num_rsps_received`**`();`	Returns number of responses received by sequencer
`function void` **`set_num_last_reqs`**`(` `  int unsigned max);`	Sets size of last request buffer (default=1,max=1024)
`function int unsigned` **`get_num_last_reqs`**`();`	Gets size of last requests buffer
`function REQ` **`last_req`**`(` `  int unsigned n = 0);`	Gets the last request from the buffer (or possition within buffer)

`function void` **`set_num_last_rsps`**`( int unsigned max);`	Sets size of last response buffer (default=1,max=1024)
`function int unsigned` **`get_num_last_rsps`**`();`	Gets size of last response buffer
`function RSP` **`last_rsp`**`( int unsigned n = 0);`	Gets the last response from the buffer (or possition within buffer)
`virtual task` **`execute_item`**`(ovm_sequence_item item);`	Execute a sequence without adding it to the library and ignoring the response

From *ovm_sequencer:*

`function` **`new`**`( string name, ovm_component parent);`	Constructor
`virtual function string` **`get_type_name`**`();`	Returns "ovm_sequencer"
`virtual function void` **`send_request`**`( ovm_sequence_base sequence_ptr, ovm_sequence_item t, bit rerandomize = 0);`	Sends a request to the sequencer. May only be called immediately after `wait_for_grant`

Members

From *ovm_sequencer_base:*

`protected rand int` **`seq_kind`**`;`	Sequence id
`string` **`sequences`**`[$];`	Queue storing the names of the available sequences
`protected string` **`default_sequence`** `= "ovm_random_sequence";`	Name of the sequence the sequencer starts automatically if the count != 0
`int` **`count`** `= -1;`	Integer value used by the ovm_random_sequence to determine the number of random sequences to execute if count = -1, then a random number of

	sequences between 1 and max_random_count are selected if count = 0, then the default_sequence is not started by sequencer if count > 0, then the specified count of random sequences are selected
`integer unsigned max_random_count=10;`	Max number of selected sequences if count = -1
`int unsigned max_random_depth = 4;`	Depth of random sequence

From *ovm_sequencer_param_base:*

`ovm_analysis_export #(RSP) rsp_export;`	Analysis export that may be used to send responses to sequencer

From *ovm_sequencer:*

`ovm_seq_item_pull_imp #(REQ, RSP, this_type) seq_item_export;`	Sequence item export (connects to driver)

Macros

Utility macros create the sequence library and/or register the sequencer with the OVM factory.

```
(1) `ovm_sequencer_utils(SEQUENCER)
```

This macro provides the infrastructure required to build the sequencer's sequence library and utilize the random sequence selection interfaces. It adds ovm_random_sequence, ovm_exhaustive_sequence, and ovm_simple_sequence to the sequencer's sequence library. It also calls `` `ovm_component_utils(SEQUENCER) ``.

```
(2) `ovm_sequencer_utils_begin(SEQUENCER)
    `ovm_sequencer_utils_end
```

This is similar to `` `ovm_sequencer_utils ``, but calls `` `ovm_component_utils_begin(SEQUENCER) ``, and `ovm_component_utils_end`. This allows the `` `ovm_field_* `` macros to be called. For example,

```
`ovm_sequencer_utils_begin(my_sequencer, my_sequencer)
    `ovm_field_int(status, OVM_ALL_ON)
    ...
`ovm_sequencer_utils_end
```

(3) **`ovm_sequencer_param_utils**(SEQUENCER)

```
`ovm_sequencer_param_utils_begin(SEQUENCER)
`ovm_sequencer_utils_end
```

These macros are provided for parameterized sequencers. They have the same behavior as (1) and (2) except that they do not add a type name when they register the sequencer with the factory.

(4) **`ovm_update_sequence_lib_and_item**(USER_ITEM_TYPE)

This macro populates the `sequences[$]` queue and builds the sequence library with the `ovm_random_sequence`, `ovm_exhaustive_sequence`, and `ovm_simple_sequence`. The `USER_ITEM_TYPE` defines the sequence item type to be used with the `ovm_simple_sequence`.

This macro should only be used with a non-virtual sequencer and placed in its constructor as follows:

```
function my_sequencer::new( string name = "",
                            ovm_component parent = null );
   super.new( name, parent );
  `ovm_update_sequence_lib_and_item( my_seq_item )
endfunction : new
```

(5) **`ovm_update_sequence_lib**

This macro populates the sequences[$] queue and builds the sequence `library with the ovm_random_sequence`, `ovm_exhaustive_sequence`. Unlike (4), it does not add `ovm_simple_sequence` to the sequence library. It is used for virtual sequencers that do not create sequence item directly (only by invoking sequences on other sequencers). See **Virtual Sequences**

Example

```
// Define an ovm_sequence_item
typedef enum { read, write } dir_t;
class my_seq_item extends ovm_sequence_item;
  bit [31:0] data;
  bit [9:0]  addr;
  dir_t      dir;
```



```
  // Register sequence item with the factory and add the
  // field automation macros
  `ovm_object_utils_begin( my_seq_item )
    `ovm_field_int( data, OVM_ALL_ON )
    `ovm_field_int( addr, OVM_ALL_ON )
    `ovm_field_enum( dir_t, dir, OVM_ALL_ON )
  `ovm_object_utils_end
endclass : my_seq_item

// Create the sequencer
class my_sequencer extends ovm_sequencer #(my_seq_item);

  // Register the sequencer with the factory
  `ovm_sequencer_utils ( my_sequencer )

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

    // Create the sequence library
    `ovm_update_sequence_lib_and_item ( my_seq_item )
  endfunction : new
endclass : my_sequencer

// Connect the sequencer to the driver
class my_env extends ovm_env;
  ...
  my_sequencer  m_seqr;
  my_driver     m_drv;

  function void connect;
    // Hook up the sequencer to the driver
    m_drv.seq_item_port.connect(m_seqr.seq_item_export);
  endfunction : connect
  ...
endclass : my_env
```

Tips

- Use `set_config_string` to set the default sequence that the sequencer should execute. For example,

  ```
  set_config_string( "*.intf_sequencer", //Sequencer name
  ```

```
                         "default_sequence",
                         "my_seq"  );        // New sequence
```

When in random mode, sequencers begin executing sequences automatically based on the setting of `default_sequence`. Using `set_config_string` simplifies a test case so that only the `default_sequence` needs to be specified:

```
class read_write_test extends ovm_test;
   ...  // register with factory, define constructor

   // Only need build to define starting sequence
   virtual function void build();
     super.build();

     // Specify the test sequence
     set_config_string( "*.intf_sequencer",
                        "default_sequence",
                        "read_write_seq" );

     // Create the environment for the test
     m_env = my_env::type_id::create(...);
   endfunction : build

endclass : read_write_test
```

- Set the `count` to 0 using `set_config_int` if the test case only needs to execute one specific sequence. For example,

```
// Execute only the "my_seq" sequence
set_config_string( "*", "default_sequence", "my_seq" );
set_config_int( "*.sequencer", "count", 0);
```

Gotchas

- As of OVM 2.0, the `ovm_sequencer` implements only PULL mode, meaning that the `ovm_driver` controls or pulls the sequence items from the sequencer. For the sequencer to control the interaction (i.e., PUSH mode), user modifications are required.
- By default, a sequencer will execute the `ovm_random_sequence`, which is a random selection of sequences from the sequencer's sequence library. The number of sequences executed will be between 1 and `max_random_count` (default 10). These sequences will execute in addition to the test case sequence unless `count` is set to 0 and the `default_sequence` is set to a sequence test sequence (see examples above in Tips section).

See also

Sequence, Virtual Sequences, ovm_sequence, ovm_sequence_item, Special Sequences, ovm_driver, Sequence Action Macros, Sequencer Interface and Ports

In OVM, the passing of transactions between sequencers and drivers happens through a sequencer interface export/port pair. A sequencer producing items or sequences contains a sequence item export and a driver consuming items or sequences contains a sequence item port.

