



Universal Verification Methodology (UVM) 1.0 EA Class Reference

May 2010

Copyright[©] 2010 Accellera. All rights reserved.

Notices

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

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

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

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

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

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

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

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

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

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

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

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

vip-tc@lists.accellera.org

The current Working Group's website address is

www.accellera.org/activities/vip

UVM Class Reference

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

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

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

Base

This basic building blocks for all environments are components, which do the actual work, transactions, which convey information between components, and ports, which provide the interfaces used to convey transactions. The UVM's core *base* classes provide these building blocks. See [Core Base Classes](#) for more information.

Reporting

The *reporting* classes provide a facility for issuing reports (messages) with consistent formatting and configurable side effects, such as logging to a file or exiting simulation. Users can also filter out reports based on their verbosity , unique ID, or severity. See [Reporting Classes](#) for more information.

Factory

As the name implies, the UVM factory is used to manufacture (create) UVM objects and components. Users can configure the factory to produce an object of a given type on a global or instance basis. Use of the factory allows dynamically configurable component hierarchies and object substitutions without having to modify their code and without breaking encapsulation. See [Factory Classes](#) for details.

Synchronization

The UVM provides event and barrier synchronization classes for process synchronization. See [Synchronization Classes](#) for more information.

Policies

Each of UVM's policy classes perform a specific task for *uvm_object*-based objects: printing, comparing, recording, packing, and unpacking. They are implemented separately from *uvm_object* so that users can plug in different ways to print, compare, etc. without modifying the object class being operated on. The user can simply apply a different printer or compare "policy" to change how an object is printed or compared. See [Policy Classes](#) for more information.

TLM

The UVM TLM library defines several abstract, transaction-level interfaces and the ports and exports that facilitate their use. Each TLM interface consists of one or more methods used to transport data, typically whole transactions (objects) at a time. Component designs that use TLM ports and exports to communicate are inherently more reusable, interoperable, and modular. See [TLM Interfaces, Ports, and Exports](#) for details.

Components

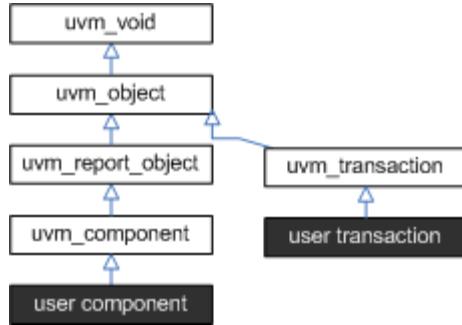
Components form the foundation of the UVM. They encapsulate behavior of drivers, scoreboards, and other objects in a testbench. The UVM library provides a set of predefined component types, all derived directly or indirectly from *uvm_component*. See [Predefined Component Classes](#) for more information.

<i>Sequencers</i>	The sequencer serves as an arbiter for controlling transaction flow from multiple stimulus generators. More specifically, the sequencer controls the flow of <code>uvm_sequence_item</code> -based transactions generated by one or more <code>uvm_sequence #(REQ,RSP)</code> -based sequences. See Sequencer Classes for more information.
<i>Sequences</i>	Sequences encapsulate user-defined procedures that generate multiple <code>uvm_sequence_item</code> -based transactions. Such sequences can be reused, extended, randomized, and combined sequentially and hierarchically in interesting ways to produce realistic stimulus to your DUT. See Sequence Classes for more information.
<i>Macros</i>	The UVM provides several macros to help increase user productivity. See Utility and Field Macros and Sequence and Do Action Macros for a complete list.
<i>Globals</i>	This category defines a small list of types, variables, functions, and tasks defined in <code>uvm_pkg</code> scope. These items are accessible from any scope that imports the <code>uvm_pkg</code> . See Types and Enumerations and Globals for details.

Core Base Classes

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

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



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

uvm_void

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

uvm_object

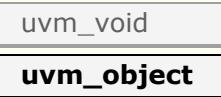
The uvm_object class is the base class for all UVM data and hierarchical classes. Its primary role is to define a set of methods for such common operations as [create](#), [copy](#), [compare](#), [print](#), and [record](#). Classes deriving from uvm_object must implement the pure virtual methods such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

Summary

uvm_object

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

CLASS HIERARCHY



CLASS DECLARATION

```
virtual class uvm_object extends uvm_void
```

new

Creates a new uvm_object with the given instance *name*.

SEEDING

use_uvm_seeding

This bit enables or disables the UVM seeding mechanism.

reseed

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

IDENTIFICATION

set_name

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

get_name

Returns the name of the object, as provided by the *name* argument in the [new](#) constructor or [set_name](#) method.

get_full_name

Returns the full hierarchical name of this object.

get_inst_id

Returns the object's unique, numeric instance identifier.

get_inst_count

Returns the current value of the instance counter, which represents the total number of uvm_object-based objects that have been allocated in simulation.

get_type

Returns the type-proxy (wrapper) for this object.

get_object_type

Returns the type-proxy (wrapper) for this object.

get_type_name

This function returns the type name of the object, which is typically the type identifier enclosed in quotes.

CREATION

create

The *create* method allocates a new object of the same type as this object and returns it via a base uvm_object handle.

clone

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

PRINTING

print

The *print* method deep-prints this object's properties in a format and manner governed by the given *printer* argument; if the *printer* argument is not provided, the global [uvm_default_printer](#) is used.

sprint

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

do_print

The *do_print* method is the user-definable hook called by [print](#) and [sprint](#) that allows users to customize what gets printed or sprinted beyond the field information

<code>convert2string</code>	provided by the <code><`uvm_field_*></code> macros. This virtual function is a user-definable hook, called directly by the user, that allows users to provide object information in the form of a string.
Fields declared in <code><`uvm_field_*></code> macros, if used, will not	automatically appear in calls to <code>convert2string</code> .
RECORDING	
<code>record</code>	The <code>record</code> method deep-records this object's properties according to an optional <i>recorder</i> policy.
<code>do_record</code>	The <code>do_record</code> method is the user-definable hook called by the <code>record</code> method.
COPYING	
<code>copy</code>	The <code>copy</code> method returns a deep copy of this object.
<code>do_copy</code>	The <code>do_copy</code> method is the user-definable hook called by the <code>copy</code> method.
COMPARING	
<code>compare</code>	The <code>compare</code> method deep compares this data object with the object provided in the <i>rhs</i> (right-hand side) argument.
<code>do_compare</code>	The <code>do_compare</code> method is the user-definable hook called by the <code>compare</code> method.
PACKING	
<code>pack</code>	
<code>pack_bytes</code>	
<code>pack_ints</code>	
<code>do_pack</code>	The <code>pack</code> methods bitwise-concatenate this object's properties into an array of bits, bytes, or ints. The <code>do_pack</code> method is the user-definable hook called by the <code>pack</code> methods.
UNPACKING	
<code>unpack</code>	
<code>unpack_bytes</code>	
<code>unpack_ints</code>	
<code>do_unpack</code>	The <code>unpack</code> methods extract property values from an array of bits, bytes, or ints. The <code>do_unpack</code> method is the user-definable hook called by the <code>unpack</code> method.
CONFIGURATION	
<code>set_int_local</code>	
<code>set_string_local</code>	
<code>set_object_local</code>	These methods provide write access to integral, string, and <code>uvm_object</code> -based properties indexed by a <i>field_name</i> string.

new

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

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

SEEDING

use_uvm_seeding

```
static bit use_uvm_seeding = 1
```

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

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

[reseed](#)

```
function void reseed ()
```

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

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

[IDENTIFICATION](#)

[set_name](#)

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

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

[get_name](#)

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

Returns the name of the object, as provided by the `name` argument in the [new](#) constructor or `set_name` method.

[get_full_name](#)

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

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

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

[get_inst_id](#)

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

Returns the object's unique, numeric instance identifier.

get_inst_count

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

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

get_type

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

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

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

For example

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

Then, to use

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

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

get_object_type

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

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

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

For example

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

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

get_type_name

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

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

This function must be defined in every derived class.

A typical implementation is as follows

```
class mytype extends uvm_object;
  ...
  const static string type_name = "mytype";
  virtual function string get_type_name();
    return type_name;
  endfunction
```

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

CREATION

create

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

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

A typical implementation is as follows

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

```
mytype t = new(name);
return t;
endfunction
```

clone

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

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

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

PRINTING

print

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

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

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

sprint

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

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

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

do_print

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

The `do_print` method is the user-definable hook called by `print` and `sprint` that allows users to customize what gets printed or sprinted beyond the field information provided by the `<`uvm_field_*>` macros.

The `printer` argument is the policy object that governs the format and content of the output. To ensure correct `print` and `sprint` operation, and to ensure a consistent output format, the `printer` must be used by all `do_print` implementations. That is, instead of

using \$display or string concatenations directly, a *do_print* implementation must call through the *printer*'s API to add information to be printed or sprinted.

An example implementation of *do_print* is as follows

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

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

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

See [uvm_printer](#) for information about the printer API.

convert2string

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

Fields declared in <`uvm_field_*> macros, if used, will not

automatically appear in calls to convert2string.

An example implementation of convert2string follows.

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

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

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

Then, to display an object, you could write

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

The output will look similar to

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

RECORDING

record

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

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

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

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

do_record

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

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

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

A typical implementation is as follows

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

COPYING

copy

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

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

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

do_copy

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

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

A typical implementation is as follows

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

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

COMPARING

compare

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

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

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

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

See [uvm_comparer](#) for more information.

[do_compare](#)

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

The `do_compare` method is the user-definable hook called by the [compare](#) method. A derived class should override this method to include its fields in a compare operation.

A typical implementation is as follows

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

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

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

[PACKING](#)

[pack](#)

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

[pack_bytes](#)

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

[pack_ints](#)

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

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

The optional `packer` argument specifies the packing policy, which governs the packing operation. If a packer policy is not provided, the global `uvm_default_packer` policy is used. See [uvm_packer](#) for more information.

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

do_pack

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

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

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

A typical example of an object packing itself is as follows

```
class mysubtype extends mysupertype;
  ...
  shortint myshort;
  obj_type myobj;
  byte myarray[];

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

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

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

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

When the `uvm_*_field macros are used, the above meta information is included provided the `uvm_packer`'s `<use_metadata>` variable is set.

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

UNPACKING

unpack

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

unpack_bytes

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

unpack_ints

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

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

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

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

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

do_unpack

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

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

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

The following implementation corresponds to the example given in [do_pack](#).

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

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

- For queues, dynamic arrays, or associative arrays, unpack the number of elements in the array from the 32 bits immediately before unpacking individual elements, as

shown above.

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

CONFIGURATION

set_int_local

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

set_string_local

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

set_object_local

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

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

The global [uvm_is_match](#) function is used to match the field names, so *field_name* may contain wildcards.

An example implementation of all three methods is as follows.

```
class mytype extends uvm_object;
  local int myint;
  local byte mybyte;
  local shortint myshort; // no access
  local string mystring;
  local obj_type myobj;

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

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

  // provide access to sub-objects
```

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

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

uvm_transaction

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

Summary

uvm_transaction

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

CLASS HIERARCHY



CLASS DECLARATION

```
virtual class uvm_transaction extends uvm_object
```

METHODS

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

METHODS

new

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

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

accept_tr

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

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

This function performs the following actions

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

do_accept_tr

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

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

begin_tr

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

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

This function performs the following actions

- The transaction's internal start time is set to the current simulation time, or to begin_time if provided and non-zero. The begin_time may be any time, past or future, but should not be less than the accept time.
- If recording is enabled, then a new database-transaction is started with the same

begin time as above. The record method inherited from uvm_object is then called, which records the current property values to this new transaction.

- The do_begin_tr method is called to allow for any post-begin action in derived classes.
- The transaction's internal begin event is triggered. Any processes waiting on this event will resume in the next delta cycle.

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

begin_child_tr

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

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

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

This function performs the following actions

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

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

do_begin_tr

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

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

end_tr

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

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

This function performs the following actions

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

do_end_tr

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

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

get_tr_handle

```
function integer get_tr_handle ()
```

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

disable_recording

```
function void disable_recording ()
```

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

enable_recording

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

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

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

is_recording_enabled

```
function bit is_recording_enabled()
```

Returns 1 if recording is currently on, 0 otherwise.

[is_active](#)

```
function bit is_active ()
```

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

[get_event_pool](#)

```
function uvm_event_pool get_event_pool ()
```

Returns the event pool associated with this transaction.

By default, the event pool contains the events: begin, accept, and end. Events can also be added by derivative objects. See [uvm_event_pool](#) for more information.

[set_initiator](#)

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

Sets initiator as the initiator of this transaction.

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

[get_initiator](#)

```
function uvm_component get_initiator ()
```

Returns the component that produced or started the transaction, as set by a previous call to [set_initiator](#).

[get_accept_time](#)

```
function time get_accept_time ()
```

[get_begin_time](#)

```
function time get_begin_time ()
```

[get_end_time](#)

```
function time get_end_time ()
```

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

[set_transaction_id](#)

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

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

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

[get_transaction_id](#)

```
function integer get_transaction_id()
```

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

When using 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.

uvm_component

The uvm_component class is the root base class for UVM components. In addition to the features inherited from [uvm_object](#) and [uvm_report_object](#), uvm_component provides the following interfaces:

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

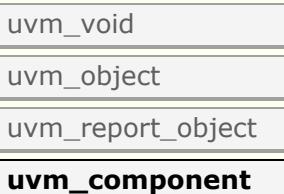
The uvm_component is automatically seeded during construction using UVM seeding, if enabled. All other objects must be manually reseeded, if appropriate. See [uvm_object::reseed](#) for more information.

Summary

uvm_component

The uvm_component class is the root base class for UVM components.

CLASS HIERARCHY



CLASS DECLARATION

