OVM Class Reference

Version 2.0
September 2008

Contents

OVM Class Definitions

The OVM Reference documents all public classes in the OVM library.

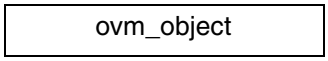
The following class index provides an alphabetized list of each OVM class and the page number to its description. Each description includes an inheritance diagram, a short overview, a method summary, and detailed method descriptions.

Class Index

Table 1-1 Class Index

Base

ovm_void



The `ovm_void` class is the base class for all OVM classes.

The `ovm_void` class is an abstract class with no data members or functions. It allows for generic containers of objects to be created, similar to a `void` pointer in the C programming language. User classes derived directly from `ovm_void` inherit none of the OVM functionality, but such classes may be placed in containers with `ovm` type objects.

Summary

```
virtual class ovm_void;
endclass
```

File

base/ovm_misc.svh

Virtual

Yes

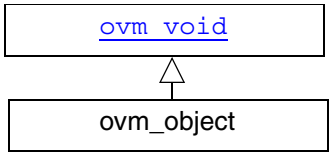
Members

None

Methods

None

ovm_object



The `ovm_object` class is the base class for all OVM data and hierarchical classes. Its primary role is to define a set of methods for such common operations as create, copy, compare, print, and record. Classes deriving from `ovm_object` must implement the pure virtual methods such as `create` and `get_type_name`. Additionally, it is strongly recommended that derived classes override the virtual methods prefixed with `do_`.

Summary

```
virtual class ovm_object extends ovm_void;
    function new (string name="");

    pure virtual function string get_type_name ();
    function string get_name ();
    virtual function string get_full_name ();
    virtual function void set_name (string name);
    virtual function int get_inst_id ();
    static  function int get_inst_count();
    static function ovm_object_wrapper get_type ();

    pure virtual function ovm_object create (string name="");
    virtual function ovm_object clone ();
    function void copy (ovm_object rhs);
    function bit  compare (ovm_object rhs, ovm_comparer comparer=null);
    function void record (ovm_recorder recorder=null);
    function int  pack (ref bit bitstream[], input ovm_packer packer=null);
    function int  unpack (ref bit bitstream[], input ovm_packer packer=null);
    function int  pack_bytes (ref byte bitstream[], input ovm_packer packer=null);
    function int  unpack_bytes (ref byte bitstream[],input ovm_packerpacker=null);
    function int  pack_ints (ref int intstream[], input ovm_packer packer=null);
    function int  unpack_ints (ref int intstream[],input ovm_packer packer=null);
    function void print (ovm_printer printer=null);
    function string sprint (ovm_printer printer=null);

    virtual function void do_print   (ovm_printer printer);
```

```
    virtual function void do_record  (ovm_recorder recorder);
    virtual function void do_copy    (ovm_object rhs);
    virtual function bit  do_compare (ovm_object rhs, ovm_comparer comparer);
    virtual function void do_pack    (ovm_packer packer);
    virtual function void do_unpack  (ovm_packer packer);


    virtual function void set_int_local (string field_name,
                                         ovm_bitstream_t value);
    virtual function void set_object_local (string field_name,
                                            ovm_object value, bit clone=1);
    virtual function void set_string_local (string field_name,
                                            string value);
    static bit use_ovm_seeding = 1;
    function void reseed ();

endclass
```

File

base/ovm_object.svh

Virtual

Yes

Members

`static bit` **`use_ovm_seeding`** `= 1;`

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

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

Methods

new

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

The name is the instance name of the object. If not supplied, the object is unnamed.

clone

```
virtual function ovm_object clone ()
```

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

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

compare

```
function bit compare (ovm_object rhs,
                      ovm_comparer comparer=null)
```

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

The `compare` method is not virtual and should not be overloaded in derived classes. To compare the fields of a derived class, that class should override the `do_compare` method. See do_compare on page 16 for more details.

The optional *`comparer`* argument specifies the comparison *policy*. It allows you to control some aspects of the comparison operation. It also stores the results of the comparison, such as field-by-field miscompare information and the total number of miscompares. If a compare policy is not provided, then the global `ovm_default_comparer` policy is used. See ovm_comparer on page 125 for more information.

copy

```
function ovm_object copy (ovm_object rhs)
```

The `copy` method returns a deep copy of this object.

The `copy` method is not virtual and should not be overloaded in derived classes. To copy the fields of a derived class, that class should override the `do_copy` method. See do_copy on page 17 for more details.

create

```
pure virtual function ovm_object create (string name="")
```

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

A typical implementation is as follows:

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

do_compare

```
virtual function void do_compare (ovm_object rhs,
                                  ovm_comparer comparer)
```

The `do_compare` method is user-definable hook called by the `compare` method on page 15. A derived class should override this method to include its fields in a compare operation.

A typical implementation is as follows:

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

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

The actual comparison should be implemented using the `ovm_comparer` object rather than direct field-by-field comparison. This enables users of your class to customize how comparisons are performed and how much miscompare information is collected. See ovm_comparer on page 125 for more details.

do_copy

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

The do_copy method is the user-definable hook called by the copy method on page 15. A derived class should override this method to include its fields in a copy operation.

A typical implementation is as follows:

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

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

do_pack

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

The do_pack method is the user-definable hook called by the pack method on page 22. A derived class should override this method to include its fields in a pack operation.

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

A typical example of an object packing itself is as follows:

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

```
  endfunction
```

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

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

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

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

do_print

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

The `do_print` method is the user-definable hook called by the `print` method on page 22. A derived class should override this method to include its fields in a print operation.

The `printer` argument is the policy object that governs the format and content of the output. A `do_print` method implementation should not call `$display` directly. It should merely call the appropriate *`printer`* methods for each of its fields. See ovm_printer on page 139 for more information.

A typical implementation is as follows:

```
class mytype extends ovm_object;
  data_obj data;
  int f1;
  function void do_print (ovm_printer printer);
    super.do_print(printer);
    printer.print_field("f1", f1, $bits(f1), OVM_DEC);
    printer.print_object("data", data);
  endfunction
```

do_record

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

The do_record method is the user-definable hook called by the record method on page 23. A derived class should override this method to include its fields in a record operation.

The recorder argument is policy object for recording this object. A do_record implementation should call the appropriate recorder methods for each of its fields. Vendor-specific recording implementations are encapsulated in the recorder policy, thereby insulating user-code from vendor-specific behavior. See ovm_recorder on page 136 for information.

A typical implementation is as follows:

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

do_unpack

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

The do_unpack method is the user-definable hook called by the unpack method on page 25. A derived class should override this method to include its fields in an unpack operation.

The *packer* argument is the policy object for both packing and unpacking. The do_unpack implementation must use the same packer policy, and it must unpack fields in the same order in which they were packed. See ovm_packer on page 130 for more information.

The following implementation corresponds to the example given in do_pack on page 17:

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

```
    packer.unpack_object(myobj);
  endfunction
```

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

- For queues, dynamic arrays, or associative arrays, unpack the number of elements in the array from the 32 bits immediately before unpacking individual elements, as shown above.
- For string data types, unpack into the new string until a null byte is encountered.
- For objects, unpack 4 bits into a byte or int variable. If the value is 0, the target object should be set to null and unpacking continues to the next property, if any. If the least significant bit is 1, then the target object should be allocated and its properties unpacked.

get_name

```
function string get_name ()
```

Returns the name of the object, as provided by the `name` argument in the new function, or as set by way of the `set_name` method.

get_full_name

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

Returns the full hierarchical name of this object. The default implementation concatenates the hierarchical name of the parent, if any, with the short name of this object, as given by `get_name`.

It may be desirable to override the default implementation. For example, some data elements have an anchor in the OVM hierarchy, and for these types of elements it is useful to provide the hierarchical context as part of the name. An example of this is the `ovm_sequence #(REQ,RSP)` type.

get_inst_count

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

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

get_inst_id

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

Returns the object's unique, numeric instance identifier.

get_type

```
static function ovm_object_wrapper get_type ()
```

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

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

For example:

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

Then, to use:

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

This function is implemented for classes that employ the These macros do NOT perform factory registration, implement get_type_name, nor implement the create method. Use this form when you need custom implementations of these two methods, or when you are setting up field macros for an abstract class (i.e. virtual class). on page 258.

get_type_name

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

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

This function must be defined in every derived class.

A typical implementation is as follows:

```
class mytype extends ovm_object;
  ...
```

```
virtual function string get_type_name();
  return "mytype";
endfunction
```

pack

pack_bytes

pack_ints

```
function int pack (ref bit bitstream[],         ovm_packer packer=null)
function int pack_bytes (ref byte bytestream[], ovm_packer packer=null)
function int pack_ints (ref byte intstream[],   ovm_packer packer=null)
```

The `pack` methods bitwise-concatenate this object's properties into an array of bits, bytes, or ints. The methods are not virtual and must not be overloaded. To include additional fields in the pack operation, derived classes should override the `do_pack` method on page 17.

The optional *`packer`* argument specifies the packing *policy*, which governs the packing operation. If a packer policy is not provided, the global `ovm_default_packer` policy is used. See ovm_packer on page 130 for more information.

The return value is the number of bits, bytes, or ints placed into the supplied array. The contents of the array are overwritten. Thus, the total size of the array after the operation is its initial size plus the return value.

print

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

The `print` method deep-prints this object's properties according to an optional `printer` policy. The method is not virtual and must not be overloaded. To include additional fields in the print operation, derived classes should override the `do_print` method on page 18.

The optional `printer` argument specifies the printer policy, which governs the format and content of the output. If a printer policy is not provided explicitly, then the global `ovm_default_printer` policy is used. See ovm_printer on page 139 for more information.

Note: The OVM library provides four predefined printers: `ovm_printer`, `ovm_line_printer`, `ovm_tree_printer`, and the `ovm_table_printer`. The default printer is the `table` printer.

record

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

The `record` method deep-records this object's properties according to an optional `recorder` policy. The method is not virtual and must not be overloaded. To include additional fields in the record operation, derived classes should override the `do_record` method on page 19.

The optional `recorder` argument specifies the recording policy, which governs how recording takes place. If a recorder policy is not provided explicitly, then the global `ovm_default_recorder` policy is used. See ovm_recorder on page 136 for information.

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

reseed

```
function void reseed ()
```

Calls `srandom` on the object to reseed the object using the OVM seeding mechanism to set the seed based on type name and instance name instead of based on instance position in a thread.

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

set_int_local

set_string_local

set_object_local

```
virtual function void set_int_local (string field_name,
                                     ovm_bitstream_t value)
virtual function void set_string_local (string field_name,
                                        string value)
virtual function void set_object_local (string field_name,
                                        ovm_object value,
                                        bit clone=1)
```

These methods provide write access to integral, string, and `ovm_object`-based properties indexed by a *`field_name`* string. The object designer choose which, if any, properties will be accessible, and overrides the appropriate methods depending on the

properties' types. For objects, the optional *clone* argument specifies whether to clone the value argument before assignment.

An example implementation of all three methods is as follows. The global ovm_is_match function is used so that the field_name may contain wildcards.

```
class mytype extends ovm_object;

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

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

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

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

```
endfunction
...
```

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

set_name

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

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

sprint

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

The `sprint` method deep-prints this object's properties just like the `print` method on page 22, except the output is to the return string. The method is not virtual and must not be overloaded. To include additional fields in the print operation, derived classes should override the `do_print` method on page 18.

The optional `printer` argument specifies the printer policy, which governs the format and content of the output. If a printer policy is not provided explicitly, the global `default_printer` policy is used. See ovm_printer on page 139 for details.

Note: The OVM library provides four predefined printers: `ovm_printer`, `ovm_line_printer`, `ovm_tree_printer`, and the `ovm_table_printer`. The default printer is the `table` printer.

unpack

unpack_bytes

unpack_ints

```
function int unpack       (ref bit  bitstream[], ovm_packer packer=null)
function int unpack_bytes (ref byte bytestream[],ovm_packer packer=null)
function int unpack_ints  (ref byte intstream[], ovm_packer packer=null)
```

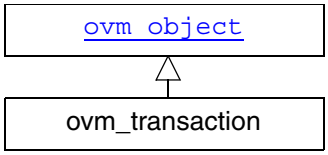
The `unpack` methods extract property values from an array of bits, bytes, or ints. The method of unpacking must exactly correspond to the method of packing. This is assured if (a) the same packer policy is used to pack and unpack, and (b) the order of unpacking is the same as the order of packing used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

The `unpack` methods are fixed (non-virtual) entry points that are called directly by the user. To include additional fields in the unpack operation, derived classes should override the `do_unpack` method on page 19.

The optional *`packer`* argument specifies the packing *policy*, which governs both the pack and unpack operation. If a packer policy is not provided, then the global `default_packer` policy is used. See ovm_packer on page 130 for more information.

The return value is the number of bits, bytes, or ints extracted from the supplied array.

ovm_transaction



The ovm_transaction class is the root base class for OVM transactions. Inheriting all the methods of ovm_object, ovm_transaction adds a timing and recording interface.

Summary

```
virtual class ovm_transaction extends ovm_object;

  function new (string name="");
  virtual function string convert2string();
  function void set_initiator (ovm_component initiator);
  function ovm_component get_initiator ();

  // Transaction recording interface
  function void accept_tr (time accept_time=0);
  function integer begin_tr (time begin_time=0);
  function integer begin_child_tr (time begin_time=0, integer parent_handle=0);
  function void end_tr (time end_time=0, bit free_handle=1);

  function integer get_tr_handle ();
  function void disable_recording ();
  function void enable_recording (string stream);
  function bit is_recording_enabled();
  function bit get_transaction_id ();

  // Methods to add action during transaction recording
  virtual protected function void do_accept_tr ();
  virtual protected function void do_begin_tr ();
  virtual protected function void do_end_tr ();

  // Access methods
  function ovm_event_pool get_event_pool ();
  function time get_begin_time ();
  function time get_end_time ();
  function time get_accept_time ();
```

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

endclass
```

File

base/ovm_transaction.svh

Virtual

Yes

Members

None

Methods

new

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

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

accept_tr

```
function void accept_tr (time accept_time=0)
```

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

This function performs the following actions:

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

- The do_accept_tr method on page 30 is called to allow for any post-accept action in derived classes.

begin_child_tr

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

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

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

This function performs the following actions:

- The transaction's internal start time is set to the current simulation time, or to *begin_time* if provided and non-zero. The *begin_time* may be any time, past or future, but should not be less than the accept time.
- If recording is enabled, then a new database-transaction is started with the same begin time as above. The record method inherited from ovm_object is then called, which records the current property values to this new transaction. Finally, the newly started transaction is linked to the parent transaction given by *parent_handle*.
- The do_begin_tr method on page 30 is called to allow for any post-begin action in derived classes.
- The transaction's internal begin event is triggered. Any processes waiting on this event will resume in the next delta cycle.

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

begin_tr

```
function integer begin_tr (time begin_time=0)
```

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

This function performs the following actions:

- The transaction's internal start time is set to the current simulation time, or to *begin_time* if provided and non-zero. The *begin_time* may be any time, past or future, but should not be less than the accept time.

- If recording is enabled, then a new database-transaction is started with the same begin time as above. The `record` method inherited from `ovm_object` is then called, which records the current property values to this new transaction.
- The `do_begin_tr` method on page 30 is called to allow for any post-begin action in derived classes.
- The transaction's internal `begin` event is triggered. Any processes waiting on this event will resume in the next delta cycle.

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

disable_recording

```
function void disable_recording ()
```

Turns off recording for the transaction.

do_accept_tr

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

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

do_begin_tr

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

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

do_end_tr

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

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

convert2string

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

This function converts a transaction to a string.

The default implementation calls `ovm_object::sprint` using the default printer.

This method can be overloaded in derived classes to provide an alternate string representation of the transaction object.

enable_recording

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

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

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

end_tr

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

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

This function performs the following actions:

- The transaction's internal end time is set to the current simulation time, or to *`end_time`* if provided and non-zero. The *`end_time`* may be any time, past or future, but should not be less than the begin time.
- If recording is enabled and a database-transaction is currently active, then the `record` method inherited from `ovm_object` is called, which records the final property values. The transaction is then ended. If *`free_handle`* is set, the transaction is released and can no longer be linked to (if supported by the implementation).
- The `do_end_tr` method on page 30 is called to allow for any post-end action in derived classes.
- The transaction's internal `end` event is triggered. Any processes waiting on this event will resume in the next delta cycle.

get_accept_time

```
function time get_accept_time ()
```

Returns the *accept* time for this transaction, as set by a previous call to `accept_tr`.

get_begin_time

```
function time get_begin_time ()
```

Returns the *begin* time for this transaction, as set by a previous call to begin_child_tr or begin_tr.

get_end_time

```
function time get_end_time ()
```

Returns the *end* time for this transaction, as set by a previous call to end_tr.

get_event_pool

```
function ovm_event_pool get_event_pool ()
```

Returns the event pool associated with this transaction.

By default, the event pool contains the events: *begin*, *accept*, and *end*. Events can also be added by derivative objects. See ovm_event_pool on page 114 for more information.

get_initiator

```
function ovm_component get_initiator ()
```

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

get_tr_handle

```
function integer get_tr_handle ()
```

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

get_transaction_id

```
function integer get_transaction_id ()
```

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

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

is_active

```
function bit is_active ()
```

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

Returns `0` if the transaction has not been started.

is_recording_enabled

```
function bit is_recording_enabled ()
```

Returns `1` if recording is currently on.

Returns `0` if recording is currently off.

set_initiator

```
function void set_initiator (ovm_component initiator)
```

Sets `initiator` as the initiator of this transaction.

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

set_transaction_id

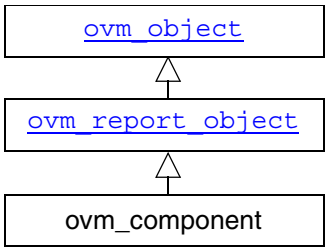
```
function void set_transaction_id (integer id)
```

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

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

Component Hierarchy

ovm_component



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

- *Hierarchy* — provides methods for searching and traversing the component hierarchy.
- *Configuration* — provides methods for configuring component topology and other parameters ahead of and during component construction.
- *Phasing* — defines a phased test flow that all components follow. Derived components implement one or more of the predefined phase callback methods to perform their function. During simulation, all components' callbacks are executed in precise order.
- *Factory* — provides a convenience interface to the ovm_factory on page 97. The factory is used to create new components and other objects based on type-wide and instance-specific configuration.
- *Reporting* — provides a convenience interface to the ovm_report_handler on page 79. All messages, warnings, and errors are processed through this interface.
- *Transaction recording* — provides methods for recording the transactions produced or consumed by the component to a transaction database (vendor specific).

Note: The `ovm_component` is automatically seeded during construction using OVM seeding, if enabled. All other objects must be manually reseeded, if appropriate. See `reseed` method on page 23 for more information.

Summary

```
virtual class ovm_component extends ovm_report_object;

  function new (string name, ovm_component parent);

    // Hierarchy information and setting
    virtual function ovm_component get_parent ();
    function int get_num_children ();
    function ovm_component get_child (string name);
    virtual function string get_full_name ()
    virtual function string get_type_name ()
    function int get_first_child (ref string name)
    function int get_next_child (ref string name)
    bit has_child (string name)
    virtual function set_name (string name)
    function ovm_component lookup (string hier_name)



    // Configuration interface
    virtual function void  See ovm_phase on page 59 for more information on
    phases.set_config_int (string inst_name,
                                          string field_name,
                                          ovm_bitstream_t value);
    virtual function void set_config_object (string inst_name,
                                          string field_name,
                                          ovm_object  value, bit clone=1);
    virtual function void set_config_string (string inst_name,
                                          string field_name,
                                          string value);
    virtual function bit  get_config_int   (string field_name,
                                         inout ovm_bitstream_t value);
    virtual function bit get_config_object (string field_name,
                                          inout ovm_object value);
    virtual function bit get_config_string (string field_name,
                                          inout string value);
    virtual function void apply_config_settings (bit verbose=0);
    function void print_config_settings (string field="",
                                       ovm_component comp=null,
                                       bit recurse=0);
    static  bit print_config_matches = 0;
```

```
// Phasing interface
virtual function void build ();
virtual function void connect ();
virtual function void end_of_elaboration ();
virtual function void start_of_simulation ();
virtual task          run ();
virtual function void extract ();
virtual function void check ();
virtual function void report ();

virtual task          suspend ();
virtual task          resume ();
virtual function void kill ();
virtual function void do_kill_all ();
virtual function void status ();

virtual  task stop (string ph_name);
protected int enable_stop_interrupt = 0;

// Factory interface
function void print_override_info (string requested_type_name,
                                   string name="");
function ovm_component create_component (string requested_type_name,
                                         string name);
function ovm_object create_object (string requested_type_name,
                                   string name="");
static function void set_type_override (string original_type_name,
                                        string override_type_name,
                                        bit replace=1);
function void set_inst_override (string relative_inst_path,
                                 string original_type_name,
                                 string override_type_name);
static function void set_type_override_by_type
                                 (ovm_object_wrapper original_type,
                                  ovm_object_wrapper override_type,
                                  bit replace=1);
function void set_inst_override_by_type (string relative_inst_path,
                                 ovm_object_wrapper original_type,
                                 ovm_object_wrapper override_type);
```

```
// Reporting interface
function void set_report_severity_action_hier (ovm_severity severity,
                                               ovm_action action);
function void set_report_id_action_hier (string id,
                                         ovm_action action);
function void set_report_severity_id_action_hier (ovm_severity severity,
                                                   string id,
                                                   ovm_action action);
function void set_report_severity_file_hier (ovm_severity severity,
                                             FILE file);
function void set_report_default_file_hier (FILE file);
function void set_report_id_file_hier (string id,
                                       FILE file);
function void set_report_severity_id_file_hier (ovm_severity severity,
                                                string id,
                                                FILE file);
function void set_report_verbosity_level_hier(int verbosity);

// Transaction interface
function void    accept_tr      (ovm_transaction tr,
                                 time accept_time=0);
function integer begin_tr        (ovm_transaction tr,
                                  string stream_name="main",
                                  string label="",
                                  string desc="",
                                  time begin_time=0);
function integer begin_child_tr (ovm_transaction tr,
                                  integer parent_handle=0,
                                  string stream_name="main",
                                  string label="",
                                  string desc="",
                                  time begin_time=0);
function void    do_accept_tr          (ovm_transaction tr,
                                  time end_time=0,
                                  bit free_handle=1);
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);
  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);
  virtual protected
    function void do_accept_tr     (ovm_transaction tr);
  virtual protected
    function void do_begin_tr      (ovm_transaction tr,
                                    string stream_name,
                                    integer tr_handle);
  virtual protected
    function void do_end_tr        (ovm_transaction tr,
                                    integer tr_handle);

endclass
```

File

base/ovm_component.svh

Virtual

Yes

Methods

new

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

All components must specify an instance name and a parent component.

The component will be inserted as a child of the parent object. If the parent is *null*, then the component will be a top-level component.

All classes derived from `ovm_component` must call `super.new()` with appropriate `name` and `parent` arguments.

If `name` is not found in the enclosing topology, then a *null* object is returned, otherwise a handle to `name` is returned.

accept_tr

```
function void accept_tr (ovm_transaction tr, time accept_time=0)
```

This function marks the acceptance of a transaction, *`tr`*, by this component. Specifically, it performs the following actions:

- Calls the transaction's `accept_tr` method, passing to it the *`accept_time`* argument. See accept_tr on page 28 for details.
- Calls the component's `do_accept_tr` method on page 42 to allow for any post-begin action in derived classes.
- Triggers the component's internal `accept_tr` event. Any processes waiting on this event will resume in the next delta cycle.

apply_config_settings

```
virtual function void apply_config_settings (bit verbose=0)
```

This is an automation function called by `ovm_component::build()` that finds all configuration overrides matching this component's full instance name.

The overrides are applied in reverse order by calling the appropriate `set_*_local` method (e.g., for an object override, `set_object_local` is called). By making the calls in reverse order, the same semantics associated with the `get_config*` calls are achieved.

Because `apply_config_settings` uses the `set_*_local` methods to apply the configuration settings, these methods must be overloaded for the component.

Note: The automation macros (`ovm_field_*) are also in effect when `apply_config_settings` is called, regardless of whether `set_*_local` is overloaded.

If you do not want `apply_config_settings` to be called for a component, then the `build()` method should be overloaded and you should not call `super.build()`. If this is done, then you must also set the `m_build_done` bit.

Likewise, `apply_config_settings()` can be overloaded, and the meaning of the automated configuration can be changed (for instance, replaced with `get_config*` calls).

When the verbose bit is set, all overrides are printed as they are applied. If the component's print_config_matches property is set, then apply_config_settings is automatically called with verbose=1.

begin_tr

```
function integer begin_tr (ovm_transaction tr,
                           string stream_name="main",
                           string label="",
                           string desc="",
                           time begin_time=0)
```

This function marks the start of a transaction, *tr*, by this component. Specifically, it performs the following actions:

- Calls the transaction's begin_tr method. The *begin_time* should be greater than or equal to the accept time. By default, when *begin_time*=0, the current simulation time is used. See begin_tr on page 29 for details.
- If recording is enabled (recording_detail != OVM_OFF), then a new database-transaction is started on the component's transaction stream given by the *stream* argument. No transaction properties are recorded at this time.
- Calls the component's do_begin_tr method on page 43 to allow for any post-begin action in derived classes.
- Triggers the component's internal begin_tr event. Any processes waiting on this event will resume in the next delta cycle.

A handle to the transaction is returned. The meaning of this handle, as well as the interpretation of the arguments *stream_name*, *label*, and *desc* are vendor specific.

begin_child_tr

```
function integer begin_child_tr (ovm_transaction tr,
                                 integer parent=0,
                                 string stream_name="main",
                                 string label="",
                                 string desc="",
                                 time begin_time=0)
```

This function marks the start of a child transaction, *tr*, by this component. Its operation is identical to that of begin_tr, except that an association is made between this transaction and the provided *parent* transaction. This association is vendor-specific.

build

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

The `build` phase callback is the first of several methods automatically called during the course of simulation. The `build` phase is the second of a two-pass construction process (the first is the built-in `new` method). The `build` phase can add additional hierarchy based on configuration information not available at time of initial construction.

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

check

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

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

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

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

This method should never be called directly.

See ovm_phase on page 59 for more information on phases.

connect

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

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

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

Generally, derived classes should override this method to make port and export connections via the `connect` method in ovm_port_base #(IF) on page 162. Any override should call `super.check`.

This method should never be called directly.

See ovm_phase on page 59 for more information on phases.

create_component

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

A convenience function for create_component_by_name in ovm_factory on page 97, this method calls upon the factory to create a new child component whose type corresponds to the preregistered type name, *requested_type_name*, and instance name, *name*. This method is equivalent to:

```
factory.create_component_by_name(requested_type_name,
                                 get_full_name(), name, this);
```

If the factory determines that a type or instance override exists, the type of the component created may be different than the requested type. See set_type_override on page 55 and set_inst_override on page 52. See also ovm_factory on page 97 for details on factory operation.

create_object

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

A convenience function for create_object_by_name in ovm_factory on page 97, this method calls upon the factory to create a new object whose type corresponds to the preregistered type name, *requested_type_name*, and instance name, *name*. This method is equivalent to:

```
factory.create_object_by_name(requested_type_name,
                              get_full_name(), name);
```

If the factory determines that a type or instance override exists, the type of the object created may be different than the requested type. See ovm_factory on page 97 for details on factory operation.

do_accept_tr

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

The accept_tr method calls this function to accommodate any user-defined post-accept action. Implementations should call `super.do_accept_tr` to ensure correct operation.

do_begin_tr

```
virtual protected function void do_begin_tr (ovm_transaction tr,
                                             string stream_name,
                                             integer tr_handle)
```

The begin_tr and begin_child_tr methods call this function to accommodate any user-defined post-begin action. Implementations should call `super.do_begin_tr` to ensure correct operation.

do_end_tr

```
virtual protected function void end_tr (ovm_transaction tr,
                                        integer tr_handle)
```

The do_accept_tr method calls this function to accommodate any user-defined post-end action. Implementations should call `super.do_begin_tr` to ensure correct operation.

do_kill_all

```
virtual function void do_kill_all ()
```

Recursively kills the process trees associated with the currently running task-based phase, e.g., `run`, for this component and all its descendants.

end_tr

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

This function marks the end of a transaction, *tr*, by this component. Specifically, it performs the following actions:

- Calls the transaction's `end_tr` method. The *end_time* must at least be greater than the begin time. By default, when *end_time*=0, the current simulation time is used. See end_tr on page 31 for details.
- The transaction's properties are recorded to the database-transaction on which it was started, and then the transaction is ended. Only those properties handled by the transaction's `record` method are recorded.
- Calls the component's do_end_tr method on page 43 to accommodate any post-end action in derived classes.

- Triggers the component's internal `end_tr` event. Any processes waiting on this event will resume in the next delta cycle.

The `free_handle` bit indicates that this transaction is no longer needed. The implementation of `free_handle` is vendor-specific.

end_of_elaboration

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

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

Starting after the `connect` phase has completed, this phase consists of calling all components' `end_of_elaboration` methods recursively in depth-first, bottom-up order, i.e., children are executed before their parents.

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

This method should never be called directly.

See ovm_phase on page 59 for more information on phases.

extract

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

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

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

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

This method should never be called directly.

See ovm_phase on page 59 for more information on phases.

get_config_int

get_config_string

get_config_object

```
virtual function bit get_config_int    (string field_name,
                                        inout ovm_bitstream_t value)
virtual function bit get_config_string (string field_name,
                                        inout ovm_bitstream_t value)
virtual function bit get_config_object (string field_name,
                                        inout ovm_object value, clone=1)
```

These methods retrieve configuration settings made by previous calls to report, set_config_string, and set_config_object. As the methods' names imply, there is direct support for integral types, strings, and objects. Settings of other types can be indirectly supported by defining an object to contain them.

Configuration settings are stored in a global table and in each component instance. With each call to a `get_config_*` method, a top-down search is made for a setting that matches this component's full name and the given *field_name*. For example, if this component's full instance name is `top.u1.u2`, then the global configuration table is searched first. If that fails, then it searches the configuration table in component `top`, followed by `top.u1`.

The first instance/field match causes *value* to be written with the value of the configuration setting and the return value is `1`. If no match is found, then `value` is unchanged and the return value is `0`.

Calling the `get_config_object` method requires special handling. Because *value* is an output of type `ovm_object`, you must provide an `ovm_object` handle to assign to (not a derived class handle). After the call, you can then `$cast` to the actual type.

For example, the following code illustrates how a component designer might call upon the configuration mechanism to assign its `data` object property. Note that we are overriding the `apply_config_settings`.

```
class mycomponent extends ovm_component;
  local myobj_t data;
  function void apply_config_settings();
    ovm_object tmp;
    if(get_config_object("data", tmp))
      if (!$cast(data, tmp))
        $display("error! config setting for 'data' not of type myobj_t");
  endfunction
  ...
```

The above example overrides the `apply_config_settings` method, which automatically configures this component's properties via the `set_*_local` methods, if implemented. See `set_object_local` method on page 23 and `apply_config_settings` method on page 39 for details on the automatic configuration mechanism.

See Members on page 57 for information on setting the global configuration table.

get_first_child

get_child

get_next_child

```
function int           get_first_child (ref string name)
function ovm_component get_child       (string name)
function int           get_next_child  (ref string name)
```

These methods are used to iterate through this component's children, if any. For example, given a component with an object handle, comp, the following code calls print for each child:

```
string name;
ovm_component child;
if (comp.get_first_child(name))
    do begin
      child = comp.get_child(name);
      child.print();
    end while (comp.get_next_child(name));
```

get_full_name

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

Returns the full hierarchical name of this object. The default implementation concatenates the hierarchical name of the parent, if any, with the leaf name of this object, as given by `get_name`.

get_num_children

```
function int get_num_children ()
```

Returns the number of this component's children.

get_parent

```
virtual function ovm_component get_parent ()
```

Returns a handle to this component's parent, or `null` if it has no parent.

get_type_name

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

Returns "`ovm_component`". Subclasses must override to return the derived type name.

has_child

```
function int has_child (string name)
```

Returns 1 if this component has a child with the given *`name`*, 0 otherwise.

kill

```
virtual function void kill ()
```

Kills the process tree associated with this component's currently running task-based phase, e.g., `run`.

An alternative mechanism for stopping the `run` phase is the *stop request*. Calling global_stop_request on page 281 causes all components' `run` processes to be killed, but only after all components have had the opportunity to complete in progress transactions and shutdown cleanly via their `stop` tasks.

lookup

```
function ovm_component lookup (string hier_name)
```

Looks for a component with the given hierarchical name relative to this component. If the given name is preceded with a '.' (dot), then the search begins relative to the top level (absolute lookup). The handle of the matching component is returned, else null. The name must not contain wildcards.

print_config_settings

```
function void print_config_settings(string field="",
                                    ovm_component comp=null,
                                    bit recurse=0)
```

Called without arguments, `print_config_settings` prints all configuration information for this component, as set by previous calls to `set_config_*`.

If *field* is specified a non-empty, then only the configuration matching the field is printed. The field cannot contain wildcards.

If *comp* is specified and non-null, then the configuration for that component is printed, not this component.

If *recurse* is set, then configuration information for all children and below are printed as well.

print_override_info

```
function void print_override_info(string type_name,
                                  string inst_name="")
```

This factory debug method performs the same lookup process as create_object and create_component, but instead of creating an object, it prints information about what type of object would be created given the provided arguments.

record_error_tr

```
function integer record_error_tr (string stream_name="main",
                                  ovm_object info=null,
                                  string label="",
                                  string desc="",
                                  time error_time=0,
                                  bit keep_active=0)
```

This function marks an error transaction by a component. Properties of the given ovm_object, *info*, as implemented in its do_record method on page 19, are recorded to the transaction database.