The OVM library provides an interface class and an associated export and port to handle the communication between sequencers and drivers. The interface is defined by class `sqr_if_base`.

An instance of class `ovm_seq_item_pull_port` named `ovm_seq_item_port` is a member of `ovm_driver`. It enables the driver to call interface methods such as `get_next_item` and `item_done`, which pull sequence items from the sequencer's item queue.

An instance of class `ovm_seq_item_pull_imp` named `ovm_seq_item_export` is a member of `ovm_sequencer`. It provides the implementation of the sequencer interface methods which manage the queuing of sequence items and sequencer synchronization.

Declarations

```
virtual class sqr_if_base #(type T1=ovm_object, T2=T1);

class ovm_seq_item_pull_port #(type REQ=int, type RSP=REQ)
  extends ovm_port_base #(sqr_if_base #(REQ, RSP));

class ovm_seq_item_pull_export #(type REQ=int,
                                 type RSP=REQ)
  extends ovm_port_base #(sqr_if_base #(REQ, RSP));

class ovm_seq_item_pull_imp #(type REQ=int,
                              type RSP=REQ, type IMP=int)
  extends ovm_port_base #(sqr_if_base #(REQ, RSP));
```

Methods

sqr_if_base

`virtual task` **`get_next_item`**`(` `  output T1 t);`	Blocks until an item is returned from the sequencer
`virtual task` **`try_next_item`**`(` `  output T1 t);`	Returns immediately an item if available, otherwise, *null*
`virtual function void` **`item_done`**`(` `  input T2 t = null);`	Indicates to the sequencer that the driver is done processing the item

`virtual task` **`wait_for_sequences`**`();`	Calls the `wait_for_sequences` task of the connected sequencer (see ovm_sequencer)
`virtual function bit` **`has_do_available`**`();`	Returns 1 if item is available, 0 if no item available
`virtual function void` **`put_response`**`(input T2 t);`	Puts a response into the sequencer queue
`virtual task` **`get`**`(output T1 t);`	Blocks until item is returned from sequencer. Call `item_done` before returning.
`virtual task` **`peek`**`(output T1 t);`	Blocks until item is returned from sequencer. Does not remove item from sequencer fifo
`virtual task` **`put`**`(input T2 t);`	Sends response back to sequencer

ovm_seq_item_pull_port

`function` **`new`** `(string name,`
`  ovm_component parent,`
`  int min_size=0, int max_size=1);`
	Constructor

Plus implementation of sqr_if_base methods

ovm_seq_item_pull_export

`function` **`new`** `(string name,`
`  ovm_component parent,`
`  int min_size=0, int max_size=1);`
	Constructor

Plus implementation of sqr_if_base methods

ovm_seq_item_pull_imp

`function` **`new`** `(string name,`
`  ovm_component parent,`
`  int min_size=0, int max_size=1);`
	Constructor

Plus implementation of sqr_if_base methods

Example

```
//
// Demonstrate the use of ovm_seq_item_pull_port and
// ovm_seq_item_pull_imp between a driver and sequencer
//
class my_driver extends ovm_driver #(my_trans);
  ...
  task run();
    forever begin
      // Pull a sequence item from the interface
      seq_item_port.get_next_item( req );

      ...     // Apply transaction to DUT interface

      // Indicate that item is done
      seq_item_port.item_done( rsp );

    end
  endtask : run
endclass : my_driver

// Connect the sequencer and driver together
class my_env extend ovm_env;
  ...

  function void connect;
    my_drv.seq_item_port.connect(
                    my_seqr.seq_item_export );
  endfunction : connect
endclass : my_env
```

Tips

A driver can call `get_next_item` multiple times before indicating `item_done` to the sequencer (in other words, there is no one-to-one correspondence of function calls).

See also

ovm_driver, ovm_sequencer, Virtual Sequences, ovm_sequence

Special Sequences

OVM defines several special sequences that are created automatically using the sequencer macros. When `` `ovm_update_sequence_lib `` or `` `ovm_update_sequence_lib_and_item(USER_ITEM) `` are declared in a sequencer, the sequence library is populated with `ovm_random_sequence` and `ovm_exhaustive_sequence`. Both special sequences operate on all the other sequences in the sequencer's sequence library (excluding each other). For example, `ovm_random_sequence` randomly selects a number of sequences between `count` and `max_random_count` (see `ovm_sequencer`). The `ovm_exhaustive_sequence` randomly executes all of the sequences in the sequencer's sequence library.

A third special sequence is available that executes a single randomized sequence item called `ovm_simple_sequence`. The `ovm_simple_sequence` is not used by virtual sequencers since virtual sequencers do not operate on sequence items. The `` `ovm_update_sequence_lib_and_item( USER_ITEM) `` macro adds this special sequence to a regular sequencer's library and registers the `USER_ITEM` as its default sequence item type.

Each sequencer's or virtual sequencer's `default_sequence` variable is set by default to "ovm_random_sequence". On startup, each sequencer executes the sequence assigned to `default_sequence`, provided its `count` variable is not set to 0. Therefore, by default, a sequencer will automatically start executing the `ovm_random_sequence` sequence even if nothing else is specified.

Declaration

```
class ovm_random_sequence
  extends ovm_sequence #(ovm_sequence_item);;

class ovm_exhaustive_sequence
  extends ovm_sequence #(ovm_sequence_item);;

class ovm_simple_sequence
  extends ovm_sequence #(ovm_sequence_item);;
```

Methods

ovm_random_sequence

`function new(` `  string name ="ovm_random_sequence");`	Constructor

ovm_exhaustive_sequence

`function new(` `  string name="ovm_exhaustive_sequence");`	Constructor

ovm_simple_sequence

`function new( string name="ovm_simple_sequence");`	Constructor

Example

```
// Use the ovm_exhaustive_sequence in an exhaustive test
class exhaustive_test extends ovm_test;
  ...
  task run();
    // Set all sequencers to exhaustively execute all
    // sequences in every sequencer's library
    set_config_string( "*",
                       "default_sequence",
                       "ovm_exhaustive_sequence");
  endtask : run
endclass : exhaustive_test
```

Tips

- Use `set_config_string` to override the `default_sequence` of a sequencer in order to the specify a specific test sequence.
- Use `set_config_int` to override `count` and `max_random_count` to control the behavior of the `ovm_random_sequence` sequence..

Gotchas

- By default, the `ovm_random_sequence` will execute even if a test case executes another set of sequences. In cases where this is not desired, set the sequencer's `count` to 0 and `default_sequence` to the test sequence.
- Take care not to use `` `ovm_update_sequence_lib_and_item( USER_TYPE) `` in a virtual sequencer since it adds `ovm_simple_sequence` to the virtual sequencer's sequence library. No `ovm_simple_sequence` should be defined for a virtual sequencer since virtual sequencers do not operate on sequence items.

See also

Sequences, ovm_sequence, ovm_sequencer, Virtual Sequences

ovm_subscriber

`ovm_subscriber` is a convenient base class for a user-defined *subscriber* (an analysis component that will receive transaction data from another component's analysis port).

A subscriber that is derived from `ovm_subscriber` must override its inherited `write` method, which will be called automatically whenever a transaction data item is written to a connected analysis port. The `analysis_export` member of a subscriber instance should be connected to the analysis port that produces the data (typically on a monitor or other verification component).