```
virtual class uvm_component extends uvm_report_object
```

`new`

Creates a new component with the given leaf instance *name* and handle to its *parent*.

HIERARCHY INTERFACE

[get_parent](#)

These methods provide user access to information about the component hierarchy, i.e., topology.

[get_full_name](#)

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

Returns the full hierarchical name of this

	object.
get_child get_next_child get_first_child	These methods are used to iterate through this component's children, if any.
get_num_children	Returns the number of this component's children.
has_child	Returns 1 if this component has a child with the given <i>name</i> , 0 otherwise.
set_name	Renames this component to <i>name</i> and recalculates all descendants' full names.
lookup	Looks for a component with the given hierarchical <i>name</i> relative to this component.
PHASING INTERFACE	Components execute their behavior in strictly ordered, pre-defined phases.
build	The build phase callback is the first of several methods automatically called during the course of simulation.
connect	The connect phase callback is one of several methods automatically called during the course of simulation.
end_of_elaboration	The end_of_elaboration phase callback is one of several methods automatically called during the course of simulation.
start_of_simulation	The start_of_simulation phase callback is one of several methods automatically called during the course of simulation.
run	The run phase callback is the only predefined phase that is time-consuming, i.e., task-based.
extract	The extract phase callback is one of several methods automatically called during the course of simulation.
check	The check phase callback is one of several methods automatically called during the course of simulation.
report	The report phase callback is the last of several predefined phase methods automatically called during the course of simulation.
suspend	Suspends the process tree spawned from this component's currently executing task-based phase, e.g.
resume	Resumes the process tree spawned from this component's currently executing task-based phase, e.g.
status	Returns the status of the parent process associated with the currently running task-based phase, e.g., run.
kill	Kills the process tree associated with this component's currently running task-based phase, e.g., run.
do_kill_all	Recursively calls kill on this component and all its descendants, which abruptly ends the currently running task-based phase, e.g., run.
stop	The stop task is called when this component's enable_stop_interrupt bit is set and global_stop_request is called during a task-based phase, e.g., run.
enable_stop_interrupt	This bit allows a component to raise an objection to the stopping of the current phase.
resolve_bindings	Processes all port, export, and imp connections.
CONFIGURATION INTERFACE	Components can be designed to be user-configurable in terms of its topology (the

<code>set_config_int</code>	type and number of children it has), mode of operation, and run-time parameters (knobs).
<code>set_config_string</code>	
<code>set_config_object</code>	
<code>get_config_int</code>	Calling <code>set_config_*</code> causes configuration settings to be created and placed in a table internal to this component.
<code>get_config_string</code>	
<code>get_config_object</code>	
<code>check_config_usage</code>	These methods retrieve configuration settings made by previous calls to their <code>set_config_*</code> counterparts. Check all configuration settings in a components configuration table to determine if the setting has been used, overridden or not used.
<code>apply_config_settings</code>	Searches for all config settings matching this component's instance path.
<code>print_config_settings</code>	Called without arguments, <code>print_config_settings</code> prints all configuration information for this component, as set by previous calls to <code>set_config_*</code> .
<code>print_config_matches</code>	Setting this static variable causes <code>get_config_*</code> to print info about matching configuration settings as they are being applied.
OBJECTION INTERFACE	
<code>raised</code>	These methods provide object level hooks into the <code>uvm_objection</code> mechanism. The <code>raised</code> callback is called when a descendant of the component instance raises the specified <i>objection</i> .
<code>dropped</code>	The <code>dropped</code> callback is called when a descendant of the component instance raises the specified <i>objection</i> .
<code>all_dropped</code>	The <code>all_dropped</code> callback is called when a descendant of the component instance raises the specified <i>objection</i> .
FACTORY INTERFACE	
<code>create_component</code>	The factory interface provides convenient access to a portion of UVM's <code>uvm_factory</code> interface. A convenience function for <code>uvm_factory::create_component_by_name</code> , this method calls upon the factory to create a new child component whose type corresponds to the preregistered type name, <code>requested_type_name</code> , and instance name, <code>name</code> .
<code>create_object</code>	A convenience function for <code>uvm_factory::create_object_by_name</code> , this method calls upon the factory to create a new object whose type corresponds to the preregistered type name, <code>requested_type_name</code> , and instance name, <code>name</code> .
<code>set_type_override_by_type</code>	A convenience function for <code>uvm_factory::set_type_override_by_type</code> , this method registers a factory override for components and objects created at this level of hierarchy or below.
<code>set_inst_override_by_type</code>	A convenience function for <code>uvm_factory::set_inst_override_by_type</code> , this method registers a factory override for components and objects created at this level of hierarchy or below.
<code>set_type_override</code>	A convenience function for <code>uvm_factory::set_type_override_by_name</code> , this method configures the factory to create an object of type <code>override_type_name</code>

<code>set_inst_override</code>	whenever the factory is asked to produce a type represented by <i>original_type_name</i> . A convenience function for <code>uvm_factory::set_inst_override_by_type</code> , this method registers a factory override for components created at this level of hierarchy or below.
<code>print_override_info</code>	This factory debug method performs the same lookup process as <code>create_object</code> and <code>create_component</code> , but instead of creating an object, it prints information about what type of object would be created given the provided arguments.
HIERARCHICAL REPORTING INTERFACE	This interface provides versions of the <code>set_report_*</code> methods in the <code>uvm_report_object</code> base class that are applied recursively to this component and all its children.
<code>set_report_severity_action_hier</code> <code>set_report_id_action_hier</code> <code>set_report_severity_id_action_hier</code>	These methods recursively associate the specified action with reports of the given <i>severity</i> , <i>id</i> , or <i>severity-id</i> pair.
<code>set_report_default_file_hier</code> <code>set_report_severity_file_hier</code> <code>set_report_id_file_hier</code> <code>set_report_severity_id_file_hier</code>	These methods recursively associate the specified FILE descriptor with reports of the given <i>severity</i> , <i>id</i> , or <i>severity-id</i> pair.
<code>set_report_verbosity_level_hier</code>	This method recursively sets the maximum verbosity level for reports for this component and all those below it.
RECORDING INTERFACE	These methods comprise the component-based transaction recording interface.
<code>accept_tr</code>	This function marks the acceptance of a transaction, <i>tr</i> , by this component.
<code>do_accept_tr</code>	The <code>accept_tr</code> method calls this function to accommodate any user-defined post-accept action.
<code>begin_tr</code>	This function marks the start of a transaction, <i>tr</i> , by this component.
<code>begin_child_tr</code>	This function marks the start of a child transaction, <i>tr</i> , by this component.
<code>do_begin_tr</code>	The <code>begin_tr</code> and <code>begin_child_tr</code> methods call this function to accommodate any user-defined post-begin action.
<code>end_tr</code>	This function marks the end of a transaction, <i>tr</i> , by this component.
<code>do_end_tr</code>	The <code>end_tr</code> method calls this function to accommodate any user-defined post-end action.
<code>record_error_tr</code>	This function marks an error transaction by a component.
<code>record_event_tr</code>	This function marks an event transaction by a component.
<code>print_enabled</code>	This bit determines if this component should automatically be printed as a child of its parent object.

new

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

Creates a new component with the given leaf instance *name* and handle to its *parent*. If the component is a top-level component (i.e. it is created in a static module or interface), *parent* should be null.

The component will be inserted as a child of the *parent* object, if any. If *parent* already has a child by the given *name*, an error is produced.

If *parent* is null, then the component will become a child of the implicit top-level component, *uvm_top*.

All classes derived from *uvm_component* must call `super.new(name,parent)`.

HIERARCHY INTERFACE

These methods provide user access to information about the component hierarchy, i.e., topology.

[get_parent](#)

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

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

[get_full_name](#)

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

Returns the full hierarchical name of this object. The default implementation concatenates the hierarchical name of the parent, if any, with the leaf name of this object, as given by [uvm_object::get_name](#).

[get_child](#)

```
function uvm_component get_child (string name)
```

[get_next_child](#)

```
function int get_next_child (ref string name)
```

[get_first_child](#)

```
function int get_first_child (ref string name)
```

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

```
string name;
uvm_component child;
if (comp.get_first_child(name))
  do begin
    child = comp.get_child(name);
```

```
    child.print();
end while (comp.get_next_child(name));
```

get_num_children

```
function int get_num_children ()
```

Returns the number of this component's children.

has_child

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

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

set_name

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

Renames this component to *name* and recalculates all descendants' full names.

lookup

```
function uvm_component lookup (string name)
```

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

PHASING INTERFACE

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

<i>new</i>	Also known as the <i>constructor</i> , the component does basic initialization of any members not subject to configuration.
<i>build</i>	The component constructs its children. It uses the <code>get_config</code> interface to obtain any configuration for itself, the <code>set_config</code> interface to set any configuration for its own children, and the factory interface for actually creating the children and other objects it might need.
<i>connect</i>	The component now makes connections (binds TLM ports and exports) from child-to-child or from child-to-self (i.e. to promote a child port or export up the hierarchy for external access. Afterward, all connections are checked via <code>resolve_bindings</code> before entering the <code>end_of_elaboration</code> phase.

<i>end_of_elaboration</i>	At this point, the entire testbench environment has been built and connected. No new components and connections may be created from this point forward. Components can do final checks for proper connectivity, and it can initiate communication with other tools that require stable, quasi-static component structure..
<i>start_of_simulation</i>	The simulation is about to begin, and this phase can be used to perform any pre-run activity such as displaying banners, printing final testbench topology and configuration information.
<i>run</i>	This is where verification takes place. It is the only predefined, time-consuming phase. A component's primary function is implemented in the run task. Other processes may be forked if desired. When a component returns from its run task, it does not signify completion of its run phase. Any processes that it may have forked <i>continue to run</i> . The run phase terminates in one of four ways:
<i>stop</i>	When a component's enable_stop_interrupt bit is set and global_stop_request is called, the component's stop task is called. Components can implement stop to allow completion of in-progress transactions, <flush> queues, etc. Upon return from stop() by all enabled components, a do_kill_all is issued. If the uvm_test_done_objection is being used, this stopping procedure is deferred until all outstanding objections on uvm_test_done have been dropped.
<i>objections dropped</i>	The uvm_test_done_objection will implicitly call global_stop_request when all objections to ending the phase are dropped. The stop procedure described above is then allowed to proceed normally.
<i>kill</i>	When called, all component's run processes are killed immediately. While kill can be called directly, it is recommended that components use the stopping mechanism, which affords a more ordered and safe shutdown.
<i>timeout</i>	If a timeout was set, then the phase ends if it expires before either of the above occur. Without a stop, kill, or timeout, simulation can continue "forever", or the simulator may end simulation prematurely if it determines that all processes are waiting.
<i>extract</i>	This phase can be used to extract simulation results from coverage collectors and scoreboards, collect status/error counts, statistics, and other information from components in bottom-up order. Being a separate phase, extract ensures all relevant data from potentially independent sources (i.e. other components) are collected before being checked in the next phase.
<i>check</i>	Having extracted vital simulation results in the previous phase, the check phase can be used to validate such data and determine the overall simulation outcome. It too executes bottom-up.
<i>report</i>	Finally, the report phase is used to output results to files and/or the screen.

All task-based phases ([run](#) is the only pre-defined task phase) will run forever until killed or stopped via [kill](#) or [global_stop_request](#). The latter causes each component's [stop](#) task

to get called back if its `enable_stop_interrupt` bit is set. After all components' stop tasks return, the UVM will end the phase.

build

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

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

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

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

The build phase of the `uvm_component` class executes the automatic configuration of fields registered in the component by calling `apply_config_settings`. To turn off automatic configuration for a component, do not call `super.build()` in the subtype's build method.

See [uvm_phase](#) for more information on phases.

connect

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

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

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

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

This method should never be called directly.

See [uvm_phase](#) for more information on phases.

end_of_elaboration

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

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

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

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

This method should never be called directly.

See [uvm_phase](#) for more information on phases.

[start_of_simulation](#)

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

The [start_of_simulation](#) phase callback is one of several methods automatically called during the course of simulation.

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

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

This method should never be called directly.

See [uvm_phase](#) for more information on phases.

[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 components' [run](#) tasks are forked as independent processes. Returning from its [run](#) task does not signify completion of a component's [run](#) phase; any processes forked by [run](#) continue to run.

The [run](#) phase terminates in one of four ways.

- 1 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). If the [uvm_test_done](#) objection> is being used, this stopping procedure is deferred until all outstanding objections on [uvm_test_done](#) have been dropped.
- 2 all objections to [uvm_test_done](#) have been dropped - When all objections on the [uvm_test_done](#) objection have been dropped, [global_stop_request](#) is called automatically, thus kicking off the stopping procedure described above. See [uvm_objection](#) for details on using the objection mechanism.
- 3 explicit call to [kill](#) or [do_kill_all](#) - When [kill](#) is called, this component's [run](#) processes are killed immediately. The [do_kill_all](#) methods applies to this component and all its descendants. Use of this method is not recommended. It is better to use the stopping mechanism, which affords a more ordered, safer shut-down.
- 4 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](#).

This run task should never be called directly.

See [uvm_phase](#) for more information on phases.

extract

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

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

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

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

This method should never be called directly.

See [uvm_phase](#) for more information on phases.

check

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

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

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

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

This method should never be called directly.

See [uvm_phase](#) for more information on phases.

report

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

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

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

executed before their parents.

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

This method should never be called directly.

See [uvm_phase](#) for more information on phases.

suspend

```
virtual task suspend ()
```

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

resume

```
virtual task resume ()
```

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

status

```
function string status ()
```

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

kill

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

Kills the process tree associated with this component's currently running task-based phase, e.g., [run](#).

An alternative mechanism for stopping the [run](#) phase is the stop request. Calling [global_stop_request](#) causes all components' run processes to be killed, but only after all components have had the opportunity to complete in progress transactions and shutdown cleanly via their [stop](#) tasks.

do_kill_all

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

Recursively calls [kill](#) on this component and all its descendants, which abruptly ends the currently running task-based phase, e.g., [run](#). See [run](#) for better options to ending a task-based phase.

stop

```
virtual task stop (string ph_name)
```

The stop task is called when this component's [enable_stop_interrupt](#) bit is set and [global_stop_request](#) is called during a task-based phase, e.g., [run](#).

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

The stop method will not be called if [enable_stop_interrupt](#) is 0.

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

This method should never be called directly.

[enable_stop_interrupt](#)

```
protected int enable_stop_interrupt = 0
```

This bit allows a component to raise an objection to the stopping of the current phase. It affects only time consuming phases (such as the [run](#) phase).

When this bit is set, the [stop](#) task in the component is called as a result of a call to [global_stop_request](#). Components that are sensitive to an immediate killing of its run-time processes should set this bit and implement the stop task to prepare for shutdown.

[resolve_bindings](#)

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

Processes all port, export, and imp connections. Checks whether each port's min and max connection requirements are met.

It is called just before the [end_of_elaboration](#) phase.

Users should not call directly.

CONFIGURATION INTERFACE

Components can be designed to be user-configurable in terms of its topology (the type and number of children it has), mode of operation, and run-time parameters (knobs). The configuration interface accommodates this common need, allowing component composition and state to be modified without having to derive new classes or new class hierarchies for every configuration scenario.

[set_config_int](#)

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

[set_config_string](#)

```
virtual function void set_config_string (string inst_name,  
                                      string field_name,  
                                      string value )
```

[set_config_object](#)

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

Calling `set_config_*` causes configuration settings to be created and placed in a table internal to this component. There are similar global methods that store settings in a global table. Each setting stores the supplied `inst_name`, `field_name`, and `value` for later use by descendant components during their construction. (The global table applies to all components and takes precedence over the component tables.)

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

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

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

- For `set_config_int`, `value` is an integral value that can be anything from 1 bit to 4096 bits.
- For `set_config_string`, `value` is a string.
- For `set_config_object`, `value` must be an `uvm_object`-based object or null. Its `clone` argument specifies whether the object should be cloned. If set, the object is cloned both going into the table (during the set) and coming out of the table (during the get), so that multiple components matched to the same setting (by way of wildcards) do not end up sharing the same object.

The following message tags are used for configuration setting. You can use the standard `uvm_report_message` interface to control these messages. `CFGNTS` -- The configuration setting was not used by any component. This is a warning. `CGOVR` -- The configuration setting was overridden by a setting above. `CFGSET` -- The configuration setting was used at least once.

See [get_config_int](#), [get_config_string](#), and [get_config_object](#) for information on getting the configurations set by these methods.

[get_config_int](#)

```
virtual function bit get_config_int (string field_name,  
                                   inout uvm_bitstream_t value )
```

[get_config_string](#)

```
virtual function bit get_config_string (string field_name,  
                                       inout string value )
```

[get_config_object](#)

```
virtual function bit get_config_object ( string field_name,
                                         inout uvm_object value,
                                         input bit clone = 1)
```

These methods retrieve configuration settings made by previous calls to their `set_config_*` counterparts. As the methods' names suggest, there is direct support for integral types, strings, and objects. Settings of other types can be indirectly supported by defining an object to contain them.

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

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

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

For example, the following code illustrates how a component designer might call upon the configuration mechanism to assign its `data` object property, whose type `myobj_t` derives from `uvm_object`.