An *error_time* of 0 indicates to use the current simulation time.

The *keep_active* bit determines if the handle should remain active. If *keep_active* is 0, then a zero-length error transaction is recorded.

A handle to the database-transaction is returned.

Interpretation of this handle, as well as the strings stream_name, label, and desc, are vendor-specific.

record_event_tr

```
function integer record_event_tr (string stream_name="main",
                                  ovm_object info=null,
                                  string label="",
```

```
                                    string desc="",
                                    time event_time=0,
                                    bit keep_active=0)
```

This function marks an event transaction by a component.

An `event_time` of `0` indicates to use the current simulation time.

A handle to the transaction is returned. The `keep_active` bit determines if the handle may be used for other vendor-specific purposes.

The strings for `stream_name`, `label`, and `desc` are vendor-specific identifiers for the transaction.

report

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

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

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

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

This method should never be called directly.

See ovm_phase on page 59 for more information on phases.

resume

```
virtual task resume ()
```

Resumes the process tree associated with this component's currently running task-based phase, e.g., `run`.

run

```
virtual task run ()
```

The `run` phase callback is the only predefined phase that is time-consuming, i.e., task-based. It executes after the `start_of_simulation` phase has completed. Derived classes should override this method to perform the bulk of its functionality, forking additional processes if needed.

In the `run` phase, all threaded components' `run` tasks are forked as independent processes. Returning from its run task does *not* signify completion of a component's `run` phase, and any processes that `run` may have forked *continue to run*.

The `run` phase terminates in one of three ways:

- *explicit call to global_stop_request* — When `global_stop_request` is called, an ordered shut-down for the currently running phase begins. First, all enabled components' `status` tasks are called bottom-up, i.e., childrens' stop tasks are called before the parent's. A component is enabled by its `enable_stop_interrupt` bit. Each component can implement `stop` to allow completion of in-progress transactions, flush queues, and other shut-down activities. Upon return from `stop` by all enabled components, the recursive `do_kill_all` is called on all top-level component(s).
- *explicit call to kill or do_kill_all* — When `kill` called, this component's run processes are killed immediately. The `do_kill_all` methods applies to this component and all its descendants. Use of this method is not recommended. It is better to use the stopping mechanism, which affords a more ordered, safer shut-down.
- *timeout* — The phase ends if the timeout expires before an explicit call to `global_stop_request` or `kill`. By default, the timeout is set to near the maximum simulation time possible. You may override this via `set_global_timeout`, but you cannot disable the timeout completely.

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

Note: The deprecated `do_test` mode has special semantics for ending the `run` phase. In this mode, once the top-level `ovm_env::run` task returns, an automatic call to `global_stop_request` is issued, effectively ending the phase.

The `run` task should never be called directly.

See ovm_phase ***on page 59*** for more information on phases.***set_config_int***

set_config_string

set_config_object

```
virtual function void set_config_int (string inst_name,
                                      string field_name,
                                      ovm_bitstream_t value)
```

```
virtual function void set_config_string (string inst_name,
                                         string field_name,
                                         string value)
virtual function void set_config_object (string inst_name,
                                         string field_name,
                                         ovm_object value,
                                         bit clone=1)
```

These methods work in conjunction with the `get_config_*` methods to provide a configuration setting mechanism for integral, string, and `ovm_object`-based types. Settings of other types, such as virtual interfaces and arrays, can be indirectly supported by defining an object to contain them.

Calling any of `set_config_*` causes a configuration setting to be created and placed in a table internal to this component. The configuration setting stores the supplied *`inst_name`*, *`field_name`*, and *`value`* for later use by descendent components during their construction.

When a descendant component calls a `get_config_*` method, the *`inst_name`* and *`field_name`* provided in the `get` call are matched against all the configuration settings stored in the global table and then in each component in the parent hierarchy, top-down. Upon the first match, the *`value`* stored in the configuration setting is returned. Thus, precedence is global, following by the top-level component, and so on down to the descendent component's parent.

Both *`inst_name`* and *`field_name`* may contain wildcards.

For `set_config_int`, *`value`* is an integral value that can be anything from `1` bit to `4096` bits.

For `set_config_string`, *`value`* is a string.

For `set_config_object`, *`value`* must be an `ovm_object`-based object or `null`. Its *`clone`* argument specifies whether the object should be cloned. If set, then the object is cloned both going into the table (during the set) and coming out of the table (during the get), so that multiple components matched to the same setting (by way of wildcards) do not end up sharing the same object.

See `get_config_int`, `get_config_string`, and get_config_object on page 45 for more information on getting and applying configuration settings. See Members on page 57 for information on setting the global configuration table.

set_inst_override

```
function void set_inst_override (string relative_inst_path,
                                 string original_type_name,
                                 string override_type_name)
```

A convenience function for set_inst_override_by_type in ovm_factory on page 97, this method registers a factory override for components created at this level of hierarchy or below. In typical usage, this method is equivalent to:

```
factory.set_inst_override_by_name({get_full_name(),".",
                                   relative_inst_path},
                                  original_type_name,
                                  override_type_name);
```

The *relative_inst_path* is relative to this component and may include wildcards. The *original_type_name* typically refers to a preregistered type in the factory. It may, however, be any arbitrary string. Subsequent calls to create_component or create_object with the same string and matching instance path will produce the type represented by *override_type_name*. The *override_type_name* must refer to a preregistered type in the factory.

set_inst_override_by_type

```
function void set_inst_override_by_type (string relative_inst_path,
                                         ovm_object_wrapper original_type,
                                         ovm_object_wrapper override_type)
```

A convenience function for set_inst_override_by_type in ovm_factory on page 97, this method registers a factory override for components and objects created at this level of hierarchy or below. In typical usage, this method is equivalent to:

```
factory.set_inst_override_by_type({get_full_name(),".",
                                   relative_inst_path},
                                  original_type,
                                  override_type);
```

The *relative_inst_path* is relative to this component and may include wildcards. The *original_type* represents the type that is being overridden. In subsequent calls to create_object or create_component, if the *requested_type* matches the *original_type* and the instance paths match, the factory will produce the *override_type*.

The original and override types are lightweight proxies to the types they represent. They can be obtained by calling *type*::get_type(), if implemented, or by directly calling *type*::type_id::get(), where *type* is the user type and type_id is the typedef to ovm_object_registry #(T,Tname) or ovm_component_registry #(T,Tname).

The following example illustrates both uses:

```
class comp extends ovm_component;
  typedef ovm_component_registry #(comp) type_id;
  static function type_id get_type();
    return type_id::get();
  endfunction
  ...
endclass
class mycomp extends ovm_component;
  typedef ovm_component_registry #(mycomp) type_id;
  static function type_id get_type();
    return type_id::get();
  endfunction
  ...
endclass
...
class block extends ovm_component;
  comp c_inst;
  virtual function void build();
    set_inst_override_by_type("c_inst",comp::get_type(),
                                       mycomp::get_type());
    set_inst_override_by_type("c_inst",comp::type_id::get(),
                                       mycomp::type_id::get());
  endfunction
  ...
endclass
```

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

set_name

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

Renames this component and recalculates all descendants' full names.

set_report_default_file_hier

set_report_id_file_hier

set_report_severity_file_hier

set_report_severity_id_file_hier

```
function void set_report_default_file_hier     (FILE file)
function void set_report_id_file_hier          (string id, FILE file)
function void set_report_severity_file_hier    (ovm_severity sev, FILE file)
function void set_report_severity_id_file_hier (ovm_severity sev,
                                                string id, FILE file)
```

These methods recursively configure the report handlers in this component and all its children to direct some or all of its output to the given *file* descriptor. The *file* argument must be a multi-channel descriptor (mcd) or file id compatible with `$fdisplay`.

- `set_report_default_file_hier` hierarchically sets a default file descriptor. It is used when no other setting applies.
- `set_report_severity_file_hier` hierarchically sets the file descriptor for reports matching the given severity. This setting takes precedence over the default setting.
- `set_report_id_file_hier` hierarchically sets the file descriptor for reports matching the given *id*. This setting takes precedence over the default and any severity settings from `set_report_severity_file_hier`.
- `set_report_severity_id_file_hier` hierarchically sets the file descriptor for reports matching both the given severity and id. This setting takes highest precedence.

For a list of severities and other information related to the report mechanism, refer to ovm_report_handler on page 79.

set_report_severity_action_hier

set_report_id_action_hier

set_report_severity_id_action_hier

```
function void set_report_severity_action_hier    (ovm_severity severity,
                                                   ovm_action action)
function void set_report_id_action_hier           (string id,
                                                   ovm_action action)
function void set_report_severity_id_action_hier (ovm_severity severity,
```

```
                                        string id,
                                        ovm_action action)
```

These methods recursively configure the report handlers in this component and all its children to perform the given *action* when issuing reports matching the given *severity*, *id*, or both *severity* and *id*.

- `set_report_severity_action_hier` hierarchically sets the *action* for reports matching the given *sev*. This setting takes precedence over the default setting.
- `set_report_id_action_hier` hierarchically sets the *action* for reports matching the given *id*. This setting takes precedence over the default and any severity settings from `set_report_severity_action_hier`.
- `set_report_severity_id_action_hier` hierarchically sets the *action* for reports matching both the given *sev* and *id*. This setting takes highest precedence.

For a list of severities and their default actions, refer to ovm_report_handler on page 79.

set_report_verbosity_level_hier

```
function void set_report_verbosity_level_hier (int verbosity)
```

This method recursively configures the report handlers in this component and all its children to output messages at the given verbosity level and below.

To be displayed, messages must have a verbosity setting equal to or less than *verbosity*. To display all messages, set *verbosity* to a large number (such as `'hffffffff`).

See ovm_report_handler on page 79 for a list of predefined message verbosity levels and their meaning.

set_type_override

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

A convenience function for calling `set_type_override_by_type` in ovm_factory on page 97, this method configures the factory to create an object of type *override_type_name* whenever the factory is asked to produce a type represented by *original_type_name*. This method is equivalent to:

```
factory.set_type_override_by_name(original_type_name,
                                  override_type_name, replace);
```

The *original_type_name* typically refers to a preregistered type in the factory. It may, however, be any arbitrary string. Subsequent calls to `create_component` or

create_object with the same string and matching instance path will produce the type represented by *override_type_name*. The *override_type_name* must refer to a preregistered type in the factory.

set_type_override_by_type

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

A convenience function for set_type_override_by_type in ovm_factory on page 97, this method registers a factory override for components and objects created at this level of hierarchy or below. This method is equivalent to:

```
factory.set_type_override_by_type(original_type,
                                  override_type,replace);
```

The *relative_inst_path* is relative to this component and may include wildcards. The *original_type* represents the type that is being overridden. In subsequent calls to create_object or create_component, if the *requested_type* matches the *original_type* and the instance paths match, the factory will produce the *override_type*.

The original and override type arguments are lightweight proxies to the types they represent. See set_inst_override_by_type method on page 52 for information on usage.

start_of_simulation

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

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

Starting after the end_of_elaboration phase has completed, this phase consists of calling all components' start_of_simulation methods recursively in depth-first, bottom-up order, i.e. children are executed before their parents.

The start_of_simulation phase starts after the end_of_elaboration phase has completed and before the run phase. In the start_of_simulation phase, all components' start_of_simulation methods are called recursively in depth-first, bottom-up order, i.e., children are executed before their parents.

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

This method should never be called directly.

See ovm_phase on page 59 for more information on phases.

status

```
function string status ()
```

Returns the status of the parent process associated with the currently running task-based phase, e.g., `run`.

stop

```
virtual task stop()
```

This component's `stop` task is called when `global_stop_request` is called during a task-based phase (e.g., `run`) and its `enable_stop_interrupt` bit is set.

Before a phase is abruptly ended, e.g., when a test deems the simulation complete, some components may need extra time to shut down cleanly. Such components may implement `stop` to finish the currently executing transaction, flush the queue, or perform other cleanup. Upon return from its `stop`, a component signals it is ready to be stopped.

The `stop` method will *not* be called if `enable_stop_interrupt`=0.

The default implementation of stop is empty, i.e., to return immediately.

The `stop` method should never be called directly.

suspend

```
virtual task suspend ()
```

Suspends the process tree associated with this component's currently running task-based phase, e.g., `run`.

Members

enable_stop_interrupt

```
bit enable_stop_interrupt = 0
```

This bit allows a component to raise an objection to the stopping of the current phase. It affects only time consuming phases (such as the *run* phase in `ovm_component`).

When this bit is set, the `stop` task in the component is called as a result of a call to `global_stop_request`.

print_config_matches

```
static bit print_config_matches = 0
```

This static bit sets up the printing of configuration matches for debug purposes.

When a `get_config_*` call is used, or when the automatic configuration mechanism finds a match, the match is printed when `print_config_matches` is `1`.

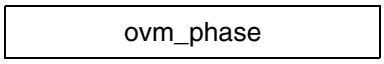
print_enabled

```
bit print_enabled = 1
```

This bit determines if this component should automatically be printed as a child of its parent object.

By default, all children are printed. However, this bit allows a parent component to disable the printing of specific children.

ovm_phase



The `ovm_phase` class is used for defining phases for ovm_component and its subclasses.

Phases are a synchronizing mechanism for the environment. They are represented by callback methods. A set of predefined phases and corresponding callbacks are provided in ovm_component. Any class deriving from ovm_component may implement any or all of these callbacks, which are executed in a particular order. Depending on the properties of any given phase, the corresponding callback is either a function or task, and it is executed in top-down or bottom-up order.

Table 1-2 on page 59 shows the predefined phases for all ovm_component-based objects:

Table 1-2 Predefined Phases

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

Summary

```
virtual class ovm_phase;
   function new (string name, bit is_top_down, bit is_task);

   function string get_name();
   virtual function string get_type_name();

   function bit is_task();
   function bit is_top_down();

   function bit is_in_progress();
   function bit is_done();
   function bit reset();

   virtual task call_task(ovm_component parent);
   virtual function void call_func(ovm_component parent);

endclass
```

File

base/ovm_phases.sv

Virtual

Yes

Methods

new

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

Creates a phase object.

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

call_task

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

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

call_func

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

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

get_name

```
function string get_name ()
```

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

get_type_name

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

Returns "ovm_phase". Subclasses must override to return the derived type name.

is_done

```
function bit is_done ()
```

Returns 1 if the phase has completed, 0 otherwise.

is_in_progress

```
function bit is_in_progress ()
```

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

is_task

```
function bit is_task ()
```

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

is_top_down

```
function bit is_top_down ()
```

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

reset

```
function void reset ()
```

Resets phase state such that `is_done` and `is_in_progress` both return 0.

Usage

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

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

Inheriting from the ovm_phase Class

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

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

   ```
   `ovm_phase_func_topdown_decl( preload )
   ```

   This convenient phase declaration macro is described below.

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

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

3. Register the phase object with `ovm_top`.

   ```
   class my_memory extends ovm_component;
     function new(string name, ovm_component parent);
       super.new(name,parent);
       ovm_top.insert_phase(preload_ph, start_of_simulation_ph);
     endfunction
     virtual function void preload();
   ```

```
        ...
      endfunction
    endclass
```

Optional Macros

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

The *PHASE_NAME* argument is used to define the name of the phase, the name of the component method that is called back during phase execution, and the prefix to the type-name of the phase class itself.

`ovm_phase_func_decl

```
`ovm_phase_func_decl (PHASE_NAME, TOP_DOWN)
```

This macro creates the following class definition.

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

`ovm_phase_task_decl

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

This macro creates the following class definition:

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

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

`ovm_phase_func_topdown_decl

`ovm_phase_func_bottomup_decl

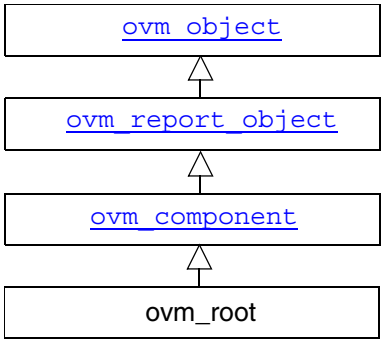
`ovm_phase_task_topdown_decl

`ovm_phase_task_bottomup_decl

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

These alternative macros have a single phase name argument. The top-down or bottom-up selection is specified in the macro name.

ovm_root



The ovm_root class provides an implicit top-level and phase control for all OVM components. A single instance of ovm_root named *ovm_top* serves as top-level container for all OVM components and controls all aspects of simulation phases. Any component whose parent is specified as NULL becomes a child of ovm_top.

Summary

```
class ovm_root extends ovm_component;
  protected function new ();
  function string        get_type_name ();

  task                   run_test      (string test_name="");
  function void          insert_phase  (ovm_phase new_phase,
                                        ovm_phase exist_phase);
  function void          stop_request  ();
  function ovm_component find          (string comp_match)
  function void          find_all      (string comp_match,
                                        ref ovm_component comps[$],
                                        input ovm_component comp=null);
  function ovm_phase     get_current_phase ();

  bit  enable_print_topology = 0;
  bit  finish_on_completion = 1;
  time phase_timeout = 0;
  time stop_timeout  = 0;
endclass
```

File

base/ovm_root.svh

Virtual

No

Methods

new

```
protected function new ()
```

Creates an instance of `ovm_root`, if not already created. Users should never call this method. It is used once as a static initializer for the `ovm_top` global variable.

get_type_name

```
function string get_type_name ()
```

Returns "`ovm_root`".

get_current_phase

```
function ovm_phase get_current_phase ()
```

Returns the handle of the currently executing phase.

run_test

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

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

insert_phase

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

This method is used to register phases for later execution by ovm_top, the singleton instance of ovm_root. The ovm_top maintains a queue of phases executed in consecutive order. This method allows you to insert new phases into that queue, where the phase given by *new_phase* will be inserted *after* the existing phase given by *exist_phase*. If *exist_phase* is null, then *new_phase* is inserted at the head of the queue, i.e., it becomes the first phase.

stop_request

```
function void stop_request ()
```

Calling this function triggers the process of shutting down the currently running task-based phase. This process involves calling all components' stop tasks for those components whose enable_stop_interrupt bit is set. Once all stop tasks return, or once the optional global_stop_timeout expires, all components' kill method is called, effectively ending the current phase. The ovm_top will then begin execution of the next phase, if any.

find

find_all

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

Returns the component handle (find) or list of components handles (find_all) matching a given string. The string may contain the wildcards, * and ?. Strings beginning with '.' are absolute path names. If optional comp arg is provided, then search begins from that component down (default=all components).

Members

enable_print_topology

```
bit  enable_print_topology = 0
```

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

finish_on_completion

```
bit finish_on_completion = 1
```

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

phase_timeout

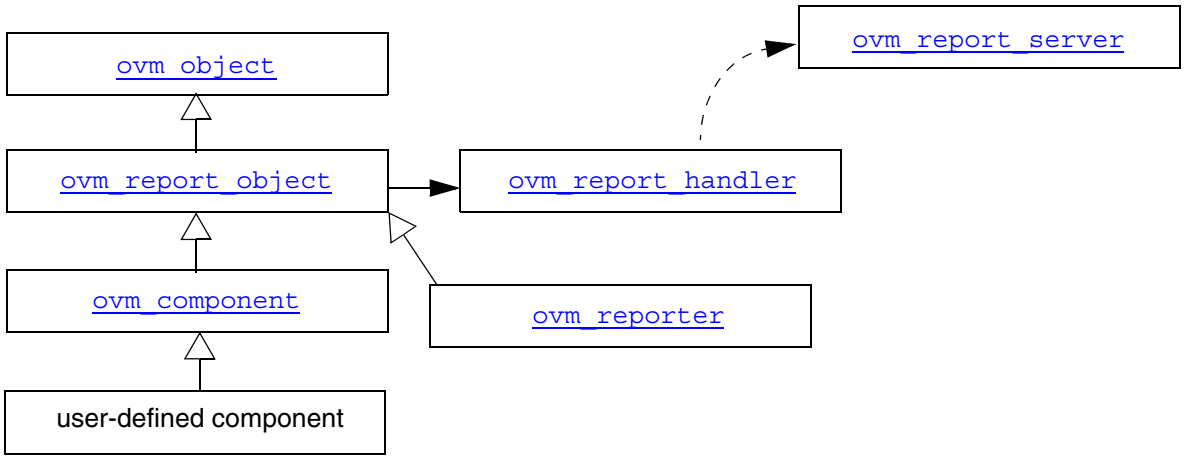
stop_timeout

```
time phase_timeout = 0
time stop_timeout  = 0
```

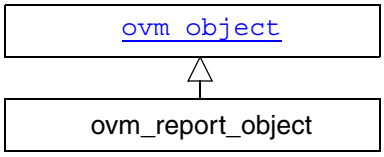
These set watchdog timers for task-based phases and stop tasks. You can not disable the timeouts. When set to 0, a timeout of the maximum time possible is applied. A timeout at this value usually indicates a problem with your testbench. You should lower the timeout to prevent "never-ending" simulations.

Reporting

The reporting classes provide a facility for issuing reports with different severities and IDs, and to different files. The primary interface to the reporting facility is `ovm_report_object`, which is inherited by `ovm_component`.



ovm_report_object



The ovm_report_object provides an interface to the OVM reporting facility. Through this interface, components issue the various messages that occur during simulation. They can also configure what actions are taken and what file(s) are output for individual messages or for all messages.

Most methods in ovm_report_object are delegated to an instance of an ovm_report_handler, which stores its component's reporting configuration and determines whether an issued message should be displayed based on the configuration. To display a message, the report handler delegates the actual formatting and production of messages to a central ovm_report_server.

Summary

```
virtual class ovm_report_object extends ovm_object;
    function new(string name="");
    function void ovm_report_fatal(string id, string message,
                                   int verbosity_level=100,
                                   tring filename="", int line=0);
    function void ovm_report_error(string id, string message,
                                   int verbosity_level=0,
                                   string filename="", int line=0);
    function void ovm_report_warning(string id, string message,
                                   int verbosity_level=300,
                                   string filename="", int line=0);
    function void ovm_report_info (string id, string message,
                                   int verbosity_level=200,
                                   string filename="", int line=0);

    virtual function void report_header(FILE file=0);
    virtual function void report_summarize(FILE file=0);

    function void set_report_handler(ovm_report_handler hndlr);
    function ovm_report_handler get_report_handler();
    function void reset_report_handler();
```

```
    virtual function bit report_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);
    virtual function bit report_error_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_info_hook (string id,
                                           string message,
                                           int verbosity,
                                           string filename, int line);
    function void set_report_max_quit_count(int m);

    function void set_report_verbosity_level (int verbosity_level);
    function void set_report_severity_action (ovm_severity severity,
                                              ovm_action action);
    function void set_report_id_action       (string id, ovm_action action);
    function void set_report_severity_id_action (ovm_severity severity,
                                              string id, ovm_action action);
    function void set_report_default_file    (FILE file);
    function void set_report_severity_file   (ovm_severity severity, FILE file);
    function void set_report_id_file         (string id, FILE file);
    function void set_report_severity_id_file(ovm_severity severity,
                                              string id,
                                              FILE file);

    function ovm_report_server  get_report_server ();
    function void  dump_report_state ();
    virtual function void die();
endclass
```

File

base/ovm_report_object.svh

Virtual

Yes

Methods

new

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

Creates a new report object with the given name. This method also creates a new `ovm_report_handler` object, which this object delegates most tasks to.

ovm_report_fatal

ovm_report_error

ovm_report_warning

ovm_report_info

```
function void ovm_report_fatal(string id, string message,
                               int verbosity_level=0,
                               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_warning(string id, string message,
                                 int verbosity_level=200,
                                 string filenamee="", int line=0)
function void ovm_report_info(string id, string message,
                              int verbosity_level=300,
                              string filename="", int line=0)
```

These methods produce reports of severity `OVM_FATAL`, `OVM_ERROR`, `OVM_WARNING`, and `OVM_INFO`. All message output should come from calls to these four methods.

The *`id`* argument is a unique identifier for a message. You can configure an individual report's actions and output file descriptor using its *`id`* string.

The *message* argument is main body of the message you want displayed.

The *verbosity* argument specifies the message's relative importance. If the *verbosity* argument is higher than the maximum verbosity setting in the report handler, this report is simply ignored. The default verbosity levels by severity are: OVM_FATAL=0, OVM_ERROR=100, warning=200, and info=300. The maximum verbosity can be set using the set_report_verbosity_level method on page 77 or set_report_verbosity_level_hier method on page 55.

The *filename* and *line* arguments allow you to provide the location of the call to the report methods. If specified, they are displayed in the output.

die

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

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

If this report object is a super-class of an ovm_component and the run phase is currently being executed, then die will issue a global_stop_request, which ends the phase and allows simulation to continue to subsequent phases.

Otherwise, die calls report_summarize and terminates simulation with $finish.

dump_report_state

```
function void dump_report_state()
```

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

get_report_handler

```
function ovm_report_handler get_report_handler()
```

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

get_report_server

```
function ovm_report_server get_report_server()
```

Returns the report server associated with this report object.

report_header

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

Prints version and copyright information. This information is sent to the command line if *file* is 0, or to the file descriptor *file* if it is not 0.

This method is called by ovm_env immediately after the construction phase and before the connect phase.

report_hook

report_info_hook

report_warning_hook

report_error_hook

report_fatal_hook

```
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)
```

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

report_summarize

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

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

reset_report_handler

```
function void reset_report_handler()
```

Re-initializes the component's report handler to the default settings.

set_report_handler

```
function void set_report_handler(ovm_report_handler hndlr)
```

Sets the report handler, thus allowing more than one component to share the same report handler.

set_report_max_quit_count

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

Sets the maximum quit count in the report handler to *max_count*. When the number of OVM_COUNT actions reaches *max_count*, the die method on page 73 is called.

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

set_report_default_file

set_report_severity_file

set_report_id_file

set_report_severity_id_file

```
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)
```

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

- `set_report_default_file` sets a default file descriptor for all reports issued by this report handler. The initial descriptor is set to 0, which means that even if the action includes a `OVM_LOG` attribute, the report is not sent to a file.
- `set_report_severity_file` sets the file descriptor for reports matching the given severity. This setting takes precedence over the default file descriptor.
- `set_report_id_file` sets the file descriptor for reports matching the given *`id`*. This setting takes precedence over the default and any severity settings from `set_report_severity_file`.
- `set_report_severity_id_file` sets the file descriptor for reports matching both the given severity and id. This setting takes highest precedence.

See ovm_report_handler on page 79 for more information.

set_report_severity_action

set_report_id_action

set_report_severity_id_action

```
function void set_report_severity_action    (ovm_severity severity,
                                              ovm_action action)
function void set_report_id_action           (string id,
                                              ovm_action action)
function void set_report_severity_id_action (ovm_severity severity,
                                              string id,
                                              ovm_action action)
```

These methods configure the report handler in this component to perform the given *`action`* when issuing reports matching the given *`severity`*, *`id`*, or *`severity-id`* pair.

- `set_report_severity_action` sets the *`action`* for reports matching the given *`severity`*. This setting takes precedence over the default setting.
- `set_report_id_action` sets the *`action`* for reports matching the given *`id`*. This setting takes precedence over settings from `set_report_severity_action`.
- `set_report_severity_id_action` sets the *`action`* for reports matching both the given *`severity`* and *`id`*. An action associated with a (`severity`, `id`) pair takes priority over an action associated with either the *`severity`* or the *`id`* alone.

The *action* argument can take the value OVM_NO_ACTION (5'b00000), or it can be a bitwise OR of any combination of OVM_DISPLAY, OVM_LOG, OVM_COUNT, OVM_EXIT, and OVM_CALL_HOOK.

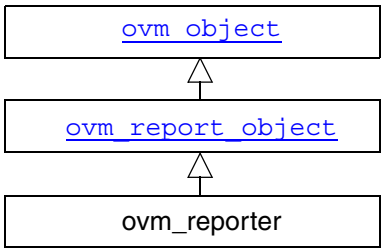
For a list of severities and their default actions, refer to ovm_report_handler on page 79.

set_report_verbosity_level

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

Sets the maximum verbosity level for the report handler. Any report whose verbosity exceeds this maximum is ignored.

ovm_reporter



The `ovm_reporter` extends `ovm_report_object` and is used as a standalone reporter. Objects that are not `ovm_components` may use this to issue reports that leverage the same configuration and formatting features as components.

Summary

```
class ovm_reporter extends ovm_report_object;
    function new(string name="reporter");
endclass
```

File

base/ovm_report_object.svh

Virtual

No

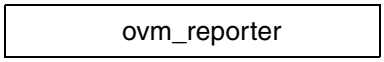
Methods

new

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

The constructor has the default name of `reporter`.

ovm_report_handler



ovm_report_handler is the class to which many of the methods in ovm_report_object are delegated. It stores the maximum verbosity, actions, and files that affect the way reports are handled.

Note: The report handler is not intended for direct use. See ovm_report_object on page 70 for information on the OVM reporting mechanism.

The relationship between ovm_report_object (a base class for ovm_component) and ovm_report_handler is usually one to one, but it can, in theory, be many to one. When a report needs processing, the report handler passes it to the central report server. The relationship between ovm_report_handler and ovm_report_server is many to one.

Summary

```
class ovm_report_handler;
    function new();
    function void initialize();
    function void set_max_quit_count(int max_count);
    function void set_verbosity_level(int verbosity_level);

    function  int get_verbosity_level     ();
    function void set_severity_action(ovm_severity severity, ovm_action action);
    function void set_id_action(string id, ovm_action action);
    function void set_severity_id_action(ovm_severity severity, string id,
                                         ovm_action action);
    function void set_default_file(FILE file);
    function void set_severity_file(ovm_severity severity, FILE file);
    function void set_id_file(string id, FILE file);
    function void set_severity_id_file(ovm_severity severity, string id,
                                       FILE file);
    function action get_action(ovm_severity severity, string id);
    function FILE get_file_handle(ovm_severity severity, string id);
    function string format_action(ovm_action action);

    function void initialize(FILE file=0);
    function void summarize(FILE file=0);
    function void report(ovm_severity severity, string name, string id,
```

```
                          string message,
                          int verbosity_level=0,
                          ovm_report_object client=null);
    virtual function bit run_hooks(ovm_report_object client,
                                   ovm_severity severity, string id,
                                   string message, int verbosity,
                                   string filename, int line);
endclass
```

File

base/ovm_report_handler.svh

Virtual

No

Default Actions

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

Severity	Actions
OVM_INFO	OVM_DISPLAY
OVM_WARNING	OVM_DISPLAY
OVM_ERROR	OVM_DISPLAY | OVM_COUNT
OVM_FATAL	OVM_DISPLAY | OVM_EXIT

Default File Handle

The default file handle is 0, which means that reports are not sent to a file even if an `OVM_LOG` attribute is set in the action associated with the report.

This can be overridden by any of the `set_*_file` methods.

Methods

new

```
function new()
```

Creates and initializes a new `ovm_report_handler` object.

format_action

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

Returns a string representation of the action, e.g., "`OVM_ERROR`".

initialize

```
function void initialize()
```

This method is called by the constructor to initialize the arrays and other variables described above to their default values.

get_action

```
function action get_action(ovm_severity severity, string id)
```

This method looks up the action associated with this `severity` and `id`.

get_file_handle

```
function FILE get_file_handle(ovm_severity severity, string id)
```

This method returns the file descriptor associated with the given `severity` and `id`.

get_verbosity_level

```
function int get_verbosity_level()
```

Returns the configured maximum verbosity level.

report

```
function void report(ovm_severity severity, string name, string id,
                     string message, int verbosity_level, string filename,
                     int line, ovm_report_object client)
```

This is the common handler method used by the four core reporting methods (e.g., `ovm_report_error`) in ovm_report_object on page 70.

report_header

```
function void report_header(FILE file=0)
```

See corresponding methods in ovm_report_object on page 70.

run_hooks

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

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

summarize

```
function void summarize(FILE file=0)
```

See corresponding methods in ovm_report_object on page 70.

set_max_quit_count

```
function void set_max_quit_count (int max_count)
```

See corresponding method in ovm_report_object on page 70.

set_verbosity_level

```
function void set_verbosity_level(int verbosity_level)
```

See corresponding method in ovm_report_object on page 70.

set_default_file

set_severity_file

set_id_file

set_severity_id_file

```
function void set_default_file(FILE file)
function void set_severity_file(ovm_severity severity, FILE file)
function void set_id_file(string id, FILE file)
```

```
function void set_severity_id_file(ovm_severity severity,
                                   string id, FILE file)
```

See the corresponding methods in ovm_report_object on page 70.

set_severity_action

set_id_action

set_severity_id_action

```
function void set_severity_action   (ovm_severity severity,
                                     ovm_action action)
function void set_id_action         (string id, ovm_action action)
function void set_severity_id_action(ovm_severity severity,
                                     string id, ovm_action action)
```

See the corresponding methods in ovm_report_object on page 70.

ovm_report_server

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

Summary

```
class ovm_report_server;
    protected function new();
    static function ovm_report_server get_server();
    function int get_max_quit_count();
    function void set_max_quit_count(int count);
    function void reset_quit_count();
    function void incr_quit_count();
    function int get_quit_count();
    function bit is_quit_count_reached();
    function void reset_severity_counts();
    function int get_severity_count(ovm_severity severity);
    function void incr_severity_count(ovm_severity severity);
    function void set_id_count(string id, int count);
    function int get_id_count(string id);
    function void incr_id_count(string id);
    function void summarize(FILE file=0);
    function void f_ovm_display(FILE file, string str);
    function void dump_server_state();
    virtual function void process_report(ovm_severity severity, string name,
                                         string id, string message,
                                         ovm_action action,
                                         FILE file,
                                         string filename, int line,
                                         ovm_report_object client );
    virtual function string compose_message(ovm_severity severity, string name,
                                            string id, string message);
endclass
```

File

base/ovm_report_server.svh

Virtual

No

Methods

new

```
protected function new ()
```

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

get_server

```
static function ovm_report_server get_server ()
```

Returns a handle to the central report server.

get_max_quit_count

set_max_quit_count

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

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

get_quit_count

incr_quit_count

is_quit_count_reached

reset_quit_count

```
function int get_quit_count ()
function void incr_quit_count ()
```

```
function bit is_quit_count_reached ()
function void reset_quit_count ()
```

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

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

get_severity_count

incr_severity_count

reset_severity_counts

```
function int get_severity_count (ovm_severity severity)
function void incr_severity_count (ovm_severity severity)
function void reset_severity_counts ()
```

Get or increment the counter for the given *severity*, or reset all severity counters to 0.

get_id_count

incr_id_count

set_id_count

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

Get, increment, or set the counter for reports with the given *id*.

summarize

```
function void summarize (FILE file=0)
```

See ovm_report_object::report_summarize method on page 74.

f_ovm_display

```
function void f_ovm_display (FILE file, string severity)
```

This method sends string *severity* to the command line if *file* is 0 and to the file(s) specified by *file* if it is not 0.

dump_server_state

```
function void dump_server_state()
```

See ovm_report_object::dump_report_state on page 73.

process_report

```
virtual function void process_report(ovm_severity severity, string name,
                                     string id, string message,
                                     ovm_action action,
                                     FILE file,
                                     string filename, int line,
                                     ovm_report_object client )
```

This method calls compose_message to construct the actual message to be output. It then takes the appropriate action according to the value of *action* and *file*.