Declaration

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

Methods

`function` **`new`**`( string name, ovm_component parent);`	Constructor
`pure virtual function void` **`write`**`(` *`transaction_type`* `t);`	Override this method to implement your subscriber's behavior when it receives a transaction data item

Members

`ovm_analysis_imp #(`*`transaction_type, this_type`*`)` **`analysis_export`**`;`	Implementation of an analysis export, ready for connection to an analysis port

Example

```
// Define a specialized subscriber class. This class does nothing except to
// report all transactions it receives.
class example_subscriber extends
    ovm_subscriber #(example_transaction);
  int transaction_count;
  function new(string name, ovm_component parent);
    super.new(name, parent);
  endfunction
  function void write (example_transaction t);
    ovm_report_info("WRITE", $psprintf(
       "Received transaction number %0d:\n%s",
       transaction_count++, t.sprint() ) );
  endfunction
endclass

// In the enclosing environment class:
class subscriber_test_env extends ovm_env;
  example_subscriber m_subscriber;
  example_monitor m_monitor;  // see article ovm_monitor
  ...
  function void build();
    ...
    $cast ( m_monitor,
         create_component("example_monitor",
                          "m_monitor") );
    $cast ( m_subscriber,
         create_component("example_subscriber",
                          "m_subscriber") );
    ...
  endfunction
  function void connect();
    // Connect monitor's analysis port (requires)
    // to subscriber's export (provides)
    m_monitor.monitor_ap.connect (
        m_subscriber.analysis_export );
  endfunction
  ...
endclass
```

Tips

- Use `ovm_subscriber` as the base for any custom class that needs to monitor a single stream of transactions. Typical uses include coverage collection, protocol checking, or data logging to a file. It is not appropriate for end-to-end checkers, since these typically need to monitor transaction streams from more than one analysis port (see *Gotchas* below). For such applications, one of the built-in comparator components such as `ovm_in_order_class_comparator` or `ovm_algorithmic_comparator` may be appropriate.

Gotchas

- Unless one of the built-in comparator components meets your need, a component that needs to subscribe to more than one analysis port must be created as a specialized extension of `ovm_component`, with a separate `ovm_analysis_export` for connection to each analysis port that will provide data. Classes derived from `ovm_subscriber` are applicable only for monitoring the output from a single analysis port (but note that a custom comparator could contain multiple subscriber members that implement the `write` function for each of its `analysis_exports`).
- The argument for the overridden `write` function MUST be named "`t`".

See also

ovm_in_order_*_comparator, ovm_analysis_export

A class derived from `ovm_test` should be used to represent each test case. A test will create and configure the environment(s) required to verify particular features of the device under test (DUT). There will typically be multiple test classes associated with a testbench: a single test object is created from one of these at the start of each simulation run. This approach separates the testbench from individual test cases and improves its reusability. `ovm_test` is itself derived from `ovm_component` so a test may include a `run` task. A test class is sometimes defined, but never explicitly instantiated, in the top-level testbench module (a test class cannot be defined in a package if it needs to include hierarchical references).

The `ovm_top.run_test` task or the global `run_test` task is called from an `initial` block in the top-level testbench module to instantiate a test (using the factory) and then run it (`run_test` is a wrapper that calls `ovm_top.run_test`.)

The test's `build` method creates the top-level environment(s).

The simulator's command-line plusarg `+OVM_TESTNAME=`*`testname`* specifies the name of the test to run (a name is associated with a test by registering the test class with the factory). If this plusarg is not used, then the `test_name` argument of `run_test` may be used to specify the test name instead. If no test name is given, a default test that does nothing will be run and no environment will be created.

Configuration and/or factory overrides may be used within the test to customize a reusable test environment (or any of its components) without having to modify the environment code.

Declaration

```
virtual class ovm_test extends ovm_component
```

Methods

`function new ( string name, ovm_component parent);`	Constructor

Also, inherited methods, including `build`, `configure`, `connect`, `run`

Example

This example shows a test being defined and run from a top-level module.

```
module top;

  class test1 extends ovm_test;
    ...
    function void build();
      // Create environment
    endfunction: build

    function void connect();
      // Connect test environment’s virtual interface to the DUT’s interface
      m_env.m_mon.m_bus_if = tf.cpu_if.mon;
    endfunction: connect

    task run();
      // Call methods in environment to control test (optional)
    endtask: run

    // Register test with factory
    `ovm_component_utils(test1)
  endclass: test1

  initial begin
    run_test("test1"); // Use test1 by default
                       // Can override using +OVM_TESTNAME
  end

  // Contains DUT, DUT interface, clock/reset, ancillary modules etc.
  test_harness tf ();

endmodule
```

Tips

- Write a new test class for each test case in your verification plan.
- Keep the test classes simple: functionality that is required for every test should be part of the testbench and not repeated in every test class.
- It is a good idea to start with a "default" test that provides random stimulus before spending time developing more directed tests.

- Use the `connect` method to connect the virtual interface in the driver to the DUT's interface.
- If you want to declare a test in a package, you will need to wrap the DUT's interface in a class (as described in the **Virtual Interface Wrapper** article). By doing this, you will avoid using hierarchical names in the test.
- The name of the test instance is "`ovm_test_top`".

Gotchas

- Do not forget to register the test with the factory, using `` `ovm_component_utils ``.
- Do not call the `set_inst_override` member function (inherited from `ovm_component`) for a top-level test.

See also

ovm_env, Configuration, Virtual Interface Wrapper, ovm_factory

tlm_analysis_fifo

Class `tlm_analysis_fifo` is provided for use as part of an analysis component that expects to receive its data from an analysis port. It is derived from `tlm_fifo` and adds a `write` member function. Only the methods directly relevant to analysis FIFOs are described here; for full details, consult the article on **tlm_fifo**.

The "put" end of an analysis FIFO is intended to be written-to by an analysis port. Consequently, an analysis FIFO is invariably set up to have unbounded depth so that it can never block on `write`. The FIFO's `try_put` method is re-packaged as a `write` function, which is then exposed through an `analysis_export` that can in its turn be connected to an `analysis_port` on a producer component. The `get_export` (or similar) at the other end of the FIFO is typically connected to an analysis component, such as a scoreboard, that may need to wait for other data before it is able to process the FIFO's output.

The type of data carried by the analysis FIFO is set by a type parameter.

Declaration

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

Methods

`function new(string name, ovm_component parent = null);`	Constructor
`function void write(input T t);`	Puts transaction on to the FIFO; cannot block (FIFO is unbounded)
`function void flush() ;`	Clears FIFO contents
`task get(output T t);`	Blocks if the FIFO is empty
`function bit try_get(output T t);`	Returns 0 if FIFO empty
`function bit try_peek(output T t);`	Returns 0 if FIFO empty
`function bit try_put(input T t);`	Returns 0 if FIFO full
`function int used();`	number of items in FIFO
`function bit is_empty();`	returns 1 if FIFO empty

Members

`ovm_analysis_imp #(T, tlm_analysis_fifo #(T)) analysis_export;`	For connection to an analysis port on another component.

blocking_get_export†; **nonblocking_get_export**†; **get_export**†;	Exports blocking/non-blocking/combined get interface
blocking_peek_export†; **nonblocking_peek_export**†; **peek_export**†;	Exports blocking/non-blocking/combined peek interface
blocking_get_peek_export†; **nonblocking_get_peek_export**†; **get_peek_export**†;	Exports blocking/non-blocking/combined get_peek interface
`ovm_analysis_port #( T )` **put_ap**; `ovm_analysis_port #( T )` **get_ap**;	Analysis ports

*†Export type omitted for clarity. `*_export` is an implementation of the corresponding `tlm_*_if` interface (see the article on **TLM Interfaces**)*

Example

Declaring and connecting an analysis FIFO so that the monitor can `write` to it, and the analyzer can `get` from it:

```
class example_env extends ovm_env;
  tlm_analysis_fifo #(example_transaction) an_fifo;
  example_monitor m_monitor;     // has analysis port
  example_analyzer m_analyzer;   // has "get" port
  ...
  virtual function void build();
    $cast(m_monitor, create_component(...));
    $cast(m_analyzer, create_component(...));
    an_fifo = new("an_fifo", this);
  endfunction
  virtual function void connect();
    m_monitor.monitor_ap.connect(an_fifo.analysis_export);
    m_analyzer.get_port.connect(an_fifo.get_export);
  endfunction
  ...
endclass
```

Tips

It is not necessary to use a `tlm_analysis_fifo` to pass data to an analysis component that does not consume time, such as a coverage checker or logger. In such a situation it is preferable to derive the analysis component from `ovm_subscriber` so that it can provide its own `write` method directly. Use analysis FIFOs to buffer analysis input to a component such as a scoreboard that may not be able to service each analysis transaction immediately, because it is waiting for a corresponding transaction on some other channel.