```
class mycomponent extends uvm_component;
    local myobj_t data;

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

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

The `clone` bit clones the data inbound. The `get_config_object` method can also clone the data outbound.

See [Members](#) for information on setting the global configuration table.

[check_config_usage](#)

```
function void check_config_usage (bit recurse = 1)
```

Check all configuration settings in a components configuration table to determine if the setting has been used, overridden or not used. When `recurse` is 1 (default), configuration for this and all child components are recursively checked. This function is automatically called in the check phase, but can be manually called at any time.

Additional detail is provided by the following message tags

- CFGOVR -- lists all configuration settings that have been overridden from above.
- CFGSET -- lists all configuration settings that have been set.

To get all configuration information prior to the run phase, do something like this in your top object:

```
function void start_of_simulation();
    set_report_id_action_hier(CFGOVR, UVM_DISPLAY);
    set_report_id_action_hier(CFGSET, UVM_DISPLAY);
    check_config_usage();
endfunction
```

apply_config_settings

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

Searches for all config settings matching this component's instance path. For each match, the appropriate `set_*_local` method is called using the matching config setting's `field_name` and value. Provided the `set_*_local` method is implemented, the component property associated with the `field_name` is assigned the given value.

This function is called by `uvm_component::build`.

The `apply_config_settings` method determines all the configuration settings targeting this component and calls the appropriate `set_*_local` method to set each one. To work, you must override one or more `set_*_local` methods to accommodate setting of your component's specific properties. Any properties registered with the optional ``uvm_*_field` macros do not require special handling by the `set_*_local` methods; the macros provide the `set_*_local` functionality for you.

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

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

print_config_settings

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

Called without arguments, `print_config_settings` prints all configuration information for this component, as set by previous calls to `set_config_*`. The settings are printing in the order of their precedence.

If `field` is specified and non-empty, then only configuration settings matching that field, if any, are printed. The field may not contain wildcards.

If `comp` is specified and non-null, then the configuration for that component is printed.

If `recurse` is set, then configuration information for all `comp`'s children and below are printed as well.

[print_config_matches](#)

```
static bit print_config_matches = 0
```

Setting this static variable causes `get_config_*` to print info about matching configuration settings as they are being applied.

[OBJECTION INTERFACE](#)

These methods provide object level hooks into the [uvm_objection](#) mechanism.

[raised](#)

```
virtual function void raised (uvm_objection objection,
                             uvm_object    source_obj,
                             string        description,
                             int          count      )
```

The `raised` callback is called when a descendant of the component instance raises the specified `objection`. The `source_obj` is the object which originally raised the object. `count` is an optional count that was used to indicate a number of objections which were raised.

[dropped](#)

```
virtual function void dropped (uvm_objection objection,
                               uvm_object    source_obj,
                               string        description,
                               int          count      )
```

The `dropped` callback is called when a descendant of the component instance raises the specified `objection`. The `source_obj` is the object which originally dropped the object. `count` is an optional count that was used to indicate a number of objections which were dropped.

[all_dropped](#)

```
virtual task all_dropped (uvm_objection objection,
                          uvm_object    source_obj,
                          string        description,
                          int          count      )
```

The `all_dropped` callback is called when a descendant of the component instance raises the specified `objection`. The `source_obj` is the object which originally all_dropped the object. `count` is an optional count that was used to indicate a number of objections which were dropped. This callback is time-consuming and the `all_dropped` conditional will not be propagated up to the object's parent until the callback returns.

[FACTORY INTERFACE](#)

The factory interface provides convenient access to a portion of UVM's [uvm_factory](#) interface. For creating new objects and components, the preferred method of accessing the factory is via the object or component wrapper (see [uvm_component_registry](#))

`#(T,Tname)` and `uvm_object_registry #(T,Tname)`). The wrapper also provides functions for setting type and instance overrides.

[create_component](#)

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

A convenience function for `uvm_factory::create_component_by_name`, this method calls upon the factory to create a new child component whose type corresponds to the preregistered type name, `requested_type_name`, and instance name, `name`. This method is equivalent to:

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

If the factory determines that a type or instance override exists, the type of the component created may be different than the requested type. See [set_type_override](#) and [set_inst_override](#). See also [uvm_factory](#) for details on factory operation.

[create_object](#)

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

A convenience function for `uvm_factory::create_object_by_name`, this method calls upon the factory to create a new object whose type corresponds to the preregistered type name, `requested_type_name`, and instance name, `name`. This method is equivalent to:

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

If the factory determines that a type or instance override exists, the type of the object created may be different than the requested type. See [uvm_factory](#) for details on factory operation.

[set_type_override_by_type](#)

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

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

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

The `relative_inst_path` is relative to this component and may include wildcards. The

original_type represents the type that is being overridden. In subsequent calls to [uvm_factory::create_object_by_type](#) or [uvm_factory::create_component_by_type](#), if the *requested_type* matches the *original_type* and the instance paths match, the factory will produce the *override_type*.

The original and override type arguments are lightweight proxies to the types they represent. See [set_inst_override_by_type](#) for information on usage.

[set_inst_override_by_type](#)

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

A convenience function for [uvm_factory::set_inst_override_by_type](#), this method registers a factory override for components and objects created at this level of hierarchy or below. In typical usage, this method is equivalent to:

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

The *relative_inst_path* is relative to this component and may include wildcards. The *original_type* represents the type that is being overridden. In subsequent calls to [uvm_factory::create_object_by_type](#) or [uvm_factory::create_component_by_type](#), if the *requested_type* matches the *original_type* and the instance paths match, the factory will produce the *override_type*.

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

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

The following example shows `uvm_*_utils usage

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

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

class block extends uvm_component;
  `uvm_component_utils(block)
  comp c_inst;
  virtual function void build();
    set_inst_override_by_type("c_inst", comp::get_type(),
                             mycomp::get_type());
  endfunction
  ...
endclass
```

[set_type_override](#)

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

A convenience function for [uvm_factory::set_type_override_by_name](#), this method configures the factory to create an object of type *override_type_name* whenever the factory is asked to produce a type represented by *original_type_name*. This method is equivalent to:

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

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

[set_inst_override](#)

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

A convenience function for [uvm_factory::set_inst_override_by_type](#), this method registers a factory override for components created at this level of hierarchy or below. In typical usage, this method is equivalent to:

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

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

[print_override_info](#)

```
function void print_override_info(string requested_type_name,  
                                 string name = " ")
```

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

HIERARCHICAL REPORTING INTERFACE

This interface provides versions of the `set_report_*` methods in the [uvm_report_object](#) base class that are applied recursively to this component and all its children.

When a report is issued and its associated action has the LOG bit set, the report will be sent to its associated FILE descriptor.

[set_report_severity_action_hier](#)

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

[set_report_id_action_hier](#)

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

[set_report_severity_id_action_hier](#)

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

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

For a list of severities and their default actions, refer to [uvm_report_handler](#).

[set_report_default_file_hier](#)

```
function void set_report_default_file_hier (UVM_FILE file)
```

[set_report_severity_file_hier](#)

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

[set_report_id_file_hier](#)

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

[set_report_severity_id_file_hier](#)

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

These methods recursively associate the specified FILE descriptor with reports of the given *severity*, *id*, or *severity-id* pair. A FILE associated with a particular severity-id pair

takes precedence over a FILE associated with id, which take precedence over an a FILE associated with a severity, which takes precedence over the default FILE descriptor.

For a list of severities and other information related to the report mechanism, refer to [uvm_report_handler](#).

[set_report_verbosity_level_hier](#)

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

This method recursively sets the maximum verbosity level for reports for this component and all those below it. Any report from this component subtree whose verbosity exceeds this maximum will be ignored.

See [uvm_report_handler](#) for a list of predefined message verbosity levels and their meaning.

[RECORDING INTERFACE](#)

These methods comprise the component-based transaction recording interface. The methods can be used to record the transactions that this component “sees”, i.e. produces or consumes.

The API and implementation are subject to change once a vendor-independent use-model is determined.

[accept_tr](#)

```
function void accept_tr (uvm_transaction tr,  
                        time accept_time = )
```

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

- Calls the *tr*'s [uvm_transaction::accept_tr](#) method, passing to it the *accept_time* argument.
- Calls this component's [do_accept_tr](#) method to allow for any post-begin action in derived classes.
- Triggers the component's internal *accept_tr* event. Any processes waiting on this event will resume in the next delta cycle.

[do_accept_tr](#)

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

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

[begin_tr](#)

```
function integer begin_tr (uvm_transaction tr,  
                           string stream_name = "main",  
                           string label = "",  
                           string desc = "");
```

```
time begin_time = 0 )
```

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

- Calls *tr*'s `uvm_transaction::begin_tr` method, passing to it the *begin_time* argument. The *begin_time* should be greater than or equal to the accept time. By default, when *begin_time* = 0, the current simulation time is used.

If recording is enabled (`recording_detail != UVM_OFF`), then a new database-transaction is started on the component's transaction stream given by the `stream` argument. No transaction properties are recorded at this time.

- Calls the component's `do_begin_tr` method to allow for any post-begin action in derived classes.
- Triggers the component's internal `begin_tr` event. Any processes waiting on this event will resume in the next delta cycle.

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

[begin_child_tr](#)

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

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

[do_begin_tr](#)

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

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

[end_tr](#)

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

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

- Calls *tr*'s `uvm_transaction::end_tr` method, passing to it the *end_time* argument. The *end_time* must at least be greater than the begin time. By default, when *end_time* = 0, the current simulation time is used.

The transaction's properties are recorded to the database-transaction on which it was started, and then the transaction is ended. Only those properties handled by the transaction's `do_record` method (and optional `uvm_*_field macros) are recorded.

- Calls the component's `do_end_tr` method to accommodate any post-end action in derived classes.
- Triggers the component's internal `end_tr` event. Any processes waiting on this event will resume in the next delta cycle.

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

[do_end_tr](#)

```
virtual protected function void do_end_tr (uvm_transaction tr,
                                         integer          tr_handle)
```

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

[record_error_tr](#)

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

This function marks an error transaction by a component. Properties of the given `uvm_object`, `info`, as implemented in its `<do_record>` method, are recorded to the transaction database.

An `error_time` of 0 indicates to use the current simulation time. The `keep_active` bit determines if the handle should remain active. If 0, then a zero-length error transaction is recorded. A handle to the database-transaction is returned.

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

[record_event_tr](#)

```
function integer record_event_tr (string      stream_name = "main",
                                  uvm_object  info        = null,
                                  string      label       = "event_tr",
                                  string      desc        = "",
                                  time        event_time = 0,
                                  bit         keep_active = 0
                                         )
```

This function marks an event transaction by a component.

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

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

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

[print_enabled](#)

```
bit print_enabled = 1
```

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

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

uvm_root

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

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

Implicit top-level

The *uvm_top* serves as an implicit top-level component. Any component whose parent is specified as NULL becomes a child of *uvm_top*. Thus, all UVM components in simulation are descendants of *uvm_top*.

Phase control

uvm_top manages the phasing for all components. There are eight phases predefined in every component: build, connect, end_of_elaboration, start_of_simulation, run, extract, check, and report. Of these, only the run phase is a task. All others are functions. UVM's flexible phasing mechanism allows users to insert any number of custom function and task-based phases. See [run_test](#), [insert_phase](#), and [stop_request](#), and others.

Search

Use *uvm_top* to search for components based on their hierarchical name. See [find](#) and [find_all](#).

Report configuration

Use *uvm_top* to globally configure report verbosity, log files, and actions. For example, *uvm_top.set_report_verbosity_level_hier(UVM_FULL)* would set full verbosity for all components in simulation.

Global reporter

Because *uvm_top* is globally accessible (in uvm_pkg scope), UVM's reporting mechanism is accessible from anywhere outside *uvm_component*, such as in modules and sequences. See [uvm_report_error](#), [uvm_report_warning](#), and other global methods.

Summary

uvm_root

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

METHODS

[run_test](#) Phases all components through all registered phases.

[stop_request](#) Calling this function triggers the process of shutting down the currently running task-based phase.

[in_stop_request](#) This function returns 1 if a stop request is currently active, and 0 otherwise.

[insert_phase](#) Inserts a new phase given by *new_phase* *after* the existing phase given by *exist_phase*.

[find](#)
[find_all](#) Returns the component handle (*find*) or list of components handles (*find_all*) matching a given string.

[get_current_phase](#) Returns the handle of the currently executing phase.

[get_phase_by_name](#) Returns the handle of the phase having the given name.

VARIABLES

`phase_timeout`
`stop_timeout`

These set watchdog timers for task-based phases and stop tasks.

`enable_print_topology`

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

`finish_on_completion`

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

`uvm_top`

This is the top-level that governs phase execution and provides component search interface.

METHODS

`raised`
`all_dropped`

METHODS

run_test

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

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

stop_request

```
function void stop_request()
```

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

in_stop_request

```
function bit in_stop_request()
```

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

insert_phase

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

Inserts a new phase given by `new_phase` after the existing phase given by `exist_phase`.

The uvm_top maintains a queue of phases executed in consecutive order. If exist_phase is null, then new_phase is inserted at the head of the queue, i.e., it becomes the first phase.

find

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

find_all

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

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

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

get_current_phase

```
function uvm_phase get_current_phase ()
```

Returns the handle of the currently executing phase.

get_phase_by_name

```
function uvm_phase get_phase_by_name (string name)
```

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

VARIABLES

phase_timeout

```
time phase_timeout = 0
```

stop_timeout

```
time stop_timeout = 0
```

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

enable_print_topology

```
bit enable_print_topology = 0
```

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

finish_on_completion

```
bit finish_on_completion = 1
```

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

uvm_top

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

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

METHODS

raised

```
function void uvm_root::raised (uvm_objection objection,
                                uvm_object     source_obj,
                                string         description,
                                int            count      )
```

all_dropped

```
task uvm_root::all_dropped (uvm_objection objection,
                            uvm_object     source_obj,
                            string         description,
                            int            count      )
```

uvm_phase

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

Summary

uvm_phase

The uvm_phase class is used for defining phases for uvm_component and its subclasses.

CLASS DECLARATION

```
virtual class uvm_phase
```

METHODS

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

METHODS

new

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

Creates a phase object.

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

get_name

```
function string get_name ()
```

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

is_task

```
function bit is_task ()
```

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

is_top_down

```
function bit is_top_down ()
```

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

get_type_name

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

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

wait_start

```
task wait_start ()
```

Waits until the phase has been started.

wait_done

```
task wait_done ()
```

Waits until the phase has been completed.

is_in_progress

```
function bit is_in_progress ()
```

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

is_done

```
function bit is_done ()
```

Returns 1 if the phase has completed, 0 otherwise.

reset

```
function void reset ()
```

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

call_task

```
virtual task call_task (uvm_component parent)
```

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

call_func

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

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

Usage

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

The UVM provides the following predefined phases for all `uvm_components`.

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

A phase is defined by an instance of an `uvm_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 UVM phase controller, `uvm_top`.

Inheriting from the *uvm_phase* Class

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

1. Define a new phase class, which must extend *uvm_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:

```
`uvm_phase_func_topdown_decl( preload )
```

This convenient phase declaration macro is described below.

2. Create a single instance of the phase in a convenient placein 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 *uvm_top*.