This method can be overloaded by expert users so that the report system processes the actions different from the way described in ovm_report_object and ovm_report_handler.

compose_message

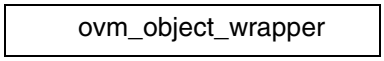
```
virtual function string compose_message(ovm_severity severity, string name,
                                        string id, string message)
```

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

Expert users can overload this method to change the formatting of the reports generated by ovm_report_object.

Factory

ovm_object_wrapper



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

Summary

```
virtual class ovm_object_wrapper;

    virtual function ovm_object    create_object    (string name="");
    virtual function ovm_component create_component (string name,
                                                     ovm_component parent);
    pure virtual function string   get_type_name();

endclass
```

File

base/ovm_factory.svh

Virtual

Yes

Methods

create_component

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

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

A component proxy (e.g. ovm_component_registry #(T,Tname) on page 90) implements this method to create a component of type T.

create_object

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

Creates a new object, passing to its constructor the optional *name*.

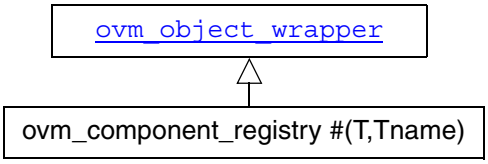
An object proxy (e.g., ovm_object_registry #(T,Tname) on page 94) implements this method to create an object of type T.

get_type_name

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

Derived classes implement this method to return the type name of the object created by `create_component` or `create_object`. The factory uses this name when matching against the requested type.

ovm_component_registry #(T,Tname)



The `ovm_component_registry` serves as a lightweight proxy for a component of type `T` and type name `Tname`, a string. The proxy enables efficient registration with the ovm_factory on page 97. Without it, registration would require an instance of the component itself.

Summary

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

  typedef ovm_component_registry #(T,Tname) this_type;

  static function this_type get();

  static function T         create            (string name,
                                               ovm_component parent,
                                               string contxt="");

  static function void      set_type_override (ovm_object_wrapper override_type,
                                               bit replace=1);

  static function void      set_inst_override (ovm_object_wrapper override_type,
                                               string inst_path,
                                               ovm_component parent=null);
  // for use by factory
  const static string type_name = Tname;
  virtual
    function ovm_component  create_component  (string name,
                                               ovm_component parent);
   virtual function string  get_type_name();
endclass
```

File

base/ovm_registry.svh

Methods

create_component

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

Creates a component of type `T` having the provided *`name`* and *`parent`*.

get_type_name

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

Returns the value `Tname`.

get

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

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

create

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

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

set_type_override

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

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

set_inst_override

```
static function void set_inst_override (ovm_object_wrapper override_type,
                                        string inst_path,
                                        ovm_component parent=null)
```

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

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

Usage

To register a particular component type, you need only typedef a specialization of its proxy class, which is typically done inside the class. For example, to register an OVM component of type mycomp:

```
class mycomp extends ovm_component;
  typedef ovm_component_registry #(mycomp,"mycomp") type_id;
endclass
```

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

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

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

For parameterized types, the type name changes with each specialization, so you can not specify a Tname inside a parameterized class and get the behavior you want; the same type

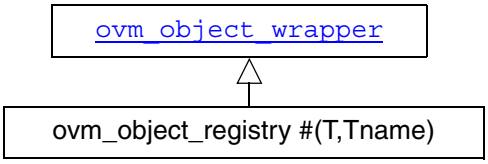
name string would be registered for all specializations of the class. (The factory would produce warnings for each specialization beyond the first.) To avoid the warnings and simulator interoperability issues with parameterized classes, must register parameterized classes with a different macro.

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

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

The `` `ovm_component_param_utils `` and `` `ovm_object_param_utils `` macros are used to register *parameterized* classes with the factory. Unlike the the non-param versions, these macros do not specify the `Tname` parameter in the `ovm_component_registry` typedef, and they do not define the `get_type_name` method for the user class. Consequently, you will not be able to use the factory's name-based methods for parameterized classes. The primary purpose for adding the factory's type-based methods was to accommodate registration of parameterized types and eliminate the many sources of errors associated with string-based factory usage.

ovm_object_registry #(T,Tname)



The `ovm_object_registry` serves as a lightweight proxy for an `ovm_object` of type `T` and type name `Tname`, a string. The proxy enables efficient registration with the ovm_factory on page 97. Without it, registration would require an instance of the object itself.

Summary

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

  static function this_type get();

  static function T         create            (string name="",
                                               ovm_component parent=null,
                                               string contxt="");

  static function void      set_type_override (ovm_object_wrapper override_type,
                                               bit replace=1);

  static function void      set_inst_override (ovm_object_wrapper override_type,
                                               string inst_path,
                                               ovm_component parent=null);

  // for use by factory
  const static string type_name = Tname;
  virtual
    function ovm_object     create_object (string name="");
  virtual function string   get_type_name ();
endclass
```

File

base/ovm_registry.svh

Methods

create_object

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

Creates an object of type T.

get_type_name

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

Returns the value Tname.

get

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

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

create

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

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

set_type_override

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

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

set_inst_override

```
static function void set_inst_override (ovm_object_wrapper override_type,
                                        string inst_path,
                                        ovm_component parent=null)
```

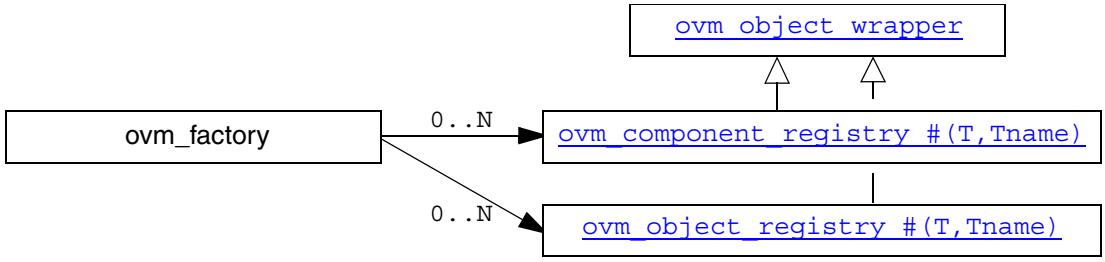
Configures the factory to create an object of the type represented by *override_type* whenever a request is made to create an object of the type represented by this proxy, T, with matching instance paths. The original type, T, is typically a super class of the override type.

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

Usage

See ovm_component_registry #(T,Tname)'s section on Usage on page 92.

ovm_factory



As the name implies, `ovm_factory` is used to manufacture (create) OVM objects and components. Only one instance of the factory is present in a given simulation (termed a singleton). Object and component types are registered with the factory using `ovm_object_registry` and `ovm_component_registry` proxy objects.

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

Summary

```
class ovm_factory;
  // type-based interface (preferred)
  function void       set_inst_override_by_type (ovm_object_wrapper original_type,
                                                 ovm_object_wrapper override_type,
                                                   string full_inst_path);

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

  function ovm_object  create_object_by_type    (ovm_object_wrapper requested_type,
                                                   string parent_inst_path="",
                                                   string name="");

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

```
                                                ovm_component parent);
  // name-based interface
  function void          set_inst_override_by_name (string original_type_name,
                                                    string override_type_name,
                                                    string full_inst_path);

  function void          set_type_override_by_type(string original_type_name,
                                                   string override_type_name,
                                                   bit replace=1);

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

  function ovm_component create_component_by_name (string requested_type_name,
                                                   string parent_inst_path="",
                                                   string name,
                                                   ovm_component parent);

  // registration and debug
  function void         register               (ovm_object_wrapper obj);
  function void         print                  (int all_types=1);

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

  function void         debug_create_by_name   (string requested_type_name,
                                               string parent_inst_path="",
                                               string name="");
  function
    ovm_object_wrapper  find_override_by_type  (ovm_object_wrapper requested_type,
                                               string full_inst_path);
  function
    ovm_object_wrapper  find_override_by_name  (string requested_type_name,
                                               string full_inst_path);
endclass
```

File

base/ovm_factory.svh

Methods

register

```
function void register (ovm_object_wrapper obj)
```

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

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

create_component_by_type

create_component_by_name

create_object_by_type

create_object_by_name

```
function ovm_component create_component_by_type
                                        (ovm_object_wrapper requested_type,
                                         string parent_inst_path="",
                                         string name,
                                         ovm_component parent)
function ovm_component create_component_by_name
                                        (string requested_type_name,
                                         string parent_inst_path="",
                                         string name,
                                         ovm_component parent)
function ovm_object create_object_by_type(ovm_object_wrapper requested_type,
                                         string parent_inst_path="",
                                         string name="")
function ovm_object create_object_by_name(string requested_type_name,
```

```
                                    string parent_inst_path="",
                                    string name="")
```

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

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

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

If the optional *parent_inst_path* is provided, then the concatenation, {*parent_inst_path*, ".",*name*}, forms an instance path (context) that is used to search for an instance override. The *parent_inst_path* is obtained via `parent.get_full_name()`.

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

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

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

print

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

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

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

debug_create_by_type

debug_create_by_name

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

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

find_override_by_name

find_override_by_type

```
function ovm_object_wrapper find_override_by_type
                (ovm_object_wrapper requested_type, string full_inst_path)
function ovm_object_wrapper find_override_by_name
                (string requested_type_name, string full_inst_path)
```

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

set_inst_override_by_type

set_inst_override_by_name

```
function void set_inst_override_by_type (ovm_object_wrapper original_type,
                                         ovm_object_wrapper override_type,
                                         string full_inst_path)
function void set_inst_override_by_name (string original_type_name,
                                         string override_type_name,
                                         string full_inst_path)
```

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

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

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

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

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

set_type_override_by_name

set_type_override_by_type

```
function void set_type_override_by_name (string original_type_name,
                                         string override_type_name,
                                         bit replace=1)
function void set_type_override_by_type (ovm_object_wrapper original_type,
                                         ovm_object_wrapper override_type,
                                         bit replace=1)
```

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

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

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

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

Usage

Using the factory involves three basic operations:

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

We'll briefly cover each of these steps here. More reference information can be found at ovm_component_registry #(T,Tname) on page 90, ovm_object_registry #(T,Tname) on page 94, and ovm_component on page 34.

1 — Registering objects and component types with the factory

When defining `ovm_object` and `ovm_component`-based classes, simply invoke the appropriate macro. Use of macros are required to ensure portability between different vendors' simulators.

For objects that are not parameterized:

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

For objects that are parameterized:

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

For components that are not parameterized:

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

For components that are parameterized:

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

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

The ‘ovm_*_param_utils macros for parameterized classes differ from ‘ovm_*_utils classes in the following ways:

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

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

2 — Designing components that defer creation to the factory

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

```
class driverB #(type T=ovm_object) extends ovm_driver;
  // parameterized classes must use the _param_utils version
  `ovm_component_param_utils(driverB #(T))
  // our packet type; this can be overridden via the factory
  T pkt;
  // standard component constructor
  function new(string name, ovm_component parent=null);
    super.new(name,parent);
  endfunction
  // get_type_name not implemented by macro for parameterized classes
  const static string type_name = {"driverB #(",T::type_name,")"};
  virtual function string get_type_name();
    return type_name;
  endfunction
```

```
  // using the factory allows pkt overrides from outside the class
  virtual function void build();
    pkt = packet::type_id::create("pkt",this);
  endfunction
  // print the packet so we can confirm its type when printing
  virtual function void do_print(ovm_printer printer);
    printer.print_object("pkt",pkt);
  endfunction
endclass
```

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

```
class driverD1 #(type T=ovm_object) extends driverB #(T);
  `ovm_component_param_utils(driverD1 #(T))
  function new(string name, ovm_component parent=null);
    super.new(name,parent);
  endfunction
  const static string type_name = {"driverD1 #(",T::type_name,")"};
  virtual function string get_type_name();
    ...return type_name;
  endfunction
endclass

class driverD2 #(type T=ovm_object) extends driverB #(T);
  `ovm_component_param_utils(driverD2 #(T))
  function new(string name, ovm_component parent=null);
    super.new(name,parent);
  endfunction
  const static string type_name = {"driverD2 #(",T::type_name,")"};
  virtual function string get_type_name();
    return type_name;
  endfunction
endclass

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

Next, we" define a non-parameterized `agent` component, which requires a different macro. Before creating the drivers using the factory, we override `driver0`'s packet type to be `packetD`.

```
class agent extends ovm_agent;
  `ovm_component_utils(agent)
  ...B_driver driver0;
  B_driver driver1;
  function new(string name, ovm_component parent=null);
    super.new(name,parent);
  endfunction
  virtual function void build();
    // override the packet type for driver0 and below
    packet::type_id::set_inst_override(packetD::get_type(),"driver0.*");
    // create using the factory; actual driver types may be different
    driver0 = B_driver::type_id::create("driver0",this);
    driver1 = B_driver::type_id::create("driver1",this);
  endfunction
endclass
```

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

```
class env extends ovm_env;
  `ovm_component_utils(env)
  agent agent0;
  agent agent1;
  function new(string name, ovm_component parent=null);
    super.new(name,parent);
  endfunction
  virtual function void build();
    // three methods to set an instance override for agent1.driver1
    // - via component convenience method...
    set_inst_override_by_type("agent1.driver1",
                              B_driver::get_type(),
                              D2_driver::get_type());
    // - via the component's proxy (same approach as create)...
    B_driver::type_id::set_inst_override(D2_driver::get_type(),
                                         "agent1.driver1",this);
    // - via a direct call to a factory method...
    factory.set_inst_override_by_type(B_driver::get_type(),
                                      D2_driver::get_type(),
```

```
                                        {get_full_name(),".agent1.driver1"});
    // create agents using the factory; actual agent types may be different
    agent0 = agent::type_id::create("agent0",this);
    agent1 = agent::type_id::create("agent1",this);
  endfunction
  // at end_of_elaboration, print topology and factory state to verify
  virtual function void end_of_elaboration();
    ovm_top.print_topology();
  endfunction
  virtual task run();
    #100 global_stop_request();
  endfunction
endclass
```

3 — Configuring the factory with type and instance overrides

In the previous step, we demonstrated setting instance overrides and creating components using the factory *within component classes*. Here, we will demonstrate setting overrides from outside components, as when initializing the environment prior to running the test.

```
module top;
  env env0;
  initial begin
    // Being registered first, the following overrides take precedence
    // over any overrides made within env0's construction & build.

    // Replace all base drivers with derived drivers...
    B_driver::type_id::set_type_override(D_driver::get_type());

    // ...except for agent0.driver0, whose type remains a base driver
    // - via the component's proxy (preferred)
    B_driver::type_id::set_inst_override(B_driver::get_type(),
                                         "env0.agent0.driver0");
    // - via a direct call to a factory method
    factory.set_inst_override_by_type(B_driver::get_type(),
                                      B_driver::get_type(),
                                    {get_full_name(),"env0.agent0.driver0"});
    // now, create the environment; factory configuration will govern topology
    env0 = new("env0");
    // run the test (will execute build phase)
    run_test();
```

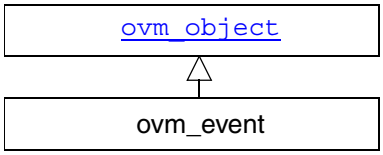
```
    end
  endmodule
```

When the above example is run, the resulting topology (displayed via a call to `ovm_top.print_topology` in `env`'s `end_of_elaboration` method) is similar to the following.

```
# OVM_INFO @ 0 [RNTST] Running test ...
# OVM_INFO @ 0 [OVMTOP] OVM testbench topology:
# ------------------------------------------------------------------------
# Name                      Type                Size                Value
# ------------------------------------------------------------------------
# env0                      env                 -                    env0@2
#   agent0                  agent               -                  agent0@4
#     driver0               driverB #(packet)   -                 driver0@8
#       pkt                 packet              -                    pkt@21
#     driver1               driverD #(packet)   -                driver1@14
#       pkt                 packet              -                    pkt@23
#   agent1                  agent               -                  agent1@6
#     driver0               driverD #(packet)   -                driver0@24
#       pkt                 packet              -                    pkt@37
#     driver1               driverD2 #(packet)  -                driver1@30
#       pkt                 packet              -                    pkt@39
# ------------------------------------------------------------------------
```

Synchronization

ovm_event



The `ovm_event` class is a wrapper class around a traditional Verilog event.

The `ovm_event` provides some services on top of a traditional Verilog event, such as setting callbacks on the event.

Note: Because of the extra overhead associated with an `ovm_event` object, these objects should be used sparingly, and should be used only in those places where traditional Verilog events are not sufficient.

Summary

```
class ovm_event extends ovm_object;
    function new (string name="");

    // waiting
    virtual task wait_on (bit delta=0);
    virtual task wait_off (bit delta=0);
    virtual task wait_trigger ();
    virtual task wait_ptrigger ();
    virtual task wait_trigger_data (output ovm_object data);
    virtual task wait_ptrigger_data(output ovm_object data);

    // triggering
    virtual function void trigger (ovm_object data=null);
    virtual function ovm_object get_trigger_data ();
    virtual function time get_trigger_time ();

    // state
    virtual function bit is_on ();
    virtual function bit is_off ();
    virtual function void reset (bit wakeup=0);
```

```
    // callbacks
    virtual function void add_callback (ovm_event_callback cb,
                                        bit append=1);
    virtual function void delete_callback (ovm_event_callback cb);

    // waiters list
    virtual function void cancel ();
    virtual function int get_num_waiters ();
endclass
```

File

base/ovm_event.svh

Virtual

No

Members

None

Methods

new

```
function new (name)
```

Creates a new event object.

add_callback

```
virtual function void add_callback (ovm_event_callback cb, bit append=1)
```

Adds a callback to the event. Callbacks have a `pre_trigger()` and `post_trigger()` function.

If `append` is set to `1`, which is the default, then the callback is added to the back of the callback list. Otherwise, the callback is put in the front of the callback list.

cancel

```
virtual function void cancel ()
```

Decrements the number of waiters on the event.

This is used if a process that is waiting on an event is disabled or activated by some other means.

delete_callback

```
virtual function void delete_callback (ovm_event_callback cb)
```

Removes a callback from the event.

get_num_waiters

```
virtual function int get_num_waiters ()
```

Returns the number of processes waiting on the event.

get_trigger_data

```
virtual function ovm_object get_trigger_data ()
```

Gets the data, if any, associated with the last trigger event.

get_trigger_time

```
virtual function time get_trigger_time ()
```

Gets the time that this event was last triggered. If the event has not been triggered, or the event has been reset, then the trigger time will be `0`.

is_on

```
virtual function bit is_on ()
```

Indicates whether the event has been triggered since it was last reset.

A return of `1` indicates that the event has triggered.

is_off

```
virtual function bit is_off ()
```

Indicates whether the event has been triggered since it was last reset.

A return of `1` indicates that the event has not been triggered.

reset

```
virtual function void reset (bit wakeup=0)
```

Resets the event to its off state. If wake-up is set, then all processes waiting for the event at the time of the reset are activated before the event is reset.

No callbacks are called during a reset.

trigger

```
virtual function void trigger (ovm_object data=null)
```

Triggers the event.

This causes all processes waiting on the event to be enabled.

An optional data argument can be supplied with the enable to provide trigger-specific information.

wait_on

```
virtual task wait_on (bit delta=0)
```

Waits for the event to be activated for the first time.

If the event has already been triggered, then this task immediately returns (if the delta bit is set, then it will cause a `#0` delay to be consumed before returning).

Once an event has been triggered, this task will always return immediately unless the event is reset.

wait_off

```
virtual task wait_off (bit delta=0)
```

Waits for the event to be reset if it has already triggered.

If the event has not already been triggered, then this task immediately returns (the delta bit will cause a `#0` delay to be consumed before returning).

wait_ptrigger

```
virtual task wait_ptrigger ()
```

Waits for the event to be triggered. Unlike `wait_trigger`, `wait_ptrigger()` views the event as persistent within a time-slice. Thus, if the waiter happens after the trigger, then the waiter will still see the event trigger during the current time-slice.

wait_ptrigger_data

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

This method is a wrapper for calling `wait_ptrigger` immediately followed by `get_trigger_data`.

wait_trigger

```
virtual task wait_trigger ()
```

Waits for the event to be triggered.

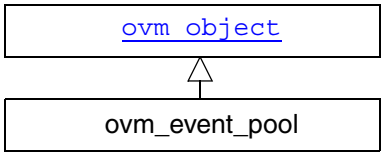
If one process calls `wait_trigger()` and another process calls `trigger()` in the same delta cycle, then a race condition occurs and there is no guarantee whether or not the waiter will see the trigger.

wait_trigger_data

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

This method is a wrapper for calling `wait_trigger` immediately followed by `get_trigger_data`.

ovm_event_pool



The ovm_event_pool is essentially an associative array of ovm_event objects, which is indexed by the string name of the event.

Summary

```
class ovm_event_pool extends ovm_object;
    function new (string name="");

    // Pool and event access
    static function ovm_event_pool get_global_pool ();
    virtual function ovm_event get (string name);

    // Iterators
    virtual function int num ();
    virtual function void delete (string name);
    virtual function int exists (string name);
    virtual function int first (ref string name);
    virtual function int last (ref string name);
    virtual function int next (ref string name);
    virtual function int prev (ref string name);

endclass
```

File

base/ovm_event.svh

Virtual

No

Members

None

Methods

new

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

Creates a new event pool.

delete

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

Removes the event `name` from the pool.

exists

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

Checks if the event `name` exists in the pool.

A return of `1` indicates that `name` is in the pool and `0` indicates that `name` is not in the pool.

first

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

Places the first event name from the pool into the variable `name`.

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

If the pool is not empty, then `name` gets the value of the first element and `1` is returned.

get

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

Returns the event with the name specified by `name`.

If no events exist with the given name, then a new event is created and returned.

get_global_pool

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

Accesses the singleton global event pool.

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

last

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

Returns the name of the last event in the pool.

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

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

num

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

Returns the number of events in the pool.

next

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

Uses the current value of `name` to find the next event name in the pool.

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

If a next name is found, then `name` is replaced with the next name and `1` is returned.

prev

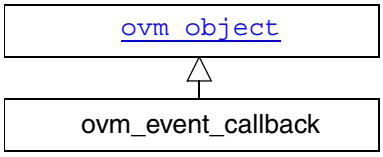
```
virtual function int prev (ref string name)
```

Uses the current value of `name` to find the previous event name in the pool.

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

If a previous name is found, then `name` is replaced with the previous name and `1` is returned.

ovm_event_callback



The `ovm_event_callback` class is an abstract class that is used to create callback objects which may be attached to events.

Callbacks are an alternative to using processes to wait on events. When a callback is attached to an event, that callback object's callback function is called each time the event is triggered.

Summary

```
virtual class ovm_event_callback extends ovm_object;

    function new (string name="");
    virtual function bit pre_trigger (ovm_event e,
                                      ovm_object data=null);
    virtual function void post_trigger (ovm_event e,
                                        ovm_object data=null);
endclass
```

File

base/ovm_event.svh

Virtual

Yes

Members

None

Methods

new

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

Creates a new callback object.

pre_trigger

```
virtual function bit pre_trigger (ovm_event e,
                                  ovm_object data=null)
```

This function implements the `pre_trigger` functionality.

If a callback returns `1`, then the event will **not** trigger its waiters. This provides a way for a callback to override an event action.

In the function, `e` is the `ovm_event` that is being triggered, and `data` is the data, if any, associated with the event trigger.

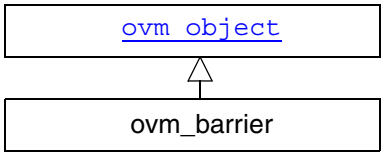
post_trigger

```
virtual function void post_trigger (ovm_event e,
                                    ovm_object data=null)
```

This function implements the `post_trigger` functionality.

In the function, `e` is the `ovm_event` that is being triggered, `and` data is the data, if any, associated with the event trigger.

ovm_barrier



The `ovm_barrier` class provides a multiprocess synchronization mechanism.

The `ovm_barrier` class enables a set of processes to block until the desired number of processes get to the synchronization point, at which time all of the processes are released.

Summary

```
class ovm_barrier extends ovm_object;

    function new (string name="");

    // waiting
    virtual task wait_for ();
    virtual function void reset (bit wakeup=1);
    virtual function void set_auto_reset (bit value=1);
    virtual function void set_threshold (int threshold);
    virtual function int get_threshold ();
    virtual function int get_num_waiters ();
    virtual function void cancel ();

endclass
```

File

base/ovm_event.svh

Virtual

No

Members

None

Methods

new

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

Creates a new barrier object.

cancel

```
virtual function void cancel ()
```

Decrements the waiter count by one. This is used when a process that is waiting on the barrier is killed or activated using some other means.

get_num_waiters

```
virtual function int get_num_waiters ()
```

Returns the number of processes currently waiting at the barrier.

get_threshold

```
virtual function int get_threshold ()
```

Gets the current threshold setting for the barrier.

reset

```
virtual function void reset (bit wakeup=1)
```

Resets the barrier. This sets the waiter count back to zero.

The threshold is unchanged. After reset, the barrier will force processes to wait for the threshold again.

If the wake-up bit is set, then currently waiting processes will be activated.

set_auto_reset

```
virtual function void set_auto_reset (bit value=1)
```

Determines if the barrier should reset itself when the threshold is reached.

The default is on, so when a barrier hits its threshold it will reset, and new processes will block until the threshold is reached again.

If auto reset is off, then once the threshold is achieved, new processes pass through without being blocked, until the barrier is reset.

set_threshold

```
virtual function void set_threshold (int threshold)
```

Sets the process threshold.

This determines how many processes must be waiting on the barrier before the processes may proceed.

Once the threshold is reached, all waiting processes are activated.

If the threshold is set to a value less than the number of waiting processes, then the barrier is reset and waiting processes are activated.

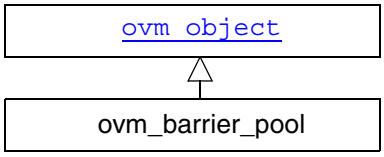
wait_for

```
virtual task wait_for ()
```

Waits for enough processes to reach the barrier before continuing.

The number of processes to wait for is set by the `set_threshold()` method.

ovm_barrier_pool



The ovm_barrier_pool is essentially an associative array of ovm_barrier objects, which is indexed by the string name of the barrier.

Summary

```
class ovm_barrier_pool extends ovm_object;

    function new (string name="");

    // Pool and barrier access
    static function ovm_barrier_pool get_global_pool ();
    virtual function ovm_barrier get (string name);

    // Iterators
    virtual function int num ();
    virtual function void delete (string name);
    virtual function int exists (string name);
    virtual function int first (ref string name);
    virtual function int last (ref string name);
    virtual function int next (ref string name);
    virtual function int prev (ref string name);

endclass
```

File

base/ovm_event.svh

Virtual

No

Members

None

Methods

new

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

Creates a new barrier pool.

delete

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

Removes the barrier `name` from the pool.

exists

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

Checks if the barrier `name` exists in the pool.

A return of `1` indicates that `name` is in the pool and `0` indicates that `name` is not in the pool.

first

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

Places the first barrier name from the pool into the variable `name`.

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

If the pool is not empty, then `name` gets the value of the first element and `1` is returned.

get

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

Returns the barrier with the name specified by `name`.

If no barriers exist with the given name, then a new barrier is created and returned.

get_global_pool

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

Accesses the singleton global barrier pool.

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

last

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

Returns the name of the last barrier in the pool.

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

Otherwise, `name` is set to the last name in the pool and `1` is returned.

num

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

Returns the number of barriers in the pool.

next

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

Uses the current value of `name` to find the next barrier name in the pool.

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

If a next name is found, then `name` is replaced with the next name and `1` is returned.

prev

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

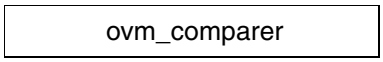
Uses the current value of `name` to find the previous barrier name in the pool.

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

If a previous name is found, then `name` is replaced with the previous name and `1` is returned.

Policies

ovm_comparer



The `ovm_comparer` class provides a policy object for doing comparisons.

The policies determine how miscompares are treated and how they are counted.

Results of a comparison are stored in the *comparer* object.

Summary

```
class ovm_comparer;

    // Comparison message settings
    int unsigned show_max = 1;
    int unsigned verbosity = 500;
    severity sev = OVM_INFO;
    string miscompares = "";

    // Comparison settings
    bit physical = 1;
    bit abstract = 1;
    bit check_type = 1;

    recursion_policy_enum policy = OVM_DEFAULT_POLICY;

    // Result of comparison
    int unsigned result = 0;

    // Methods used checking for printing information
    virtual function bit compare_field (string name,
                                        ovm_bitstream_t lhs,
                                        ovm_bitstream_t rhs,
                                        int size,
                                        radix_enum radix=OVM_NORADIX);
    virtual function bit compare_field_int (string name,
                                            logic[63:0] lhs,
                                            logic[63:0] rhs,
```

```
                                          int size,
                                          radix_enum radix=OVM_NORADIX);
    virtual function bit compare_object (string name,
                                         ovm_void lhs,
                                         ovm_void rhs);
    virtual function bit compare_string (string name,
                                         string lhs,
                                         string rhs);
    function void print_msg (string msg);
endclass
```

File

base/ovm_object.svh

Virtual

No

Members

`bit` **`abstract`** `= 1`

This bit provides a filtering mechanism for fields.

The abstract and physical settings allow an object to distinguish between two different classes of fields.

It is up to you, in the `ovm_object::do_compare()` routine, to test the setting of this field if they want to use it as a filter.

`bit` **`check_type`** `= 1`

This bit determines whether the type, given by `ovm_object::get_type_name()`, is used to verify that the types of two objects are the same.

This bit is used by the `compare_object()` method. In some cases it is useful to set this to `0` when it is legitimate for one side, for example the *rhs*, to contain a derivative of the other side (the *lhs*).

`int unsigned` **`result`** `= 0;`

This bit stores the number of miscompares for a given compare operation. You can use the result to determine the number of miscompares that were found.

`string` **`miscompares`** `= “”`

This string is reset to an empty string when a comparison is started.

The string holds the last set of miscompares that occurred during a comparison.

```
bit physical = 1
```

This bit provides a filtering mechanism for fields.

The abstract and physical settings allow an object to distinguish between two different classes of fields.

It is up to you, in the `ovm_object::do_compare()` routine, to test the setting of this field if they want to use it as a filter.

```
severity sev = OVM_INFO
```

Sets the severity for printed messages.

The severity setting is used by the messaging mechanism for printing and filtering messages.

```
int unsigned show_max = 1
```

Sets the maximum number of messages to send to the *messager* for miscompares of an object.

All miscompares are stored into the miscompares string.

```
int unsigned verbosity = 500
```

Sets the verbosity for printed messages.

The verbosity setting is used by the messaging mechanism to determine whether messages should be suppressed or shown.

Methods

compare_field

```
virtual function bit compare_field (string name,
                                    ovm_bitstream_t lhs,
                                    ovm_bitstream_t rhs,
                                    int size,
                                    radix_enum radix=OVM_NORADIX)
```

Compares two integral values.

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

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

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

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

compare_field_int

```
virtual function bit compare_field_int (string name,
                                        logic [63:0] lhs,
                                        logic [63:0] rhs,
                                        int size,
                                        radix_enum radix=OVM_NORADIX)
```

This method is the same as `compare_field` except that the arguments are small integers, less than or equal to 64 bits.

compare_object

```
virtual function bit compare_object (string name,
                                     ovm_void lhs,
                                     ovm_void rhs)
```

Compares two class objects using the policy value to determine whether the comparison should be *deep*, *shallow*, or *reference*.

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

The *lhs* and *rhs* objects are the two objects used for comparison.

The `check_type` bit is used to determine whether or not to verify the object types match (the return from `lhs.get_type_name()` matches `rhs.get_type_name()`).

compare_string

```
virtual function bit compare_string (string name,
                                     string lhs,
                                     string rhs)
```

Compares two string variables.

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

The *lhs* and *rhs* objects are the two objects used for comparison.

print_msg

```
function void print_msg (string msg)
```

Causes the error count to be incremented and the message, `msg`, to be appended to the miscompares string (a *newline* is used to separate messages).