Gotchas

If the type parameter of a `tlm_analysis_fifo` is a class, the FIFO stores object handles. If an object is updated after it has been put into a FIFO but before it is retrieved, peek and get will return the updated object. If the FIFO is required to transport an object with its state preserved, the object should first be "cloned" and the clone written to the FIFO instead.

Do not use the `put` or `try_put` methods inherited from `tlm_fifo` to write data to an analysis FIFO (they are not members of `ovm_analysis_port`).

See also

tlm_fifo, TLM Interfaces, ovm_subscriber, ovm_analysis_port, ovm_analysis_export

Class `tlm_fifo` is provided for use as a standard channel. Data is written and read in first-in first-out order. The FIFO depth may be set by a constructor argument – the default depth is 1.

The `tlm_fifo` channel has both blocking and non-blocking *put* and *get* methods. A put operation adds one data item to the front of the FIFO. A get operation retrieves and removes the last data item from the FIFO. Blocking and non-blocking peek methods are also provided that retrieve the last data item without removing it from the FIFO. Exports are provided to access the channel methods. Analysis ports enable the data associated with each put or get operation to be monitored.

The type of data carried is set by a type parameter.

Declaration

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

Methods

`function new(string name, ovm_component parent = null, int size_ = 1);`	Constructor
`function bit can_get();`	Returns 1 if FIFO is not empty
`function bit can_peek();`	Returns 1 if FIFO is not empty
`function bit can_put();`	Returns 1 if FIFO is not full
`function void flush() ;`	Clears FIFO contents
`task get(output T t);`	Blocks if the FIFO is empty
`task peek(output T t);`	Blocks if the FIFO is empty
`task put(input T t);`	Blocks if the FIFO is full
`function bit try_get(output T t);`	Returns 0 if FIFO empty
`function bit try_peek(output T t);`	Returns 0 if FIFO empty
`function bit try_put(input T t);`	Returns 0 if FIFO full
`function int size();`	Returns FIFO depth
`function int used();`	Number of items in FIFO
`function bit is_empty();`	Returns 1 if FIFO empty
`function bit is_full();`	Returns 1 if FIFO full

Members

blocking_put_export†; **nonblocking_put_export**†; **put_export**†;	Exports blocking/non-blocking/combined put interface
blocking_get_export†; **nonblocking_get_export**†; **get_export**†;	Exports blocking/non-blocking/combined get interface
blocking_peek_export†; **nonblocking_peek_export**†; **peek_export**†;	Exports blocking/non-blocking/combined peek interface
blocking_get_peek_export†; **nonblocking_get_peek_export**†; **get_peek_export**†;	Exports blocking/non-blocking/combined get_peek interface
ovm_analysis_port #(T) **put_ap**; ovm_analysis_port #(T) **get_ap**;	Analysis ports

†*Export type omitted for clarity.* `*_export` *is an implementation of the corresponding* `tlm_*_if` *interface (see the article on* ***TLM Interfaces****)*

Example

Declaring port to connect to `tlm_fifo`

```
ovm_get_port #(basic_transaction) m_trans_in;
```

Calling `tlm_fifo` method via port

```
m_trans_in.get(tx);
```

Using `tlm_fifo` between `ovm_random_stimulus` and a driver

```
class verif_env extends ovm_env;
  ovm_random_stimulus #(basic_transaction) m_stimulus;
  dut_driver m_driver;
  tlm_fifo #(basic_transaction) m_fifo;
  ...
  virtual function void build();
   super.build();
   m_stimulus = new ("m_stimulus",this);
   m_fifo = new("m_fifo",this);
```

```
   $cast(m_driver,
         ovm_factory::create_component("dut_driver","",
                                       "m_driver",this) );
  endfunction: build

  virtual function void connect;
   m_stimulus.blocking_put_port.connect(m_fifo.put_export);
   m_driver.m_trans_in.connect(m_fifo.get_export);
  endfunction: connect
  ...
endclass: verif_env
```

Tips

The `tlm_fifo` uses a SystemVerilog mailbox class as its internal buffer so an infinite depth FIFO can be created by setting the size constructor argument to 0.

Gotchas

If the type parameter of a `tlm_fifo` is a class, the FIFO stores object handles. If an object is updated after it has been put into a FIFO but before it is retrieved, peek and get will return the updated object. If the FIFO is required to transport an object with its state preserved, the object should first be “cloned” and the clone written to the FIFO instead.

See also

tlm_analysis_fifo, TLM Interfaces

TLM Interfaces

OVM provides a set of interface classes (not to be confused with SystemVerilog interfaces) that provide standard transaction-level communication methods for ports and exports (they are based on the SystemC TLM 1.0 library). These methods exist in "blocking" and "non-blocking" forms. Blocking methods may wait before returning and are always tasks. Non-blocking methods are not allowed to wait and are implemented as functions. The interface methods are virtual tasks and functions that are not intended to be called directly by applications.

Three types of operation are supported. The semantics are as follows:

- *Putting* a transaction into a channel, e.g. a request message from an initiator. This adds the transaction to the current channel but does not overwrite the existing contents.
- *Getting* a transaction from a channel, e.g. a response message from a target. This removes the transaction from the channel.
- *Peeking* at a transaction in the channel without removing it.

Unidirectional "blocking interfaces" contain a single task. Unidirectional "non-blocking interfaces" contain one function to access a transaction and typically one or more other functions to test the state of the channel conveying the transaction. For convenience, "combined interfaces" are provided which include all of the methods from related blocking and non-blocking interfaces.

Bidirectional interfaces combine a pair of related unidirectional interfaces so that request and response transactions can be sent between an initiator and target using a single bidirectional channel.

TLM ports and exports are associated with each of the TLM interfaces. They provide a mechanism to decouple the initiator and target of a transaction, providing encapsulation and improving the reusability.

The type of transaction carried by an interface, port or export is set by a type parameter.

Declarations

Unidirectional Interfaces

```
virtual class tlm_put_if #( type T = int )
extends tlm_if_base #(T, T);
```

Bidirectional Interfaces

```
virtual class tlm_master_if
#( type REQ = int ,type RSP = int )
extends tlm_if_base #(REQ, RSP);
```

Methods

Blocking unidirectional interfaces

`tlm_blocking_put_if`

`virtual task **put**(input T t);`	Blocks until t can be accepted

`tlm_blocking_get_if`

`virtual task **get**(output T t);`	Blocks until t can be fetched

`tlm_blocking_peek_if`

`virtual task **peek**(output T t);`	Blocks until t is available

`tlm_blocking_get_peek_if`

`virtual task **get**(output T t);`	Blocks until t can be fetched
`virtual task **peek**(output T t);`	Blocks until t is available

Non-blocking unidirectional interfaces

`tlm_nonblocking_put_if`

`virtual function bit **try_put**( input T t);`	Returns 1 if successful, otherwise 0
`virtual function bit **can_put**();`	Returns 1 if transaction would be accepted, otherwise 0

`tlm_nonblocking_get_if`

`virtual function bit **try_get**( output T t);`	Returns 1 if t successful, otherwise 0
`virtual function bit **can_get**();`	Returns 1 if transaction is available, otherwise 0