```
class my_memory extends uvm_component;
  function new(string name, uvm_component parent);
    super.new(name,parent);
    uvm_top.insert_phase(preload_ph, start_of_simulation_ph);
  endfunction
  virtual function void preload(); // our new phase
  endfunction
endclass
```

Phase Macros (Optional)

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

Summary

Usage

Phases are a synchronizing mechanism for the environment.

MACROS

``uvm_phase_func_decl`

``uvm_phase_func_decl (PHASE_NAME,
TOP_DOWN)`

``uvm_phase_task_decl`

``uvm_phase_func_topdown_decl`

``uvm_phase_func_bottomup_decl`

``uvm_phase_task_topdown_decl`

``uvm_phase_task_bottomup_decl`

These alternative macros have a single phase name argument.

MACROS

[`uvm_phase_func_decl](#)

```
`uvm_phase_func_decl (PHASE_NAME, TOP_DOWN)
```

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

The above macro creates the following class definition.

```
class PHASE_NAME``_phase #(type PARENT=int) extends uvm_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(uvm_component parent);
    assert($cast(m_parent,parent));
    call_func();
  endtask
endclass
```

[`uvm_phase_task_decl](#)

```
`uvm_phase_task_decl (PHASE_NAME, TOP_DOWN)
```

The above macro creates the following class definition.

```
class PHASE_NAME``_phase #(type PARENT=int) extends uvm_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(uvm_component parent);
    assert($cast(m_parent,parent));
    call_task();
  endtask
endclass
```

[`uvm_phase_func_topdown_decl](#)

[`uvm_phase_func_bottomup_decl](#)

[`uvm_phase_task_topdown_decl](#)

[`uvm_phase_task_bottomup_decl](#)

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

```
'define uvm_phase_func_topdown_decl  `uvm_phase_func_decl (PHASE_NAME,1)
`define uvm_phase_func_bottomup_decl `uvm_phase_func_decl (PHASE_NAME,0)
`define uvm_phase_task_topdown_decl  `uvm_phase_task_decl (PHASE_NAME,1)
`define uvm_phase_task_bottomup_decl `uvm_phase_task_decl (PHASE_NAME,0)
```

uvm_port_base #(IF)

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

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

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

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

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

uvm_port_base possesses the properties of components in that they have a hierarchical instance path and parent. Because SystemVerilog does not support multiple inheritance, uvm_port_base can not extend both the interface it implements and [uvm_component](#). Thus, uvm_port_base contains a local instance of [uvm_component](#), to which it delegates such commands as get_name, get_full_name, and get_parent.

Summary

uvm_port_base #(IF)

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

CLASS HIERARCHY



CLASS DECLARATION

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

METHODS

[new](#)

The first two arguments are the normal [uvm_component](#) constructor arguments.

[get_name](#)

Returns the leaf name of this port.

[get_full_name](#)

Returns the full hierarchical name of this port.

[get_parent](#)

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

[get_comp](#)

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

[get_type_name](#)

Returns the type name to this port.

[min_size](#)

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

[max_size](#)

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

<code>is_unbounded</code>	Returns 1 if this port has no maximum on the number of implementation (imp) ports this port can connect to.
<code>is_port</code> <code>is_export</code> <code>is_imp</code>	Returns 1 if this port is of the type given by the method name, 0 otherwise.
<code>size</code>	Gets the number of implementation ports connected to this port.
<code>set_default_index</code>	Sets the default implementation port to use when calling an interface method.
<code>connect</code> <code>debug_connected_to</code>	Connects this port to the given <i>provider</i> port. The <code>debug_connected_to</code> method outputs a visual text display of the port/export/imp network to which this port connects (i.e., the port's fanout).
<code>debug_provided_to</code>	The <code>debug_provided_to</code> method outputs a visual display of the port/export network that ultimately connect to this port (i.e., the port's fanin).
<code>resolve_bindings</code>	This callback is called just before entering the <code>end_of_elaboration</code> phase.
<code>get_if</code>	Returns the implementation (imp) port at the given index from the array of imps this port is connected to.

METHODS

new

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

The first two arguments are the normal `uvm_component` constructor arguments.

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

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

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

get_name

```
function string get_name()
```

Returns the leaf name of this port.

get_full_name

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

Returns the full hierarchical name of this port.

get_parent

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

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

get_comp

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

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

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

get_type_name

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

Returns the type name to this port. Derived port classes must implement this method to return the concrete type. Otherwise, only a generic "uvm_port", "uvm_export" or "uvm_implementation" is returned.

min_size

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

max_size

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

is_unbounded

```
function bit is_unbounded ()
```

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

is_port

```
function bit is_port ()
```

[is_export](#)

```
function bit is_export ()
```

[is_imp](#)

```
function bit is_imp ()
```

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

[size](#)

```
function int size ()
```

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

[set_default_index](#)

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

Sets the default implementation port to use when calling an interface method. This method should only be called on `UVM_EXPORT` types. The value must not be set before the `end_of_elaboration` phase, when port connections have not yet been resolved.

[connect](#)

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

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

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

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

Relationships, when enabled, are checked are as follows

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

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

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

[debug_connected_to](#)

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

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

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

[debug_provided_to](#)

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

The `debug_provided_to` method outputs a visual display of the port/export network that ultimately connect to this port (i.e., the port's fanin).

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

[resolve_bindings](#)

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

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

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

[get_if](#)

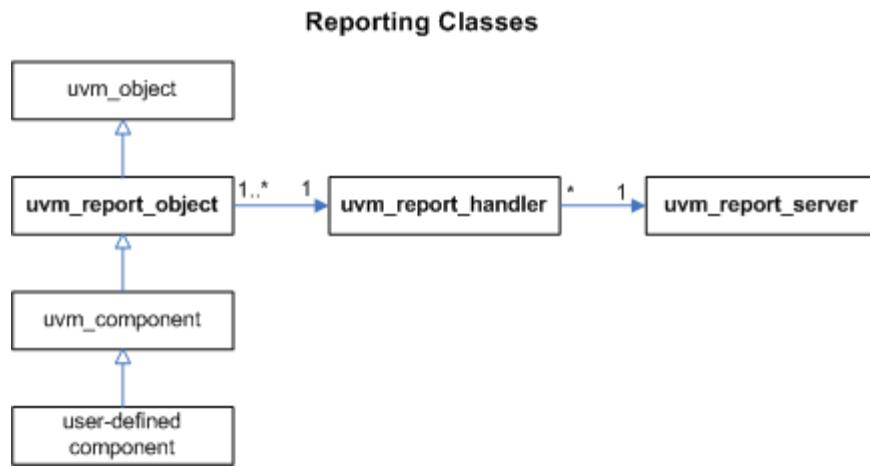
```
function uvm_port_base #(IF) get_if(int index=0)
```

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

REPORTING CLASSES

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

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



uvm_report_object

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

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

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

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

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

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

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

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

Summary

uvm_report_object

The uvm_report_object provides an interface to the UVM reporting facility.

CLASS HIERARCHY

uvm_void

uvm_object

uvm_report_object

CLASS DECLARATION

```
class uvm_report_object extends uvm_object
```

`new`

Creates a new report object with the given name.

REPORTING

`uvm_report_info`
`uvm_report_warning`
`uvm_report_error`
`uvm_report_fatal`

These are the primary reporting methods in the UVM.

CALLBACKS

`report_info_hook`
`report_error_hook`
`report_warning_hook`
`report_fatal_hook`
`report_hook`

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

`report_header`
`report_summarize`

Prints version and copyright information.
Outputs statistical information on the reports issued by the central report server.

`die`

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

CONFIGURATION

`set_report_verbosity_level`

This method sets the maximum verbosity level for reports for this component.

`set_report_severity_action`
`set_report_id_action`
`set_report_severity_id_action`

These methods associate the specified action or actions with reports of the given *severity*, *id*, or *severity-id* pair.

`set_report_default_file`
`set_report_severity_file`
`set_report_id_file`
`set_report_severity_id_file`

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

`get_report_verbosity_level`

Gets the verbosity level in effect for this object.

`get_report_action`

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

`get_report_file_handle`

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

`uvm_report_enabled`

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

`set_report_max_quit_count`

Sets the maximum quit count in the report handler to *max_count*.

SETUP

`set_report_handler`

Sets the report handler, overwriting the default instance.

`get_report_handler`

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

`reset_report_handler`

Resets the underlying report handler to its default settings.

`get_report_server`

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

`dump_report_state`

This method dumps the internal state of the report handler.

new

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

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

REPORTING

[uvm_report_info](#)

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

[uvm_report_warning](#)

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

[uvm_report_error](#)

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

[uvm_report_fatal](#)

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

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

- | | |
|------------------|--|
| <i>id</i> | a unique id for the report or report group that can be used for identification and therefore targeted filtering. You can configure an individual report's actions and output file(s) using this id string. |
| <i>message</i> | the message body, preformatted if necessary to a single string. |
| <i>verbosity</i> | the verbosity of the message, indicating its relative importance. If this number is less than or equal to the effective verbosity level (see <code><set_report_verbosity_level></code>), |

then the report is issued, subject to the configured action and file descriptor settings. Verbosity is ignored for warnings, errors, and fatals to ensure users do not inadvertently filter them out. It remains in the methods for backward compatibility.

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

CALLBACKS

[report_info_hook](#)

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

[report_error_hook](#)

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

[report_warning_hook](#)

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

[report_fatal_hook](#)

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

[report_hook](#)

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

These hook methods can be defined in derived classes to perform additional actions when reports are issued. They are called only if the UVM_CALL_HOOK bit is specified in the action associated with the report. The default implementations return 1, which allows

the report to be processed. If an override returns 0, then the report is not processed.

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

[report_header](#)

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

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

[report_summarize](#)

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

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

The `run_test` method in `uvm_top` calls this method.

[die](#)

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

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

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

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

CONFIGURATION

[set_report_verbosity_level](#)

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

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

[set_report_severity_action](#)

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

[set_report_id_action](#)

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

[set_report_severity_id_action](#)

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

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

The *action* argument can take the value [UVM_NO_ACTION](#), or it can be a bitwise OR of any combination of [UVM_DISPLAY](#), [UVM_LOG](#), [UVM_COUNT](#), <UVM_STOP>, [UVM_EXIT](#), and [UVM_CALL_HOOK](#).

[set_report_default_file](#)

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

[set_report_severity_file](#)

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

[set_report_id_file](#)

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

[set_report_severity_id_file](#)

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

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

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

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

[get_report_verbosity_level](#)

```
function int get_report_verbosity_level()
```

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

[get_report_action](#)

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

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

[get_report_file_handle](#)

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

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

[uvm_report_enabled](#)

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

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

See also [get_report_verbosity_level](#) and [get_report_action](#), and the global version of [uvm_report_enabled](#).

[set_report_max_quit_count](#)

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

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

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

SETUP

[set_report_handler](#)

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

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

[get_report_handler](#)

```
function uvm_report_handler get_report_handler()
```

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

[reset_report_handler](#)

```
function void reset_report_handler
```

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

[get_report_server](#)

```
function uvm_report_server get_report_server()
```

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

[dump_report_state](#)

```
function void dump_report_state()
```

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

uvm_report_handler

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

The report handler is not intended for direct use. See [uvm_report_object](#) for information on the UVM reporting mechanism.

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

The relationship between uvm_report_handler and [uvm_report_server](#) is many to one.

Summary

uvm_report_handler

The uvm_report_handler is the class to which most methods in [uvm_report_object](#) delegate.

CLASS DECLARATION

```
class uvm_report_handler
```

METHODS

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

METHODS

[new](#)

```
function new()
```

Creates and initializes a new uvm_report_handler object.

[run_hooks](#)

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

```

    string      message,
    int        verbosity,
    string     filename,
    int        line
)

```

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

[get_verbosity_level](#)

```
function int get_verbosity_level()
```

Returns the configured maximum verbosity level.

[get_action](#)

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

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

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

[get_file_handle](#)

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

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

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

[report](#)

```

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

```

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

[format_action](#)

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

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

uvm_report_server

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

Summary

uvm_report_server

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

VARIABLES

`id_count`

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

METHODS

`new`

Creates the central report server, if not already created.

`set_max_quit_count`
`get_max_quit_count`

Get or set the maximum number of COUNT actions that can be tolerated before an UVM_EXIT action is taken.

`set_quit_count`
`get_quit_count`
`incr_quit_count`
`reset_quit_count`

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

`is_quit_count_reached`

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

`set_severity_count`
`get_severity_count`
`incr_severity_count`
`reset_severity_counts`

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

`set_id_count`
`get_id_count`
`incr_id_count`

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

`process_report`

Calls `compose_message` to construct the actual message to be output.

`compose_message`

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

`summarize`

See `uvm_report_object::report_summarize` method.

`dump_server_state`
`get_server`

Dumps server state information.
Returns a handle to the central report server.

VARIABLES

id_count

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

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

METHODS

new

```
function new()
```

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

set_max_quit_count

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

get_max_quit_count

```
function int get_max_quit_count()
```

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

set_quit_count

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

get_quit_count

```
function int get_quit_count()
```

incr_quit_count

```
function void incr_quit_count()
```

reset_quit_count

```
function void reset_quit_count()
```

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

is_quit_count_reached

```
function bit is_quit_count_reached()
```

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

set_severity_count

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

get_severity_count

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

incr_severity_count

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

reset_severity_counts

```
function void reset_severity_counts()
```

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

set_id_count

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

get_id_count

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

incr_id_count

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

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

process_report

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

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

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

[compose_message](#)

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

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

Expert users can overload this method to customize report formatting.

[summarize](#)

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

See [uvm_report_object::report_summarize](#) method.

[dump_server_state](#)

```
function void dump_server_state()
```

Dumps server state information.

[get_server](#)

```
function uvm_report_server get_server()
```

Returns a handle to the central report server.

uvm_report_catcher

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

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

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

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

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

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

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

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

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

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

Summary

[uvm_report_catcher](#)

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

CLASS DECLARATION

```
typedef class uvm_report_catcher
```

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

new

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

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

CURRENT MESSAGE STATE

get_client

```
function uvm_report_object get_client()
```

Returns the [uvm_report_object](#) that has generated the message that is currently being processed.

[get_severity](#)

```
function uvm_severity get_severity()
```

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

[get_verbosity](#)

```
function int get_verbosity()
```

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

[get_id](#)

```
function string get_id()
```

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

[get_message](#)

```
function string get_message()
```

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

[get_action](#)

```
function uvm_action get_action()
```

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

[get_fname](#)

```
function string get_fname()
```

Returns the file name of the message.

[get_line](#)