If the message count is less than the `show_max` setting, then the message is printed to standard-out using the current verbosity and severity settings. See the `verbosity` and `severity` variables for more information.

ovm_packer

ovm_packer

The ovm_packer class provides a policy object for packing and unpacking ovm_objects. The policies determine how packing and unpacking should be done. Packing an object causes the object to be placed into a bit (byte or int) array. By default, no metadata information is stored for the packing of dynamic objects (strings, arrays, class objects). Therefore, and in general, it is not possible to automatically unpack into an object which contains dynamic data (Note that this is only a concern when using the field macros to automate packing and unpacking).

Summary

```
class ovm_packer;

    bit use_metadata  = 0;
    bit big_endian    = 1;
    bit physical = 1;
    bit abstract = 0;
    recursion_policy_enum policy = OVM_DEFAULT_POLICY;

    virtual function void pack_field_int (logic[63:0] value,
                                          int size);
    virtual function void pack_field (ovm_bitstream_t value,
                                      int size);
    virtual function void pack_string (string value);
    virtual function void pack_time (time value);
    virtual function void pack_real (real value);
    virtual function void pack_object (ovm_void value);

    virtual function bit is_null ();
    virtual function logic[63:0] unpack_field_int (int size);
    virtual function ovm_bitstream_t unpack_field (int size);
    virtual function string unpack_string (int num_chars=-1);
    virtual function time unpack_time ();
    virtual function real unpack_real ();
    virtual function void unpack_object (ovm_void value);

    virtual function int get_packed_size();

endclass
```

File

base/ovm_packer.svh

Virtual

No

Members

`bit` **`abstract`** `= 0`

This bit provides a filtering mechanism for fields.

The abstract and physical settings allow an object to distinguish between two different classes of fields. It is up to you, in the `ovm_object::do_pack()` and `ovm_object::do_unpack()` routines, to test the setting of this field if you want to use it as a filter.

`bit` **`big_endian`** `= 1`

This bit determines the order that integral data is packed (using `pack_field`, `pack_field_int`, `pack_time`, or `pack_real`) and how the data is unpacked from the pack array (using `unpack_field`, `unpack_field_int`, `unpack_time`, or `unpack_real`). When the bit is set, data is associated *msb* to *lsb*; otherwise, it is associated *lsb* to *msb*.

The following code illustrates how data can be associated *msb* to *lsb* and *lsb* to *msb*:

```
class mydata extends ovm_object;
    logic[15:0] value = 'h1234;
    function void do_pack (ovm_packer packer);
    packer.pack_field_int(value, 16);
endfunction
function void do_unpack (ovm_packer packer);
    value = packer.unpack_field_int(16);
endfunction
endclass
mydata d = new;
bit bits[];
initial begin
    d.pack(bits);  // results in 'b0001001000110100
    ovm_default_packer.big_endian = 0;
    d.pack(bits); // results in `b0010110001001000
end
```

`bit` **`physical`** `= 1`

This bit provides a filtering mechanism for fields.

The abstract and physical settings allow an object to distinguish between two different classes of fields. It is up to you, in the `ovm_object::do_pack()` and `ovm_object::do_unpack()` routines, to test the setting of this field if you want to use it as a filter.

`bit` **`use_metadata`** `= 0`

This flag indicates whether to encode metadata when packing dynamic data, or to decode metadata when unpacking. Implementations of `do_pack` and `do_unpack` should regard this bit when performing their respective operation. When set, metadata should be encoded as follows:

- For strings, pack an additional `null` byte after the string is packed.
- For objects, pack 4 bits prior to packing the object itself. Use 4'b0000 to indicate the object being packed is null, otherwise pack 4'b0001 (the remaining 3 bits are reserved).
- For queues, dynamic arrays, and associative arrays, pack 32 bits indicating the size of the array prior to to packing individual elements.

Methods

get_packed_size

`virtual function int` **`get_packed_size`** `()`

This method returns an `int` value that represents the number of bits that were packed.

is_null

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

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

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

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

pack_field

```
virtual function void pack_field (ovm_bitstream_t value,
                                  int size)
```

Packs an integral field (less than or equal to `4096` bits) into the pack array.

`value` is the value of the field to pack. `size` is the number of bits to pack.

pack_field_int

```
virtual function void pack_field_int (logic[63:0] value,
                                      int size)
```

Packs an integral field (less than or equal to `64` bits) into the pack array.

`value` is the value of the field to pack. `size` is the number of bits to pack.

This specialized version of `pack_field()` provides a higher performance mechanism for packing small vectors.

pack_string

```
virtual function void pack_string (string value)
```

Packs a string field into the pack array.

`value` is the value of the field to pack.

A 32-bit header is inserted ahead of the string to indicate the size of the string that was packed.

This is useful for mixed language communication where unpacking may occur outside of SystemVerilog OVM.

pack_object

```
virtual function void pack_object (ovm_object value)
```

Packs an object field into the pack array.

`value` is the value of the field to pack.

A 4-bit header is inserted ahead of the string to indicate the number of bits that was packed. If a *null* object was packed, then this header will be `0`.

This is useful for mixed-language communication where unpacking may occur outside of SystemVerilog OVM.

pack_real

```
virtual function void pack_real (real value)
```

Packs a real value as `64` bits into the pack array.

`value` is the value of the field to pack.

The `real` value is converted to a 6-bit scalar value using the function `$real2bits` before it is packed into the array.

pack_time

```
virtual function void pack_time (time value)
```

Packs a time value as `64` bits into the pack array. `value` is the value of the field to pack.

unpack_field

```
virtual function ovm_bitstream_t unpack_field (int size)
```

Unpacks bits from the pack array and returns the bit-stream that was unpacked. `size` is the number of bits to unpack; the maximum is `4096` bits.

unpack_field_int

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

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

`size` is the number of bits to unpack; the maximum is `64` bits.

This is a more efficient variant than `unpack_field` when unpacking into smaller vectors.

unpack_string

```
virtual function string unpack_string ()
```

Unpacks a string.

The first 32 bits are used to determine the number of characters that the packed string contains.

If the first 32 bits are `0`, then an empty string is returned.

unpack_object

```
virtual function void unpack_object (ovm_void value)
```

Unpacks an object and stores the result into `value`.

`value` must be an allocated object that has enough space for the data being unpacked. The first four bits of packed data are used to determine if a *null* object was packed into the array.

The `is_null()` function can be used by you to peek at the next four bits in the pack array before calling `unpack_object`.

unpack_real

```
virtual function real unpack_real ()
```

Unpacks the next 64 bits of the pack array and places them into a *real* variable.

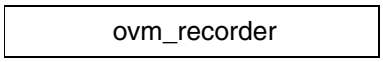
The 64 bits of packed data are converted to a real using the `$bits2real` system function.

unpack_time

```
virtual function time unpack_time ()
```

Unpacks the next 64 bits of the pack array and places them into a *time* variable.

ovm_recorder



The `ovm_recorder` class provides a policy object for recording `ovm_objects`. The policies determine how recording should be done.

A default recorder instance, `default_recorder`, is provided so the `ovm_object::record()` may be called without specifying a recorder.

Summary

```
class ovm_recorder;
    integer tr_handle = 0;
    radix_enum default_radix = OVM_HEX;
    bit physical = 1;
    bit abstract = 1;
    bit identifier = 1;
    recursion_policy_enum policy = OVM_DEFAULT_POLICY;
    virtual function void record_field (string name,
                                        ovm_bitstream_t value,
                                        int size,
                                        radix_enum radix=OVM_NORADIX);
    virtual function void record_object (string name,
                                         ovm_void value);
    virtual function void record_string (string name,
                                         string value);
    virtual function void record_time (string name,
                                       time value);
endclass
```

File

base/ovm_object.svh

Virtual

No

Members

`bit` **`abstract`** `= 1`

This bit provides a filtering mechanism for fields.

The abstract and physical settings allow an object to distinguish between two different classes of fields.

It is up to you, in the `ovm_object::do_pack()` and `ovm_object::do_unpack()` routines, to test the setting of this field if they want to use it as a filter.

`radix_enum` **`default_radix`** `= OVM_HEX`

This is the default radix setting if `record_field()` is called without a radix.

`bit` **`identifier`** `= 1`

This bit is used to specify whether or not an object's reference should be recorded when the object is recorded.

`bit` **`physical`** `= 1`

This bit provides a filtering mechanism for fields.

The abstract and physical settings allow an object to distinguish between two different classes of fields.

It is up to you, in the `ovm_object::do_pack()` and `ovm_object::do_unpack()` routines, to test the setting of this field if you want to use it as a filter.

`recursion_policy_enum` **`policy`** `= OVM_DEFAULT_POLICY`

Sets the recursion policy for recording objects.

The default policy is deep (which means to recurse an object).

`integer` **`tr_handle`** `= 0`

This is an integral handle to a transaction object. Its use is vendor specific.

A handle of `0` indicates there is no active transaction object.

Methods

new

record_field

```
virtual function void record_field (string name,
                                    ovm_bitstream_t value,
                                    int size,
```

```
                                        radix_enum radix=OVM_NORADIX)
```

Records an integral field (less than or equal to `4096` bits). `name` is the name of the field.

`value` is the value of the field to record.

`size` is the number of bits of the field which apply. `radix` is the radix to use for recording.

record_string

```
virtual function void record_string (string name,
                                     string value)
```

Records a string. `value` is the value of the field to record.

record_object

```
virtual function void record_object (string name,
                                     ovm_void value)
```

Record an object field. `name` is the name of the field.

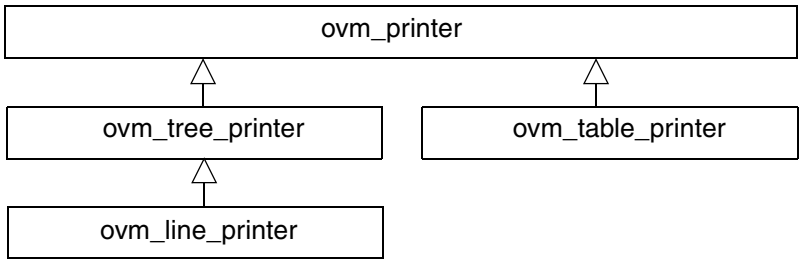
This method uses the recursion policy to determine whether or not to recurse into the object when it records.

record_time

```
virtual function void record_time (string name,
                                   time value)
```

Records a time value. `name` is the name of the field.

ovm_printer



The `ovm_printer` class provides a policy object for printing `ovm_objects` in various formats.

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

- The generic `ovm_printer` provides a raw, essentially un-formatted, dump of the object.
- The `ovm_table_printer` prints the object in a tabular form.
- The `ovm_tree_printer` prints the object in a tree form.
- The `ovm_line_printer` prints the information on a single line, but uses the same object separators as the tree printer.

Printers have knobs that you use to control the various settings. These knobs are contained in separate knob classes.

The following set of default printers are instantiated at time 0:

- `ovm_default_printer` (set to the `default_table_printer`)
- `ovm_default_tree_printer`
- `ovm_default_line_printer`
- `ovm_default_table_printer`

Summary

```
class ovm_printer;
    ovm_printer_knobs knobs = new;

    // Primary user level functions called from ovm_object::do_print()
    virtual function void print_field (string name,
                                       ovm_bitstream_t value,
```

```
                                    int size,
                                    radix_enum radix=OVM_NORADIX,
                                    byte scope_separator=".",
                                    string type_name="");
virtual function void print_object_header (string name,
                                           ovm_object value,
                                           byte scope_separator=".");
virtual function void print_object (string name,
                                    ovm_object value,
                                    byte scope_separator=".");
virtual function void print_string (string name,
                                    string value,
                                    byte scope_separator=".");
virtual function void print_time (string name,
                                  time value,
                                  byte scope_separator=".");
virtual function void print_generic (string name,
                                     string type_name,
                                     int size,
                                     string value,
                                     byte scope_separator=".");

virtual function void print_array_header (string name,
                                          int size,
                                          string arraytype="array",
                                          byte scope_separator=".");
virtual function void print_array_range (int min,
                                         int max);
virtual function void print_array_footer (int size=0);

// Primary derived class overrides for creating new printers.
virtual function void print_header ();
virtual function void print_footer ();
virtual protected function void print_id (string id,
                                          byte scope_separator=".");
virtual protected function void print_type_name (string name,
                                                 bit is_object=0);
virtual protected function void print_size (int size=-1);
virtual protected function void print_newline (bit do_global_indent=1);
virtual protected function void print_value (ovm_bitstream_t value,
                                             int size,
```

```
                                                    radix_enum radix=OVM_NORADIX);
    virtual protected function void print_value_object (ovm_object value);
    virtual protected function void print_value_string (string value);
    virtual protected function void print_value_array (string value="",
                                                       int size=0);
endclass
```

Note: The derived printer classes (`ovm_tree_printer`, `ovm_line_printer`, and `ovm_table_printer`) do not add any new user level APIs. However, they do add new knobs, but the visible API of printing is the same for all printers.

File

base/ovm_printer.svh

Virtual

No

Members

`ovm_printer_knobs` **`knobs`** `= new`

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

Each derivative printer class overloads the knobs variable with the specific knob class that applies to that printer. In this way, you always have direct access to the knobs by way of the `knobs` variable.

Global Variables

ovm_default_line_printer

```
ovm_line_printer default_line_printer = new
```

The line printer is a global object that can be used with `ovm_object::do_print()` to get single-line style printing.

ovm_default_tree_printer

```
ovm_tree_printer default_tree_printer = new
```

The tree printer is a global object that can be used with `ovm_object::do_print()` to get multi-line tree style printing.

ovm_default_table_printer

```
ovm_table_printer default_table_printer = new
```

The table printer is a global object that can be used with `ovm_object::do_print()` to get tabular style printing.

ovm_default_printer

```
ovm_printer default_printer = default_table_printer
```

The default printer is a global object that is used by `ovm_object::print()` or `ovm_object::sprint()` when no specific printer is set.

The default printer may be set to any legal `ovm_printer` derived type, including the global line, tree, and table printers described above.

Methods

print_array_header

```
virtual function void print_array_header (string name,
                                          int size,
                                          string arraytype="array",
                                          byte scope_separator=".")
```

Prints the header of an array. This function is called before each individual element is printed. `print_array_footer()` is called to mark the completion of array printing.

print_array_footer

```
virtual function void print_array_footer (int size=0)
```

Prints the header of a footer. This function marks the end of an array print. Generally, there is no output associated with the array footer, but this method lets the printer know that the array printing is complete.

print_array_range

```
virtual function void print_array_range (int min,
                                         int max)
```

Prints a range using ellipses for values. This method is used when honoring the array knobs for partial printing of large arrays.

This function should be called after the start elements have been printed and before the end elements have been printed.

print_field

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

Prints an integral field. `name` is the name of the field.

`value` is the value of the field. `size` is the number of bits of the field (maximum is `4096`).

`radix` is the radix to use for printing—the printer knob for radix is used if no `radix` is specified.

`scope_separator` is used to find the leaf name since many printers only print the *leaf* name of a field.

Typical values for the separator are `.` (dot) or `[` (open bracket).

print_generic

```
virtual function void print_generic (string name,
                                     string type_name,
                                     int size,
                                     string value,
                                     byte scope_separator=".")
```

Prints a generic value.

The value is specified as a *string* and the *type* name is supplied.

print_footer

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

When creating a new printer type, this method is used to print footer information.

The method is called when the current depth is `0`, after all fields have been printed.

print_header

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

When creating a new printer type, this method is used to print header information.

The method is called when the current depth is `0`, before any fields have been printed.

print_id

```
virtual protected function void print_id (string id,
                                          byte scope_separator=".")
```

When creating a new printer type, this method is used to print a field's name.

The intent of the separator is to mark where the leaf name starts if the printer only prints the leaf name of the identifier.

This function is called with a fully qualified name for the field.

print_newline

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

When creating a new printer type, this method is used to indicate a new line. It is up to the printer to determine how to display new lines.

The `do_global_indent` bit indicates whether or not the call to `print_newline()` should honor the indent knob.

print_object

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

Prints an object. Whether the object is recursed depends on a variety of knobs, such as the *depth* knob; if the current depth is at or below the depth setting, then the object is not recursed.

Note: By default, the children of `ovm_components` are printed. To turn this behavior off, you must set the `ovm_component::print_enabled` bit to `0` for the specific children you do not want automatically printed.

print_object_header

```
virtual function void print_object_header (string name,
                                           ovm_object value,
```

```
                                                byte scope_separator=".")
```

Prints the header of an object.

This function is called when an object is printed by reference.

For this function, the object will not be recursed.

print_size

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

When creating a new printer type, this method is used to print a field's size.

A `size` value of `-1` indicates that no size is available.

print_string

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

Prints a string field.

print_time

```
virtual function void print_time (string name,
                                  time value,
                                  byte scope_separator=".")
```

Prints a time value. `name` is the name of the field, and `value` is the value to print.

The print is subject to the `$timeformat` system task for formatting time values.

print_type_name

```
virtual protected function void print_type_name (string name,
                                                 bit is_object=0)
```

When creating a new printer type, this method is used to print a field's type.

The `is_object` bit indicates that the item being printed is a `ovm_object` derived type.

print_value

```
virtual protected function void print_value (ovm_bitstream_t value,
                                             int size,
```

```
                                                    radix_enum radix=OVM_NORADIX)
```

When creating a new printer type, this method is used to print an integral field's value.

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

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

print_value_array

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

When creating a new printer type, this method is used to print an array's value.

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

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

print_value_object

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

When creating a new printer type, this method is used to print a unique identifier associated with an object.

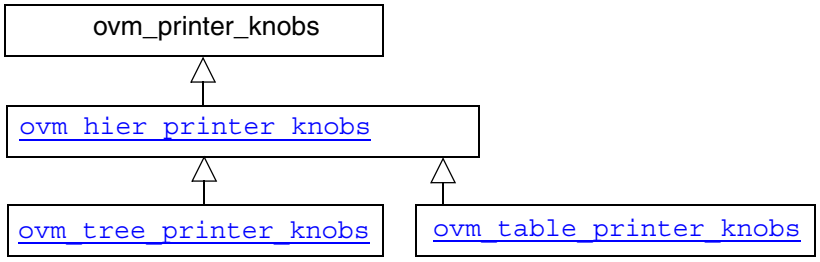
print_value_string

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

When creating a new printer type, this method is used to print a string field's value.

Policy Knobs

ovm_printer_knobs



The `ovm_printer_knobs` classes provide you with formatting control over the various printers. Each printer has a knob object that can be set to modify how the printer formats information.

All of the knobs are variables. Most of the knobs exist in the `ovm_printer_knobs` base class. The derivative classes provide extra controls that apply only to those printer types.

Summary

```
class ovm_printer_knobs;
    int column = 0;
    int max_width = 999;
    string truncation = "+";
    bit header = 1;
    bit footer = 1;
    int global_indent = 0;
    bit full_name = 1;
    bit identifier = 1;
    int depth = -1;
    bit reference = 1;
    bit type_name = 1;
    bit size = 1;
    radix_enum default_radix = OVM_HEX;

    int begin_elements = 5;
    int end_elements = 5;
    bit show_radix = 1;
```

```
    int mcd = OVM_STDOUT;

    string bin_radix = "'b";
    string oct_radix = "'o";
    string dec_radix = "'d";
    string unsigned_radix = "'d";
    string hex_radix = "'h";

    function string get_radix_str(radix_enum radix);
endclass

class ovm_hier_printer_knobs extends ovm_printer_knobs; string indent_str = "";
    bit show_root = 0;
    string indent_str = "  ";
endclass

class ovm_table_printer_knobs extends ovm_hier_printer_knobs;
    int name_width = 25;
    int type_width = 20;
    int size_width = 5;
    int value_width = 20;
endclass

class ovm_tree_printer_knobs extends ovm_hier_printer_knobs;
    string separator = "{}";
endclass
```

File

base/ovm_printer.svh

Virtual

No

Members

ovm_printer_knobs

```
int begin_elements = 5
```

This defines the number of elements at the head of a list that should be printed.

`string` **`bin_radix`** `= "'b"`

This string should be prepended to the value of an integral type when a radix of `OVM_BIN` is used for the radix of the integral object.

`int` **`column`** `= 0`

This is the column pointer, which is the current column that the printer is pointing to.

This is useful for derivative printers where column information is important, such as the `ovm_table_printer`.

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

This string should be prepended to the value of an integral type when a radix of `OVM_DEC` is used for the radix of the integral object.

Note: When a negative number is printed, the radix is not printed since only signed decimal values can print as negative.

`radix_enum` **`default_radix`** `= OVM_HEX`

This knob sets the default radix to use for integral values when a radix of `OVM_NORADIX` is supplied to the `print_field()` method.

`int` **`depth`** `= -1`

This knob indicates how deep to recurse when printing objects.

A depth of `-1` means to print everything.

`int` **`end_elements`** `= 5`

This defines the number of elements at the end of a list that should be printed.

`bit` **`footer`** `= 1`

This bit indicates whether the `print_footer()` function should be called when an object is printed.

If it is desired for a footer to be suppressed, then this bit should be set to `0`.

`bit` **`full_name`** `= 1`

This bit indicates whether the printer should print the full name of an identifier or just the leaf name when `print_id()` is called.

The line, table, and tree printers ignore this bit and always print only the *leaf* name.

`int` **`global_indent`** `= 0`

This is the number of columns of indentation to add whenever a *newline* is printed.

`bit` **`header`** `= 1`

This bit indicates whether the `print_header()` function should be called when an object is printed.

If it is desired for a header to be suppressed, then this bit should be set to `0`.

`string` **`hex_radix`** `= "'h"`

This string should be prepended to the value of an integral type when a radix of `OVM_HEX` is used for the radix of the integral object.

`bit` **`identifier`** `= 1`

This bit indicates whether the printer should print an identifier when `print_id()` is called.

This is useful in cases where you just want the values of an object, but no identifiers.

`int` **`max_width`** `= 999`

This is the maximum column width to use for a printer. If the current column reaches the maximum width, then nothing is printed until a *newline* is printed.

`integer` **`mcd`** `= OVM_STDOUT`

This is a file descriptor, or multi-channel descriptor, that specifies where the print output should be directed.

By default, the output goes to the standard output of the simulator.

`string` **`oct_radix`** `= "'o"`

This string should be prepended to the value of an integral type when a radix of `OVM_OCT` is used for the radix of the integral object.

`bit` **`reference`** `= 1`

This bit indicates whether the printer should print a unique reference ID for an `ovm_object` type.

The behavior of this knob is simulator dependent.

`bit` **`show_radix`** `= 1`

Indicates whether the radix string (`'h`, and so on) should be prepended to an integral value when one is printed.

`bit` **`size`** `= 1`

This bit indicates whether the printer should print the size of the fields that it is printing.

In some cases, printing the size obscures important aspects of the data being printed, so this information can be turned off.

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

Used to define the truncation character when a field is too large for the output.

For example, the table printer uses this character to truncate fields so that columns do not overflow.

`bit` **`type_name`** `= 1`

This bit indicates whether the printer should print the type name of the fields that it is printing.

In some cases, printing of the `type_name` obscures the important aspects of the data being printed, so this information can be turned off.

`string` **`unsigned_radix`** `= "'d"`

This is the string which should be prepended to the value of an integral type when a radix of `OVM_UNSIGNED` is used for the radix of the integral object.

ovm_hier_printer_knobs

`string` **`indent_str`** `= "  "`

This knob specifies the string to use for indentations.

By default, two spaces are used to indent each depth level.

The string can be set to any string and the string will be replicated for the current depth when indentation is done.

`bit` **`show_root`** `= 0`

This setting indicates whether or not the initial object that is printed (when current depth is `0`) prints the full path name. By default, the first object is treated like all other objects and only the *leaf* name is printed.

ovm_table_printer_knobs

`int` **`name_width`** `= 25`

This knob sets the width of the name column in the table. If this knob is set to `0`, then the name column will not be printed.

`int` **`size_width`** `= 5`

This knob sets the width of the size column in the table. If this knob is set to `0`, then the size column will not be printed.

`int` **`type_width`** `= 20`

This knob sets the width of the type column in the table. If this knob is set to `0`, then the type column will not be printed.

`int` **`value_width`** `= 20`

This knob sets the width of the value column in the table. If this knob is set to `0`, then the value column will not be printed.

ovm_tree_printer_knobs

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

The separator string is a two character string. The first character is printed when an object is traversed; it represents the start of the object value.

The second character is printed after the last field in the object has been printed; it represents the end of the object value.

Methods

get_radix_str

```
function string get_radix_str (radix_enum radix)
```

Converts the `radix` from an enumerated to a printable radix according to the radix printing knobs (`bin_radix`, and so on).

Printer Examples

The following examples show the output of a simple data object using the four styles of printer:

- Generic
- Line
- Tree
- Table

Example 1-1 on page 152 shows the output from a *generic* printer.

Example 1-1 Generic Printer, ovm_printer

```
c1 (container)(@1013)
c1.d1 (mydata)(@1022)
c1.d1.v1 (integral) (32) 'hcb8f1c97
c1.d1.e1 (enum) (32) THREE
c1.d1.str (string) (2) hi
c1.value (integral) (12) 'h2d
```

Example 1-2 on page 153 shows the output from a *line* printer.

Example 1-2 Line Printer, ovm_line_printer

```
c1: (container@1013) { d1: (mydata@1022) { v1: 'hcb8f1c97 e1: THREE str: hi }
value: 'h2d }
```

Example 1-3 on page 153 shows the output from a *tree* printer.

Example 1-3 Tree Printer, ovm_tree_printer

```
c1: (container@1013) {
    d1: (mydata@1022) {
         v1: 'hcb8f1c97
         e1: THREE
         str: hi

    value: 'h2d
}
```

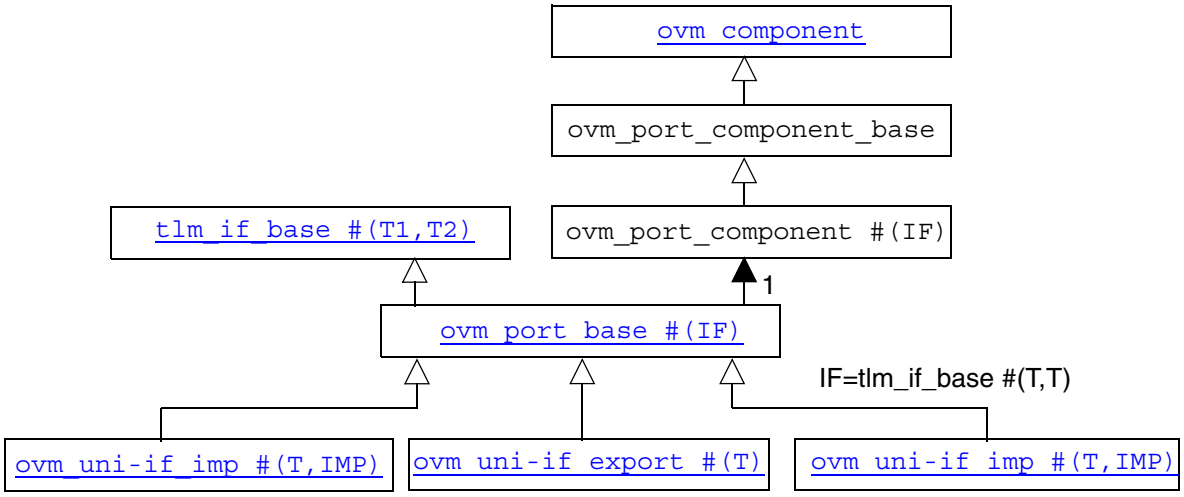
Example 1-4 on page 153 shows the output from a *table* printer.

Example 1-4 Table Printer, ovm_table_printer

```
-------------------------------------------------------------------------
Name        Type            Size        Value
-------------------------------------------------------------------------
c1          container       -           @1013
d1          mydata          -           @1022
v1          integral        32          'hcb8f1c97
e1          enum            32          THREE
str         string          2           hi
value       integral        12          'h2d
-------------------------------------------------------------------------
```

TLM Interfaces

Figure 1-1 Classes for TLM Communication



Unidirectional shown. For bidirectional interfaces, IF is tlm_if_base #(REQ,RSP).

The OVM TLM library defines several abstract, transaction-level interfaces. Each TLM interface consists of one or more methods used to transport data. TLM specifies the required behavior (semantic) of each method, but does not define their implementation. Classes (components) that implement a TLM interface must meet the specified semantic.

tlm_if_base #(T1,T2)

Summary

```
virtual class tlm_if_base #(type T1=int, type T2=int);
    virtual task put( T t );
    virtual task get( output T t );
    virtual task peek( output T t );
    virtual function bit try_put( T t );
    virtual function bit can_put();
    virtual function bit try_get( output T t );
    virtual function bit can_get();
    virtual function bit try_peek( output T t );
    virtual function bit can_peek();
    virtual task transport( REQ req , output RSP rsp );
    virtual function bit nb_transport( REQ req,output RSP rsp);
    virtual function void write( T t );
endclass
```

Virtual

Yes

Members

None

Methods

put

```
virtual task put (T t)
```

Sends a user-defined transaction of type *T*.

Components implementing the `put` method will block the calling thread if it cannot immediately accept delivery of the transaction.

get

```
virtual task get (output T t)
```

Provides a new transaction of type `T`.

The calling thread is blocked if the requested transaction cannot be provided immediately. The new transaction is returned in the provided output argument.

The implementation of `get` must regard the transaction as consumed. Subsequent calls to `get` must return a different transaction instance.

peek

```
virtual task peek (output T t)
```

Obtain a new transaction without consuming it.

If a transaction is available, then it is written to the provided output argument. If a transaction is not available, then the calling thread is blocked until one is available.

The returned transaction is not consumed. A subsequent `peek` or `get` will return the same transaction.

try_put

```
virtual function bit try_put (T t)
```

Sends a transaction of type `T`, if possible.

If the component is ready to accept the transaction argument, then it does so and returns `1`, otherwise it returns `0`.

can_put

```
virtual function bit can_put()
```

Returns `1` if the component is ready to accept the transaction; `0` otherwise.

try_get

```
virtual function bit try_get(output T t)
```

Provides a new transaction of type `T`. If a transaction is immediately available, then it is written to the provided output argument and 1 is returned. Otherwise, the output argument is not modified and `0` is returned.

can_get

```
virtual function bit can_get()
```

Returns 1 if a new transaction can be provided immediately upon request, `0` otherwise.

try_peek

```
virtual function bit try_peek(output T t)
```

Provides a new transaction without consuming it.

If available, a transaction is written to the output argument and `1` is returned. A subsequent `peek` or `get` will return the same transaction. If a transaction is not available, then the argument is unmodified and `0` is returned.

can_peek

```
virtual function bit can_peek()
```

Returns `1` if a new transaction is available; `0` otherwise.

transport

```
virtual task transport (REQ request, output RSP response)
```

Sends a transaction `request` for immediate execution.

A `response` is provided in the output argument upon return. The calling thread may block until the `response` is provided.

nb_transport

```
virtual function bit nb_transport (REQ request, output REQ response)
```

Sends a transaction `request` for immediate execution.

Execution must occur without blocking—for example, no waits and no simulation time passes. The `response` is provided in the output argument. If for any reason the `request` could not be delivered immediately, then a `0` must be returned; otherwise `1`.

write

```
virtual function void write (T t)
```

Broadcasts a user-defined transaction of type `T` to any number of listeners. The calling thread must not be blocked.

Port and Export Connectors

All of the methods of the TLM API are contained in the `tlm_if_base #(T1,T2)` class. Various subsets of these methods are combined to form primitive TLM interfaces, which are then paired in various ways to form more abstract "combination" TLM interfaces. Components that *require* a particular interface use *ports* to convey that requirement. Components that *provide* a particular interface use *exports* to convey its availability.

Communication between components is established by connecting ports to compatible exports, much like connecting module signal-level output ports to compatible input ports. The difference is that OVM ports and exports define communication at a much higher level of abstraction than low-level signals; they convey groups of functions and tasks that pass data as whole transactions (objects). The set of primitve and combination TLM interfaces afford many choices for designing components that communicate at the transaction level.

Uni-directional interfaces

Typically, the various forms of the `put`, `get`, and `peek` interfaces are used at one each end of a uni-directional channel to enable two or more components to communicate.