`tlm_nonblocking_peek_if`

`virtual function bit **try_peek**( output T t);`	Returns 1 if successful, otherwise 0

`virtual function bit` **`can_peek`**`();`	Returns 1 if transaction is available, otherwise 0

`tlm_nonblocking_get_peek_if`

`virtual function bit` **`try_get`**`( output T t);`	Returns 1 if t successful, otherwise 0
`virtual function bit` **`can_get`**`();`	Returns 1 if transaction is available, otherwise 0
`virtual function bit` **`try_peek`**`( output T t);`	Returns 1 if successful, otherwise 0
`virtual function bit` **`can_peek`**`();`	Returns 1 if transaction is available, otherwise 0

Combined Interfaces

`tlm_put_if`

`virtual task` **`put`**`(input T t);`	Blocks until t can be accepted
`virtual function bit` **`try_put`**`( input T t);`	Returns 1 if successful, otherwise 0
`virtual function bit` **`can_put`**`();`	Returns 1 if transaction would be accepted, otherwise 0

`tlm_get_if`

`virtual task` **`get`**`(output T t);`	Blocks until t can be fetched
`virtual function bit` **`try_get`**`( output T t);`	Returns 1 if t successful, otherwise 0
`virtual function bit` **`can_get`**`();`	Returns 1 if transaction is available, otherwise 0

`tlm_peek_if`

`virtual task` **`peek`**`(output T t);`	Blocks until t is available
`virtual function bit` **`try_peek`**`( output T t);`	Returns 1 if successful, otherwise 0
`virtual function bit` **`can_peek`**`();`	Returns 1 if transaction is available, otherwise 0

`tlm_get_peek_if`

`virtual task get(output T t);`	Blocks until t can be fetched
`virtual function bit try_get( output T t);`	Returns 1 if t successful, otherwise 0
`virtual function bit can_get();`	Returns 1 if transaction is available, otherwise 0
`virtual task peek(output T t);`	Blocks until t is available
`virtual function bit try_peek( output T t);`	Returns 1 if successful, otherwise 0
`virtual function bit can_peek();`	Returns 1 if transaction is available, otherwise 0

Bidirectional Interfaces

`tlm_blocking_master_if`

`virtual task put(input REQ t);`	Blocks until t can be accepted
`virtual task get(output RSP t);`	Blocks until t can be fetched
`virtual task peek(output RSP t);`	Blocks until t is available

`tlm_nonblocking_master_if`

`virtual function bit try_put( input REQ t);`	Returns 1 if successful, otherwise 0
`virtual function bit can_put();`	Returns 1 if request would be accepted, otherwise 0
`virtual function bit try_get( output RSP t);`	Returns 1 if t successful, otherwise 0
`virtual function bit can_get();`	Returns 1 if response is available, otherwise 0
`virtual function bit try_peek( output RSP t);`	Returns 1 if successful, otherwise 0
`virtual function bit can_peek();`	Returns 1 if response is available, otherwise 0

`tlm_master_if`

`virtual task put(input REQ t);`	Blocks until t can be accepted

`virtual function bit` **`try_put`**`( input REQ t);`	Returns 1 if successful, otherwise 0
`virtual function bit` **`can_put`**`();`	Returns 1 if request would be accepted, otherwise 0
`virtual task` **`get`**`(output RSP t);`	Blocks until t can be fetched
`virtual function bit` **`try_get`**`( output RSP t);`	Returns 1 if t successful, otherwise 0
`virtual function bit` **`can_get`**`();`	Returns 1 if response is available, otherwise 0
`virtual task` **`peek`**`(output RSP t);`	Blocks until t is available
`virtual function bit` **`try_peek`**`( output RSP t);`	Returns 1 if successful, otherwise 0
`virtual function bit` **`can_peek`**`();`	Returns 1 if response is available, otherwise 0

`tlm_blocking_slave_if`

`virtual task` **`put`**`(input RSP t);`	Blocks until t can be accepted
`virtual task` **`get`**`(output REQ t);`	Blocks until t can be fetched
`virtual task` **`peek`**`(output REQ t);`	Blocks until t is available

`tlm_nonblocking_slave_if`

`virtual function bit` **`try_put`**`( input RSP t);`	Returns 1 if successful, otherwise 0
`virtual function bit` **`can_put`**`();`	Returns 1 if response would be accepted, otherwise 0
`virtual function bit` **`try_get`**`( output REQ t);`	Returns 1 if t successful, otherwise 0
`virtual function bit` **`can_get`**`();`	Returns 1 if request is available, otherwise 0
`virtual function bit` **`try_peek`**`( output REQ t);`	Returns 1 if successful, otherwise 0
`virtual function bit` **`can_peek`**`();`	Returns 1 if request is available, otherwise 0

`tlm_slave_if`

`virtual task **put**(input RSP t);`	Blocks until t can be accepted
`virtual function bit **try_put**( input RSP t);`	Returns 1 if successful, otherwise 0
`virtual function bit **can_put**();`	Returns 1 if response would be accepted, otherwise 0
`virtual task **get**(output REQ t);`	Blocks until t can be fetched
`virtual function bit **try_get**( output REQ t);`	Returns 1 if t successful, otherwise 0
`virtual function bit **can_get**();`	Returns 1 if request is available, otherwise 0
`virtual task **peek**(output REQ t);`	Blocks until t is available
`virtual function bit **try_peek**( output REQ t);`	Returns 1 if successful, otherwise 0
`virtual function bit **can_peek**();`	Returns 1 if request is available, otherwise 0

`tlm_blocking_transport_if`

`virtual task **transport**( input REQ request, output RSP response );`	Blocks until request accepted and response is returned

`tlm_nonblocking_transport_if`

`virtual function bit **nb_transport**( input REQ request, output RSP response);`	Returns 1 if request accepted and response returned immediately, otherwise 0

`tlm_transport_if`

`virtual task **transport**( input REQ request, output RSP response);`	Blocks until request accepted and response is returned
`virtual function bit **nb_transport**( input REQ request, output RSP response);`	Returns 1 if request accepted and response returned immediately, otherwise 0

Example

Declaring port to connect to `tlm_req_rsp_channel`

```
ovm_master_port #(my_req,my_rsp) m_master;
```

Calling `tlm_req_rsp_channel` methods via port

```
m_master.put(req1);
m_master.get(req1);
```

Use of `master_imp` and `slave_imp` in `tlm_req_rsp_channel` to implement exports.

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

  typedef tlm_req_rsp_channel #( REQ , RSP ) this_type;

  protected  tlm_fifo #( REQ ) m_request_fifo;
  protected tlm_fifo #( RSP ) m_response_fifo;

  ...
  ovm_master_imp
  #( REQ , RSP , this_type , tlm_fifo #( REQ ) ,
     tlm_fifo #( RSP ) ) master_export;

  ovm_slave_imp
  #( REQ , RSP ,this_type ,tlm_fifo #( REQ ) ,
    tlm_fifo #( RSP ) ) slave_export;
  ...
```

Exports instantiated in function called from `tlm_req_rsp_channel` constructor:

```
master_export = new(  "master_export" , this ,
                  m_request_fifo, m_response_fifo );

slave_export = new(   "slave_export" , this ,
                  m_request_fifo , m_response_fifo );
```

Tips

- It is often easier to use the combined exports when creating a channel since these can be connected to blocking, non-blocking or combined ports..

Gotchas

- Remember that the export that provides the actual implementation of the interface methods should use `ovm_*_imp` rather than `ovm_*export`.
- An `ovm_*_imp` instance requires a type parameter to set the type of the class that will define its interface methods (this is often its parent class). This object should also be passed as an argument to its constructor

See also

tlm_fifo, Ports and Exports

ovm_transaction

ovm_transaction is a virtual class that should be used as the base class for transactions in a OVM environment. It is derived from ovm_object. It adds functions for managing transactions and hooks to support transaction recording.

Transactions are often used as the stimulus in an OVM testbench. The name of the component that initiates a transaction may be recorded as a member field. Knowing where each transaction in a complex testbench originates from can be a useful debugging aid. The initiator name can be set as a constructor argument or by calling the set_initiator function. A get_initiator function is provided to retrieve the initiator name.

The start and end time of a transaction may be recorded and stored within the transaction using the functions begin_tr and end_tr respectively. By default, the time recorded is the current simulation time: if a different time is required, this can be specified as a function argument. Many transaction-level models support the concept of a transaction being accepted by a target some time after it has been sent. An accept_tr function is provided to indicate when this occurs. The start, end and acceptance of a transaction are notified by events named "begin", "end" and "accept" respectively. These events are contained within a pool of events managed by each transaction. The event pool can be accessed by calling the get_event_pool function. Callback functions are provided that correspond to the "begin", "end" and "accept" events. They are virtual functions that by default do nothing but may be overridden in a derived class.

A transaction recording interface is provided but is not implemented in OVM itself. This is intended to be implemented by OVM tools, where required.

Declaration

```
virtual class ovm_transaction extends ovm_object;
```

Methods

function **new**(string name="", ovm_component initiator=null);	Constructor
function void **set_initiator**(ovm_component initiator);	Sets initiator (component that creates transaction)
function ovm_component **get_initiator**();	Get initiator
function void **accept_tr**(time accept_time=0);	Accept transaction (triggers "accept" event)
function integer **begin_tr**(time begin_time=0);	Indicate start of transaction (triggers "begin" event)

`function integer begin_child_tr( time begin_time=0, integer parent_handle=0);`	Indicate start of child transaction (triggers `"begin"` event)
`function void end_tr( time end_time=0, bit free_handle=1);`	Indicate end of transaction (triggers `"end"` event)
`function integer get_tr_handle();`	Returns transaction handle
`function void enable_recording( string stream);`	Enable (start) recording to specified stream
`function void disable_recording();`	Disable (stop) recording
`function bit is_recording_enabled();`	Check if recording enabled
`function bit is_active();`	Returns 1 if transaction started but not yet ended, otherwise 0
`virtual protected function void do_accept_tr();`	User-defined callback function
`virtual protected function void do_begin_tr();`	User-defined callback function
`virtual protected function void do_end_tr();`	User-defined callback function
`function ovm_event_pool get_event_pool();`	Returns the local event pool
`function time get_begin_time();`	Return transaction begin time
`function time get_end_time();`	Return transaction end time
`function time get_accept_time();`	Returns time that transaction was accepted
`virtual function string convert2string();`	Return transaction as string (calls `sprint()` by default)

Example

```
class basic_transaction extends ovm_transaction;
  rand bit[7:0] m_var1, m_var2;
  static int tx_count = 0;

  function new (string name = "",
                ovm_component initiator=null);
    super.new(name,initiator);
    tx_count++;
```

```
  endfunction: new

  virtual protected function void do_accept_tr();
    ovm_report_info("TRX",$psprintf(
                    "Transaction %0d accepted",tx_count));
  endfunction: do_accept_tr

  virtual protected function void do_end_tr();
    ovm_report_info("TRX",$psprintf(
                    "Transaction %0d ended",tx_count));
  endfunction: do_end_tr

  `ovm_object_utils_begin(basic_transaction)
    `ovm_field_int(m_var1,OVM_ALL_ON + OVM_DEC)
    `ovm_field_int(m_var2,OVM_ALL_ON + OVM_DEC)
  `ovm_object_utils_end
endclass : basic_transaction
```

Generating constrained random transactions:

```
virtual task generate_stimulus(
    basic_transaction t = null, input int max_count = 30 );
  basic_transaction temp;
  ovm_event_pool tx_epool;
  ovm_event tx_end;
  if( t == null ) t = new("trans",this);
  for( int i = 0; (max_count == 0 || i < max_count-1);i++ )
  begin
    assert( t.randomize() );
    $cast( temp , t.clone() );
    ovm_report_info("stimulus generation" ,
                     temp.convert2string() );
    tx_epool = temp.get_event_pool();
    blocking_put_port.put( temp );
    tx_end = tx_epool.get("end");
    tx_end.wait_trigger();
  end
endtask: generate_stimulus
```

Tips

The functions to accept, begin and end transactions are optional – they are only really useful when transaction recording is active.

Gotchas

Each transaction object maintains its own event pool. If an initiator needs to wait for a transaction to be accepted/ended before continuing, it needs to save a copy of the transaction handle to access the associated events.

See also

ovm_object, ovm_sequence_item, ovm_random_stimulus, Component

Utility Macros

OVM defines a set of utility macros for objects and components. These register a class with the OVM factory and define functions required by the factory. Two of these functions are visible to users:

```
function ovm_object create (string name="");
```

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

These macros also provide a wrapper around the *field automation macros*.

Macros

Macros used only with objects:

\`ovm_object_utils(T)	Registers simple object T with factory and defines factory methods
\`ovm_object_utils_begin(T)	Registers simple object T with factory and defines factory methods. May be followed by list of field automation macros
\`ovm_object_utils_end	Terminates list of field automation macros
\`ovm_object_param_utils(T)	Registers parameterized object T with factory and defines factory methods
\`ovm_object_param_utils_begin(T)	Registers parameterized object T with factory and defines factory methods. May be followed by list of field automation macros
\`ovm_object_param_utils_end	Terminates list of field automation macros

Macros used only with components:

\`ovm_component_utils(T)	Registers simple component T with factory and defines factory methods

`ovm_component_utils_begin(T)	Registers simple component T with factory and defines factory methods. May be followed by list of field automation macros
`ovm_component_utils_end	Terminates list of field automation macros
`ovm_component_param_utils(T)	Registers parameterized component T with factory and defines factory methods
`ovm_component_param_utils_begin(T)	Registers parameterized component T with factory and defines factory methods. May be followed by list of field automation macros
`ovm_component_param_utils_end	Terminates list of field automation macros

Macros that do not register objects with the factory and do not define factory methods (used to call field automation macros in abstract base classes or where the default factory methods are not suitable):

`ovm_field_utils_begin(T)	May be followed by list of field automation macros
`ovm_field_utils_end	Terminates list of field automation macros

Example

Registering a transaction with the factory and calling the field automation macros on its fields

```
class basic_transaction extends ovm_sequence_item;
  rand bit[7:0] addr, data;
  ...

  `ovm_object_utils_begin(basic_transaction)
    `ovm_field_int(addr,OVM_ALL_ON)
    `ovm_field_int(data,OVM_ALL_ON)
  `ovm_object_utils_end
endclass : basic_transaction
```

Registering a driver component with the factory and calling the field automation macro for its virtual interface wrapper:

```
class dut_driver extends ovm_driver;
  ovm_get_port #(basic_transaction) m_trans_in;
  if_wrapper if_wr;
  ...
  `ovm_component_utils_begin(dut_driver)
    `ovm_field_object(if_wr,OVM_ALL_ON)
  `ovm_component_utils_end
endclass: dut_driver
```

Tips

- Call the macros at the end of the class definition. This will ensure any members that are referenced by the field automation macros will have been declared.

Gotchas

- Do not use a semicolon after a macro or a compiler error may result.
- Make sure you call `` `ovm_object_utils[_begin/_end] `` with objects and `` `ovm_component[_begin/_end] `` with components.
- Classes that call `` `ovm_field_utils_* `` cannot be built by the factory unless they have `create` and `get_type_name` member functions.
- Parameterized classes must use the `ovm_object_param_utils*` or `ovm_component_param_utils*` versions. See **ovm_component** and **ovm_object**. for more details and examples.

See also

ovm_object, ovm_component, ovm_factory, Field Macros

Virtual interfaces are used in an OVM testbench to connect a class-based environment to a module-based test harness. A virtual interface that is a member of a class (typically a driver or monitor) may be assigned to a SystemVerilog interface that is instantiated in a test harness (or the top-level module that also calls `run_test` to create the OVM environment). The interface is bound to the ports of the device under test (another module).

A virtual interface must be initialized to an interface instance before it is used. If the test class is defined in the top level module, its `connect` method can be used to assign the virtual interfaces to the actual interfaces (this cannot be done from a class within a package as SystemVerilog does not allow hierarchical references within a package). An alternative approach to connect a virtual interface is to include a member function in its parent class that can be called from the top-level module.

Unfortunately, a virtual interface member of a component cannot be configured using the OVM configuration mechanism. A virtual interface wrapper class that is derived from `ovm_object` overcomes this limitation. An easy way of associating the wrapper's virtual interface with an actual interface is to create a wrapper instance in the top-level module and pass the virtual interface as an argument to the wrapper constructor. The wrapper instance can then be associated with a wrapper handle member of one or more components by calling `set_config_object` before `run_test`.

The use of a wrapper enables all of the testbench classes to be defined within a package.

Example

A simple virtual interface wrapper class

```
class if_wrapper extends ovm_object;
  virtual chip_if if1;
  function new(string name,virtual chip_if if_);
    super.new(name);
    if1 = if_;
  endfunction : new
endclass : if_wrapper
```

A driver component that uses the virtual interface wrapper

```
class dut_driver extends ovm_driver;
  if_wrapper if_wr;
  ...
  task run();
    ...
    if_wr.if1.driver.addr = addr;
    if_wr.if1.driver.data = data;
```

```
    ...
  endtask: run

  `ovm_component_utils_begin(dut_driver)
    `ovm_field_object(if_wr,OVM_ALL_ON)
  `ovm_component_utils_end

endclass: dut_driver
```

Creating virtual interface wrapper and configuring the driver component in the top-level module

```
module top;
  ...
  chip_if dut_if();
  chip dut ( .dut_if );
  if_wrapper if_wr = new("if_wr",dut_if);

  initial
    begin
      set_config_object("*.m_driver","if_wr",if_wr,0);
      run_test();
    end
endmodule : top
```

An alternative virtual interface wrapper that can be built using the factory. The virtual interface must be set by calling a member function rather than being passed as a constructor argument.

```
class if_wrapper extends ovm_object;
  virtual chip_if if1;
  function new(string name = "");
    super.new(name);
  endfunction : new

  function void set_if(virtual chip_if if_);
    if1 = if_;
  endfunction : set_if
  `ovm_object_utils(if_wrapper)
endclass : if_wrapper
```

Creating virtual interface wrapper using the factory

```
module top;
  ...
  initial
    begin
      if_wrapper if_wr;
      $cast(if_wr,ovm_factory::create_object(
                                  "if_wrapper",,"if_wr"));
      if_wr.set_if(dut_if);
      set_config_object("*.m_driver","if_wr",if_wr,0);
      run_test();
    end
endmodule : top
```

Tips

- Use virtual interface wrappers rather than virtual interfaces within components.
- Use `set_config_object` to assign the virtual interface wrapper instance to the virtual interface wrapper member of a component. If you make the assignment manually, e.g. in the test class `connect` function, you need to define the test class in the top-level module rather than a package.
- Use modports to manage multiple connections to an interface (e.g. separate modports for driver and monitor classes).
- If multiple interfaces are instantiated within a `generate` loop, you should also create the wrapper and call `set_config_object` within the same `generate`, e.g.

```
genvar i;
generate
  for(i=0; i < `NUM; i++)
    begin: c_gen
      chip_if dut_if(.clk);
      chip dut ( .dut_if, .clk );
      if_wrapper if_wr  = new(
              $psprintf("if_wrapper_%0d",i),dut_if);
      initial set_config_object(
              $psprintf("env%0d.*",i),"if_wr",if_wr,0);
    end
endgenerate
```

Gotchas

- Common mistakes are forgetting to call `set_config_object` to setup the automatic configuration of the component that uses the virtual interface wrapper, or calling it with an incorrect hierarchical name.
- If you forget to call the field automation macro for the virtual interface wrapper member of a component, or you do not call `set_config_object` correctly, you will not see any error message until the component tries to access the virtual interface wrapper (usually during its `run` task). The error reported will be a null reference since the wrapper in the component is not connected.
- You cannot search for a virtual interface wrapper instance by name if it was derived from `ovm_object`. If you need to do this for any reason, you should use `ovm_component` as the base class for the wrapper instead.
- You must call `create_component` rather than `create_object` when using the OVM factory if the wrapper's base class is `ovm_object`.

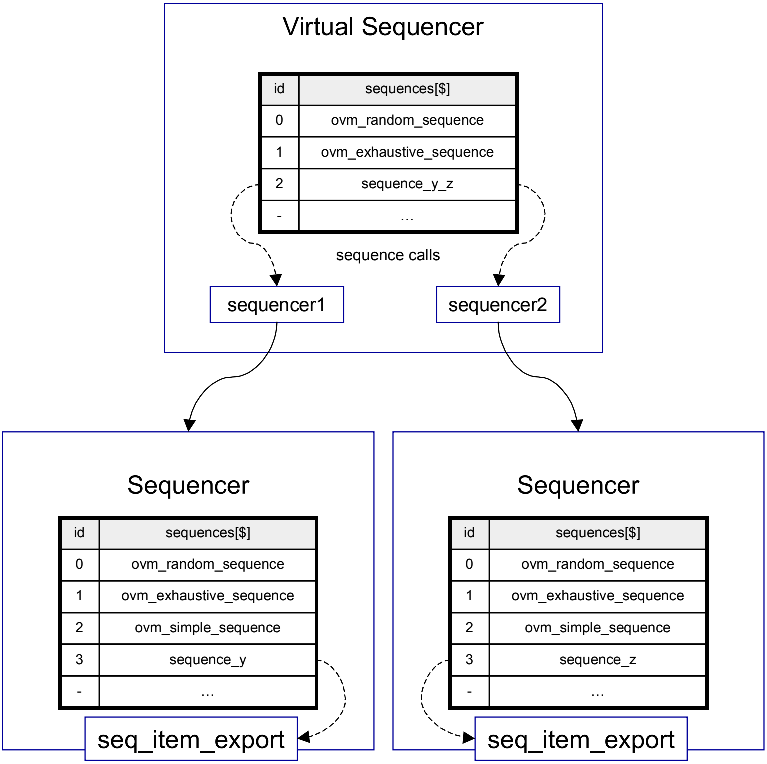
See also

ovm_component, ovm_test, Configuration

A *virtual sequence* is a sequence whose purpose is to manage the behaviour of other sequencers. Like ordinary sequences, a virtual sequence runs on a sequencer. However, the sequencer for a virtual sequence is not linked to a driver, and the virtual sequence does not itself have data items. A sequencer used in this way is commonly called a *virtual sequencer*. A virtual sequencer acts only upon sequences (not sequence items) and is used to coordinate the execution of sequences on one or more other sequencers. A (non-virtual) sequencer can only be associated with a single driver, which typically only controls one device interface. A virtual sequencer provides a mechanism to control the interaction of multiple device interfaces by managing the sequencers that generate their transactions.

For each sequencer that it controls, a virtual sequencer must have a variable holding a reference to that sequencer. It is your responsibility to populate these variables with references to the appropriate subsequencer instances; this should be done as part of the `connect` method of the enclosing component or environment. Note that a virtual sequencer does *not* use TLM interfaces to control its subsequencers.

The following diagram illustrates the connections of a virtual sequencer with multiple sequencers:



Note that no `ovm_virtual_sequence` or `ovm_virtual_sequencer` classes exist; rather, both sequencers and virtual sequencers use sequences derived from the `ovm_sequence` class.

Creating a new virtual sequence is closely similar to creating any ordinary new sequence: : However, virtual sequences invoke an alternative set of sequence action macros: `` `ovm_do_on `` and `` `ovm_do_on_with ``. These macros require an additional argument to specify the sequencer instance that should execute the sequence (there could be multiple instances of a sequencer). This is specified by providing a reference to the element of the virtual sequencer's consumer interface that is has been connected to the desired sequencer producer interface.

Declarations

```
class ovm_sequencer
class ovm_sequence
```

Virtual sequences and sequencers use the same base classes as ordinary sequences and sequencers.

Macros

Utility macros create the sequence library and/or register the sequencer with the OVM factory. The `` `ovm_sequencer_utils* `` macros are used in precisely the same way as they would be for an ordinary sequencer.