```
function int get_line()
```

Returns the line number of the message.

CHANGE MESSAGE STATE

[set_verbosity](#)

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

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

[set_id](#)

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

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

[set_message](#)

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

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

[set_action](#)

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

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

DEBUG

[get_report_catcher](#)

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

Returns the first report catcher that has *name*.

[print_catcher](#)

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

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

CALLBACK INTERFACE

catch

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

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

REPORTING

uvm_report_fatal

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

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

uvm_report_error

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

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

uvm_report_warning

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

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

uvm_report_info

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

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

issue

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

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

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

summarize_report_catcher

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

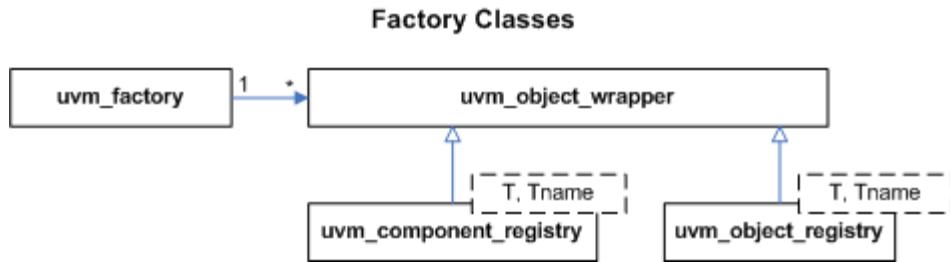
This function is called automatically by [uvm_report_server::summarize\(\)](#). It prints the statistics for the active catchers.

Factory Classes

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

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

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



uvm_component_registry #(T,Tname)

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

See Usage section below for information on using uvm_component_registry.

Summary

uvm_component_registry #(T,Tname)

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

CLASS HIERARCHY

uvm_object_wrapper

uvm_component_registry#(T,Tname)

CLASS DECLARATION

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

METHODS

create_component	Creates a component of type <i>T</i> having the provided <i>name</i> and <i>parent</i> .
get_type_name	Returns the value given by the string parameter, <i>Tname</i> .
get	Returns the singleton instance of this type.
create	Returns an instance of the component type, <i>T</i> , represented by this proxy, subject to any factory overrides based on the context provided by the <i>parent's</i> full name.
set_type_override	Configures the factory to create an object of the type represented by <i>override_type</i> whenever a request is made to create an object of the type, <i>T</i> , represented by this proxy, provided no instance override applies.
set_inst_override	Configures the factory to create a component of the type represented by <i>override_type</i> whenever a request is made to create an object of the type, <i>T</i> , represented by this proxy, with matching instance paths.

METHODS

[create_component](#)

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

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

[get_type_name](#)

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

Returns the value given by the string parameter, *Tname*. This method overrides the method in [uvm_object_wrapper](#).

[get](#)

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

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

[create](#)

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

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

[set_type_override](#)

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

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

[set_inst_override](#)

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

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

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

uvm_object_registry #(T,Tname)

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

See Usage section below for information on using uvm_component_registry.

Summary

uvm_object_registry #(T,Tname)

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

CLASS HIERARCHY



CLASS DECLARATION

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

[create_object](#) Creates an object of type *T* and returns it as a handle to an [uvm_object](#).

[get_type_name](#) Returns the value given by the string parameter, *Tname*.
[get](#) Returns the singleton instance of this type.

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

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

[set_inst_override](#) Configures the factory to create an object of the type represented by *override_type* whenever a request is made to create an object of the type represented by this proxy, with matching instance paths.

USAGE This section describes usage for the uvm_*_registry classes.

create_object

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

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

get_type_name

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

Returns the value given by the string parameter, *Tname*. This method overrides the method in [uvm_object_wrapper](#).

get

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

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

create

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

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

set_type_override

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

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

set_inst_override

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

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

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

USAGE

This section describes usage for the uvm_*_registry classes.

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

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

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

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

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

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

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

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

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

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

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

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

UVM Factory

This page covers the following classes.

- [uvm_factory](#) - creates objects and components according to user-defined type and instance-based overrides.
- [uvm_object_wrapper](#) - a lightweight substitute (proxy) representing a user-defined object or component.

Summary

UVM Factory

This page covers the following classes.

uvm_factory

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

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

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

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

Summary

uvm_factory

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

CLASS DECLARATION

```
class uvm_factory
```

REGISTERING TYPES

```
register
```

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

TYPE & INSTANCE OVERRIDES

```
set_inst_override_by_type  
set_inst_override_by_name
```

Configures the factory to create an object of the override's type whenever a request is

[set_type_override_by_type](#)
[set_type_override_by_name](#)

made to create an object of the original type using a context that matches *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, provided no instance override applies.

CREATION

[create_object_by_type](#)
[create_component_by_type](#)
[create_object_by_name](#)
[create_component_by_name](#)

Creates and returns a component or object of the requested type, which may be specified by type or by name.

DEBUG

[debug_create_by_type](#)
[debug_create_by_name](#)

These methods perform the same search algorithm as the *create_** methods, but they do not create new objects.

[find_override_by_type](#)
[find_override_by_name](#)

These methods return the proxy to the object that would be created given the arguments.

[print](#)

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

USAGE

Using the factory involves three basic operations

REGISTERING TYPES

register

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

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

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

TYPE & INSTANCE OVERRIDES

set_inst_override_by_type

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

[set_inst_override_by_name](#)

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

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

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

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

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

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

[set_type_override_by_type](#)

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

[set_type_override_by_name](#)

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

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

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

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

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

[create_object_by_type](#)

```
function uvm_object create_object_by_type (uvm_object_wrapper requested_type,
                                         string parent_inst_path
                                         string name)
```

[create_component_by_type](#)

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

[create_object_by_name](#)

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

[create_component_by_name](#)

```
function uvm_component create_component_by_name (string requested_type_name,
                                                 string parent_inst_path
                                                 string name,
                                                 uvm_component parent)
```

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

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

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

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

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

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

Override searches are recursively applied, with instance overrides taking precedence over type overrides. If *foo* overrides *bar*, and *xyz* overrides *foo*, then a request for *bar* will

produce xyz. Recursive loops will result in an error, in which case the type returned will be that which formed the loop. Using the previous example, if bar overrides xyz, then bar is returned after the error is issued.

DEBUG

debug_create_by_type

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

debug_create_by_name

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

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

find_override_by_type

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

find_override_by_name

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

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

print

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

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

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

USAGE

Using the factory involves three basic operations

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

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

1 -- Registering objects and component types with the factory

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

Objects that are not parameterized are declared as

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

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

Objects that are parameterized are declared as

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

Components that are not parameterized are declared as

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

Components that are parameterized are declared as

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

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

The `uvm_*_param_utils macros for parameterized classes differ from `uvm_*_utils

classes in the following ways:

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

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

2 -- Designing components that defer creation to the factory

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

```
class driverB #(type T=uvm_object) extends uvm_driver;
  // parameterized classes must use the _param_utils version
  `uvm_component_param_utils(driverB #(T))

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

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

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

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

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

endclass
```

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

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

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

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

```

endclass

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

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

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

endclass

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

```

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

```

class agent extends uvm_agent;
  `uvm_component_utils(agent)
  ...
  B_driver driver0;
  B_driver driver1;

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

  virtual function void build();

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

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

  endfunction

endclass

```

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

```

class env extends uvm_env;
  `uvm_component_utils(env)

  agent agent0;
  agent agent1;

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

  virtual function void build();

    // 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();
    uvm_top.print_topology();
endfunction

virtual task run();
    #100 global_stop_request();
endfunction

endclass

```

3 -- Configuring the factory with type and instance overrides

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

```

module top;
env env0;

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

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

    // ...except for agent0.driver0, whose type remains a base driver.
    //     (Both methods below have the equivalent result.)

    // - via the component's proxy (preferred)
    B_driver::type_id::set_inst_override(B_driver::get_type(),
                                         "env0.agent0.driver0");

    // - via a direct call to a factory method
    factory.set_inst_override_by_type(B_driver::get_type(),
                                       B_driver::get_type(),
                                       {get_full_name(),"env0.agent0.driver0"});

    // now, create the environment; our factory configuration will
    // govern what topology gets created
    env0 = new("env0");

    // run the test (will execute build phase)
    run_test();
end
endmodule

```

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

```

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

```

uvm_object_wrapper

The uvm_object_wrapper provides an abstract interface for creating object and component proxies. Instances of these lightweight proxies, representing every [uvm_object](#)-based and [uvm_component](#)-based object available in the test environment, are registered with the [uvm_factory](#). When the factory is called upon to create an object or component, it finds and delegates the request to the appropriate proxy.

Summary

uvm_object_wrapper

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

CLASS DECLARATION

```
virtual class uvm_object_wrapper
```

METHODS

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

METHODS

[create_object](#)

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

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

[create_component](#)

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

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

[get_type_name](#)

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

Derived classes implement this method to return the type name of the object created by [create_component](#) or [create_object](#). The factory uses this name when matching against the requested type in name-based lookups.

Synchronization Classes



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

- [uvm_event](#) - UVM's event class augments the SystemVerilog event datatype with such services as setting callbacks and data delivery.
- [uvm_barrier](#) - A barrier is used to prevent a pre-configured number of processes from continuing until all have reached a certain point in simulation.
- [uvm_event_pool](#) and [uvm_barrier_pool](#) - The event and barrier pool classes are used to store collections of events and barriers, respectively, all indexed by string name. Each pool class contains a static, "global" pool instance for sharing across all processes.
- [uvm_event_callback](#) - The event callback is used to create callback objects that may be attached to [uvm_events](#).

uvm_event

The uvm_event class is a wrapper class around the SystemVerilog event construct. It provides some additional services such as setting callbacks and maintaining the number of waiters.

Summary

uvm_event

The uvm_event class is a wrapper class around the SystemVerilog event construct.

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_event extends uvm_object
```

METHODS

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

METHODS

new

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

Creates a new event object.

[wait_on](#)

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

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

If the event has already been triggered, this task returns immediately. If *delta* is set, the caller will be forced to wait a single delta #0 before returning. This prevents the caller from returning before previously waiting processes have had a chance to resume.

Once an event has been triggered, it will remain "on" until the event is [reset](#).

[wait_off](#)

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

If the event has already triggered and is "on", this task waits for the event to be turned "off" via a call to [reset](#).

If the event has not already been triggered, this task returns immediately. If *delta* is set, the caller will be forced to wait a single delta #0 before returning. This prevents the caller from returning before previously waiting processes have had a chance to resume.

[wait_trigger](#)

```
virtual task wait_trigger ()
```

Waits for the event to be triggered.

If one process calls [wait_trigger](#) in the same delta as another process calls [trigger](#), a race condition occurs. If the call to [wait](#) occurs before the [trigger](#), this method will return in this delta. If the [wait](#) occurs after the [trigger](#), this method will not return until the next [trigger](#), which may never occur and thus cause deadlock.

[wait_ptrigger](#)

```
virtual task wait_ptrigger ()
```

Waits for a persistent trigger of the event. Unlike [wait_trigger](#), this views the trigger as persistent within a given time-slice and thus avoids certain race conditions. If this method is called after the [trigger](#) but within the same time-slice, the caller returns immediately.

[wait_trigger_data](#)

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

This method calls [wait_trigger](#) followed by [get_trigger_data](#).

[wait_ptrigger_data](#)

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

This method calls [wait_ptrigger](#) followed by [get_trigger_data](#).

[trigger](#)

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

Triggers the event, resuming all waiting processes.

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

[get_trigger_data](#)

```
virtual function uvm_object get_trigger_data ()
```

Gets the data, if any, provided by the last call to [trigger](#).

[get_trigger_time](#)

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

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

[is_on](#)

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

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

A return of 1 indicates that the event has triggered.

[is_off](#)

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

Indicates whether the event has been triggered or been reset.

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

[reset](#)

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

Resets the event to its off state. If *wakeup* is set, then all processes currently waiting for the event are activated before the reset.

No callbacks are called during a reset.

add_callback

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

Registers a callback object, *cb*, with this event. The callback object may include *pre_trigger* and *post_trigger* functionality. If *append* is set to 1, the default, *cb* is added to the back of the callback list. Otherwise, *cb* is placed at the front of the callback list.

delete_callback

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

Unregisters the given callback, *cb*, from this event.

cancel

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

Decrementsthe number of waiters on the event.

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

get_num_waiters

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

Returns the number of processes waiting on the event.

uvm_event_callback

The uvm_event_callback class is an abstract class that is used to create callback objects which may be attached to [uvm_events](#). To use, you derive a new class and override any or both [pre_trigger](#) and [post_trigger](#).

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

Summary

uvm_event_callback

The uvm_event_callback class is an abstract class that is used to create callback objects which may be attached to [uvm_events](#).

CLASS HIERARCHY

uvm_void

uvm_object

uvm_event_callback

CLASS DECLARATION

```
virtual class uvm_event_callback extends uvm_object
```

METHODS

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

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

[post_trigger](#) This callback is called after triggering the associated event.

METHODS

[new](#)

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

Creates a new callback object.

[pre_trigger](#)

```
virtual function bit pre_trigger (uvm_event e,  
                                 uvm_object data = null)
```

This callback is called just before triggering the associated event. In a derived class, override this method to implement any pre-trigger functionality.

If your callback returns 1, then the event will not trigger and the post-trigger callback is not called. This provides a way for a callback to prevent the event from triggering.

In the function, *e* is the [uvm_event](#) that is being triggered, and *data* is the optional data

associated with the event trigger.

post_trigger

```
virtual function void post_trigger (uvm_event e,
                                  uvm_object data = null)
```

This callback is called after triggering the associated event. In a derived class, override this method to implement any post-trigger functionality.

In the function, *e* is the [uvm_event](#) that is being triggered, and *data* is the optional data associated with the event trigger.

uvm_barrier

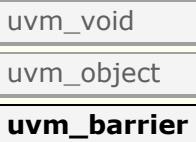
The uvm_barrier class provides a multiprocess synchronization mechanism. It enables a set of processes to block until the desired number of processes get to the synchronization point, at which time all of the processes are released.

Summary

uvm_barrier

The uvm_barrier class provides a multiprocess synchronization mechanism.

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_barrier extends uvm_object
```

METHODS

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

METHODS

new

```
function new (string name = "", int threshold = 0)
```

Creates a new barrier object.

wait_for

```
virtual task wait_for()
```

Waits for enough processes to reach the barrier before continuing.

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

reset

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

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

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

If the *wakeup* bit is set, any currently waiting processes will be activated.

set_auto_reset

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

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

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

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

set_threshold

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

Sets the process threshold.

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

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

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

get_threshold

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

Gets the current threshold setting for the barrier.

get_num_waiters

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

Returns the number of processes currently waiting at the barrier.

cancel

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

Decrement the waiter count by one. This is used when a process that is waiting on the

barrier is killed or activated by some other means.

uvm_objection_cb

This class allows for external consumers to attach to the various objection callbacks, [uvm_objection::raised](#), [uvm_objection::dropped](#) and [uvm_objection::all_dropped](#).