Bi-directional interfaces

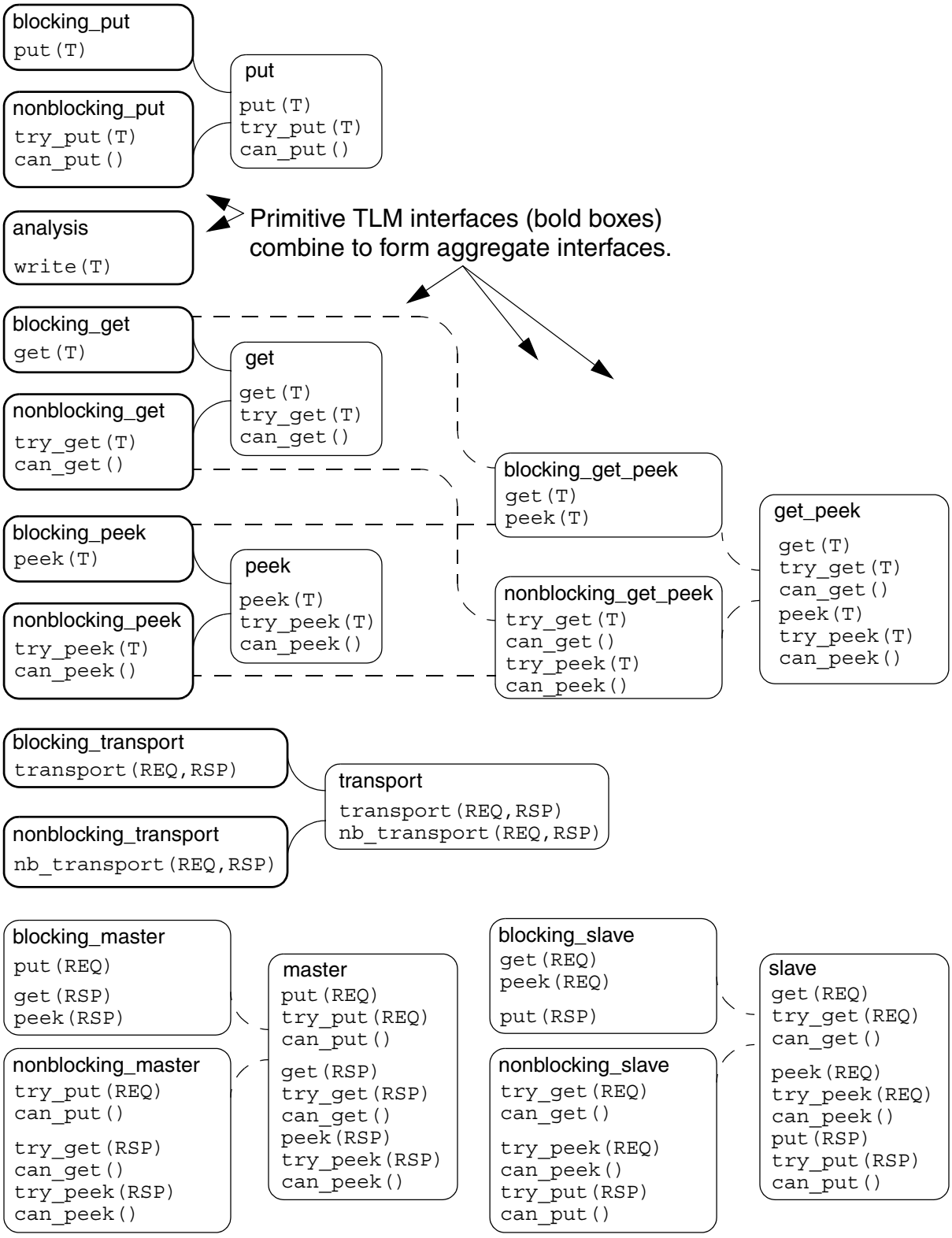
These primitive interfaces are combined to form the `master` and `slave` interfaces used in bi-directional channel communication.

The `transport` interface is used for a bi-directional channel where requests and responses are linked together in a non-pipelined fashion.

Analysis interface

An `analysis` interface is used by a component such as a monitor to publish a transaction to zero, one, or more subscribers. Typically, it will be used inside a monitor to publish a transaction observed on a bus to scoreboards and coverage objects.

Figure 1-2 TLM Interface Methods Map



Uni-Directional Interfaces

```
class ovm_uni-if_export #( type T = int )
    extends ovm_port_base #( tlm_if_base #(T,T) );
class ovm_uni-if_port #( type T = int )
    extends ovm_port_base #( tlm_if_base #(T,T) );
class ovm_uni-if_imp #( type T = int )
    extends ovm_port_base #( tlm_if_base #(T,T) );
```

Table 1-3 *uni-if*

blocking_put
nonblocking_put
put
blocking_get
nonblocking_get
get
blocking_peek
nonblocking_peek
peek
blocking_get_peek
nonblocking_get_peek
get_peek
analysis

Bi-Directional Interfaces

```
class ovm_bi-if_export #( type REQ = int , type RSP = int )
    extends ovm_port_base #( tlm_if_base #(REQ, RSP) );
class ovm_bi-if_port #( type REQ = int , type RSP = int )
    extends ovm_port_base #( tlm_if_base #(REQ, RSP) );
class ovm_bi-if_imp #( type REQ = int , type RSP = int )
    extends ovm_port_base #( tlm_if_base #(REQ, RSP) );
```

Table 1-4 *bi-if*

blocking_master
nonblocking_master
master
blocking_slave
nonblocking_slave
slave
blocking_transport
nonblocking_transport
transport

Ports and Exports

ovm_port_base #(IF)

`ovm_port_base` is the base class for all ports, exports, and implementations (`ovm_*_port`, `ovm_*_export`, and `ovm_*_imp`). The `ovm_port_base` extends `IF`, which is the type of the interface implemented by derived port, export, or implementation.

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

Summary

```
class ovm_port_base #(type IF=ovm_object) extends ovm_port_base_baseIF;
    function new(string name,
                 ovm_component parent,
                 ovm_port_type_e port_type,
                 int min_size=1, int max_size=1);
      function string get_name();
      virtual function string get_full_name();
      virtual function ovm_component get_parent();
      virtual function ovm_port_component_base get_comp();
      virtual function string get_type_name();
      function int max_size ();
      function int min_size ();
      function bit is_unbounded ();
      function bit is_port ();
      function bit is_export ();
      function bit is_imp ();
      function int size ();
      function ovm_port_base #(IF) get_if(int index=0);
      function void get_type_name (int index);
      function void debug_connected_to (int level=0, int max_level=-1);
      function void debug_provided_to  (int level=0, int max_level=-1);
      function void resolve_bindings();
      function void connect(this_type provider);
endclass
```

File

base/ovm_port_base.svh

Virtual

Yes

Parameters

`type IF = ovm_void`

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

Methods

new

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

The first two arguments are the normal `ovm_component` constructor arguments. The *`port_type`* can be one of `OVM_PORT`, `OVM_EXPORT`, or `OVM_IMPLEMENTATION`. The *`min_size`* and *`max_size`* specify the minimum and maximum number of implementation (imp) ports that must be connected to this port base by the end of elaboration. Setting *`max_size`* to `OVM_UNBOUNDED_CONNECTIONS` sets no maximum, i.e., an unlimited number of connections are allowed.

connect

```
function void connect (ovm_port_base #(IF) provider)
```

Connects this port to the given *`provider`* port.

If this port is an `OVM_PORT` type, the *`provider`* can be a parent port, or a sibling export or implementation port. If the parent / sibling relationship is violated, a warning is issued.

If this port is an `OVM_EXPORT` type, the *`provider`* can be a child export or implementation port. If the parent / sibling relationship is violated, a warning is issued.

If this port is an OVM_IMPLEMENTATION port, an error is produced, as imp ports can only be bound to the component that implements the interface. You may not call connect on an implementation port.

debug_connected_to

debug_provided_to

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

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

The debug_provided_to method outputs a visual display of the port/export network that ultimately connect to this port (i.e., the port's fanin).

These methods must not be called before the end_of_elaboration phase, as port connections have not yet been resolved.

get_name

get_full_name

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

Returns the leaf name and full path name to this port.

get_parent

```
virtual function ovm_component get_parent()
```

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

get_comp

```
virtual function ovm_port_component_base get_comp()
```

Ports are considered components. However, they do not inherit ovm_component. They *contain* a special component instance that serves as a proxy to this port. This method returns a handle to the internal instance of the proxy component.

get_type_name

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

Returns the type name to this port. Derived port classes must implement this method to return the concrete type. Otherwise, only a generic "ovm_port", "ovm_export" or "ovm_implementation" is returned.

is_port

is_export

is_imp

```
function bit is_port()
function bit is_export()
function bit is_imp()
```

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

is_unbounded

```
function bit is_unbounded()
```

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

max_size

min_size

```
function int max_size()
function int min_size()
```

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

set_default_index

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

Sets the default implementation port to use when calling an interface method. This method should only be called on OVM_EXPORT types. The value must not be set before the end_of_elaboration phase, when port connections have not yet been resolved.

size

```
function int size()
```

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

resolve_bindings

```
function void resolve_bindings()
```

Resolves all port connections, producing errors for any port whose *`min_size`* or *`max_size`* limits have not been met. This method is automatically called just before the start of the `end_of_elaboration` phase.

get_if

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

Returns the implementation (imp) port at the given index from the array of imps this port is connected to. Use `size` to get the valid range for index. This method must not be called before the `end_of_elaboration` phase, as port connections have not yet been resolved.

ovm_*uni-if*_port #(T)

An ovm_*uni-if*_port is a uni-directional connector that requires interfaces from other components. It gets these interfaces by connecting to an ovm_*uni-if*_*if*_port in a parent component or an ovm_*uni-if*_imp in a sibling component. There is one export class for each uni-directional interface. The ovm_*uni-if*_port classes inherit all methods (e.g., connect) from its ovm_port_base class.

Summary

```
class ovm_uni-if_port (type T=int) extends ovm_port_base #(tlm_if_base #(T,T));
    function new(string name, ovm_component parent,
                 int min_size=1,
                int max_size=1);
endclass
```

File

tlm/ovm_ports.svh

Parameters

```
type T = int
```

The type of transaction to be communicated across the export.

Methods

new

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

The *name* and *parent* are the standard ovm_component constructor arguments. The min_size and max_size specify the minimum and maximum number of interfaces that must have been supplied to this port by the end of elaboration.

ovm_*bi-if*_port #(REQ,RSP)

An ovm_*bi-if*_port is a bi-directional connector that requires interfaces from other components. It gets these interfaces by connecting to an ovm_*bi-if*_port in a parent component or an ovm_*bi-if*_imp in a sibling component. There is one export class for each bi-directional interface. The ovm_*bi-if*_port classes inherit all methods (e.g., connect) from its ovm_port_base class.

Summary

```
class ovm_bi-if_port #(type REQ=int, RSP=int)
                            extends ovm_port_base #(tlm_if_base #(REQ,RSP));
    function new(string name, ovm_component parent,
                 int min_size=1,
                 int max_size=1);
endclass
```

File

tlm/ovm_ports.svh

Parameters

`type REQ = int`

The type of request transaction to be communicated across the export.

`type RSP = int`

The type of response transaction to be communicated across the export.

Methods

new

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

The *name* and *parent* are the standard ovm_component constructor arguments. The min_size and max_size specify the minimum and maximum number of interfaces that must have been supplied to this port by the end of elaboration.

ovm_*uni-if*_export #(T)

An ovm_*uni-if*_export is a uni-directional connector that provides interfaces to other components. It provides these interfaces by connecting to a compatible ovm_*uni-if*_export or ovm_*uni-if*_imp in a child component. There is one export class for each uni-directional interface. The ovm_*uni-if*_export classes inherit all methods (e.g., the connect) from its ovm_port_base class.

Summary

```
class ovm_uni-if_export #(type T=int) extends ovm_port_base  #(tlm_if_base #(T,T));
    function new(string name, ovm_component parent,
                 int min_size=1,
                int max_size=1);

endclass
```

File

tlm/ovm_exports.svh

Parameters

`type T = int`

The type of transaction to be communicated across the export.

Methods

new

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

The *name* and *parent* are the standard ovm_component constructor arguments. The *min_size* and *max_size* specify the minimum and maximum number of interfaces that must have been supplied to this port by the end of elaboration.

ovm_*bi-if*_export #(REQ,RSP)

An ovm_*bi-if*_export is a bi-directional connector that provides interfaces to other components. It provides these interfaces by connecting to an ovm_*bi-if*_export or ovm_*bi-if*_imp in a child component. There is one port class for each bi-directional interface. The ovm_*bi-if*_export classes inherit all methods (e.g., connect) from its ovm_port_base class.

Summary

```
class ovm_bi-if_export #(type REQ=int, RSP=int)
                              extends ovm_port_base #(tlm_if_base #(REQ,RSP));
    function new(string name, ovm_component parent,
                 int min_size=1,
                 int max_size=1);
endclass
```

File

tlm/ovm_exports.svh

Parameters

`type REQ = int`

The type of request transaction to be communicated across the export.

`type RSP = int`

The type of response transaction to be communicated across the export.

Methods

new

`function` **`new`**`(string name, ovm_component parent, int min_size=1, int max_size=1)`

The *name* and *parent* are the standard ovm_component constructor arguments. The min_size and max_size specify the minimum and maximum number of interfaces that must have been supplied to this port by the end of elaboration.

ovm_*uni-if*_imp #(T,IMP)

ovm_*uni-if*_imp provides the implementations of the methods in tlm_if_base to ports and exports that require it. The actual implementation of the methods that comprise tlm_if_base are defined in an object of type IMP, a handle to which is passed in to the constructor.

Summary

```
class ovm_uni-if_imp #(type T=int, type IMP=int) extends ovm_port_base
                     #(tlm_if_base #(T));
    function new(string name, IMP imp);
endclass
```

File

tlm/ovm_imps.svh

Parameters

`type T = int`

Type of transactions to be communicated across the underlying interface.

`type IMP = int`

Type of the parent of this implementation.

Methods

new

```
function new(string name, IMP imp)
```

The *name* is the normal first argument to an ovm_component constructor. The imp is a slightly different form for the second argument to the ovm_component constructor, which is of type IMP and defines the type of the parent.

Since it is the purpose of an *imp* class to provide an implementation of a set of interface tasks and functions, the particular set of tasks and functions available for each ovm_*uni-if*_imp class is dependent on the type of the interface it implements, which is the particular TLM interface it extends.

ovm_*bi-if*_imp #(REQ,RSP,IMP)

`ovm_bi-if_imp` provides the implementations of the methods in `tlm_if_base` to ports and exports that require it. The actual implementation of the methods that comprise `tlm_if_base` are defined in an object of type `IMP`, a handle to which is passed in to the constructor.

Summary

```
class ovm_bi-if_imp (type REQ=int, type RSP=int,
                     type IMP=int,
                     type REQ_IMP=IMP, type RSP_IMP=IMP)
                     extends ovm_port_base #(tlm_blocking_master_if #(REQ, RSP));
    function new(string name, IMP imp, REQ_IMP req_imp=imp, RSP_IMP rsp_imp=imp);
endclass
```

File

tlm/ovm_imps.svh

Parameters

`type REQ = int`

Type of transactions to be sent by the master or received by the slave.

`type RSP = int`

Type of transactions to be received by the master or sent by the slave.

`type IMP = int`

Type of the parent of this implementation.

`type REQ_IMP = IMP`

Type of the object that implements the request side of the interface.

`type RSP_IMP = IMP`

Type of the object that implements the response side of the interface.

Methods

new

```
    function new(string name,
```

```
                IMP imp,
                REQ_IMP req_imp=imp,
                RSP_IMP rsp_imp=imp)
```

The `name` is the normal first argument to an `ovm_component` constructor. The `imp` is a slightly different form for the second argument to the `ovm_component` constructor, which is of type `IMP` and defines the type of the parent. The `req_imp` and `rsp_imp` are optional. If they are specified, then they must point to the underlying implementation of the request and response methods; `tlm_req_rsp_channel`, `req_imp` and `rsp_imp` are the request and response FIFOs.

sqr_if_base #(REQ,RSP)

seq_if_base #(REQ,RSP)

This class defines an interface for sequence drivers to communicate with sequencers. The driver requires the interface via a port, and the sequencer implements it and provides it via an export.

Summary

```
virtual class sqr_if_base #(type REQ=ovm_object, RSP=REQ);
  virtual task          get_next_item      (output REQ request);
  virtual task          try_next_item      (output REQ request);
  virtual function void item_done          (input RSP response=null);
  virtual task          wait_for_sequences ();
  virtual function bit  has_do_available   ();
  virtual task          get                (output REQ request);
  virtual task          peek               (output REQ request);
  virtual task          put                (input RSP response);
endclass
```

File

tlm/sqr_ifs.svh

Parameters

`type REQ = int`

Type of the request transaction.

`type RSP = REQ`

Type of the response transaction.

Methods

get

```
task get (output REQ request)
```

Retrieves the next available item from a sequence. The call blocks until an item is available. The following steps occur on this call:

1 — Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, wait for a requesting unlocked relevant sequence, then re-arbitrate.

2 — The chosen sequence will return from `wait_for_grant`

3 — The chosen sequence `pre_do` is called

4 — The chosen sequence item is randomized

5 — The chosen sequence `post_do` is called

6 — Indicate `item_done` to the sequencer

7 — Return with a reference to the item

When `get` is called, `item_done` may not be called. A new item can be obtained by calling `get` again, or a response may be sent using either `put`, or `rsp_port.write`.

get_next_item

```
task get_next_item (output REQ request)
```

Retrieves the next available item from a sequence. The call will block until an item is available. The following steps occur on this call:

1 — Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, wait for a requesting unlocked relevant sequence, then re-arbitrate.

2 — The chosen sequence will return from `wait_for_grant`

3 — The chosen sequence `pre_do` is called

4 — The chosen sequence item is randomized

5 — The chosen sequence `post_do` is called

6 — Return with a reference to the item

Once `get_next_item` is called, `item_done` must be called to indicate the completion of the request to the sequencer. This will remove the request item from the sequencer fifo.

has_do_available

```
function bit has_do_available ()
```

Indicates whether a sequence item is available for immediate processing. Implementations should return 1 if an item is available, 0 otherwise.

item_done

```
function void item_done (RSP response = null)
```

Indicates that the request is completed to the sequencer. Any `wait_for_item_done` calls made by a sequence for this item will return.

The current item is removed from the sequencer fifo.

If a response item is provided, then it will be sent back to the requesting sequence. The response item must have it's sequence ID and transaction ID set correctly, using the `set_id_info` method:

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

Before `item_done` is called, any calls to `peek` will retrieve the current item that was obtained by `get_next_item`. After `item_done` is called, `peek` will cause the sequencer to arbitrate for a new item.

peek

```
task peek (output REQ request)
```

Returns the current request item if one is in the sequencer fifo. If no item is in the fifo, then the call will block until the sequencer has a new request.

The following steps will occur if the sequencer fifo is empty:

1 — Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, wait for a requesting unlocked relevant sequence, then re-arbitrate.

2 — The chosen sequence will return from `wait_for_grant`

3 — The chosen sequence `pre_do` is called

4 — The chosen sequence item is randomized

5 — The chosen sequence `post_do` is called

Once a request item has been retrieved and is in the sequencer fifo, subsequent calls to `peek` will return the same item. The item will stay in the fifo until either `get` or `item_done` is called.

put

```
task put (RSP response)
```

Sends a response back to the sequence that issued the request. Before the response is put, it must have it's sequence ID and transaction ID set to match the request. This can be done using the `set_id_info` call:

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

This task will not block. The response will be put into the sequence response_queue or it will be sent to the sequence response handler.

try_next_item

```
task try_next_item (output REQ request)
```

Retrieves the next available item from a sequence if one is available. Otherwise, the function returns immediately with *`request`* set to `null`.

The following steps occur on this call:

1 — Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, return `null`.

2 — The chosen sequence will return from `wait_for_grant`

3 — The chosen sequence `pre_do` is called

4 — The chosen sequence item is randomized

5 — The chosen sequence `post_do` is called

6 — Return with a reference to the item

Once `try_next_item` is called, `item_done` must be called to indicate the completion of the request to the sequencer. This will remove the request item from the sequencer fifo.

wait_for_sequences

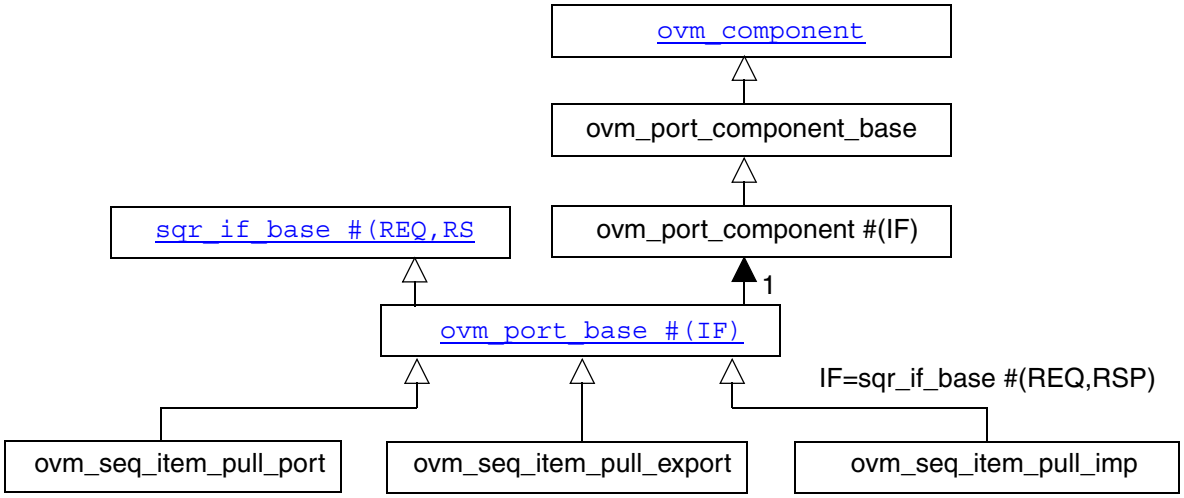
```
task wait_for_sequences ()
```

Waits for a sequence to have a new item available. The default implementation in the sequencer delays *`pound_zero_count`* delta cycles. (This variable is defined in ovm_sequencer_base.) User-derived sequencers may override its `wait_for_sequences` implementation to perform some other application-specific implementation.

ovm_seq_item_pull_*port_type* #(REQ,RSP)

OVM provides a port, export, and imp connector for use in sequencer-driver communication. All have standard port connector constructors, except that `ovm_seq_item_pull_port`'s default *`min_size`* argument is 0; it can be left unconnected.

Figure 1-3 Port Classes for Sequencer Communication



Summary

```
class class ovm_seq_item_pull_port #(type REQ=int, type RSP=REQ)
                                  extends ovm_port_base #(sqr_if_base #(REQ, RSP));
    function new(string name, ovm_component parent,
                 int min_size=0, int max_size=1);
endclass
class ovm_seq_item_pull_export #(type REQ=int, type RSP=REQ)
                                  extends ovm_port_base #(sqr_if_base #(REQ, RSP));
    function new(string name, ovm_component parent,
                 int min_size=1, int max_size=1);
endclass
class ovm_seq_item_pull_imp #(type REQ=int, type RSP=REQ)
                                extends ovm_port_base #(sqr_if_base #(REQ, RSP));
    function new(string name, ovm_component parent,
                 int min_size=1, int max_size=1);
endclass
```

File

tlm/sqr_connections.svh

Parameters

`type REQ = int`

Type of the request transaction.

`type RSP = REQ`

Type of the response transaction.

Methods

new

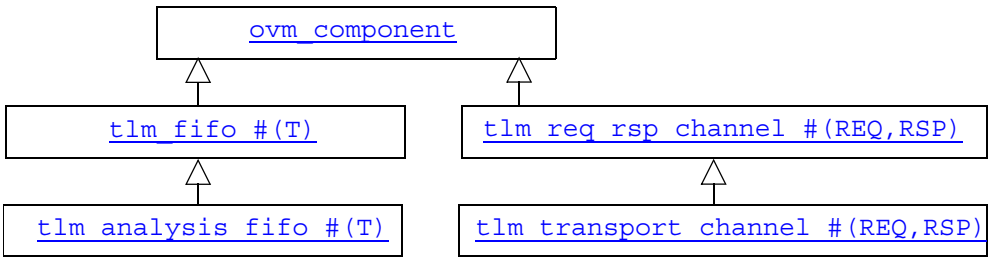
```
function new (string name, ovm_component parent=null,
              int min_size=1, int max_size=1)
```

Constructor method. The *`name`* and *`parent`* are the normal `ovm_component` constructor arguments. The *`min_size`* and *`max_size`* arguments are the normal port connector arguments. For `ovm_seq_item_pull_port`, the *`min_size`* default is `0`, which means it can be left unconnected.

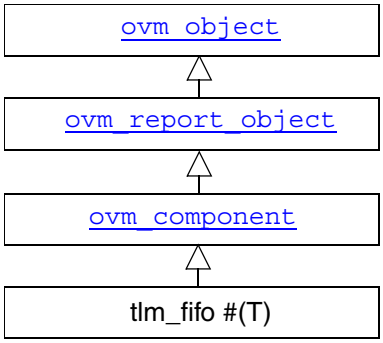
Built-In TLM Channels

The OVM supplies a FIFO channel and a variety of interfaces to access it. The interfaces have both blocking and non-blocking forms. Because SystemVerilog does not support multiple inheritance, the FIFO has a collection of *imps* implementations of abstract interfaces that are used to access the FIFO. The FIFO is a named component and thus has a name and a location in the component hierarchy.

Figure 1-4 Predefined TLM Channels



tlm_fifo #(T)



The `tlm_fifo` is a FIFO that implements all the uni-directional TLM interfaces.

Summary

```
class tlm_fifo #(type T=int) extends ovm_component;
    function new (string name, ovm_component parent=null, int size=1);

    ovm_put_imp #(T, this_type)      put_export;
    ovm_get_peek_imp #(T, this_type) get_peek_export;

    ovm_analysis_port #(T) put_ap;
    ovm_analysis_port #(T) get_ap;

    function void flush ();
    function int  size ();
endclass
```

File

tlm/tlm_fifos.svh

Virtual

No

Parameters

```
type T = int
```

Type of transactions to be stored in the FIFO.

Members

put_export

```
ovm_put_imp #(T, tlm_fifo #(T)) put_export
```

The `put_export` provides both the blocking and non-blocking `put` interface methods:

```
task put (input T t)
function bit can_put ()
function bit try_put (input T t)
```

Any `put` port variant can connect and send transactions to the FIFO via this export, provided the transaction types match.

get_peek_export

```
ovm_get_peek_imp #(T, tlm_fifo #(T)) get_peek_export
```

The `get_peek_export` provides all the blocking and non-blocking `get` and `peek` interface methods:

```
task get (output T t)
function bit can_get ()
function bit try_get (output T t)
task peek (output T t)
function bit can_peek ()
function bit try_peek (output T t)
```

Any `get` or `peek` port variant can connect to and retrieve transactions from the FIFO via this export, provided the transaction types match.

put_ap

```
ovm_analysis_port #(T) put_ap
```

Transactions passed via `put` or `try_put` (via any port connected to the `put_export`) are sent out this port via its `write` method.

```
function void write (T t)
```

All connected analysis exports and imps will receive these transactions.

get_ap

```
ovm_analysis_port #(T) get_ap
```

Transactions passed via `get`, `try_get`, `peek`, or `try_peek` (via any port connected to the `get_peek_export`) are sent out this port via its `write` method.

```
function void write (T t)
```

All connected analysis exports and imps will receive these transactions.

Methods

new

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

The *`name`* and *`parent`* are the normal `ovm_component` constructor arguments. The *`parent`* should be *`null`* if the `tlm_fifo` is going to be used in a statically elaborated construct (e.g., a `module`). The *`size`* indicates the maximum size of the FIFO; a value of zero indicates no upper bound.

flush

```
function void flush()
```

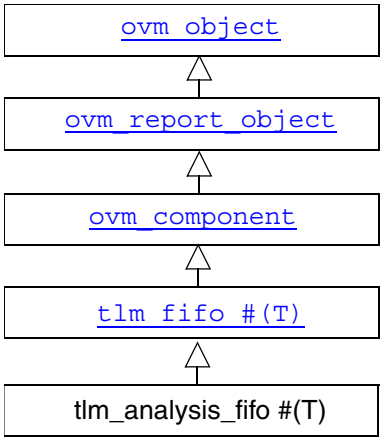
Removes all entries from the FIFO, after which `size` returns 0.

size

```
function int size()
```

This returns the number of entries in the FIFO.

tlm_analysis_fifo #(T)



An `analysis_fifo` is a `tlm_fifo` with an unbounded size and a `write` interface. It can be used any place an `ovm_subscriber` is used. Typical usage is as a buffer between an `analysis_port` in a monitor and an analysis component (a component derived from `ovm_subscriber`).

Summary

```
class tlm_analysis_fifo #(type T=int) extends tlm_fifo #(T);
    function new (string name, ovm_component parent=null);
    ovm_analysis_imp #(T, tlm_analysis_fifo #(T)) analysis_export;
endclass
```

File

tlm/tlm_fifos.svh

Virtual

No

Parameters

`type T = int`

Type of transactions to be stored in the FIFO.

Members

analysis_export

```
ovm_analysis_imp #(T, tlm_analysis_fifo #(T)) analysis_export
```

The `analysis_export` provides the `write` method to all connected analysis ports and parent exports:

```
function void write (T t)
```

Access via ports bound to this export is the normal mechanism for writing to an analysis FIFO. See `write` method on page 157 for more information.

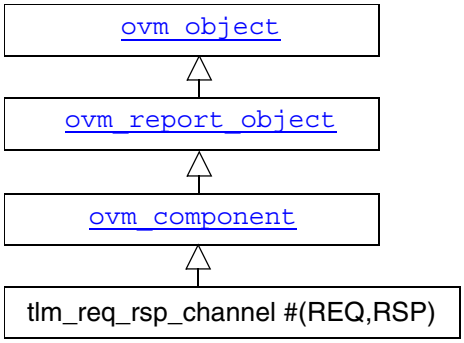
Methods

new

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

This is the standard `ovm_component` constructor. The *`name`* is the local name of this component. The `parent` should be left unspecified when this component is instantiated in statically elaborated constructs and must be specified when this component is a child of another OVM component.

tlm_req_rsp_channel #(REQ,RSP)



The `tlm_req_rsp_channel` contains a request FIFO of type `REQ` and a response FIFO of type `RSP`. These FIFOs can be of any size. This channel is particularly useful for dealing with pipelined protocols where the request and response are not tightly coupled.

Summary

```
class tlm_req_rsp_channel #(type REQ=int, type RSP=int) extends ovm_component;
  function new (string name, ovm_component parent=null,
                int request_fifo_size=1, int response_fifo_size=1);
  ovm_put_export #(REQ)      put request export;
  ovm_get_peek_export #(REQ) get peek response export;
  ovm_put_export #(RSP)      put response export;
  ovm_get_peek_export #(RSP) get peek response export;
  ovm_master_imp #(REQ, RSP, this_type,...) master export;
  ovm_slave_imp #(REQ, RSP, this_type,...)  slave export;
  ovm_analysis_port #(REQ)   request ap;
  ovm_analysis_port #(RSP)   response ap;
endclass
```

File

tlm/tlm_req_rsp.svh

Virtual

No

Parameters

`type REQ = int`

Type of the request transactions conveyed by this channel.

`type RSP = int`

Type of the reponse transactions conveyed by this channel.

Members

put_request_export

`ovm_put_export #(REQ)` **`put_request_export`**

The `put_export` provides both the blocking and non-blocking `put` interface methods to the request FIFO:

```
task put (input T t);
function bit can_put ();
function bit try_put (input T t);
```

Any `put` port variant can connect and send transactions to the request FIFO via this export, provided the transaction types match.

get_peek_response_export

`ovm_get_peek_export #(RSP)` **`get_peek_response_export`**

The `get_peek_export` provides all the blocking and non-blocking `get` and `peek` interface methods to the response FIFO:

```
task get (output T t)
function bit can_get ()
function bit try_get (output T t)
task peek (output T t)
function bit can_peek ()
function bit try_peek (output T t)
```

Any `get` or `peek` port variant can connect to and retrieve transactions from the response FIFO via this export, provided the transaction types match.

get_peek_request_export

`ovm_get_peek_export #(REQ)` **`get_peek_request_export`**

The `get_peek_export` provides all the blocking and non-blocking `get` and `peek` interface methods to the request FIFO:

```
task get (output T t)
function bit can_get ()
function bit try_get (output T t)
task peek (output T t)
function bit can_peek ()
function bit try_peek (output T t)
```

Any `get` or `peek` port variant can connect to and retrieve transactions from the request FIFO via this export, provided the transaction types match.

put_response_export

`ovm_put_export #(RSP)` **`put_response_export`**

The `put_export` provides both the blocking and non-blocking `put` interface methods to the response FIFO:

```
task put (input T t);
function bit can_put ();
function bit try_put (input T t);
```

Any `put` port variant can connect and send transactions to the response FIFO via this export, provided the transaction types match.

master_export

`ovm_master_imp #(REQ, RSP, this_type,...)` **`master_export`**

Exports a single interface that allows a master to `put` requests and `get` or `peek` responses. It is a combination of the `put_request_export` and `get_peek_response_export`.

slave_export

`ovm_slave_imp #(REQ, RSP, this_type,...)` **`slave_export`**

Exports a single interface that allows a slave to `get` or `peek` requests and to `put` responses. It is a combination of the `get_peek_request_export` and `put_response_export`.

request_ap

`ovm_analysis_port #(REQ)` **`request_ap`**

Transactions passed via `put` or `try_put` (via any port connected to the `put_request_export`) are sent out this port via its `write` method.

```
function void write (T t)
```

All connected analysis exports and imps will receive these transactions.

response_ap

```
ovm_analysis_port #(RSP) response_ap
```

Transactions passed via `put` or `try_put` (via any port connected to the `put_response_export`) are sent out this port via its `write` method.

```
function void write (T t)
```

All connected analysis exports and imps will receive these transactions.

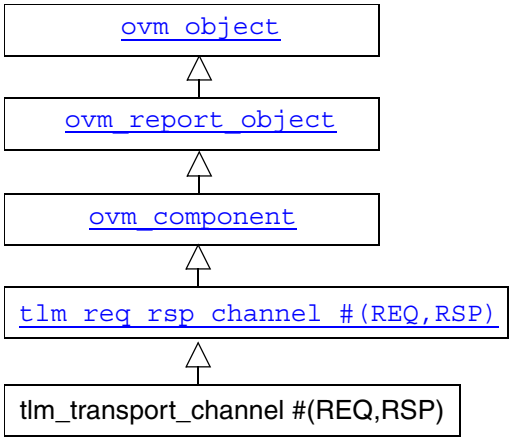
Methods

new

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

The `name` and `parent` are the standard `ovm_component` constructor arguments. The `parent` must be `null` if this component is defined within a static component such as a module, program block, or interface. The last two arguments specify the request and response FIFO sizes, which have default values of `1`.

tlm_transport_channel #(REQ,RSP)



A `tlm_transport_channel` is a `tlm_req_rsp_channel` that implements the `transport` interface. It is useful when modeling a non-pipelined bus at the transaction level. Because the requests and responses have a tightly coupled one-to-one relationship, the request and response FIFO sizes are both set to one.