```
(1) `ovm_sequencer_utils(sequencer_class_name)

(2) `ovm_sequencer_utils_begin(sequencer_class_name)
    `ovm_sequencer_utils_end
```

These macros allow the `` `ovm_field_* `` macros to be used. For example,

```
`ovm_sequencer_utils_begin(my_sequencer)
   `ovm_field_int(status, OVM_ALL_ON)
  ...
`ovm_sequencer_utils_end
```

`` `ovm_update_sequence_lib `` is specific to virtual sequencers and sequences.

```
(3) `ovm_update_sequence_lib
```

This macro builds the virtual sequence library with the `ovm_random_sequence`, and `ovm_exhaustive_sequence`. Unlike the similar macro `` `ovm_update_sequence_lib_and_item `` that is used for ordinary sequencers, it does not create an `ovm_simple_sequence` because a virtual sequencer cannot operate on sequence items.

This macro must be placed in the constructor of the virtual sequencer as follows:

```
function my_virtual_sequencer::new( string name = "",
                              ovm_component parent = null );
  super.new( name, parent );
  `ovm_update_sequence_lib
endfunction : new
```

Example

```
// Create a virtual sequencer
class my_virtual_sequencer extends ovm_sequencer;
  // Variables to reference the sequencers we will control
  Write_sequencer m_write_sqr;
  Read_sequencer  m_read_sqr;

  // Register this sequencer with the factory
  `ovm_sequencer_utils ( my_virtual_sequencer )

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

      // Create the virtual sequence library
      `ovm_update_sequence_lib
  endfunction : new

endclass : my_virtual_sequencer

// Create a virtual sequence
class read_write_seq extends ovm_sequence;
  my_read_seq    read_seq;  // Sequences on sequencers
  my_write_seq   write_seq;

  // Register sequence in virtual sequencer's library
  `ovm_sequence_utils(read_write_seq, my_virtual_sequencer)

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

```
  // Define the virtual sequence functionality.
  // NOTE the use of `ovm_do_on() instead of `ovm_do() !!
  virtual task body();
    // Write to a bunch of register locations
    for ( int i = 0; i < 32; i += 4 ) begin
      `ovm_do_on_with ( write_seq,
                        p_sequencer.m_write_sqr,
                      { addr == i; })
    end

    // Now read the results on another interface
    `ovm_do ( read_seq, p_sequencer.m_read_sqr )
    ...
  endtask : body
endclass : read_write_seq

...

// Connect the sequencer to the driver
class my_env extends ovm_env;
  ...
  my_virtual_sequencer    m_vseqr;
  Write_sequencer         m_seqr_w;
  Read_sequencer          m_seqr_r;

  // Build the components
  function void build();
    super.build();
    $cast( m_vseqr, create_component( ... ));
    $cast( m_seqr_w, create_component( ... ));
    $cast( m_seqr_r, create_component( ... ));
  endfunction : build

  // Connect up the sequencers to the virtual sequencer
  function void connect;
    m_vseqr.m_write_sqr = m_seqr_w;
    m_vseqr.m_read_sqr = m_seqr_r;
  endfunction : connect
  ...
endclass : my_env
```

Tips

- By default, sequences done by a virtual sequence on a subsequencer will be interleaved with sequences created by the subsequencer itself. It is often useful to suppress the subsequencer's normal activity by configuring its `count` member to zero. For more flexible control over the relationship between virtual sequences and ordinary sequences, your virtual sequencer can use the `grab` and `ungrab` methods of its subsequencers to interrupt their normal activity.

- Use `set_config_string()` to set the default sequence that the sequencer should execute. For example,

  ```
  set_config_string( "*.virtual_seqr",  // Sequencer name
                     "default_sequence",
                     "my_seq" );        // New sequence
  ```

  When in random mode, sequencers begin executing sequences automatically based on the setting of `default_sequence`. Using `set_config_string()` simplifies a test case so that only the `default_sequence` needs to be specified:

  ```
  class read_write_test extends ovm_test;
    ...      // Register with factory, define constructor

    // Only need build() to define starting sequence
    virtual function void build();
      super.build();

      // Specify the test sequence
      set_config_string( "*.virtual_seqr",
        "default_sequence",
        "read_write_vseq" );

      // Create the environment for the test
      $cast( m_env, create_component(...));

    endfunction : build

  endclass : read_write_test
  ```

- Set the `count` to 0 using `set_config_int()` if the test case only needs to execute 1 specific sequence. For example,

  ```
  // Execute only the "my_seq" sequence
  set_config_string( "*", "default_sequence", "my_seq" );
  set_config_int( "*.virtual_seqr", "count", 0);
  ```

- Multiple virtual sequencers can be connected together in the same way that a sequencer connects to a virtual sequencer. A higher-level virtual sequencers contains references to lower-level virtual sequencers, just as a regular virtual sequencer contains references to ordinary sequencers. In this way, multiple layers of stimulus and control can be configured using a hierarchy of virtual sequencers.

Gotchas

- Do not use `` `ovm_do ``, `` `ovm_do_with ``, `` `ovm_rand_send ``, ***etc.*** with virtual sequencers. Instead, use the `` `ovm_do_on `` and `` `ovm_do_on_with `` macros in virtual sequences.
- Use `ovm_sequence` and `ovm_sequencer` for virtual sequences. No `ovm_virtual_sequence` class exists. The `ovm_virtual_sequencer` class (from OVM versions earlier than 2.0) is no longer useful, and is deprecated.
- Virtual sequence action macros such as `` `ovm_do_on `` automatically create and randomize a sequence object. If instead you call `start_sequence` manually, the argument `this_item` must first be allocated (using `new` or *`sequence_type`*`::type_id::create`) and randomized.
- By default, a sequencer will execute the `ovm_random_sequence`, which is a random selection of sequences from the virtual sequencer's sequence library. The number of sequences executed will be between 1 and `max_random_count` (default 10). These sequences will execute in addition to the test case sequence unless `count` is set to zero.
- A virtual sequencer's sequence library does not include `ovm_simple_sequence` because it deals with sequence items.

See also

Sequences, ovm_sequencer, ovm_sequence, Sequence Action Macros, Sequencer Interface and Ports

Index

Free OVM Tutorials

To assist new users in understanding and applying OVM, Doulos has created a number of tutorials, which are available on our website. Please visit **www.doulos.com/knowhow**

The Golden Reference Guide (GRG) series

- SystemC
- SystemVerilog
- ***e***
- PSL
- VHDL
- Verilog

About Doulos

Doulos is the global leader for the development and delivery of world class training solutions for SoC, FPGA and ASIC design and verification. Established in 1990 and fully independent, Doulos sets the industry standard for high quality training in SystemC™, SystemVerilog, ***e***, PSL, Verilog®, VHDL, Perl & Tcl/Tk.

Doulos know-how is delivered worldwide through regularly scheduled classes in major locations in the U.S and Europe, and through in-house training at customer locations. To find out more about the Doulos training portfolio please visit our website **www.doulos.com**