```
class my_objection_cb extends uvm_objection_cb;
  virtual function void raised (uvm_object obj, uvm_object source_obj,
    string description, int count);
    if(obj == source_obj)
      $display("Got raise: %s from object %s", description,
    obj.get_full_name());
  endfunction
endclass

my_objection_cb cb = new;

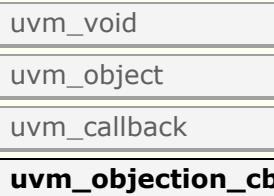
//add to every type of objection
initial uvm_callbacks#(uvm_objection)::add(null,cb);
```

Summary

uvm_objection_cb

This class allows for external consumers to attach to the various objection callbacks, [uvm_objection::raised](#), [uvm_objection::dropped](#) and [uvm_objection::all_dropped](#).

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_objection_cb extends uvm_callback
```

uvm_objection

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

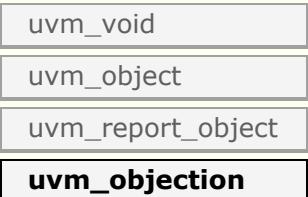
Tracing of objection activity can be turned on to follow the activity of the objection mechanism. It may be turned on for a specific objection instance with [uvm_objection::trace_mode](#), or it can be set for all objections from the command line using the option +UVM_OBJECTION_TRACE.

Summary

uvm_objection

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

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_objection extends uvm_report_object
```

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

OBJECTION CONTROL

<code>raise_objection</code>	Raises the number of objections for the source <i>object</i> by <i>count</i> , which defaults to 1.
<code>drop_objection</code>	Drops the number of objections for the source <i>object</i> by <i>count</i> , which defaults to 1.
<code>set_drain_time</code>	Sets the drain time on the given <i>object</i> to <i>drain</i> .

CALLBACK HOOKS

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

OBJECTION STATUS

<code>get_objection_count</code>	Returns the current number of objections raised by the given <i>object</i> .
<code>get_objection_total</code>	Returns the current number of objections raised by the given <i>object</i> and all descendants.
<code>get_drain_time</code>	Returns the current drain time set for the given <i>object</i> (default: 0 ns).
<code>display_objections</code>	Displays objection information about the given <i>object</i> .

new

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

Creates a new objection instance. Accesses the command line argument `+UVM_OBJECTION_TRACE` to turn tracing on for all objection objects.

trace_mode

```
function bit trace_mode (int mode = -1)
```

Set or get the trace mode for the objection object. If no argument is specified (or an argument other than 0 or 1) the current trace mode is unaffected. A trace_mode of 0 turns tracing off. A trace mode of 1 turns tracing on. The return value is the mode prior to being reset.

raise_objection

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

Raises the number of objections for the source *object* by *count*, which defaults to 1. The *object* is usually the *this* handle of the caller. If *object* is not specified or null, the implicit top-level component, *uvm_top*, is chosen.

Raising an objection causes the following.

- The source and total objection counts for *object* are increased by *count*. *description* is a string that marks a specific objection and is used in tracing/debug.
- The objection's [raised](#) virtual method is called, which calls the [uvm_component::raised](#) method for all of the components up the hierarchy.

drop_objection

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

Drops the number of objections for the source *object* by *count*, which defaults to 1. The *object* is usually the *this* handle of the caller. If *object* is not specified or null, the implicit top-level component, *uvm_top*, is chosen.

Dropping an objection causes the following.

- The source and total objection counts for *object* are decreased by *count*. It is an error to drop the objection count for *object* below zero.
- The objection's [dropped](#) virtual method is called, which calls the [uvm_component::dropped](#) method for all of the components up the hierarchy.
- If the total objection count has not reached zero for *object*, then the drop is propagated up the object hierarchy as with [raise_objection](#). Then, each object in the hierarchy will have updated their *source* counts--objections that they originated--and *total* counts--the total number of objections by them and all their descendants.

If the total objection count reaches zero, propagation up the hierarchy is deferred until a configurable drain-time has passed and the [uvm_component::all_dropped](#) callback for the current hierarchy level has returned. The following process occurs for each instance up the hierarchy from the source caller:

A process is forked in a non-blocking fashion, allowing the *drop* call to return. The forked process then does the following:

- If a drain time was set for the given *object*, the process waits for that amount of time.
- The objection's [all_dropped](#) virtual method is called, which calls the [uvm_component::all_dropped](#) method (if *object* is a component).
- The process then waits for the *all_dropped* callback to complete.
- After the drain time has elapsed and *all_dropped* callback has completed, propagation of the dropped objection to the parent proceeds as described in [raise_objection](#), except as described below.

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

As an optimization, if the *object* has no set drain-time and no registered callbacks, the forked process can be skipped and propagation proceeds immediately to the parent as described.

set_drain_time

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

Sets the drain time on the given *object* to *drain*.

The drain time is the amount of time to wait once all objections have been dropped before calling the all_dropped callback and propagating the objection to the parent.

If a new objection for this *object* or any of its descendants is raised during the drain time or during execution of the all_dropped callbacks, the drain_time/all_dropped execution is terminated.

CALLBACK HOOKS

raised

```
virtual function void raised (uvm_object obj,
                            uvm_object source_obj,
                            string      description,
                            int         count      )
```

Objection callback that is called when a [raise_objection](#) has reached *obj*. The default implementation calls [uvm_component::raised](#).

dropped

```
virtual function void dropped (uvm_object obj,
                             uvm_object source_obj,
                             string      description,
                             int         count      )
```

Objection callback that is called when a [drop_objection](#) has reached *obj*. The default implementation calls [uvm_component::dropped](#).

all_dropped

```
virtual task all_dropped (uvm_object obj,
                         uvm_object source_obj,
                         string      description,
                         int         count      )
```

Objection callback that is called when a [drop_objection](#) has reached *obj*, and the total count for *obj* goes to zero. This callback is executed after the drain time associated with *obj*. The default implementation calls [uvm_component::all_dropped](#).

OBJECTION STATUS

[get_objection_count](#)

```
function int get_objection_count (uvm_object obj)
```

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

[get_objection_total](#)

```
function int get_objection_total (uvm_object obj = null)
```

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

[get_drain_time](#)

```
function time get_drain_time (uvm_object obj)
```

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

[display_objections](#)

```
protected function string m_display_objections(uvm_object obj = null,  
                                              bit show_header = 1)
```

Displays objection information about the given *object*. If *object* is not specified or *null*, the implicit top-level component, <uvm_top>, is chosen. The *show_header* argument allows control of whether a header is output.

uvm_test_done_objection

Built-in end-of-test coordination

Summary

[uvm_test_done_objection](#)

Built-in end-of-test coordination

CLASS HIERARCHY

```
uvm_void
```

```
uvm_object
```

uvm_report_object

uvm_objection

uvm_test_done_objection

CLASS DECLARATION

```
class uvm_test_done_objection extends uvm_objection
```

METHODS

qualify

Checks that the given *object* is derived from either [uvm_component](#) or [uvm_sequence_base](#).

all_dropped

This callback is called when the given *object*'s objection count reaches zero; if the *object* is the implicit top-level, `<uvm_top>` then it means there are no more objections raised for the *uvm_test_done* objection.

raise_objection

Calls [uvm_objection::raise_objection](#) after calling [qualify](#).

drop

Calls [uvm_objection::drop_objection](#) after calling [qualify](#).

force_stop

Forces the propagation of the `all_dropped()` callback, even if there are still outstanding objections.

METHODS

qualify

```
virtual function void qualify(uvm_object obj = null,  
                           bit      is_raise,  
                           string   description )
```

Checks that the given *object* is derived from either [uvm_component](#) or [uvm_sequence_base](#).

all_dropped

```
virtual task all_dropped (uvm_object obj,  
                         uvm_object source_obj,  
                         string   description,  
                         int      count )
```

This callback is called when the given *object*'s objection count reaches zero; if the *object* is the implicit top-level, `<uvm_top>` then it means there are no more objections raised for the *uvm_test_done* objection. Thus, after calling [uvm_objection::all_dropped](#), this method will call [global_stop_request](#) to stop the current task-based phase (e.g. run).

raise_objection

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

Calls [uvm_objection::raise_objection](#) after calling [qualify](#). If the *object* is not provided or is `null`, then the implicit top-level component, *uvm_top*, is chosen.

drop

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

Calls [uvm_objection::drop_objection](#) after calling [qualify](#). If the *object* is not provided or is *null*, then the implicit top-level component, *uvm_top*, is chosen.

force_stop

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

Forces the propagation of the *all_dropped()* callback, even if there are still outstanding objections. The net effect of this action is to forcibly end the current phase.

uvm_heartbeat

Heartbeats provide a way for environments to easily ensure that their descendants are alive. A uvm_heartbeat is associated with a specific objection object. A component that is being tracked by the heartbeat object must raise (or drop) the synchronizing objection during the heartbeat window.

The uvm_heartbeat object has a list of participating objects. The heartbeat can be configured so that all components (UVM_ALL_ACTIVE), exactly one (UVM_ONE_ACTIVE), or any component (UVM_ANY_ACTIVE) must trigger the objection in order to satisfy the heartbeat condition.

Summary

uvm_heartbeat

Heartbeats provide a way for environments to easily ensure that their descendants are alive.

METHODS

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

METHODS

new

```
function new(string name,  
            uvm_component cntxt,  
            uvm_objection objection = null)
```

Creates a new heartbeat instance associated with *cntxt*. The context is the hierarchical locationa that the heartbeat objections will flow through and be monitored at. The *objection* associated with the heartbeat is optional, if it is left null then the uvm_test_done objection is used.

```
uvm_objection myobjection = new("myobjection"); //some shared objection  
class myenv extends uvm_env;  
    uvm_heartbeat hb = new("hb", this, myobjection);  
    ...  
endclass
```

hb_mode

```
function uvm_heartbeat_modes hb_mode (uvm_heartbeat_modes mode = UVM_NO_HB_MC
```

Sets or retrieves the heartbeat mode. The current value for the heartbeat mode is returned. If an argument is specified to change the mode then the mode is changed to the new value.

[set_heartbeat](#)

```
function void set_heartbeat ( uvm_event e,  
                           ref uvm_component comps[$] )
```

Sets up the heartbeat event and assigns a list of objects to watch. The monitoring is started as soon as this method is called. Once the monitoring has been started with a specific event, providing a new monitor event results in an error. To change trigger events, you must first [stop](#) the monitor and then [start](#) with a new event trigger.

If the trigger event *e* is null and there was no previously set trigger event, then the monitoring is not started. Monitoring can be started by explicitly calling [start](#).

[add](#)

```
function void add (uvm_component comp)
```

Add a single component to the set of components to be monitored. This does not cause monitoring to be started. If monitoring is currently active then this component will be immediately added to the list of components and will be expected to participate in the currently active event window.

[remove](#)

```
function void remove (uvm_component comp)
```

Remove a single component to the set of components being monitored. Monitoring is not stopped, even if the last component has been removed (an explicit stop is required).

[start](#)

```
function void start (uvm_event e = null)
```

Starts the heartbeat monitor. If *e* is null then whatever event was previously set is used. If no event was previously set then a warning is issued. It is an error if the monitor is currently running and *e* is specifying a different trigger event from the current event.

[stop](#)

```
function void stop ()
```

Stops the heartbeat monitor. Current state information is reset so that if [start](#) is called again the process will wait for the first event trigger to start the monitoring.

uvm_pool #(T)

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

Summary

uvm_pool #(T)

Implements a class-based dynamic associative array.

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_pool #(type KEY = int,  
                 T     = uvm_void) extends uvm_object
```

METHODS

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

METHODS

new

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

Creates a new pool with the given *name*.

get_global_pool

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

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

This allows items to be shared amongst components throughout the verification

environment.

get_global

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

Returns the specified item instance from the global item pool.

get

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

Returns the item with the given *key*.

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

add

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

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

num

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

Returns the number of uniquely keyed items stored in the pool.

delete

```
virtual function void delete (KEY key)
```

Removes the item with the given *key* from the pool.

exists

```
virtual function int exists (KEY key)
```

Returns 1 if a item with the given *key* exists in the pool, 0 otherwise.

first

```
virtual function int first (ref KEY key)
```

Returns the key of the first item stored in the pool.

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

If the pool is not empty, then *key* is key of the first item and 1 is returned.

last

```
virtual function int last (ref KEY key)
```

Returns the key of the last item stored in the pool.

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

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

next

```
virtual function int next (ref KEY key)
```

Returns the key of the next item in the pool.

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

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

prev

```
virtual function int prev (ref KEY key)
```

Returns the key of the previous item in the pool.

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

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

uvm_object_string_pool #(T)

This provides a specialization of the generic <uvm_pool #(KEY,T) class for an associative array of [uvm_object](#)-based objects indexed by string. Specializations of this class include the *uvm_event_pool* and *uvm_barrier_pool* classes.

Summary

uvm_object_string_pool #(T)

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

CLASS HIERARCHY

```
uvm_pool#(string,T)
```

```
uvm_object_string_pool#(T)
```

CLASS DECLARATION

```
class uvm_object_string_pool #(  
    type T = uvm_object  
) extends uvm_pool #(string,T)
```

METHODS

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

METHODS

new

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

Creates a new pool with the given *name*.

get_type_name

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

Returns the type name of this object.

get_global_pool

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

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

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

get

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

Returns the object item at the given string *key*.

If no item exists by the given *key*, a new item is created for that key and returned.

delete

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

Removes the item with the given string *key* from the pool.

uvm_queue #(T)

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

Summary

uvm_queue #(T)

Implements a class-based dynamic queue.

CLASS HIERARCHY

uvm_void

uvm_object

uvm_queue#(T)

CLASS DECLARATION

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

METHODS

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

METHODS

new

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

Creates a new queue with the given *name*.

get_global_queue

```
static function this_type get_global_queue ()
```

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

This allows items to be shared amongst components throughout the verification

environment.

get_global

```
static function T get_global (int index)
```

Returns the specified item instance from the global item queue.

get

```
virtual function T get (int index)
```

Returns the item at the given *index*.

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

size

```
virtual function int size ()
```

Returns the number of items stored in the queue.

insert

```
virtual function void insert (int index,  
                           T      item  )
```

Inserts the item at the given *index* in the queue.

delete

```
virtual function void delete (int index = -1)
```

Removes the item at the given *index* from the queue; if *index* is not provided, the entire contents of the queue are deleted.

pop_front

```
virtual function T pop_front()
```

Returns the first element in the queue (index=0), or *null* if the queue is empty.

pop_back

```
virtual function T pop_back()
```

Returns the last element in the queue (index=size()-1), or *null* if the queue is empty.

push_front

```
virtual function void push_front(T item)
```

Inserts the given *item* at the front of the queue.

push_back

```
virtual function void push_back(T item)
```

Inserts the given *item* at the back of the queue.

uvm_callbacks #(T,CB)

The *uvm_callbacks* class provides a base class for implementing callbacks, which are typically used to modify or augment component behavior without changing the component class. To work effectively, the developer of the component class defines a set of "hook" methods that enable users to customize certain behaviors of the component in a manner that is controlled by the component developer. The integrity of the component's overall behavior is intact, while still allowing certain customizable actions by the user.

To enable compile-time type-safety, the class is parameterized on both the user-defined callback interface implementation as well as the object type associated with the callback. The object type-callback type pair are associated together using the `uvm_register_cb macro to define a valid pairing; valid pairings are checked when a user attempts to add a callback to an object.

To provide the most flexibility for end-user customization and reuse, it is recommended that the component developer also define a corresponding set of virtual method hooks in the component itself. This affords users the ability to customize via inheritance/factory overrides as well as callback object registration. The implementation of each virtual method would provide the default traversal algorithm for the particular callback being called. Being virtual, users can define subtypes that override the default algorithm, perform tasks before and/or after calling super.<method> to execute any registered callbacks, or to not call the base implementation, effectively disabling that particular hook. A demonstration of this methodology is provided in an example included in the kit.