Summary

```
class tlm_transport_channel #(type REQ=int, type RSP=int)
                                          extends tlm_req_rsp_channel #(REQ, RSP);
  function new (string name, ovm_component parent=null);
  ovm_transport_imp #(REQ,RSP,tlm_transport_channel #(REQ,RSP)) transport_export;
endclass
```

File

tlm/tlm_req_rsp.svh

Parameters

`type REQ = int`

Type of transactions to be passed to/from the request FIFO.

`type RSP = int`

Type of transactions to be passed to/from the response FIFO.

Members

transport_export

```
ovm_transport_imp#(REQ,RSP,tlm_transport_channel #(REQ,RSP)) transport_export
```

The `put_export` provides both the blocking and non-blocking `transport` interface methods to the response FIFO:

```
task transport(REQ request, output RSP response);
function bit nb_transport(REQ request, output RSP response);
```

Any `transport` port variant can connect to and send requests and retrieve responses via this export, provided the transaction types match. Upon return, the *`response`* argument carries the response to the *`request`*.

Methods

new

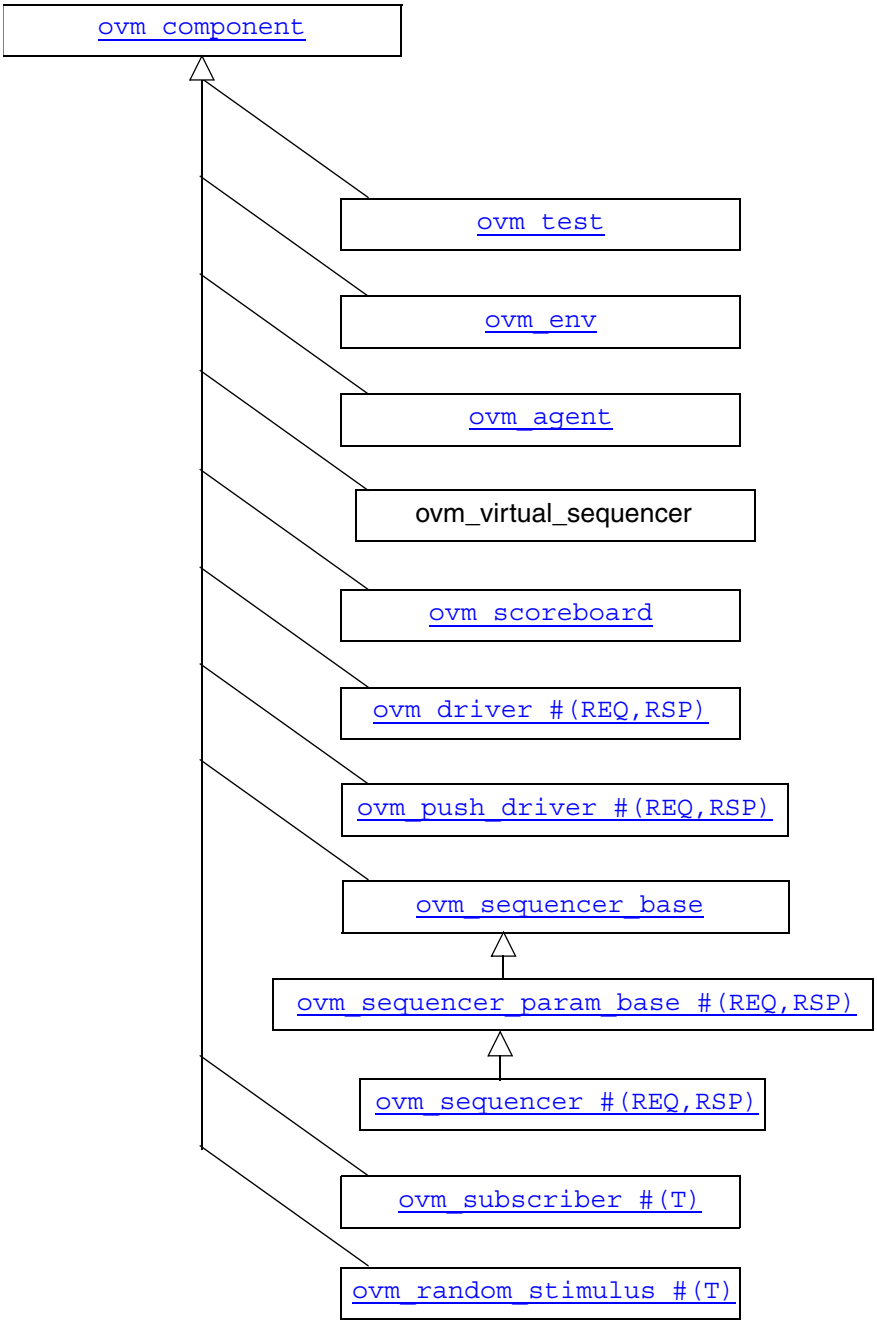
```
function new(string name, ovm_component parent=null)
```

The `name` and `parent` are the standard `ovm_component` constructor arguments. The *`parent`* must be `null` if this component is defined within a statically elaborated construct such as a module, program block, or interface.

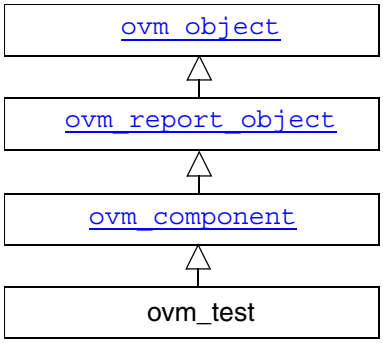
Components

Components form the foundation of the OVM. They encapsulate behavior of transactors, scoreboards, and other objects in a testbench. The `ovm_component` is the base class from which all component classes are derived.

Figure 1-5 Predefined Components and Specialized Component Base Classes



ovm_test



The `ovm_test` class is the virtual base class for the user-defined tests.

Summary

```
virtual class ovm_test extends ovm_component;
  function new (input string name, ovm_component parent);
endclass
```

File

methodology/ovm_test.svh

Virtual

Yes

Members

None

Methods

new

new (input string name, ovm_component parent)

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

Usage

The ovm_test virtual class should be used as the base class for the user-defined tests. Doing so provides the ability to select which test to execute by using the OVM_TESTNAME command line argument when used in conjunction with the run_test() task. For example:

```
> 'simulator command and switches' +OVM_TESTNAME=test_bus_retry
```

The run_test() task should be specified inside an initial block such as:

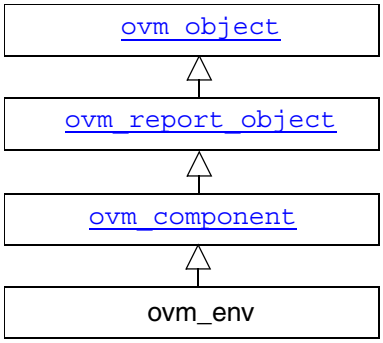
```
initial begin
  run_test();
end
```

This allows multiple tests to be compiled in and then selected for execution from the command line with random seeding—preventing the need for a recompilation.

If run_test() is used and +OVM_TESTNAME=*test_name* is specified, then the specified *test_name* is created by factory and executed. If the specified *test_name* cannot be created by the factory, then a fatal error occurs. If run_test() is used and OVM_TESTNAME is not specified, then all constructed components will be cycled through their simulation phases.

Deriving from ovm_test will allow you to distinguish tests from other component types using its inheritance. Also, tests will automatically inherit any new test-specific features that are added to ovm_test.

ovm_env



The ovm_env class is a top-level container component that provides phasing control for a hierarchy of components.

Summary

```
virtual class ovm_env extends ovm_component
    function new (string name=env, ovm_component parent=null);
    virtual function string get_type_name ();
endclass
```

File

base/ovm_env.svh

Virtual

Yes

Methods

new

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

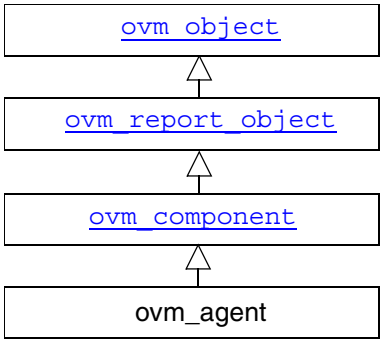
Creates and initializes an instance of this class using the normal constructor arguments for ovm_component: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

get_type_name

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

Returns “ovm_env”. Subclasses must override to return the derived type name.

ovm_agent



The `ovm_agent` virtual class should be used as the base class for the user-defined agents. Deriving from `ovm_agent` will allow you to distinguish agents from other component types also using its inheritance. Also, agents will automatically inherit any new agent-specific features that are added to `ovm_agent`.

While an agent's `build` function, inherited from `ovm_component`, can be implemented to define any agent topology, an agent typically contains three subcomponents: a driver, sequencer, and monitor. If the agent is active (signified by the `is_active` control field), the agent contains all three subcomponents. If the agent is passive, it contains only the monitor.

Summary

```
virtual class ovm_agent extends ovm_component;
  function new (input string name, ovm_component parent);
endclass
```

File

methodology/ovm_agent.svh

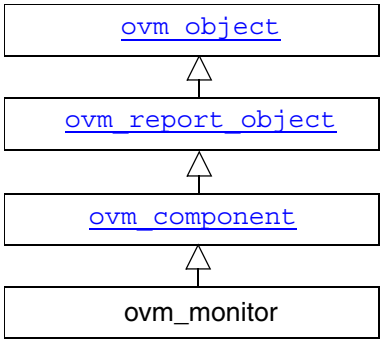
Methods

new

new (input string name, ovm_component parent)

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

ovm_monitor



The ovm_monitor virtual class should be used as the base class for the user-defined monitors. Deriving from ovm_monitor allows you to distinguish monitors from other component types also using its inheritance. Also, monitors will automatically inherit any new monitor-specific features that are added to ovm_monitor.

Summary

```
virtual class ovm_monitor extends ovm_component;
  function new (input string name, ovm_component parent);
endclass
```

File

methodology/ovm_monitor.svh

Virtual

Yes

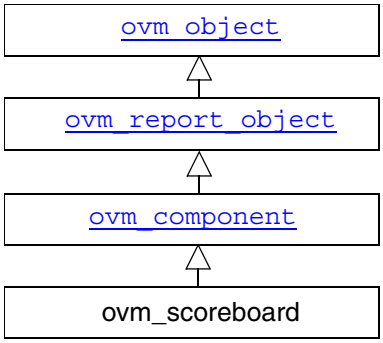
Methods

new

```
function new (input string name, ovm_component parent)
```

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

ovm_scoreboard



The ovm_scoreboard virtual class should be used as the base class for the user-defined scoreboards. Deriving from ovm_scoreboard will allow you to distinguish scoreboards from other component types using its inheritance. Also, scoreboards will automatically inherit any new scoreboard-specific features that are added to ovm_scoreboard.

Summary

```
virtual class ovm_scoreboard extends ovm_component;
  function new (input string name, ovm_component parent);
endclass
```

File

methodology/ovm_scoreboard.svh

Virtual

Yes

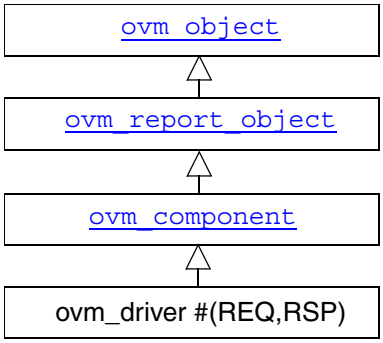
Methods

new

```
function new (input string name, ovm_component parent)
```

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

ovm_driver #(REQ,RSP)



The ovm_push_driver class provides base driver class with port connectors for communicating with a sequencer.

Summary

```
class ovm_driver #(type REQ=ovm_sequence_item, RSP=REQ) extends ovm_component;
  function new (input string name, ovm_component parent);
  ovm_seq_item_pull_port #(REQ, RSP) seq_item_port;
  ovm_analysis_port #(RSP) rsp_port;
endclass
```

File

methodology/ovm_driver.svh

Virtual

No

Members

seq_item_port

```
ovm_seq_item_pull_port #(REQ, RSP) seq_item_port
```

Derived driver classes should use this port to request items from the sequencer, and it may also use it to put responses.

rsp_port

```
ovm_analysis_port #(RSP) rsp_port
```

The `rsp_port` analysis port allows responses to be sent to the sequencer as another way to route them to the originating sequence.

Methods

new

```
new (input string name, ovm_component parent)
```

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

Usage

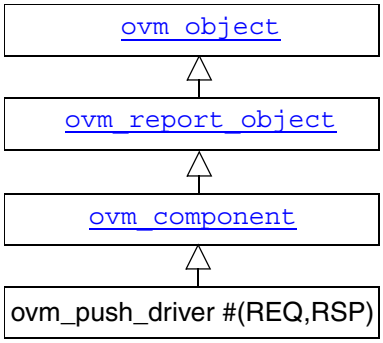
Sequencer to Driver port connections

- A Pull sequencer has two ports that may be connected to the driver sequencer.

```
driver.seq_item_port.connect(sequencer.seq_item_export);
driver.rsp_port.connect(sequencer.rsp_export);
```

The `rsp_port` is only needed if the driver is going to use the `rsp_port` to write responses to the analysis export in the sequencer

ovm_push_driver #(REQ,RSP)



The `ovm_push_driver` class provides base driver class TLM port connectorsfor communicating with a sequencer.

Summary

```
class ovm_push_driver #(type REQ=ovm_sequence_item,RSP=REQ) extends ovm_component;
  function new (string name, ovm_component parent);
  ovm_blocking_put_imp #(REQ, this_type) req_export;
  ovm_analysis_port #(RSP) rsp_port;
endclass
```

File

methodology/ovm_push_driver.svh

Virtual

No

Members

req_export

```
ovm_blocking_put_imp #(REQ, this_type) req_export
```

This export provides the blocking put method, `put`, whose default implementation produces an error. Derived components must override `put` with an appropriate implementation (and not call `super.put`).

rsp_port

```
ovm_analysis_port #(RSP) rsp_port
```

The `rsp_port` analysis port allows responses to be sent to the sequencer as another way to route them to the originating sequence.

Methods

new

```
new (input string name, ovm_component parent)
```

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

Usage

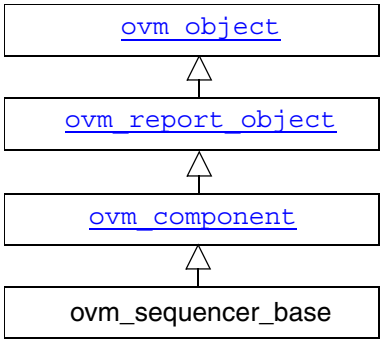
Sequencer to Driver port connections

- A Push sequencer has two ports that may be connected to the driver:

```
sequencer.req_port.connect(driver.req_export);
driver.rsp_port.connect(sequencer.rsp_export);
```

The `rsp_port` is only needed if the driver is going to use the `rsp_port` to write responses to the analysis export in the sequencer

ovm_sequencer_base



The ovm_sequencer_base class provides the methods used to create streams of sequence items and other sequences.

Summary

```
class ovm_sequencer_base extends ovm_component;
    function new (string name, ovm_component parent);
    task                      wait_for_grant (ovm_sequence_base sequence_ptr,
                                               integer item_priority = -1,
                                              bit lock_request = 0);
    task                   set_arbitration (ovm_sequence_base sequence_ptr,
                                              integer transaction_id);
    virtual function void       send_request (ovm_sequence_base sequence_ptr,
                                               ovm_sequence_item t,
                                               bit rerandomize = 0);
    function bit                is_child     (ovm_sequence_base parent,
                                              ovm_sequence_base child);
    function bit                is_blocked   (ovm_sequence_base sequence_ptr);
    function bit                is_locked    (ovm_sequence_base sequence_ptr);
    virtual function bit        is_grabbed   ();
    task                        lock         (ovm_sequence_base sequence_ptr);
    task                        grab         (ovm_sequence_base sequence_ptr);
    function void               unlock       (ovm_sequence_base sequence_ptr);
    function void               ungrab        (ovm_sequence_base sequence_ptr);
    function void               stop_sequences ();
    virtual
     function ovm_sequence_base has_do_available ();
    function bit                has_do_available();
    function void               set_arbitration (SEQ_ARB_TYPE val);
```

```
    virtual function
          integer user_priority_arbitration (integer avail_sequences[$]);

    function void               add_sequence    (string type_name);
    function void               remove_sequence (string type_name);
    static function bit         add_typewide_sequence (string type_name);
    static function bit         remove_typewide_sequence (string type_name);
    function integer            get_seq_kind    (string type_name);
    function ovm_sequence_base  get_sequence    (integer req_kind);
    function integer            num_sequences   ();

    protected string default_sequence;
endclass
```

Members

default_sequence

```
protected string default_sequence
```

This sequencer's default sequence. It may be configured through the `ovm_component`'s `set_config_string` method using the field name, `"default_sequence"`.

Methods

add_sequence

```
function void add_sequence (string type_name)
```

This function allows users to add sequence strings to the sequence library of a given user sequencer instance. This function must be called after the instance(s) of the user sequencer types are created.

remove_sequence

```
function void remove_sequence(string type_name)
```

This function allows users to remove sequence strings to the sequence library of a given user sequencer instance. This function must be called after the instance(s) of the user sequencer types are created.

add_typewide_sequence

```
static function bit add_typewide_sequence(string type_name)
```

This function is provided to the ovm_sequencer class if the `ovm_sequencer_utils macro is used. This allows users to add sequence strings to the sequence library of a given user sequencer type. This static function must be called before the instance(s) of the user sequencer types are created.

remove_typewide_sequence

```
static function bit remove_typewide_sequence(string type_name)
```

This function is provided to the ovm_sequencer class if the `ovm_sequencer_utils macro is used. This allows users to remove sequence strings to the sequence library of a given user sequencer type.

current_grabber

```
virtual function ovm_sequence_base current_grabber()
```

current_grabber returns a reference to the sequence that currently has a lock or grab on the sequence. If multiple hierarchical sequences have a lock, it returns the child that is currently allowed to perform operations on the sequencer.

is_blocked

```
function bit is_blocked (ovm_sequence_base sequence_ptr)
```

Returns 1 if the sequence referred to by *sequence_ptr* is currently locked out of the sequencer. It will return 0 if the sequence is currently allowed to issue operations.

Note that even when a sequence is not blocked, it is possible for another sequence to issue a lock before this sequence is able to issue a request or lock.

is_child

```
function bit is_child (ovm_sequence_base parent, ovm_sequence_base child)
```

Returns 1 if the *child* sequence is a child of the *parent* sequence, 0 otherwise.

is_grabbed

```
virtual function bit is_grabbed()
```

Returns 1 if any sequence currently has a lock or grab on this sequencer, 0 otherwise.

is_locked

```
function bit is_locked (ovm_sequence_base sequence_ptr)
```

Returns 1 if the sequence refered to in the parameter currently has a lock on this sequencer, 0 otherwise.

Note that even if this sequence has a lock, a child sequence may also have a lock, in which case the sequence is still blocked from issueing operations on the sequencer

has_do_available

```
function bit has_do_available ()
```

Determines if a sequence is ready to supply a transaction. A sequence that obtains a transaction in pre-do must determine if the upstream object is ready to provide an item

Returns 1 if a sequence is ready to issue an operation. Returns 0 if no unblocked, relevant sequence is requesting.

lock

```
task lock (ovm_sequence_base sequence_ptr)
```

Requests a lock for the sequence specified by *sequence_ptr*.

A lock request will be arbitrated the same as any other request. A lock is granted after all earlier requests are completed and no other locks or grabs are blocking this sequence.

The lock call will return when the lock has been granted.

grab

```
task grab (ovm_sequence_base sequence_ptr)
```

Requests a lock for the sequence specified by *sequence_ptr*.

A grab request is put in front of the arbitration queue. It will be arbitrated before any other requests. A grab is granted when no other grabs or locks are blocking this sequence.

The grab call will return when the grab has been granted.

ungrab

```
function ungrab (ovm_sequence_base sequence_ptr)
```

Removes any locks and grabs obtained by the specified *sequence_ptr*.

unlock

```
function unlock (ovm_sequence_base sequence_ptr)
```

Removes any locks and grabs obtained by the specified *sequence_ptr*.

get_seq_kind

```
function integer get_seq_kind (string type_name)
```

Returns an integer seq_kind correlating to the sequence of type *type_name* in the sequencer's sequence library.

get_sequence

```
function ovm_sequence_base get_sequence (integer seq_kind)
```

Returns a reference to a sequence specified by the *seq_kind* integer. The *seq_kind* integer may be obtained using the `get_seq_kind()` method.

num_sequences

```
function integer num_sequences()
```

Returns the number of sequences in the sequencer's sequence library.

send_request

```
function send_request (ovm_sequence_base sequence_ptr,
                       ovm_sequence_item request,
                       bit rerandomize = 0)
```

This function may only be called after a `wait_for_grant` call. This call will send the request item to the sequencer, which will forward it to the driver. If the *rerandomize* bit is set, the item will be randomized before being sent to the driver.

set_arbitration

```
function void set_arbitration (SEQ_ARB_TYPE val)
```

Specify the arbitration mode for the sequencer. the arbitration mode must be one of:

`SEQ_ARB_FIFO`
All requests are granted in FIFO order

SEQ_ARB_WEIGHTED
Requests are granted randomly by weight

SEQ_ARB_RANDOM
Requests are granted randomly

SEQ_ARB_STRICT_FIFO
All requests at the highest priority are granted in fifo order

SEQ_ARB_STRICT_RANDOM
All requests at the highest priority are granted in random order

SEQ_ARB_USER
The user function user_priority_arbitration is called. That function will specify the next sequence to grant. The default user function specifies FIFO order

stop_sequences

```
function void stop_sequences()
```

Tells the sequencer to kill all sequences and child sequences currently operating on the sequencer, and remove all requests, locks and responses that are currently queued. This essentially resets the sequencer to an idle state.

user_priority_arbitration

```
virtual function integer user_priority_arbitration(integer avail_sequences[$])
```

If the sequencer arbitration mode is set to SEQ_ARB_USER (via the set_arbitration method), then the sequencer will call this function each time that it needs to arbitrate among sequences.

Derived sequencers may override this method to perform a custom arbitration policy. Such an override must return one of the entries from the *avail_sequences* queue, which are integer indexes into an internal queue, *arb_sequence_q*.

The default implementation behaves like SEQ_ARB_FIFO, which returns the entry at *avail_sequences[0]*.

wait_for_grant

```
task wait_for_grant (ovm_sequence_base sequence_ptr,
                     integer item_priority = -1,
                     bit lock_request = 0)
```

This task issues a request for the specified sequence. If *item_priority* is not specified, then the current sequence priority will be used by the arbiter. If a

lock_request is made, then the sequencer will issue a lock immediately before granting the sequence. (Note that the lock may be granted without the sequence being granted if is_relevant is not asserted).

When this method returns, the sequencer has granted the sequence, and the sequence must call send_request without inserting any simulation delay other than delta cycles. The driver is currently waiting for the next item to be sent via the send_request call.

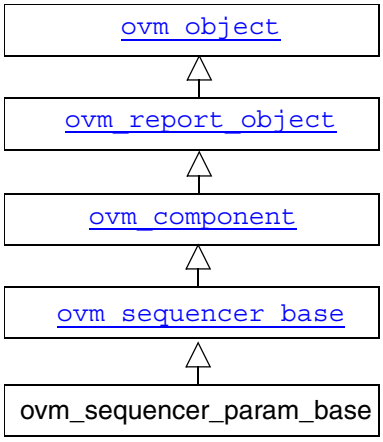
wait_for_item_done

```
task wait_for_item_done (ovm_sequence_base sequence_ptr,
                         integer transaction_id = -1)
```

A sequence may optionally call wait_for_item_done. This task will block until the driver calls item_done() or put() on a transaction issued by the specified sequence. If no transaction_id parameter is specified, then the call will return the next time that the driver calls item_done() or put(). If a specific transaction_id is specified, then the call will only return when the driver indicates that it has completed that specific item.

Note that if a specific transaction_id has been specified, and the driver has already issued an item_done or put for that transaction, then the call will hang waiting for that specific transaction_id.

ovm_sequencer_param_base #(REQ,RSP)



This class provides the base parameterized code used by the ovm_sequencer and ovm_push_sequencer.

Summary

```
class ovm_sequencer_param_base #(type REQ=ovm_sequence_item, RSP=REQ)
                                  extends ovm_sequencer_base;
  function      new (string name, ovm_component parent);

  ovm_analysis_export #(RSP) rsp_export;

  function REQ          get_current_item  ();
  function int          get_num_reqs_sent ();
  function int          get_num_rsps_received();
  function void         set_num_last_reqs (int unsigned max);
  function int unsigned get_num_last_reqs ();
  function REQ          last_req          (int unsigned n = 0);
  function void         set_num_last_rsps (int unsigned max);
  function int unsigned get_num_last_rsps ();
  function RSP          last_rsp          (int unsigned n = 0);
  task                  start_default_sequence();
  virtual task          execute_item      (ovm_sequence_item item);
  virtual function void send_request      (ovm_sequence_base sequence_ptr,
                                           ovm_sequence_item t,
                                           bit rerandomize = 0);
endclass
```

Members

rsp_export

```
ovm_analysis_export #(RSP) rsp_export
```

This is the analysis export used by drivers or monitors to send responses to the sequencer. When a driver wishes to send a response, it may do so through exactly one of three methods:

```
seq_item_port.item_done(response)
seq_item_done.put(response)
rsp_port.write(response)
```

The `rsp_port` in the driver and/or monitor must be connected to the `rsp_export` in this sequencer in order to send responses through the response analysis port.

Methods

new

```
new (input string name, ovm_component parent)
```

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

send_request

```
function void send_request (ovm_sequence_base sequence_ptr,
                            ovm_sequence_item t,
                            bit rerandomize = 0)
```

The `send_request` function may only be called after a `wait_for_grant` call. This call will send the request item, *`t`*, to the sequencer pointed to by *`sequence_ptr`*. The sequencer will forward it to the driver. If *`rerandomize`* is set, the item will be randomized before being sent to the driver.

get_current_item

```
function REQ get_current_item()
```

Returns the request_item currently being executed by the sequencer. If the sequencer is not currently executing an item, this method will return null.

The sequencer is executing an item from the time that `get_next_item` or `peek` is called until the time that `get` or `item_done` is called.

Note that a driver that only calls get() will never show a current item, since the item is completed at the same time as it is requsted.

last_req

```
function REQ last_req (int unsigned n = 0)
```

Returns the last request item by default. If *n* is not 0, then it will get the n'th before last request item. If *n* is greater than the last request buffer size, the function will return null.

set_num_last_reqs

```
function void set_num_last_reqs (int unsigned max)
```

Sets the size of the last_requests buffer. Note that the maximum buffer size is 1024. If *max* is greater than 1024, a warning is issued, and the buffer is set to 1024. The default value is 1.

get_num_last_reqs

```
function int unsigned get_num_last_reqs()
```

Returns the size of the last requests buffer, as set by `set_num_last_reqs`.

get_num_reqs_sent

```
function int get_num_reqs_sent()
```

Returns the number of requests that have been sent by this sequencer.

last_rsp

```
function RSP last_rsp (int unsigned n = 0)
```

Returns the last response item by default. If *n* is not 0, then it will get the *n*'th before last response item. If *n* is greater than the last response buffer size, the function will return null.

set_num_last_rsps

```
function void set_num_last_rsps (int unsigned max)
```

Sets the size of the last_responses buffer. The maximum buffer size is 1024. If *max* is greater than 1024, a warning is issued, and the buffer is set to 1024. The default value is 1.

get_num_last_rsps

```
function int unsigned get_num_last_rsps()
```

Returns the max size of the last responses buffer, as set by `set_num_last_rsps`.

get_num_rsps_received

```
function int get_num_rsps_received()
```

Returns the number of responses received thus far by this sequencer.

execute_item

```
task execute_item (ovm_sequence_item item)
```

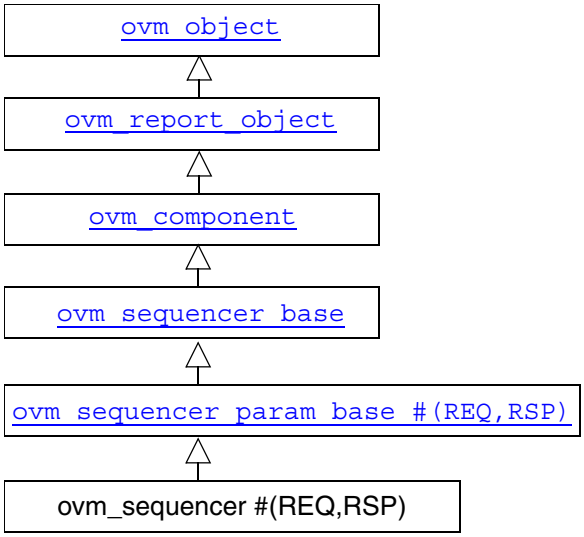
This task allows the user to supply an item or sequence to the sequencer and have it be executed procedurally. The parent sequence for the item or sequence is a temporary sequence that is automatically created. There is no capability to retrieve responses. The sequencer will drop responses to items done using this interface.

start_default_sequence

```
task start_default_sequence()
```

Called when the `run` phase begins, this method starts the default sequence, as specified by the `default_sequence` member variable.

ovm_sequencer #(REQ,RSP)



Summary

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

    function new (string name, ovm_component parent);

    ovm_seq_item_pull_imp #(REQ, RSP,
                            ovm_sequencer #(REQ,RSP)) seq_item_export;

    int unsigned pound_zero_count;

    virtual function void send_request(ovm_sequence_base sequence_ptr,
                                       ovm_sequence_item t,
                                       bit rerandomize = 0);
endclass
```

Members

seq_item_export

```
    ovm_seq_item_pull_imp #(REQ, RSP, ovm_sequencer #(REQ,RSP)) seq_item_export
```

This export provides access to this sequencer's implementation of the sequencer interface, `sqr_if_base`, which defines the following methods:

```
virtual task          get_next_item      (output REQ request);
virtual task          try_next_item      (output REQ request);
virtual function void item_done          (input RSP response=null);
virtual task          wait_for_sequences ();
virtual function bit  has_do_available   ();
virtual task          get                (output REQ request);
virtual task          peek               (output REQ request);
virtual task          put                (input RSP response);
```

See sqr_if_base #(REQ,RSP) on page 174 for information about this interface.

pound_zero_count

```
int unsigned pound_zero_count;
```

Methods

new

```
function new (string name, ovm_component parent);
```

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

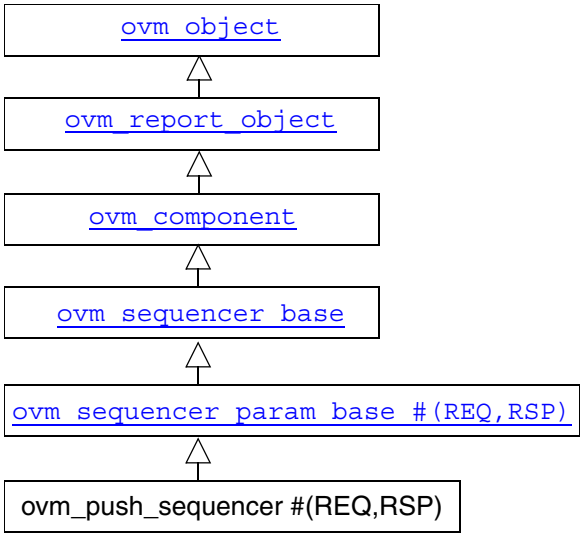
send_request

```
virtual function send_request (ovm_sequence_base sequence_ptr,
                               ovm_sequence_item request,
                               bit rerandomize = 0);
```

Sends the *`request`* item to the sequencer pointed to by *`sequencer_ptr`*. This sequencer will then forward it to the driver. If *`rerandomize`* is set, the item will be randomized before being sent to the driver.

The `send_request` function may only be called after a `wait_for_grant` call (from `ovm_sequencer_base`).

ovm_push_sequencer #(REQ,RSP)



Summary

```
class ovm_push_sequencer #(type REQ=ovm_sequence_item, RSP=REQ
                                 extends ovm_sequencer_param_base #(REQ, RSP);
  function new (string name, ovm_component parent);
  ovm_blocking_put_port #(REQ) req_port;
  virtual task run();
endclass
```

Members

req_port

```
ovm_blocking_put_port #(REQ) req_port
```

The push sequencer requires access to a blocking put interface. Continual sequence items, based on the list of available sequences loaded into this sequencer, are sent out this port.

Methods

new

```
function new (string name, ovm_component parent);
```

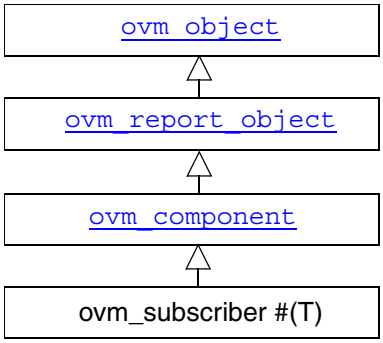
Creates and initializes an instance of this class using the normal constructor arguments for `ovm_component`: *`name`* is the name of the instance, and *`parent`* is the handle to the hierarchical parent, if any.

run

```
virtual task run();
```

The push sequencer continuously selects from its list of available sequences and sends the next item from the selected sequence out its `req_port` using `req_port.put(item)`. Typically, the `req_port` would be connected to the `req_export` on an instance of an `ovm_push_driver`, which would be responsible for executing the item.

ovm_subscriber #(T)



A subclass of `ovm_subscriber` can be used to connect to an `ovm_analysis_port` that writes transactions of type `T`.

`ovm_subscriber` has a single pure virtual method, `write()`, which is made available to the outside by way of an `analysis_export`. This is particularly useful when writing a coverage object that needs to be attached to a monitor.

Summary