Summary

uvm_callbacks #(T,CB)

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

T

This type parameter specifies the base object type with which the CB callback objects will be registered.

CB

This type parameter specifies the base callback type that will be managed by this callback class.

Add/Delete

INTERFACE

add

Registers the given callback object, cb, with the given obj handle.

add_by_name

Registers the given callback object, cb, with one or more uvm_components.

delete

Deletes the given callback object, cb, from the queue associated with the given obj handle.

delete_by_name

Removes the given callback object, cb, associated with one or more uvm_component callback queues.

Iterator Interface

This set of functions provide an iterator interface for callback queues.

get_first

returns the first enabled callback of type CB which resides in the queue for obj.

get_next

returns the next enabled callback of type CB which resides in the queue for obj, using itr as the starting point.

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

CB

This type parameter specifies the base callback type that will be managed by this callback class. The callback type is typically a interface class, which defines one or more virtual method prototypes that users can override in subtypes. This type must be a derivative of [uvm_callback](#).

ADD / DELETE INTERFACE

add

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

Registers the given callback object, *cb*, with the given *obj* handle. The *obj* handle can be null, which allows registration of callbacks without an object context. If *ordreing* is UVM_APPEND (default), the callback will be executed after previously added callbacks, else the callback will be executed ahead of previously added callbacks. The *cb* is the callback handle; it must be non-null, and if the callback has already been added to the object instance then a warning is issued. Note that the CB parameter is optional. For example, the following are equivalent:

```
uvm_callbacks#(my_comp)::add(comp_a, cb);
uvm_callbacks#(my_comp, my_callback)::add(comp_a, cb);
```

add_by_name

```
static function void add_by_name(string      name,
                                 uvm_callback cb,
                                 uvm_component root,
                                 uvm_appprepend ordering = UVM_APPEND)
```

Registers the given callback object, *cb*, with one or more [uvm_components](#). The components must already exist and must be type T or a derivative. As with [add](#) the CB parameter is optional. *root* specifies the location in the component hierarchy to start the search for *name*. See [uvm_root::find_all](#) for more details on searching by name.

delete

```
static function void delete(T          obj,
                           uvm_callback cb )
```

Deletes the given callback object, *cb*, from the queue associated with the given *obj* handle. The *obj* handle can be null, which allows de-registration of callbacks without an object context. The *cb* is the callback handle; it must be non-null, and if the callback has already been removed to the object instance then a warning is issued. Note that the CB parameter is optional. For example, the following are equivalent:

```
uvm_callbacks#(my_comp)::delete(comp_a, cb);
uvm_callbacks#(my_comp, my_callback)::delete(comp_a, cb);
```

[delete_by_name](#)

```
static function void delete_by_name(string name,
                                    uvm_callback cb,
                                    uvm_component root )
```

Removes the given callback object, *cb*, associated with one or more `uvm_component` callback queues. As with [delete](#) the CB parameter is optional. *root* specifies the location in the component hierarchy to start the search for *name*. See [uvm_root::find_all](#) for more details on searching by name.

[ITERATOR INTERFACE](#)

This set of functions provide an iterator interface for callback queues. A facade class, [uvm_callback_iter](#) is also available, and is the generally preferred way to iterate over callback queues.

[get_first](#)

```
static function CB get_first ( ref int itr,
                             input T obj )
```

returns the first enabled callback of type CB which resides in the queue for *obj*. If *obj* is null then the typewide queue for T is searched. *itr* is the iterator; it will be updated with a value that can be supplied to [get_next](#) to get the next callback object.

If the queue is empty then null is returned.

The iterator class [uvm_callback_iter](#) may be used as an alternative, simplified, iterator interface.

[get_next](#)

```
static function CB get_next ( ref int itr,
                            input T obj )
```

returns the next enabled callback of type CB which resides in the queue for *obj*, using *itr* as the starting point. If *obj* is null then the typewide queue for T is searched. *itr* is the iterator; it will be updated with a value that can be supplied to [get_next](#) to get the next callback object.

If no more callbacks exist in the queue, then null is returned. [get_next](#) will continue to return null in this case until [get_first](#) has been used to reset the iterator.

The iterator class [uvm_callback_iter](#) may be used as an alternative, simplified, iterator interface.

uvm_callback_iter

The `uvm_callback_iter` class is an iterator class for iterating over callback queues of a specific callback type. The typical usage of the class is:

```
uvm_callback_iter#(mycomp,mycb) iter = new(this);
for(mycb cb = iter.first(); cb != null; cb = iter.next())
    cb.dosomething();
```

The callback iteration macros, ``uvm_do_callbacks`` and ``uvm_do_callbacks_exit_on`` provide a simple method for iterating callbacks and executing the callback methods.

Summary

uvm_callback_iter

The `uvm_callback_iter` class is an iterator class for iterating over callback queues of a specific callback type.

CLASS DECLARATION

```
class uvm_callback_iter#(type T = uvm_object,
                           type CB = uvm_callback)
```

METHODS

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

METHODS

new

```
function new(T obj)
```

Creates a new callback iterator object. It is required that the object context be provided.

first

```
function CB first()
```

Returns the first valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object. If the queue is empty then null is returned.

next

```
function CB next()
```

Returns the next valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object. If there are no more valid callbacks in the queue, then null is returned.

get_cb

```
function CB get_cb()
```

Returns the last callback accessed via a first() or next() call.

uvm_callback

The *uvm_callback* class is the base class for user-defined callback classes. Typically, the component developer defines an application-specific callback class that extends from this class. In it, he defines one or more virtual methods, called a *callback interface*, that represent the hooks available for user override.

Methods intended for optional override should not be declared *pure*. Usually, all the callback methods are defined with empty implementations so users have the option of overriding any or all of them.

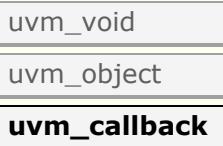
The prototypes for each hook method are completely application specific with no restrictions.

Summary

uvm_callback

The *uvm_callback* class is the base class for user-defined callback classes.

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_callback extends uvm_object
```

METHODS

`new` Creates a new *uvm_callback* object, giving it an optional *name*.

`callback_mode` Enable/disable callbacks (modeled like *rand_mode* and *constraint_mode*).

`is_enabled` Returns 1 if the callback is enabled, 0 otherwise.

`get_type_name` Returns the type name of this callback object.

METHODS

new

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

Creates a new uvm_callback object, giving it an optional *name*.

callback_mode

```
function bit callback_mode(int on = -1)
```

Enable/disable callbacks (modeled like rand_mode and constraint_mode).

is_enabled

```
function bit is_enabled()
```

Returns 1 if the callback is enabled, 0 otherwise.

get_type_name

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

Returns the type name of this callback object.

Policy Classes

Each of UVM's policy classes perform a specific task for [uvm_object](#)-based objects: printing, comparing, recording, packing, and unpacking. They are implemented separately from *uvm_object* so that users can plug in different ways to print, compare, etc. without modifying the object class being operated on. The user can simply apply a different printer or compare "policy" to change how an object is printed or compared.

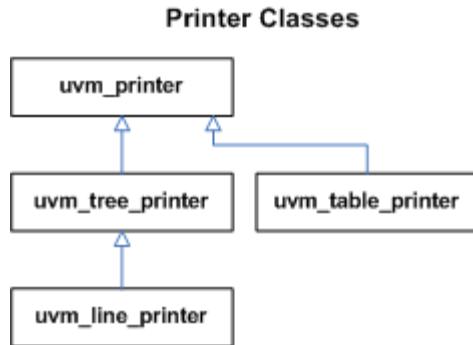
Each policy class includes several user-configurable parameters that control the operation. Users may also customize operations by deriving new policy subtypes from these base types. For example, the UVM provides four different [uvm_printer](#)-based policy classes, each of which print objects in a different format.

- [uvm_printer](#) - performs deep printing of *uvm_object*-based objects. The UVM provides several subtypes to [uvm_printer](#) that print objects in a specific format: [uvm_table_printer](#), [uvm_tree_printer](#), and [uvm_line_printer](#). Each such printer has many configuration options that govern what and how object members are printed.
- [uvm_comparer](#) - performs deep comparison of *uvm_object*-based objects. Users may configure what is compared and how miscompares are reported.
- [uvm_recorder](#) - performs the task of recording *uvm_object*-based objects to a transaction data base. The implementation is vendor-specific.
- [uvm_packer](#) - used to pack (serialize) and unpack *uvm_object*-based properties into bit, byte, or int arrays and back again.

uvm_printer

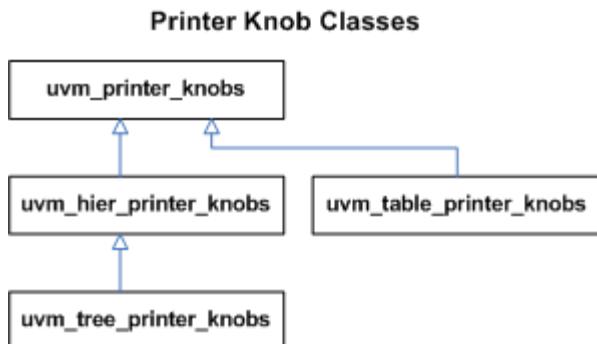
The `uvm_printer` class provides an interface for printing `uvm_objects` in various formats. Subtypes of `uvm_printer` implement different print formats, or policies.

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



- `uvm_printer` - provides raw, essentially un-formatted output
- `uvm_table_printer` - prints the object in a tabular form.
- `uvm_tree_printer` - prints the object in a tree form.
- `uvm_line_printer` - prints the information on a single line, but uses the same object separators as the tree printer.

Printers have knobs that you use to control what and how information is printed. These knobs are contained in separate knob classes:



- `uvm_printer_knobs` - common printer settings
- `uvm_hier_printer_knobs` - settings for printing hierarchically
- `uvm_table_printer_knobs` - settings specific to the table printer
- `uvm_tree_printer_knobs` - settings specific to the tree printer

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

- `uvm_default_tree_printer`
- `uvm_default_line_printer`
- `uvm_default_table_printer`
- `uvm_default_printer` (set to `default_table_printer` by default)

The `uvm_default_printer` is used by `uvm_object::print` and `uvm_object::sprint` when the optional `uvm_printer` argument to these methods is not provided.

Summary

uvm_printer

The uvm_printer class provides an interface for printing [uvm_objects](#) in various formats.

CLASS DECLARATION

```
class uvm_printer
```

knobs

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

METHODS FOR PRINTER

USAGE

print_field	Prints an integral field.
print_object_header	Prints the header of an object.
print_object	Prints an object.
print_string	Prints a string field.
print_time	Prints a time value.

METHODS FOR PRINTER

SUBTYPING

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

knobs

```
uvm_printer_knobs knobs = new
```

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

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

METHODS FOR PRINTER USAGE

print_field

```
virtual function void print_field (string name,  
                                  uvm_bitstream_t value,  
                                  int size,  
                                  uvm_radix_enum radix = UVM_NORM  
                                  byte scope_separator = " . ",  
                                  string type_name = "")
```

Prints an integral field.

<i>name</i>	The name of the field.
<i>value</i>	The value of the field.
<i>size</i>	The number of bits of the field (maximum is 4096).
<i>radix</i>	The radix to use for printing the printer knob for radix is used if no radix is specified.
<i>scope_separator</i>	is used to find the leaf name since many printers only print the leaf name of a field. Typical values for the separator are . (dot) or [(open bracket).

[print_object_header](#)

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

Prints the header of an object.

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

[print_object](#)

```
virtual function void print_object (string      name,
                                   uvm_object   value,
                                   byte        scope_separator = ".")
```

Prints an object. Whether the object is recursed depends on a variety of knobs, such as the depth knob; if the current depth is at or below the depth setting, then the object is not recursed.

By default, the children of [uvm_components](#) are printed. To turn this behavior off, you must set the [uvm_component::print_enabled](#) bit to 0 for the specific children you do not want automatically printed.

[print_string](#)

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

Prints a string field.

[print_time](#)

```
virtual function void print_time (string name,
                                  time   value,
                                  byte   scope_separator = ".")
```

Prints a time value. name is the name of the field, and value is the value to print.

The print is subject to the *\$timeformat* system task for formatting time values.

METHODS FOR PRINTER SUBTYPING

print_header

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

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

print_footer

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

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

print_id

```
virtual protected function void print_id (string id,  
                                         byte scope_separator = ".")
```

Prints a field's name, or *id*, which is the full instance name.

The intent of the separator is to mark where the leaf name starts if the printer is configured to print only the leaf name of the identifier.

print_type_name

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

Prints a field's type name.

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

print_size

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

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

print_newline

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

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

[print_value](#)

```
virtual protected function void print_value (uvm_bitstream_t value,
                                         int          size,
                                         uvm_radix_enum radix = UVM_NOF
```

Prints an integral field's value.

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

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

[print_value_object](#)

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

Prints a unique handle identifier for the given object.

[print_value_string](#)

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

Prints a string field's value.

[print_value_array](#)

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

Prints an array's value.

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

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

[print_array_header](#)

```
virtual function void print_array_header(string name,
                                         int      size,
                                         string arraytype = "array",
                                         byte    scope_separator = ".")
```

Prints the header of an array. This function is called before each individual element is printed. [print_array_footer](#) is called to mark the completion of array printing.

[print_array_range](#)

```
virtual function void print_array_range (int min,
```

```
int max )
```

Prints a range using ellipses for values. This method is used when honoring the array knobs for partial printing of large arrays, [uvm_printer_knobs::begin_elements](#) and [uvm_printer_knobs::end_elements](#).

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

[print_array_footer](#)

```
virtual function void print_array_footer (int size = )
```

Prints the header of a footer. This function marks the end of an array print. Generally, there is no output associated with the array footer, but this method lets the printer know that the array printing is complete.

uvm_table_printer

The table printer prints output in a tabular format.

The following shows sample output from the table printer.

Name	Type	Size	Value
c1	container	-	@1013
d1	mydata	-	@1022
v1	integral	32	'hcb8f1c97
e1	enum	32	THREE
str	string	2	hi
value	integral	12	'h2d

Summary

uvm_table_printer

The table printer prints output in a tabular format.

CLASS HIERARCHY

```
uvm_printer
```

```
uvm_table_printer
```

CLASS DECLARATION

```
class uvm_table_printer extends uvm_printer
```

VARIABLES

- | | |
|--------------------|--|
| <code>new</code> | Creates a new instance of <i>uvm_table_printer</i> . |
| <code>knobs</code> | An instance of uvm_table_printer_knobs , which govern the content and format of the printed table. |

VARIABLES

new

```
function new()
```

Creates a new instance of *uvm_table_printer*.

knobs

```
uvm_table_printer_knobs knobs = new
```

An instance of *uvm_table_printer_knobs*, which govern the content and format of the printed table.

uvm_tree_printer

By overriding various methods of the *uvm_printer* super class, the tree printer prints output in a tree format.

The following shows sample output from the tree printer.