```
virtual class ovm_subscriber #(type T=int) extends ovm_component;
  function new (string name, ovm_component parent);
  ovm_analysis_imp #(T, ovm_subscriber #(T)) analysis_export;
  pure virtual function void write (T t);
endclass
```

File

utils/ovm_subscriber.svh

Parameters

`type T = int`

Specifies the type of transaction to be received.

Members

analysis_export

```
ovm_analysis_imp #(T, ovm_subscriber #(T)) analysis_export
```

This export provides access to the `write` method.

Methods

new

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

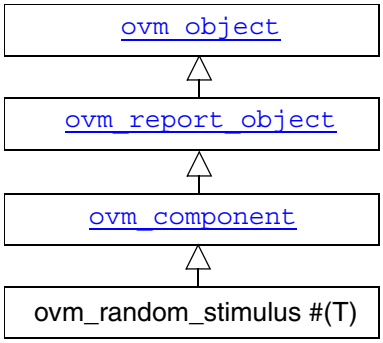
Creates and initializes an instance of this class using the normal constructor arguments for `ovm_component`: *`name`* is the name of the instance, and *`parent`* is the handle to the hierarchical parent, if any.

write

```
pure virtual function void write (T t)
```

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

ovm_random_stimulus #(T)



This is a general purpose uni-directional random stimulus generator. It is a very useful component in its own right, but can also be used as a template to define other stimulus generators, or it can be extended to add additional stimulus generation methods to simplify test writing.

The `ovm_random_stimulus` class generates streams of `trans_type` transactions. These streams may be generated by the `randomize()` method of `trans_type`, or the `randomize()` method of one of its subclasses, depending on the type of the argument passed into the `generate_stimulus()` method. The stream may go indefinitely, until terminated by a call to `stop_stimulus_generation()`, or you may specify the maximum number of transactions to be generated.

By using inheritance, we can add directed initialization or tidy up sequences to the random stimulus generation.

Summary

```
class ovm_random_stimulus #(type T=ovm_transaction) extends ovm_component;
    function new(string name, ovm_component parent);
    ovm_blocking_put_port #(T) blocking_put_port;
    virtual task generate_stimulus(trans_type t=null, int max_count=0);
    virtual function void stop_stimulus_generation();
endclass
```

File

base/ovm_random_stimulus.svh

Parameters

`type trans_type=ovm_transaction`

Specifies the type of transaction to be generated.

Members

blocking_put_port

`ovm_blocking_put_port #(type T=int)` **`blocking_put_port`**

The port through which transactions come out of the stimulus generator.

Methods

new

`function` **`new`** `(string name, ovm_component parent)`

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

The constructor displays the string obtained from `get_randstate()` during construction. The `set_randstate()` can then be used to regenerate precisely the same sequence of transactions for debugging purposes.

generate_stimulus

`virtual task` **`generate_stimulus`** `(trans_type t=null, int max_count=0)`

The main user-visible method. If `t` is not specified, then it will generate random transactions of type `trans_type`. If `t` is specified, then it will use the `randomize()` method in `t` to generate transactions—so `t` must be a subclass of `trans_type`. The `max_count` is the maximum number of transactions to be generated. A value of zero indicates no maximum—in this case, `generate_stimulus()` will go on indefinitely unless stopped by some other process. The transactions are cloned before they are sent out over the `blocking_put_port`.

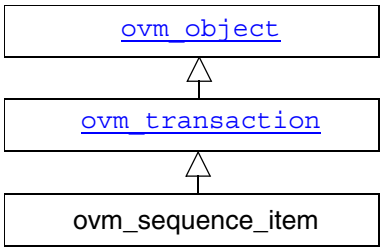
stop_stimulus_generation

`virtual function void` **`stop_stimulus_generation`** `()`

Stops the generation of stimulus.

Sequences

ovm_sequence_item



The `ovm_sequence_item` class is the base class for user-defined sequence items and also the base class for the `ovm_sequence`.

The `ovm_sequence_item` class provides the basic functionality for objects, both sequence items and sequences, to operate in the sequence mechanism.

Summary

```
class ovm_sequence_item extends ovm_transaction;
    function new (string name = "ovm_sequence_item",
                  ovm_sequencer_base sequencer = null,
                  ovm_sequence_base parent_sequence = null);
    function void        set_sequence_id       (integer id);
    function integer     get_sequence_id       ();
    function void        set_use_sequence_info (bit value);
    function bit         get_use_sequence_info ();
    function void        set_id_info           (ovm_sequence_item item);
    function void        set_sequencer         (ovm_sequencer_base sequencer);
    function ovm_sequencer_base get_sequencer   ();
    function void        set_parent_sequence    (ovm_sequence_base parent);
    function ovm_sequence_base get_parent_sequence();
    function void        set_depth              (integer value);
    function integer     get_depth              ();
    virtual function bit is_item                ();
    function string      get_root_sequence_name ();
    function ovm_sequence_base get_root_sequence();
    function string      get_sequence_path      ();
endclass
```

File

sequences/ovm_sequence_item.svh

Virtual

No

Methods

new

```
function new (string name = "ovm_sequence_item",
              ovm_sequencer_base sequencer = null,
              ovm_sequence_base  parent_sequence = null)
```

The constructor method for `ovm_sequence_item`. The *`sequencer`* and *`parent_sequence`* may be specified in the constructor, or directly using `ovm_sequence_item` methods.

set_sequence_id

get_sequence_id

```
function void    set_sequence_id (integer value)
function integer get_sequence_id ()
```

These methods allow access to the `sequence_item` sequence and transaction IDs. `get_transaction_id` and `set_transaction_id` are methods on the `ovm_transaction` base_class. These IDs are used to identify sequences to the sequencer, to route responses back to the sequence that issued a request, and to uniquely identify transactions.

The `sequence_id` is assigned automatically by a sequencer when a sequence initiates communication through any sequencer calls (i.e. `ovm_do_xxx, `wait_for_grant`). The `sequence_id` will remain unique for this sequence until it ends or it is killed. Should a sequence start again after it has ended, it will be given a new unique `sequence_id`.

The `transaction_id` is assigned automatically by the sequence each time a transaction is sent to the sequencer with the `transaction_id` in its default (-1) value. If the user sets the `transaction_id` to any non-default value, that value will be maintained.

Responses are routed back to this sequences based on sequence_id. The sequence may use the transaction_id to correlate responses with their requests.

set_use_sequence_info

get_use_sequence_info

```
function void set_use_sequence_info (bit value)
function bit  get_use_sequence_info ()
```

These methods are used to set and get the status of the use_sequence_info bit. Use_sequence_info controls whether the sequence information (*sequencer*, *parent_sequence*, *sequence_id*, etc.) is printed, copied, or recorded. When use_sequence_info is the default value of 0, then the sequence information is not used. When use_sequence_info is set to 1, the sequence information will be used in printing and copying.

set_id_info

```
function void set_id_info (ovm_sequence_item item);
```

Copies the sequence_id and transaction_id from the referenced item into the calling item. This routine should always be used by drivers to initialize responses for future compatibility.

set_sequencer

get_sequencer

```
function void set_sequencer (ovm_sequencer_base sequencer)
function ovm_sequencer_base get_sequencer()
```

These routines set and get the reference to the *sequencer* to which this sequence_item communicates.

set_parent_sequence

```
function void set_parent_sequence (ovm_sequence_base parent)
```

Sets the *parent* sequence of this sequence_item. This is used to identify the source sequence of a sequence_item.

get_parent_sequence

```
function ovm_sequence_base get_parent_sequence()
```

Returns a reference to the parent sequence of any sequence on which this method was called. If this is a parent sequence, the method will return null.

get_depth

```
function integer get_depth()
```

Returns the depth of a sequence from it's parent. A parent sequence will have a depth of 1, it's child will have a depth of 2, and it's grandchild will have a depth of 3.

set_depth

```
function void set_depth (integer value)
```

The depth of any sequence is calculated automatically. However, the user may use `set_depth` to specify the depth of a particular sequence. This method will override the automatically calculated depth, even if it is incorrect.

is_item

```
virtual function is_item()
```

This function may be called on any `sequence_item` or sequence. It will return 1 for items and 0 for sequences.

get_root_sequence_name

```
function string get_root_sequence_name()
```

Provides the string name of the root sequence (the top-most parent sequence).

get_root_sequence

```
function ovm_sequence_base get_root_sequence()
```

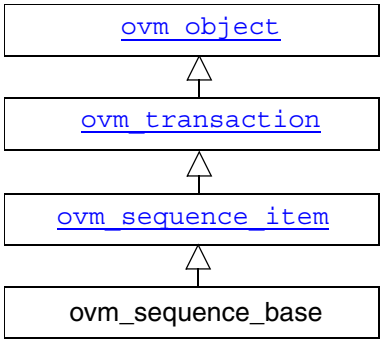
Provides a reference to the root sequence (the top-most parent sequence).

get_sequence_path

```
function string get_sequence_path()
```

Provides a string of names of each sequence in the full hierarchical path. A "." is used as the separator between each sequence.

ovm_sequence_base



The ovm_sequence_base class provides the interfaces necessary in order to create streams of sequence items and/or other sequences.

Summary

```
virtual class ovm_sequence_base extends ovm_sequence_item;

    function new (string name = "ovm_sequence",
                  ovm_sequencer_base sequencer_ptr = null,
                  ovm_sequence_base parent_seq = null);

    virtual task         start     (ovm_sequencer_base sequencer,
                                    ovm_sequence_base parent_sequence = null,
                                    integer this_priority = 100,
                                     bit call_pre_post = 1);
    task                  do_sequence_kind (integer unsigned req_kind);

    virtual task          pre_body ();
    virtual task          body     ();
    virtual task          post_body();
    virtual task          pre_do   (bit is_item);
    virtual function void mid_do   (ovm_sequence_item this_item);
    virtual function void post_do  (ovm_sequence_item this_item);

    virtual function bit  is_item           ();
    function integer      num_sequences     ();
    function integer      get_seq_kind      (string type_name);
    function ovm_sequence_base get_sequence (integer unsigned req_kind);
```

```
    function void         set_priority      (integer value);
    function integer      get_priority      ();

    virtual task          wait_for_relevant ();
    virtual function bit  is_relevant       ();
    function bit          is_blocked        ();
    task                  lock              (ovm_sequencer_base sequencer=null);
    task                  grab              (ovm_sequencer_base sequencer=null);
    function void         unlock            (ovm_sequencer_base sequencer=null);
    function void          ungrab             (ovm_sequencer_base sequencer=null);

    virtual task          wait_for_grant     (integer item_priority = -1,
                                              bit lock_request = 0);
    virtual function void send_request       (ovm_sequence_item request,
                                              bit rerandomize = 0);
    virtual task          wait_for_item_done (integer transaction_id = -1);

    virtual function void set_sequencer      (ovm_sequencer_base sequencer);
    virtual function ovm_sequencer_base get_sequencer();

    function void         use_response_handler(bit enable);
    function bit          get_use_response_handler();
    virtual function void response_handler(ovm_sequence_item response);

    function void kill();
    function ovm_sequence_state_enum get_sequence_state();
    task wait_for_sequence_state(ovm_sequence_state_enum state);
endclass
```

File

sequences/ovm_sequence_base.svh

Virtual

Yes

Members

seq_kind

```
integer unsigned seq_kind
```

Used as an identifier in constraints for a specific sequence type.

Methods

new

```
function new (string name = "ovm_sequence",
              ovm_sequencer_base sequencer_ptr = null,
              ovm_sequence_base parent_seq = null)
```

The constructor for the sequence.

Although generally set in the `start` method, *`sequencer_ptr`*, if set, specifies the default sequencer at initialization time.

Although generally set in the `start` method, *`parent_seq`*, if set, specifies this sequence's parent sequence at initialization.

start

```
virtual task start (ovm_sequencer_base sequencer,
                    ovm_sequence_base parent_sequence = null,
                    integer this_priority = 100,
                    bit call_pre_post = 1)
```

The `start` task is called to begin execution of a sequence.

If *`parent_sequence`* is null, then the sequence is a parent, otherwise it is a child of the specified parent.

By default, the priority of a sequence is 100. A different priority may be specified by *`this_priority`*. Higher numbers indicate higher priority.

If *`call_pre_post`* is set to 1, then the `pre_body` and `post_body` tasks will be called before and after the sequence `body` is called.

get_sequence_state

```
function sequence_state_enum get_sequence_state()
```

Returns the sequence state as an enumerated value.

wait_for_sequence_state

```
task wait_for_sequence_state(ovm_sequence_state_enum state)
```

This method will wait until the sequence reaches the state specified by the *state* argument.

kill

```
function kill()
```

This function will kill the sequence, and cause all current locks and requests in the sequence's default sequencer to be removed.

Note: If a sequence has issued locks, grabs, or requests on sequences other than the default sequence, then care must be taken to unregister the sequence with the other sequencer using the sequencer `unregister_sequence()` method.

pre_body

```
virtual task pre_body()
```

This task is a user-definable callback task that is called before the execution of the body, unless the sequence is started with `call_pre_post` = 0.. This method should not be called by the user.

post_body

```
virtual task post_body()
```

This task is a user-definable callback task that is called after the execution of the body, unless the sequence is started with `call_pre_post` = 0. This method should not be called by the user.

pre_do

```
virtual task pre_do (bit is_item)
```

This task is a user-definable callback task that is called after the sequence has issued a `wait_for_grant()` call and after the sequencer has selected this sequence, and before the item is randomized. This method should not be called by the user.

Although `pre_do` is a task, consuming simulation cycles may result in unexpected behavior on the driver.

body

```
virtual task body()
```

This is the user-defined task where the main sequence code resides. This method should not be called directly by the user.

mid_do

```
virtual function mid_do (ovm_sequence_item this_item)
```

This function is a user-definable callback function that is called after the sequence item has been randomized, and just before the item is sent to the driver. This mehod should not be called by the user.

post_do

```
virtual function void post_do (ovm_sequence_item this_item)
```

This function is a user-definable callback function that is called after the driver has indicated that it has completed the item, using either this `item_done` or put methods. This method should not be called by the user.

is_item

```
virtual function is_item()
```

This function may be called on any `sequence_item` or sequence object. It will return 1 on items and 0 on sequences.

num_sequences

```
function integer num_sequences()
```

This function returns the number of sequences in the sequencer's sequence library.

get_seq_kind

```
function integer get_seq_kind (string type_name);
```

This function returns an integer representing the sequence kind that has been registerd with the sequencer. The `seq_kind` integer may be used with the `get_sequence` or `do_sequence_kind` methods.

get_sequence

```
function ovm_sequence_base get_sequence (integer req_kind)
```

This function returns a reference to a sequence specified by *req_kind*, which can be obtained using the get_seq_kind method.

do_sequence_kind

```
task do_sequence_kind (integer req_kind)
```

This task will start a sequence of kind specified by *req_kind*, which can be obtained using the get_seq_kind method.

set_priority

```
function set_priority (integer value)
```

The priority of a sequence may be changed at any point in time. When the priority of a sequence is changed, the new priority will be used by the sequencer the next time that it arbitrates between sequences.

The default priority value for a sequence is 100. Higher values result in higher priorities.

get_priority

```
function integer get_priority()
```

This function returns the current priority of a the sequence.

wait_for_relevant

```
virtual task wait_for_relevant()
```

This method is called by the sequencer when all available sequences are not relevant. When wait_for_relevant returns the sequencer attempt to re-arbitrate.

Returning from this call does not guarantee a sequence is relevant, although that would be the ideal. The method provide some delay to prevent an infinite loop.

If a sequence defines is_relevant so that it is not always relevant (by default, a sequence is always relevant), then the sequence must also supply a wait_for_relevant method.

is_relevant

```
function function bit is_relevant()
```

The default is_relevant implementation returns 1, indicating that the sequence is always relevant.

Users may choose to override with their own virtual function to indicate to the sequencer that the sequence is not currently relevant after a request has been made.

When the sequencer arbitrates, it will call `is_relevant` on each requesting, unblocked sequence to see if it is relevant. If a 0 is returned, then the sequence will not be chosen.

If all requesting sequences are not relevant, then the sequencer will call `wait_for_relevant` on all sequences and re-arbitrate upon its return.

Any sequence that implements `is_relevant` must also implement `wait_for_relevant` so that the sequencer has a way to wait for a sequence to become relevant.

is_blocked

```
function bit is_blocked()
```

Returns a bit indicating whether this sequence is currently prevented from running due to another lock or grab. A 1 is returned if the sequence is currently blocked. A 0 is returned if no lock or grab prevents this sequence from executing. Note that even if a sequence is not blocked, it is possible for another sequence to issue a lock or grab before this sequence can issue a request.

lock

```
task lock (ovm_sequencer_base sequencer = null)
```

Requests a lock on the specified *`sequencer`*. If *`sequencer`* is null, the lock will be requested on the current default sequencer.

A lock request will be arbitrated the same as any other request. A lock is granted after all earlier requests are completed and no other locks or grabs are blocking this sequence.

The lock call will return when the lock has been granted.

grab

```
task grab (ovm_sequencer_base sequencer = null)
```

Requests a lock on the specified *`sequencer`*. If no parameter is supplied, the lock will be requested on the current default sequencer.

A grab equest is put in front of the arbitration queue. It will be arbitrated before any other requests. A grab is granted when no other grabs or locks are blocking this sequence.

The grab call will return when the grab has been granted.

unlock

```
function unlock (ovm_sequencer_base sequencer = null)
```

Removes any locks or grabs obtained by this sequence on the specified *sequencer*. If *sequencer* is null, then the unlock will be done on the current default sequencer.

ungrab

```
function ungrab (ovm_sequencer_base sequencer = null)
```

Removes any locks or grabs obtained by this sequence on the specified *sequencer*. If *sequencer* is null, then the unlock will be done on the current default sequencer.

wait_for_grant

```
task wait_for_grant (integer item_priority = -1, bit lock_request = 0)
```

This task issues a request to the current sequencer. If item_priority is not specified, then the current sequence priority will be used by the arbiter. If a lock_request is made, then the sequencer will issue a lock immediately before granting the sequence. (Note that the lock may be granted without the sequence being granted if is_relevant is not asserted).

When this method returns, the sequencer has granted the sequence, and the sequence must call send_request without inserting any simulation delay other than delta cycles. The driver is currently waiting for the next item to be sent via the send_request call.

wait_for_item_done

```
task wait_for_item_done (integer transaction_id = -1)
```

A sequence may optionally call wait_for_item_done. This task will block until the driver calls item_done or put. If no *transaction_id* parameter is specified, then the call will return the next time that the driver calls item_done or put. If a specific transaction_id is specified, then the call will return when the driver indicates completion of that specific item.

Note that if a specific transaction_id has been specified, and the driver has already issued an item_done or put for that transaction, then the call will hang, having missed the earlier notification.

send_request

```
function send_request (ovm_sequence_item request, bit rerandomize = 0)
```

The send_request function may only be called after a wait_for_grant call. This call will send the request item to the sequencer, which will forward it to the driver. If the rerandomize bit is set, the item will be randomized before being sent to the driver.

set_sequencer

```
function set_sequencer (ovm_sequencer_base sequencer)
```

Sets the default sequencer for the sequence to *sequencer*. It will take effect immediately, so it should not be called while the sequence is actively communicating with the sequencer.

get_sequencer

```
function ovm_sequencer_base get_sequencer()
```

Returns a reference to the current default sequencer of the sequence.

use_response_handler

```
function void use_response_handler (bit enable)
```

When called with *enable* set to 1, responses will be sent to the response handler. Otherwise, responses must be retrieved using get_response.

By default, responses from the driver are retrieved in the sequence by calling get_response.

An alternative method is for the sequencer to call the response_handler function with each response.

get_use_response_handler

```
function bit get_use_response_handler()
```

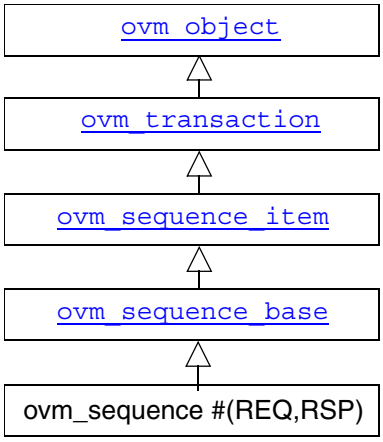
Returns the state of the use_response_handler bit.

response_handler

```
virtual function void response_handler (ovm_sequence_item response)
```

When the use_reponse_handler bit is set to 1, this virtual task is called by the sequencer for each *response* that arrives for this sequence.

ovm_sequence #(REQ,RSP)



The ovm_sequence class provides the interfaces necessary in order to create streams of sequence items and/or other sequences.

Summary

```
virtual class ovm_sequence #(type REQ = ovm_sequence_item,
                             type RSP = REQ) extends ovm_sequence_base;
  function         new              (string name = "ovm_sequence",
                                     ovm_sequencer_base sequencer_ptr = null,
                                     ovm_sequence_base parent_seq = null);
  function void    send_request     (ovm_sequence_item request
                                     bit rerandomize = 0);
  task             get_response     (output RSP response,
                                     input integer transaction_id = -1);
  function REQ     get_current_item();
  virtual function void set_sequencer (ovm_sequencer_base sequencer);
  function void    set_response_queue_error_report_disabled(bit value);
  function bit     get_response_queue_error_report_disabled();
  function void    set_response_queue_depth(integer value);
  function integer get_response_queue_depth();
endclass
```

File

sequences/ovm_sequence.svh

Virtual

Yes

Members

req

```
REQ req
```

rsp

```
RSP rsp
```

Methods

new

```
function new (string name = "ovm_sequence",
              ovm_sequencer_base sequencer_ptr = null,
              ovm_sequence_base parent_seq = null)
```

The constructor for the sequence.

Although generally set in the `start` method, *`sequencer_ptr`*, if set, specifies the default sequencer at initialization time.

Although generally set in the `start` method, *`parent_seq`*, if set, specifies this sequence's parent sequence at initialization.

send_request

```
function send_request (ovm_sequence_item request, bit rerandomize = 0)
```

The `send_request` function may only be called after a `wait_for_grant` call. This call will send the *`request`* item to the sequencer, which will forward it to the driver. If the *`rerandomize`* bit is set, the item will be randomized before being sent to the driver.

get_response

```
task get_response (output RSP response, input integer transaction_id = -1)
```

By default, sequences must retrieve responses by calling `get_response`. If no *`transaction_id`* is specified, this task will return the next response sent to this

sequence. If no response is available in the response queue, the method will block until a response is recieved.

If a *transaction_id* parameter is specified, the task will block until a response with that *transaction_id* is received in the response queue.

The default size of the response queue is 8. The get_response method must be called soon enough to avoid an overflow of the response queue to prevent responses from being dropped.

If a response is dropped in the response queue, an error will be reported unless the error reporting is disabled via set_response_queue_error_report_disabled.

set_sequencer

```
function set_sequencer (ovm_sequencer_base sequencer)
```

Sets the default sequencer for the sequence to *sequencer*. It will take effect immediately, so it should not be called while the sequence is actively communicating with the sequencer.

set_response_queue_error_report_disabled

```
function void set_response_queue_error_report_disabled (bit value)
```

By default, if the response_queue overflows, an error is reported. The response_queue will overflow if more responses are sent to this sequence from the driver than get_response calls are made. Setting *value* to 0 disables these errors, while setting it to 1 enables them.

get_response_queue_error_report_disabled

```
function bit get_response_queue_error_report_disabled()
```

When this bit is 0 (default value), error reports are generated when the response queue overflows. When this bit is 1, no such error reports are generated.

set_response_queue_depth

get_response_queue_depth

```
function void    set_response_queue_depth (integer value)
function integer get_response_queue_depth ()
```

The default maximum depth of the response queue is 8. These method is used to examine or change the maximum depth of the response queue.

Setting the response_queue_depth to -1 indicates an arbitrarily deep response queue. No checking is done.

get_current_item

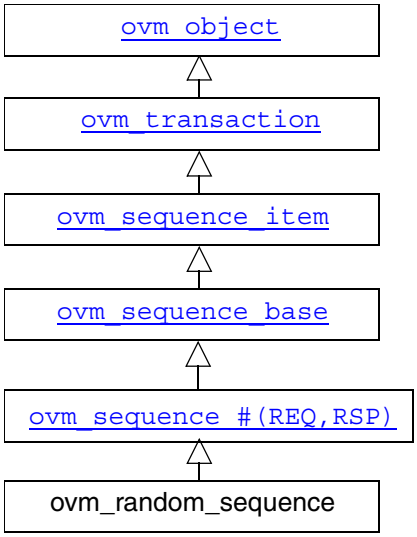
```
function REQ get_current_item()
```

Returns the request item currently being executed by the sequencer. If the sequencer is not currently executing an item, this method will return null.

The sequencer is executing an item from the time that `get_next_item` or `peek` is called until the time that `get` or `item_done` is called.

Note that a driver that only calls `get` will never show a current item, since the item is completed at the same time as it is requested.

ovm_random_sequence



The `ovm_random_sequence` class is a built-in sequence that is preloaded into every sequencer's and virtual sequencer's sequence library.

This sequence is registered in the sequence library as `ovm_random_sequence`. This sequence randomly selects and executes a sequence from the sequencer's sequence library, excluding `ovm_random_sequence` itself, and `ovm_exhaustive_sequence`.

The number of selections and executions is determined by the count property of the sequencer (or virtual sequencer) on which `ovm_random_sequence` is operating. See ovm_sequencer_base on page 204 for more information.

Summary

```
class ovm_random_sequence extends ovm_sequence #(ovm_sequence_item);
    function new (input string name="ovm_random_sequence",
                  ovm_sequencer_base sequencer = null,
                  ovm_sequence parent_seq = null);
endclass
```

File

sequences/ovm_sequence_builtin.svh

Virtual

No

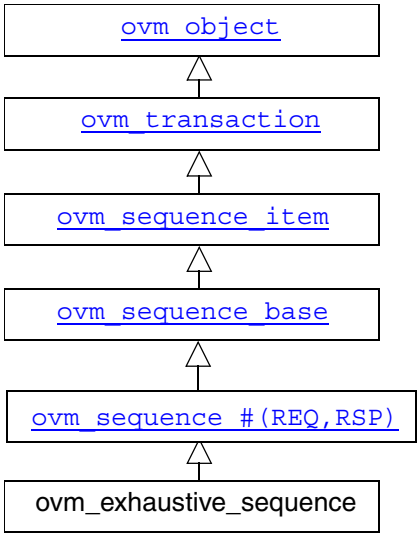
Members

None

Methods

None

ovm_exhaustive_sequence



The `ovm_exhaustive_sequence` class is a built-in sequence that is preloaded into every sequencer's and virtual sequencer's sequence library.

This sequence is registered in the sequence library as `ovm_exhaustive_sequence`. This sequence randomly selects and executes each sequence from the sequencer's sequence library once, excluding `ovm_exhaustive_sequence` itself, and `ovm_random_sequence`.

Summary

```
class ovm_exhaustive_sequence extends ovm_sequence #(ovm_sequence_item);
    function new (input string name="ovm_exhaustive_sequence",
                  ovm_sequencer_base sequencer = null,
                  ovm_sequence parent_seq = null);
endclass
```

File

sequences/ovm_sequence_builtin.svh

Virtual

No

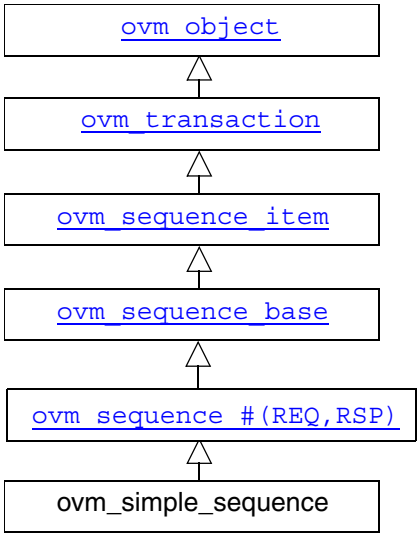
Members

None

Methods

None

ovm_simple_sequence



The `ovm_simple_sequence` class is a built-in sequence that is preloaded into every sequencer's (but not virtual sequencer's) sequence library.

This sequence is registered in the sequence library as `ovm_simple_sequence`. This sequence simply executes a single sequence item.

The item parameterization of the sequencer that the `ovm_simple_sequence` is executed on defines the actual type of the item executed. See ovm_sequencer #(REQ,RSP) on page 215 for more information.

Summary

```
class ovm_simple_sequence extends ovm_sequence #(ovm_sequence_item);
    function new (input string name="ovm_simple_sequence",
                  ovm_sequencer_base sequencer = null,
                  ovm_sequence parent_seq = null);
endclass
```

File

sequences/ovm_sequence_builtin.svh

Virtual

No

Members

None

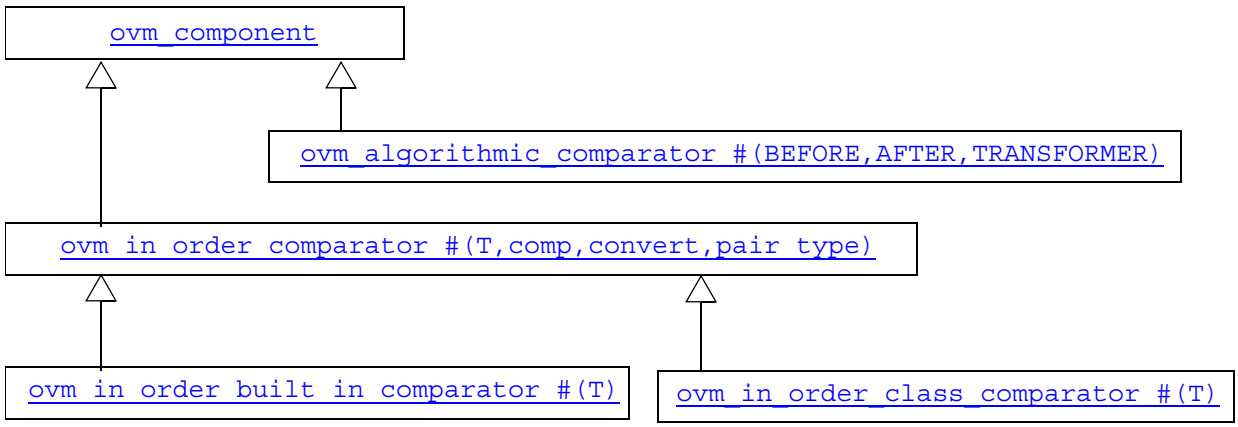
Methods

None

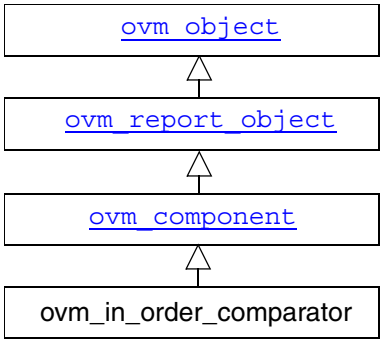
Comparators

A common function of testbenches is to compare streams of transactions for equivalence. For example, a testbench may compare a stream of transactions from a DUT with expected results. The OVM library provides a base class called `ovm_in_order_comparator` and two derived classes, which are `ovm_in_order_built_in_comparator` for comparing streams of built-in types and `ovm_in_order_class_comparator` for comparing streams of class objects. The `ovm_algorithmic_comparator` also compares two streams of transactions; however, the transaction streams might be of different type objects. This device will use a user-written transformation function to convert one type to another before performing a comparison.

Figure 1-6 UML Diagram for OVM Comparator Classes



ovm_in_order_comparator #(T,comp,convert,pair_type)



Compares two streams of transactions. These transactions may either be classes or built-in types. To be successfully compared, the two streams of data must be in the same order. Apart from that, there are no assumptions made about the relative timing of the two streams of data.

Summary

```
class ovm_in_order_comparator #(type T=int,
                                comp=ovm_built_in_comp #(T),
                                convert=ovm_built_in_converter #(T),
                               pair_type=ovm_built_in_pair #(T)) extends ovm_component;
  function new (string name, ovm_component parent);
  ovm_analysis_export #(T)       before_export;
  ovm_analysis_export #(T)       after_export;
  ovm_analysis_port #(pair_type) pair_ap;
  function void flush ();
  task run ();
endclass
```

File

methodology/ovm_in_order_comparator.svh

Parameters

`type T = int`

Specifies the type of transactions to be compared.

`type comp = ovm_built_in_comp #(T)`

The type of the comparator to be used to compare the two transaction streams.

```
type convert = ovm_built_in_converter #(T)
```

A policy class to allow `convert2string()` to be called on the transactions being compared. If *`T`* is an extension of `ovm_transaction`, then it uses *`T::convert2string()`*. If *`T`* is a built-in type, then the policy provides a `convert2string()` method for the comparator to call.

```
type pair_type = ovm_built_in_pair #(T)
```

A policy class to allow pairs of transactions to be handled as a single `ovm_transaction` type.

Members

before_export

```
ovm_analysis_export #(T) before_export
```

The export to which one stream of data is written.

after_export

```
ovm_analysis_export #(T) after_export
```

The export to which the other stream of data is written.

pair_ap

```
ovm_analysis_port #(pair_type) pair_ap
```

The comparator sends out pairs of transactions across this analysis port. Both matched and unmatched pairs are published.

Methods

new

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

The normal `ovm_component` constructor.

flush

```
function void flush()
```

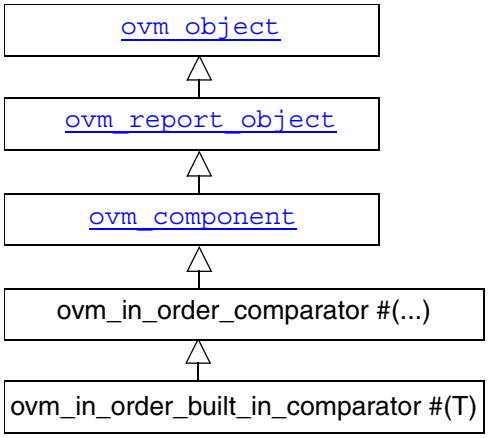
This method sets `m_matches` and `m_mismatches` back to zero. The `tlm_fifo::flush` takes care of flushing the FIFOs.

run

```
task run()
```

Takes pairs of `before` and `after` transactions and compares them. Status information is updated according to the results of the comparison and pairs are published using the analysis port.

ovm_in_order_built_in_comparator #(T)



A subclass of `ovm_in_order_comparator` that is used to compare two streams of built-in types.

Summary

```
class ovm_in_order_built_in_comparator #(type T=int) extends
                                                    ovm_in_order_comparator #(T);
  function new (string name,ovm_component parent);
endclass
```

File

methodology/ovm_in_order_comparator.svh

Parameters

`type T = int`

Specifies the type of transactions to be compared.

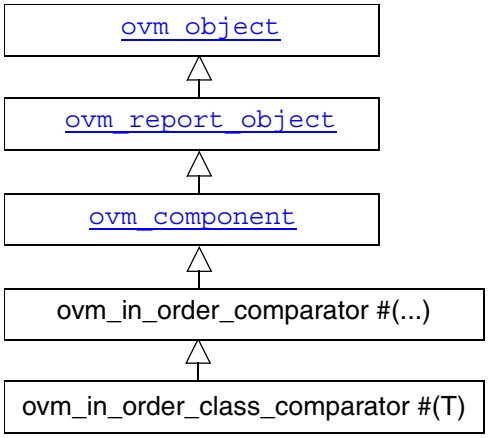
Methods

new

`function` **`new`**`(string name,ovm_component parent)`

This is the normal `ovm_component` constructor.

ovm_in_order_class_comparator #(T)



A subclass of `ovm_in_order_comparator` that is used to compare two streams of built-in types.

Summary

```
class ovm_in_order_built_in_comparator #(type T=int) extends
                                                    ovm_in_order_comparator #(T);
  function new(string name,ovm_component parent);
endclass
```

File

methodology/ovm_in_order_comparator.svh

Parameters

`type T = int`

Specifies the type of transactions to be compared.

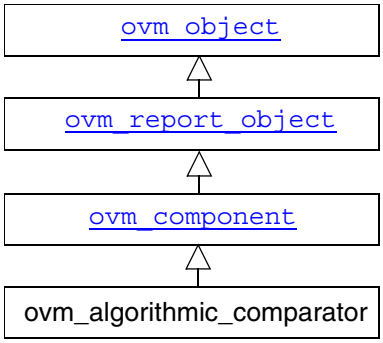
Methods

new

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

This is the normal `ovm_component` constructor.

ovm_algorithmic_comparator #(BEFORE,AFTER,TRANSFORMER)



The algorithmic comparator is a wrapper around `ovm_in_order_class_comparator`. Like the in-order comparator, the algorithmic comparator compares two streams of transactions, the “before” stream and the “after” stream. It is often the case when two streams of transactions need to be compared that the two streams are in different forms. That is, the type of the before transaction stream is different than the type of the after transaction stream.

The `ovm_algorithmic_comparator` provides a transformer that transforms before transactions into after transactions. The transformer is supplied to the algorithmic comparator as a *policy class* via the class parameter `TRANSFORMER`. The transformer policy must provide a `transform()` method with the following prototype:

```
    AFTER transform (BEFORE b);
```

Summary

```
class ovm_algorithmic_comparator #(type BEFORE=int,
                                   type AFTER=int,
                                   type TRANSFORMER=int_transform)
                                   extends ovm_component;
    function new(TRANSFORMER transformer, string name, ovm_component parent);
    ovm_analysis_export #(AFTER) after_export;
    ovm_analysis_imp #(BEFORE, this_type) before_export;
endclass
```

File

methodology/ovm_algorithmic_comparator.svh

Virtual

no

Parameters

`type AFTER = int`

The type of the transaction against which the transformed `BEFORE` transactions will be compared.

`type BEFORE = int`

The type of incoming transaction to be transformed prior to comparing against the `AFTER` transactions.

`type TRANSFORMER = int_transform`

The type of the class that contains the `transform()` method.

Members

`typedef ovm_algorithmic_comparator #(BEFORE, AFTER, TRANSFORMER) this_type`

after_export

`ovm_analysis_export #(AFTER) after_export`

Provides a `write (AFTER t)` method so that publishers (*monitors*) can send in an ordered stream of transactions against which the transformed `BEFORE` transactions will be compared.

before_export

`ovm_analysis_imp #(BEFORE, this_type) before_export`

Provides a `write (BEFORE t)` method so that publishers (*monitors*) can send in an ordered stream of transactions to be transformed and compared to the `AFTER` transactions.

Methods

new

`function new(TRANSFORMER transformer, string name, ovm_component parent)`

The constructor takes a handle to an externally constructed *transformer*, a *name*, and a *parent*. The last two arguments are the normal arguments for an ovm_component constructor.

We create an instance of the transformer (rather than making it a genuine policy class with a static transform method) because we might need to do reset and configuration on the transformer itself.

write

```
function void write(BEFORE b)
```

This method handles incoming BEFORE transactions. It is usually accessed via the before_export, and it transforms the BEFORE transaction into an AFTER transaction before passing it to the in_order_class_comparator.

OVM Macros

OVM provides a number of macros to make common code easier to write. It is never necessary to use the macros, but in many cases the macros can save a substantial amount of user written code.

The OVM macros include:

- Utility Macros
- Sequence Macros
- Sequencer Macros
- Field Macros
- Array Printing Macros

Utility Macros

The utility macros provide overrides for the `create` method, which is needed for cloning, and the `get_type_name` method, which is needed for a number of debugging features. They also register the type with the `ovm_factory`, and they implement the `get_type` method, which eases usage of the factory's type-based methods.

Below is an example usage of the utility macros with the field macros. By using the macros, you do not have to implement any of the data methods to get all of the capabilities of an `ovm_object`.

```
class mydata extends ovm_object;
    string str;
    mydata subdata;
    int field;
    myenum e1;
    int queue[$];
    `ovm_object_utils_begin(mydata) //requires ctor with default args
        `ovm_field_string(str, OVM_DEFAULT)
        `ovm_field_object(subdata, OVM_DEFAULT)
        `ovm_field_int(field, OVM_DEC) //use decimal radix
        `ovm_field_enum(e1, OVM_DEFAULT)
        `ovm_field_queue_int(queue, OVM_DEFAULT)
    `ovm_object_utils_end
endclass
```

`ovm_object_utils

`ovm_object_param_utils

`ovm_object_utils_begin

`ovm_object_param_utils_begin

`ovm_object_utils_end

```
// for simple objects with no field macros
`ovm_object_utils(TYPE)

// for simple objects with field macros
`ovm_object_utils_begin(TYPE)
  // `ovm_field_* macro invocations here
`ovm_object_utils_end

// for parameterized objects with no field macros
`ovm_object_param_utils(TYPE)s

// for parameterized objects, with field macros
`ovm_object_param_utils_begin(TYPE)
  // `ovm_field_* macro invocations here
`ovm_object_utils_end
```

`ovm_object`-based class declarations may contain one of the above forms of utility macros.

Simple (non-parameterized) objects use the `ovm_object_utils*` versions, which do the following:

- Implements `get_type_name`, which returns `TYPE` as a string
- Implements `create`, which allocates an object of type `TYPE` by calling its constructor with no arguments. `TYPE`'s constructor, if defined, must have default values on all it arguments.
- Registers the `TYPE` with the factory, using the string `TYPE` as the factory lookup string for the type.

Parameterized classes must use the `ovm_object_param_utils*` versions. They differ from `` `ovm_object_utils `` only in that they do not supply a type name when registering the object with the factory. As such, name-based lookup with the factory for parameterized classes is not possible.

The macros with `_begin` suffixes are the same as the non-suffixed versions except that they also start a block in which `` `ovm_field_* `` macros can be placed. The block must be terminated by `` `ovm_object_utils_end ``.

Objects deriving from `ovm_sequence` must use the `` `ovm_sequence_* `` macros instead of these macros. See `ovm_sequence_utils on page 259 for details.

`ovm_component_utils

`ovm_component_param_utils

`ovm_component_utils_begin

`ovm_component_param_utils_begin

`ovm_component_utils_end

```
// for simple components with no field macros
`ovm_component_utils(TYPE)

// for simple components with field macros
`ovm_component_utils_begin(TYPE)
  // `ovm_field_* macro invocations here
`ovm_component_utils_end

// for parameterized components with no field macros
`ovm_component_param_utils(TYPE)

// for parameterized components with field macros
`ovm_component_param_utils_begin(TYPE)
  // `ovm_field_* macro invocations here
`ovm_component_utils_end
```

`ovm_component`-based class declarations may contain one of the above forms of utility macros.

Simple (non-parameterized) components must use the `ovm_components_utils*` versions, which do the following:

- Implements `get_type_name`, which returns `TYPE` as a string.

- Implements `create`, which allocates a component of type `TYPE` using a two argument constructor. `TYPE`'s constructor must have a name and a parent argument.
- Registers the `TYPE` with the factory, using the string `TYPE` as the factory lookup string for the type.

Parameterized classes must use the `ovm_object_param_utils*` versions. They differ from `` `ovm_object_utils `` only in that they do not supply a type name when registering the object with the factory. As such, name-based lookup with the factory for parameterized classes is not possible.

The macros with `_begin` suffixes are the same as the non-suffixed versions except that they also start a block in which `` `ovm_field_* `` macros can be placed. The block must be terminated by `` `ovm_component_utils_end ``.

Components deriving from `ovm_sequencer` must use the `` `ovm_sequencer_* `` macros instead of these macros. See `ovm_sequencer_utils on page 262 for details.

`ovm_field_utils_begin

`ovm_field_utils_end

```
`ovm_field_utils_begin(TYPE)
  // `ovm_field_* macro invocations here
`ovm_field_utils_end
```

These macros form a block in which `` `ovm_field_* `` macros can be placed.

These macros do NOT perform factory registration, implement `get_type_name`, nor implement the `create` method. Use this form when you need custom implementations of these two methods, or when you are setting up field macros for an abstract class (i.e. `virtual class`).

Sequence Macros

`ovm_register_sequence

```
`ovm_register_sequence(TYPE_NAME, SQR_TYPE_NAME)
```

This macro registers the sequence of type *`TYPE_NAME`* with the sequence library of the given sequencer type, *`SQR_TYPE_NAME`*.

`ovm_sequence_utils

`ovm_sequence_param_utils

`ovm_sequence_utils_begin

`ovm_sequence_param_utils_begin

`ovm_sequence_utils_end

```
// for simple sequences, no field macros
`ovm_sequence_utils(TYPE_NAME,SQR_TYPE_NAME)

// for simple sequences, with field macros
`ovm_sequence_utils_begin(TYPE_NAME,SQR_TYPE_NAME)
  // `ovm_field_* macro invocations here
`ovm_sequence_utils_end

// for parameterized sequences, no field macros
`ovm_sequence_param_utils(TYPE_NAME,SQR_TYPE_NAME)

// for parameterized sequences, with field macros
`ovm_sequence_param_utils_begin(TYPE_NAME,SQR_TYPE_NAME)
  // `ovm_field_* macro invocations here
`ovm_sequence_utils_end
```

One of the above four macro forms can be used in `ovm_sequence`-based class declarations.

The sequence-specific macros perform the same function as the set of`ovm_object_*_utils` macros except that they also register the sequence's type, *`TYPE_NAME`*, with the given sequencer type, *`SQR_TYPE_NAME`*, and define the *`p_sequencer`* variable and `m_set_p_sequencer` method.

Use `` `ovm_sequence_utils[_begin] `` for non-parameterized classes and `` `ovm_sequence_param_utils[_begin] `` for parameterized classes.

Sequence Action Macros

`ovm_do

```
`ovm_do(item_or_sequence)
```

The ‘ovm_do macro initiates activity by creating a new item or sequence of the type passed in, randomizing it, and then executing it. In the case of a sequence a sub-sequence is spawned. In the case of an item, the item is sent to the driver through the associated sequencer.

`ovm_do_pri

`ovm_do_pri(*item_or_sequence*, *prioirity*)

This is the same as `ovm_do except that the sequene item or sequence is executed with the priority specified in the argument

`ovm_do_with

`ovm_do_with(*item_or_sequence*, *constraint_block*)

This is the same as `ovm_do except that the given constraint block is applied to the item or sequence in a `randomize with` statement before execution.

`ovm_do_pri_with

`ovm_do_pri_with(*item_or_sequence*, *prioirity*, *constraint_block*)

This is the same as `ovm_do_pri except that the given constraint block is applied to the item or sequence in a `randomize with` statement before execution.

`ovm_create

`ovm_create(*item_or_sequence*)

This action creates the item or sequence using the factory. It intentionally does zero processing. After this action completes, the user can manually set values, manipulate `rand_mode` and `constraint_mode`, etc.

`ovm_send

`ovm_send(*item_or_sequence*)

This macro processes the item or sequence that has been created using `ovm_create`. The processing is done without randomization. Essentially, an `ovm_do without the create or randomization.

`ovm_send_pri

`ovm_send_pri(*item_or_sequence*, *prioirity*)

This is the same as `ovm_send except that the sequene item or sequence is executed with the priority specified in the argument

`ovm_rand_send

`ovm_rand_send(*item_or_sequence*)

This macro processes the item or sequence that has been created using `ovm_create. The processing is done with randomization. Essentially, an `ovm_do without the create.

`ovm_rand_send_pri

`ovm_rand_send_pri(*item_or_sequence*, *prioirity*)

This is the same as `ovm_rand_send except that the sequene item or sequence is executed with the priority specified in the argument

`ovm_rand_send_with

`ovm_rand_send_with(*item_or_sequence*, *constraint_block*)

This is the same as `ovm_rand_send except that the given constraint block is applied to the item or sequence in a `randomize with` statement before execution.

`ovm_rand_send_pri_with

`ovm_rand_send_pri_with(*item_or_sequence*, *prioirity*, *constraint_block*)

This is the same as `ovm_rand_send_pri except that the given constraint block is applied to the item or sequence in a `randomize with` statement before execution.

`ovm_create_on

`ovm_create_on(*item_or_sequence*, *seqr_ref*)

This is the same as `ovm_create except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *seqr_ref* argument.

`ovm_do_on

`ovm_do_on(*item_or_sequence*, *seqr_ref*)

This is the same as `ovm_do except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *seqr_ref* argument.

`ovm_do_on_pri

`ovm_do_on_pri(*item_or_sequence*, *seqr_ref*, *priority*)

This is the same as `ovm_do_pri except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *seqr_ref* argument.

`ovm_do_on_pri_with

`ovm_do_on_pri_with(*item_or_sequence*,*seqr_ref*,*priority*,*constraint_block*)

This is the same as `ovm_do_pri_with except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *seqr_ref* argument.

Sequencer Macros

`ovm_sequencer_utils

`ovm_sequencer_param_utils

`ovm_sequencer_utils_begin

`ovm_sequencer_param_utils_begin

`ovm_sequencer_utils_end

```
// for simple sequencers, no field macros
`ovm_sequencer_utils(SQR_TYPE_NAME)

// for simple sequencers, with field macros
`ovm_sequencer_utils_begin(SQR_TYPE_NAME)
  // `ovm_field_* macros here
`ovm_sequencer_utils_end

// for parameterized sequencers, no field macros
`ovm_sequencer_param_utils(SQR_TYPE_NAME)

// for parameterized sequencers, with field macros
`ovm_sequencer_param_utils_begin(SQR_TYPE_NAME)
  // `ovm_field_* macros here
`ovm_sequencer_utils_end
```

One of the above four macro forms can be used in `ovm_sequencer`-based class declarations.

The sequencer-specific macros perform the same function as the set of `` `ovm_componenent_*utils `` macros except that they also declare the plumbing necessary for creating the sequencer's sequence library. This includes:

1. Declaring the type-based static queue of strings registered on the sequencer type.

2. Declaring the static function to add strings to item #1 above.

3. Declaring the static function to remove strings to item #1 above.

4. Declaring the function to populate the instance specific sequence library for a sequencer.

Use `` `ovm_sequencer_utils[_begin] `` for non-parameterized classes and `` `ovm_sequencer_param_utils[_begin] `` for parameterized classes.

`ovm_update_sequence_lib

```
`ovm_update_sequence_lib
```

This macro populates the instance-specific sequence library for a sequencer. It should be invoked inside the sequencer's constructor.

`ovm_update_sequence_lib_and_item

```
`ovm_update_sequence_lib_and_item(ITEM_TYPE_NAME)
```

This macro does two things:

1. Populates the instance specific sequence library for a sequencer.

2. Registers *`ITEM_TYPE_NAME`* as the instance override for the simple sequence's *`item`* variable.

The macro should be invoked inside the sequencer's constructor.

Field Macros

The `` `ovm_field_* `` macros are invoked inside of the `` `ovm_*_utils_begin `` and `` `ovm_*_utils_end `` macro blocks to form "automatic" implementations of the core data methods: *copy*, *compare*, *pack*, *unpack*, *record*, *print*, and *sprint*. For example:

```
class my_trans extends ovm_transaction;
```

```
  string my_string;
  `ovm_object_utils_begin(my_trans)
    `ovm_field_string(my_string, OVM_ALL_ON)
  `ovm_object_utils_end
endclass
```

Each `ovm_field_* macro is named to correspond to a particular data type: integrals, strings, objects, queues, etc., and each has at least two arguments: *ARG* and *FLAG*.

ARG is the instance name of the variable, whose type must be compatible with the macro being invoked. In the example, class variable my_string is of type string, so we use the `ovm_field_string macro.

If *FLAG* is set to OVM_ALL_ON, as in the example, the *ARG* variable will be included in all data methods. The *FLAG*, if set to something other than OVM_ALL_ON or OVM_DEFAULT, specifies which data method implementations will NOT include the given variable. Thus, if *FLAG* is specified as NO_COMPARE, the *ARG* variable will not affect comparison operations, but it will be included in everything else.

All possible values for *FLAG* are listed and described below. Multiple flag values can be bitwise ORed together (in most cases they may be added together as well, but care must be taken when using the + operator to ensure that the same bit is not added more than once).

Table 1-5 Field Macro Flags

Flag	Meaning
OVM_DEFAULT	Use the default flag settings.
OVM_ALL_ON	Set all operations on (default).
OVM_COPY	Do a copy for this field (default).
OVM_NOCOPY	Do not copy this field.
OVM_COMPARE	Do a compare for this field (default).
OVM_NOCOMPARE	Do not compare this field.
OVM_PRINT	Print this field (default).
OVM_NOPRINT	Do not print this field.

OVM_NODEFPRINT	Do not print the field if it is the same as its default value.
OVM_PACK	Pack and unpack this field (default).
OVM_NOPACK	Do not pack or unpack this field.
OVM_PHYSICAL	Treat as a physical field. Use physical setting in policy class for this field.
OVM_ABSTRACT	Treat as an abstract field. Use the abstract setting in the policy class for this field.
OVM_READONLY	Do not allow setting of this field from the set_*_local methods.
OVM_BIN, OVM_DEC, OVM_UNSIGNED, OVM_OCT, OVM_HEX, OVM_STRING, OVM_TIME, OVM_NORADIX	Radix settings for integral types. Hex is the default radix if none is specified.

`ovm_field_int

```
`ovm_field_int(ARG, FLAG)
```

This macro implements the data operations for packed integral types.

`ovm_field_enum

```
`ovm_field_enum(TYPE, ARG, FLAG)
```

This macro implements the data operations for enumerated types.

For *enums*, the *TYPE* argument is necessary to specify the enumerated type. This is needed because of SystemVerilog strong typing rules with respect to enumerated types.

`ovm_field_object

```
`ovm_field_object(ARG, FLAG)
```

This macro implements the data operations for ovm_object derived objects.

`ovm_field_string

```
`ovm_field_string(ARG, FLAG)
```

This macro implements the data operations for `string` types.

`ovm_field_array_int

```
`ovm_field_array_int(ARG, FLAG)
```

This macro implements the data operations for dynamic arrays of `integral` types.

`ovm_field_array_object

```
`ovm_field_array_object(ARG, FLAG)
```

This macro implements the data operations for dynamic arrays of `ovm_object` types.

`ovm_field_array_string

```
`ovm_field_array_string(ARG, FLAG)
```

This macro implements the data operations for dynamic arrays of `string` types.

`ovm_field_queue_int

```
`ovm_field_queue_int(ARG, FLAG)
```

This macro implements the data operations for queues of `integral` types.

`ovm_field_queue_object

```
`ovm_field_queue_object(ARG, FLAG)
```

This macro implements the data operations for queues of `ovm_object` types.

`ovm_field_queue_string

```
`ovm_field_queue_string(ARG, FLAG)
```

This macro implements the data operations for queues of `string` types.

`ovm_field_aa_int_string

```
`ovm_field_aa_int_string(ARG, FLAG)
```

This macro implements the data operations for associative arrays of `integral` types with `string` keys.

`ovm_field_aa_object_string

```
`ovm_field_aa_object_string(ARG, FLAG)
```

This macro implements the data operations for associative arrays of `ovm_object` types with `string` keys.

`ovm_field_aa_string_string

```
`ovm_field_aa_string_string(ARG, FLAG)
```

This macro implements the data operations for associative arrays of `string` types with `string` keys.

`ovm_field_aa_int_<key_type>

```
`ovm_field_aa_int_<key_type>(ARG, FLAG)
```

These macros implement the data operations for associative arrays of integral types with integral keys.

The key type can be any of the following: `int`, `integer`, `int_unsigned`, `integer_unsigned`, `byte`, `byte_unsigned`, `shortint`, `shortint_unsigned`, `longint`, `longint_unsigned`.

`ovm_field_aa_string_int

```
`ovm_field_aa_string_int(ARG, FLAG)
```

This macro implements the data operations for associative arrays of `string` types with `int` keys.

`ovm_field_aa_object_int

```
`ovm_field_aa_object_int(ARG, FLAG)
```

This macro implements the data operations for associative arrays of `ovm_object` types with `int` keys.

Array Printing Macros

The array printing macros can be used inside of the `do_print()` method of an `ovm_object` derived class to add the appropriate code from printing a queue, dynamic array or associative array.

`ovm_print_aa_int_object2

`ovm_print_aa_int_object2(*field*, *printer*)

This macro implements array printing for an associative array of ovm_object types with an int key.

field is the field to print and is also used as the name of the field.

printer is the printer to use.

`ovm_print_aa_int_key4

`ovm_print_aa_int_key4(*key_type*, *field*, *radix*, *printer*)

This macro implements array printing for an associative array of integral types with an arbitrary key type.

key_type is the type of the indexing variable for the array.

field is the field to print and is also used as the name of the field.

radix is the radix to use for each element.

printer is the printer to use.

`ovm_print_aa_string_int3

`ovm_print_aa_string_int3(*field*, *radix*, *printer*)

This macro implements array printing for an associative array of integral types with a string key.

field is the field to print and is also used as the name of the field.

radix is the radix to use for the elements.

printer is the printer to use.

`ovm_print_aa_string_object2

`ovm_print_aa_string_object2(*field*, *printer*)

This macro implements array printing for an associative array of ovm_object types with a string key.

field is the field to print and is also used as the name of the field.

printer is the printer to use.

`ovm_print_aa_string_string2

`ovm_print_aa_string_string2(*field*, *printer*)

This macro implements array printing for an associative array of string types with a `string` key.

field is the field to print and is also used as the name of the field.

printer is the printer to use.

`ovm_print_object_qda3

`ovm_print_object_qda3(*field*, *printer*, *arraytype*)

This macro implements array printing for an `ovm_object` array type.

field is the field to print and is also used as the name of the field.

printer is the printer to use.

arraytype is the type name to use when printing the array (no quotes are used).

`ovm_print_qda_int4

`ovm_print_qda_int4(*field*, *radix*, *printer*, *arraytype*)

This macro implements array printing for an integral array type.

field is the field to print and is also used as the name of the field.

radix is the radix to use for the elements.

printer is the printer to use.

`arraytype` is the type name to use when printing the array (no quotes are used).

`ovm_print_string_qda3

`ovm_print_string_qda3(*field*, *printer*, *arraytype*)

This macro implements array printing for a string array type.

field is the field to print and is also used as the name of the field.

printer is the printer to use.

arraytype is the type name to use when printing the array (no quotes are used).

Transactions

ovm_built_in_clone #(T)

This policy class is used to clone built-in types. It is used to build generic components that will work with either classes or built-in types.

Summary

```
class ovm_built_in_clone #(type T=int);
    static function T clone(input T from);
endclass
```

File

methodology/ovm_policies.svh

Virtual

No

Parameters

```
type T = int
```

The return type of the `clone()` method.

Members

None

Methods

clone

```
static function T clone(input T from)
```

Returns the value of from.

ovm_built_in_comp #(T)

This policy class is used to compare built-in types. It is used to build generic components that work with either classes or built-in types.

Summary

```
class ovm_built_in_comp #(type T=int);
    static function bit comp(input T a, input T b);
endclass
```

File

methodology/ovm_policies.svh.

Virtual

No

Parameters

`type T = int`

The type of the items to be compared.

Members

None

Methods

comp

```
static function bit comp(input T a, input T b)
```

Returns the value of *a==b*.

ovm_built_in_converter #(T)

This policy class is used to convert built-in types to strings. It is used to build generic components that will work with either classes or built-in types.

Summary

```
class ovm_built_in_converter #(type T=int);
    static function string convert2string (input T t);
endclass
```

File

methodology/ovm_policies.svh

Virtual

No

Parameters

`type T = int`

The type of the item to be converted.

Members

None

Methods

convert2string

```
static function string convert2string(input T t);
```

Returns the value of `t` as a string.

ovm_built_in_pair #(T1,T2)

This class represents a pair of built-in types.

Summary

```
class ovm_built_in_pair #(type T1=int, type T2=T1) extends ovm_transaction;
    virtual function string convert2string();
    function bit comp (this_type t);
    function void copy (input this_type t);
    function ovm_transaction clone ();
endclass
```

File

utils/ovm_pair.svh

Virtual

No

Parameters

`type T1 = int`

The type of the first element of the pair.

`type T2 = T1`

The type of the second element of the pair. By default, the two types are the same.

Members

`typedef ovm_built_in_pair #(T1, T2) this_type`

`T1 first`

The first element of the pair.

`T2 second`

The second element of the pair.

Methods

Since `ovm_built_in_pair` is a transaction class, it provides the four compulsory methods as defined by `ovm_object`.

convert2string

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

comp

```
function bit comp(this_type t)
```

copy

```
function void copy(input this_type t)
```

clone

```
function ovm_transaction clone()
```

ovm_class_clone #(T)

This policy class is used to clone classes. It is used to build generic components that work with either classes or built-in types.

Summary

```
class ovm_class_clone #(type T=int);
    static function ovm_transaction clone (input T from);
endclass
```

File

methodology/ovm_policies.svh

Virtual

No

Members

None

Methods

clone

```
static function ovm_transaction clone(input T from)
```

This method returns `from.clone()`.

ovm_class_comp #(T)

This policy class is used to compare classes. It is used to build generic components that work with either built-in types or classes.

Summary

```
class ovm_class_comp #(type T=int);
    static function bit comp (input T a, input T b);
endclass
```

File

methodology/ovm_policies.svh

Virtual

No

Members

None

Methods

comp

```
static function bit comp (input T a, input T b)
```

This method returns `a.comp(b)`.

ovm_class_converter #(T)

This policy class is used to convert classes to strings. It is used to build generic components that work with either built-in types or classes.

Summary

```
class ovm_class_converter #(type T=int);
    static function string convert2string (input T t);
endclass
```

File

base/ovm_policies.svh

Virtual

No

Members

None

Methods

convert2string

```
static function string convert2string(input T t)
```

This method returns `t.convert2string()`.

ovm_class_pair #(T1,T2)

This class represents a pair of classes.

Summary

```
class ovm_class_pair #(type T1=int, type T2=T1) extends ovm_transaction;
    typedef ovm_class_pair #(T1, T2) this_type;
    function new(input T1 f=null, input T2 s=null);
    function string convert2string;
    function bit comp(this_type t);
    function void copy(input this_type t);
    function ovm_transaction clone;
endclass
```

File

methodology/ovm_pairs.svh

Virtual

No

Members

`T1 first`

This is the first element in the pair.

`T2 second`

This is the second element in the pair.

Methods

new

```
function new(input T1 f=null, input T2 s=null)
```

A constructor, with optional arguments for first and second. No cloning is performed for nondefault values.

convert2string

```
function string convert2string
```

comp

```
function bit comp(this_type t)
```

copy

```
function void copy(input this_type t)
```

clone

```
function ovm_transaction clone
```

Since `ovm_built_in_pair` is a transaction class, it provides the four compulsory methods as defined by `ovm_object`.

Global Functions and Variables

The following functions and variables are defined in `ovm_pkg` space. They are globally visible to all OVM classes and any user code that imports `ovm_pkg`.

ovm_top

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

This is the implicit top-level that governs phase execution and provides component search interface. See ovm_root on page 65 for more information.

factory

```
const ovm_factory factory = ovm_factory::get();
```

The singleton instance of ovm_factory on page 97, which is used to create objects and components based on type and instance overrides.

run_test

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

Convenience function for `ovm_top.run_test()`. See ovm_root on page 65 for more information.

global_stop_request

```
function void global_stop_request()
```

Convenience function for `ovm_top.stop_request()`. See ovm_root on page 65 for more information.

set_global_timeout

```
function void set_global_timeout(time timeout)
```

Convenience function for `ovm_top.phase_timeout = timeout`. See ovm_root on page 65 for more information.

set_global_stop_timeout

```
function void set_global_stop_timeout(time timeout)
```

Convenience function for `ovm_top.stop_timeout = timeout`. See ovm_root on page 65 for more information.

ovm_is_match

```
function bit ovm_is_match (string expr, string str)
```

Returns 1 if the two strings match, 0 otherwise.

The first string, *expr*, is a string that may contain '*' and '?' characters. A * matches zero or more characters, and ? matches any single character.

build_ph

connect_ph

end_of_elaboration_ph

start_of_simulation_ph

run_ph

extract_ph

check_ph

report_ph

```
`ovm_phase_func_topdown_decl(build)
build_phase #(ovm_component) build_ph = new();
`ovm_phase_task_bottomup_decl(run)
run_phase   #(ovm_component) run_ph  = new();
...
```

These objects represent all the predefined OVM phases. Two examples of their declaration and initialization are given. See ovm_phase on page 59 for more information.

set_config_int

set_config_string

set_config_object

```
function void set_config_int    (string inst_name,
                                 string field_name,
```

```
                                    ovm_bitstream_t value)
function void set_config_string (string inst_name,
                                    string field_name,
                                    string value)
function void set_config_object (string inst_name,
                                    string field_name,
                                    ovm_object value,
                                    bit clone=1)
```

These are the global versions of the report, set_config_string, and set_config_object in ovm_component on page 34. They place the configuration settings in the global override table, which has highest precedence over any component-level setting.

ovm_bitstream_t

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

The bitstream type is used as a argument type for passing integral values in such methods as set_int_local, get_int_local, get_config_int, report, pack and unpack.

Printing

ovm_default_printer

ovm_default_table_printer

ovm_default_tree_printer

ovm_default_line_printer

```
ovm_table_printer ovm_default_table_printer = new();
ovm_tree_printer  ovm_default_tree_printer  = new();
ovm_line_printer  ovm_default_line_printer  = new();
ovm_printer       ovm_default_printer       = ovm_default_table_printer;
```

Reporting

ovm_severity / ovm_severity_type

```
typedef bit [1:0] ovm_severity;
typedef enum ovm_severity
{
  OVM_INFO,
  OVM_WARNING,
  OVM_ERROR,
  OVM_FATAL
} ovm_severity_type;
```

Defines all possible values for report severity.

ovm_action / ovm_action_type

```
typedef bit [5:0] ovm_action;
typedef enum ovm_action
{
  OVM_NO_ACTION = 6'b000000,
  OVM_DISPLAY   = 6'b000001, // send report to standard output
  OVM_LOG       = 6'b000010, // send report to one or more file(s)
  OVM_COUNT     = 6'b000100, // increment report counter
  OVM_EXIT      = 6'b001000, // terminate simulation immediately
  OVM_CALL_HOOK = 6'b010000, // call report_hook methods
  OVM_STOP      = 6'b100000  // issue a stop_request, ending current phase
} ovm_action_type;
```

Defines all possible values for report actions. Each report is configured to execute one or more actions, determined by the bitwise OR of any or all of the following enumeration constants.

ovm_verbosity

Verbosity values are just integers. This enum provides some predefined verbosity levels.

```
typedef enum {
  OVM_NONE   = 0,
  OVM_LOW    = 10000,
  OVM_MEDIUM = 20000,
  OVM_HIGH   = 30000,
  OVM_FULL   = 40000
```

```
} ovm_verbosity;
```

OVM_FILE

```
typedef int OVM_FILE;
```

_global_reporter

```
ovm_reporter _global_reporter
```

The `_global_reporter` is an instance of `ovm_report_object` that can be used by non-component-based code, including modules and interfaces.

ovm_report_fatal

ovm_report_error

ovm_report_warning

ovm_report_info

```
function void ovm_report_fatal  (string id, string message,
                                 int verbosity=0,
                                 string filename="", int line=0)
function void ovm_report_error  (string id, string message,
                                 int verbosity=100,
                                 string filename="", int line=0)
function void ovm_report_warning(string id, string message,
                                 int verbosity=200,
                                 string filenamee="", int line=0)
function void ovm_report_info   (string id, string message,
                                 int verbosity=300,
                                 string filename="", int line=0)
```

These methods, defined in package scope, are convenience functions that delegate to the corresponding methods in `_global_reporter`. See ovm_report_object on page 70 for details on their behavior.

Index

A

B

C

F

G

M

P

R

S

T

U

Utility Macros