```
c1: (container@1013) {  
    d1: (mydata@1022) {  
        v1: 'hcb8f1c97  
        e1: THREE  
        str: hi  
    }  
    value: 'h2d  
}
```

Summary

uvm_tree_printer

By overriding various methods of the *uvm_printer* super class, the tree printer prints output in a tree format.

CLASS HIERARCHY

```
uvm_printer
```

```
uvm_tree_printer
```

CLASS DECLARATION

```
class uvm_tree_printer extends uvm_printer
```

VARIABLES

new

Creates a new instance of *uvm_tree_printer*.

knobs

An instance of *uvm_tree_printer_knobs*, which govern the content and format of the printed tree.

VARIABLES

new

```
function new()
```

Creates a new instance of *uvm_tree_printer*.

knobs

```
uvm_tree_printer_knobs knobs = new
```

An instance of [uvm_tree_printer_knobs](#), which govern the content and format of the printed tree.

uvm_line_printer

The line printer prints output in a line format.

The following shows sample output from the line printer.

```
c1: (container@1013) { d1: (mydata@1022) { v1: 'hcb8f1c97 e1: THREE str: hi  
} value: 'h2d }
```

Summary

uvm_line_printer

The line printer prints output in a line format.

CLASS HIERARCHY

```
uvm_printer
```

```
uvm_tree_printer
```

```
uvm_line_printer
```

CLASS DECLARATION

```
class uvm_line_printer extends uvm_tree_printer
```

VARIABLES

new Creates a new instance of *uvm_line_printer*.

METHODS

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

VARIABLES

new

```
function new()
```

Creates a new instance of *uvm_line_printer*.

METHODS

[print_newline](#)

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

Overrides [uvm_printer::print_newline](#) to not print a newline, effectively making everything appear on a single line.

uvm_printer_knobs

The *uvm_printer_knobs* class defines the printer settings available to all printer subtypes. Printer subtypes may subtype this class to provide additional knobs for their specific format. For example, the [uvm_table_printer](#) uses the [uvm_table_printer_knobs](#), which defines knobs for setting table column widths.

Summary

uvm_printer_knobs

The *uvm_printer_knobs* class defines the printer settings available to all printer subtypes.

CLASS DECLARATION

```
class uvm_printer_knobs
```

VARIABLES

<code>max_width</code>	The maximum width of a field.
<code>truncation</code>	Specifies the character to use to indicate a field was truncated.
<code>header</code>	Indicates whether the <code><print_header></code> function should be called when printing an object.
<code>footer</code>	Indicates whether the <code><print_footer></code> function should be called when printing an object.
<code>global_indent</code>	Specifies the number of spaces of indentation to add whenever a newline is printed.
<code>full_name</code>	Indicates whether <code><print_id></code> should print the full name of an identifier or just the leaf name.
<code>identifier</code>	Indicates whether <code><print_id></code> should print the identifier.
<code>depth</code>	Indicates how deep to recurse when printing objects.
<code>reference</code>	Controls whether to print a unique reference ID for object handles.
<code>type_name</code>	Controls whether to print a field's type name.
<code>size</code>	Controls whether to print a field's size.
<code>begin_elements</code>	Defines the number of elements at the head of a list to print.
<code>end_elements</code>	This defines the number of elements at the end of a list that should be printed.

<code>show_radix</code>	Indicates whether the radix string ('h, and so on) should be prepended to an integral value when one is printed.
<code>prefix</code>	Specifies the string prepended to each output line
<code>mcd</code>	This is a file descriptor, or multi-channel descriptor, that specifies where the print output should be directed.
<code>default_radix</code>	This knob sets the default radix to use for integral values when no radix enum is explicitly supplied to the <code>print_field()</code> method.
<code>dec_radix</code>	This string should be prepended to the value of an integral type when a radix of <code>UVM_DEC</code> is used for the radix of the integral object.
<code>bin_radix</code>	This string should be prepended to the value of an integral type when a radix of <code>UVM_BIN</code> is used for the radix of the integral object.
<code>oct_radix</code>	This string should be prepended to the value of an integral type when a radix of <code>UVM_OCT</code> is used for the radix of the integral object.
<code>unsigned_radix</code>	This is the string which should be prepended to the value of an integral type when a radix of <code>UVM_UNSIGNED</code> is used for the radix of the integral object.
<code>hex_radix</code>	This string should be prepended to the value of an integral type when a radix of <code>UVM_HEX</code> is used for the radix of the integral object.

METHODS

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

VARIABLES

max_width

```
int max_width = 999
```

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

truncation

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

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

header

```
bit header = 1
```

Indicates whether the <print_header> function should be called when printing an object.

footer

```
bit footer = 1
```

Indicates whether the <print_footer> function should be called when printing an object.

global_indent

```
int global_indent = 0
```

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

full_name

```
bit full_name = 1
```

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

identifier

```
bit identifier = 1
```

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

depth

```
int depth = -1
```

Indicates how deep to recurse when printing objects. A depth of -1 means to print everything.

reference

```
bit reference = 1
```

Controls whether to print a unique reference ID for object handles. The behavior of this knob is simulator-dependent.

type_name

```
bit type_name = 1
```

Controls whether to print a field's type name.

size

```
bit size = 1
```

Controls whether to print a field's size.

begin_elements

```
int begin_elements = 5
```

Defines the number of elements at the head of a list to print. Use -1 for no max.

[end_elements](#)

```
int end_elements = 5
```

This defines the number of elements at the end of a list that should be printed.

[show_radix](#)

```
bit show_radix = 1
```

Indicates whether the radix string ('h, and so on) should be prepended to an integral value when one is printed.

[prefix](#)

```
string prefix = ""
```

Specifies the string prepended to each output line

[mcd](#)

```
int mcd = UVM_STDOUT
```

This is a file descriptor, or multi-channel descriptor, that specifies where the print output should be directed.

By default, the output goes to the standard output of the simulator.

[default_radix](#)

```
uvm_radix_enum default_radix = UVM_HEX
```

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

[dec_radix](#)

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

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

When a negative number is printed, the radix is not printed since only signed decimal values can print as negative.

bin_radix

```
string bin_radix = "'b"
```

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

oct_radix

```
string oct_radix = "'o"
```

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

unsigned_radix

```
string unsigned_radix = "'d"
```

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

hex_radix

```
string hex_radix = "'h"
```

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

METHODS

get_radix_str

```
function string get_radix_str (uvm_radix_enum radix)
```

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

uvm_hier_printer_knobs

The *uvm_hier_printer_knobs* is a simple container class that extends <uvm_printer::uvm_printer_knobs> with settings for printing information hierarchically.

Summary

uvm_hier_printer_knobs

The *uvm_hier_printer_knobs* is a simple container class that extends <uvm_printer::uvm_printer_knobs> with settings for printing information

hierarchically.

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_hier_printer_knobs extends uvm_printer_knobs
```

VARIABLES

indent_str	This knob specifies the string to use for level indentation.
show_root	This setting indicates whether or not the initial object that is printed (when current depth is 0) prints the full path name.

VARIABLES

indent_str

```
string indent_str = " "
```

This knob specifies the string to use for level indentation. The default level indentation is two spaces.

show_root

```
bit show_root = 0
```

This setting indicates whether or not the initial object that is printed (when current depth is 0) prints the full path name. By default, the first object is treated like all other objects and only the leaf name is printed.

uvm_table_printer_knobs

The *uvm_table_printer_knobs* is a simple container class that extends <uvm_printer:::uvm_hier_printer_knobs> with settings specific to printing in table format.

Summary

uvm_table_printer_knobs

The *uvm_table_printer_knobs* is a simple container class that extends <uvm_printer:::uvm_hier_printer_knobs> with settings specific to printing in table format.

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_table_printer_knobs extends  
uvm_hier_printer_knobs
```

VARIABLES

name_width	Sets the width of the <i>name</i> column.
type_width	Sets the width of the <i>type</i> column.
size_width	Sets the width of the <i>size</i> column.
value_width	Sets the width of the <i>value</i> column.

VARIABLES

name_width

```
int name_width = 25
```

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

type_width

```
int type_width = 20
```

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

size_width

```
int size_width = 5
```

Sets the width of the *size* column. If set to 0, the column is not printed.

value_width

```
int value_width = 20
```

Sets the width of the *value* column. If set to 0, the column is not printed.

uvm_tree_printer_knobs

The *uvm_tree_printer_knobs* is a simple container class that extends <uvm_printer::uvm_hier_printer_knobs> with settings specific to printing in tree format.

Summary

uvm_tree_printer_knobs

The *uvm_tree_printer_knobs* is a simple container class that extends <uvm_printer::uvm_hier_printer_knobs> with settings specific to printing in tree

format.

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_tree_printer_knobs extends  
uvm_hier_printer_knobs
```

VARIABLES

separator Determines the opening and closing separators used for nested objects.

VARIABLES

separator

```
string separator = "{}"
```

Determines the opening and closing separators used for nested objects.

uvm_comparer

The uvm_comparer class provides a policy object for doing comparisons. The policies determine how miscompares are treated and counted. Results of a comparison are stored in the comparer object. The [uvm_object::compare](#) and [uvm_object::do_compare](#) methods are passed an uvm_comparer policy object.

Summary

uvm_comparer

The uvm_comparer class provides a policy object for doing comparisons.

CLASS DECLARATION

```
class uvm_comparer
```

VARIABLES

policy	Determines whether comparison is UVM_DEEP, UVM_REFERENCE, or UVM_SHALLOW.
show_max	Sets the maximum number of messages to send to the messenger for miscompares of an object.
verbosity	Sets the verbosity for printed messages.
sev	Sets the severity for printed messages.
miscompares	This string is reset to an empty string when a comparison is started.
physical	This bit provides a filtering mechanism for fields.
abstract	This bit provides a filtering mechanism for fields.
check_type	This bit determines whether the type, given by uvm_object::get_type_name , is used to verify that the types of two objects are the same.
result	This bit stores the number of miscompares for a given compare operation.

METHODS

compare_field	Compares two integral values.
compare_field_int	This method is the same as compare_field except that the arguments are small integers, less than or equal to 64 bits.
compare_field_real	This method is the same as compare_field except that the arguments are real numbers.
compare_object	Compares two class objects using the policy knob to determine whether the comparison should be deep, shallow, or reference.
compare_string	Compares two string variables.
print_msg	Causes the error count to be incremented and the message, <i>msg</i> , to be appended to the miscompares string (a newline is used to separate messages).

VARIABLES

policy

```
uvm_recursion_policy_enum policy = UVM_DEFAULT_POLICY
```

Determines whether comparison is UVM_DEEP, UVM_REFERENCE, or UVM_SHALLOW.

show_max

```
int unsigned show_max = 1
```

Sets the maximum number of messages to send to the messenger for miscompares of an object.

verbosity

```
int unsigned verbosity = UVM_LOW
```

Sets the verbosity for printed messages.

The verbosity setting is used by the messaging mechanism to determine whether messages should be suppressed or shown.

sev

```
uvm_severity sev = UVM_INFO
```

Sets the severity for printed messages.

The severity setting is used by the messaging mechanism for printing and filtering messages.

miscompares

```
string miscompares = ""
```

This string is reset to an empty string when a comparison is started.

The string holds the last set of miscompares that occurred during a comparison.

physical

```
bit physical = 1
```

This bit provides a filtering mechanism for fields.

The abstract and physical settings allow an object to distinguish between two different classes of fields.

It is up to you, in the [uvm_object::do_compare](#) method, to test the setting of this field if you want to use the physical trait as a filter.

abstract

```
bit abstract = 1
```

This bit provides a filtering mechanism for fields.

The abstract and physical settings allow an object to distinguish between two different classes of fields.

It is up to you, in the `uvm_object::do_compare` method, to test the setting of this field if you want to use the abstract trait as a filter.

check_type

```
bit check_type = 1
```

This bit determines whether the type, given by `uvm_object::get_type_name`, is used to verify that the types of two objects are the same.

This bit is used by the `compare_object` method. In some cases it is useful to set this to 0 when the two operands are related by inheritance but are different types.

result

```
int unsigned result = 0
```

This bit stores the number of miscompares for a given compare operation. You can use the result to determine the number of miscompares that were found.

METHODS

compare_field

```
virtual function bit compare_field (string name,  
                                  uvm_bitstream_t lhs,  
                                  uvm_bitstream_t rhs,  
                                  int size,  
                                  uvm_radix_enum radix = UVM_NORADIX)
```

Compares two integral values.

The *name* input is used for purposes of storing and printing a miscompare.

The left-hand-side *lhs* and right-hand-side *rhs* objects are the two objects used for comparison.

The size variable indicates the number of bits to compare; size must be less than or equal to 4096.

The radix is used for reporting purposes, the default radix is hex.

compare_field_int

```
virtual function bit compare_field_int (string name,  
                                       logic[63:0] lhs,  
                                       logic[63:0] rhs,  
                                       int size,  
                                       uvm_radix_enum radix = UVM_NORADIX)
```

This method is the same as `compare_field` except that the arguments are small integers, less than or equal to 64 bits. It is automatically called by `compare_field` if the operand size is less than or equal to 64.

[compare_field_real](#)

```
virtual function bit compare_field_real (string name,  
                                      real    lhs,  
                                      real    rhs   )
```

This method is the same as [compare_field](#) except that the arguments are real numbers.

[compare_object](#)

```
virtual function bit compare_object (string      name,  
                                    uvm_object  lhs,  
                                    uvm_object  rhs   )
```

Compares two class objects using the [policy](#) knob to determine whether the comparison should be deep, shallow, or reference.

The *name* input is used for purposes of storing and printing a miscompare.

The *lhs* and *rhs* objects are the two objects used for comparison.

The *check_type* determines whether or not to verify the object types match (the return from *lhs.get_type_name()* matches *rhs.get_type_name()*).

[compare_string](#)

```
virtual function bit compare_string (string name,  
                                    string lhs,  
                                    string rhs   )
```

Compares two string variables.

The *name* input is used for purposes of storing and printing a miscompare.

The *lhs* and *rhs* objects are the two objects used for comparison.

[print_msg](#)

```
function void print_msg (string msg)
```

Causes the error count to be incremented and the message, *msg*, to be appended to the [miscompares](#) string (a newline is used to separate messages).

If the message count is less than the [show_max](#) setting, then the message is printed to standard-out using the current verbosity and severity settings. See the [verbosity](#) and [sev](#) variables for more information.

uvm_recorder

The uvm_recorder class provides a policy object for recording [uvm_objects](#). The policies determine how recording should be done.

A default recorder instance, [uvm_default_recorder](#), is used when the [uvm_object::record](#) is called without specifying a recorder.

Summary

uvm_recorder

The uvm_recorder class provides a policy object for recording [uvm_objects](#).

CLASS DECLARATION

```
class uvm_recorder
```

VARIABLES

tr_handle	This is an integral handle to a transaction object.
default_radix	This is the default radix setting if record_field is called without a radix.
physical	This bit provides a filtering mechanism for fields.
abstract	This bit provides a filtering mechanism for fields.
identifier	This bit is used to specify whether or not an object's reference should be recorded when the object is recorded.
recursion_policy	Sets the recursion policy for recording objects.

METHODS

record_field	Records an integral field (less than or equal to 4096 bits).
record_field_real	Records an real field.
record_object	Records an object field.
record_string	Records a string field.
record_time	Records a time value.
record_generic	Records the name-value pair, where value has been converted to a string, e.g.

VARIABLES

tr_handle

```
integer tr_handle = 0
```

This is an integral handle to a transaction object. Its use is vendor specific.

A handle of 0 indicates there is no active transaction object.

default_radix

```
uvm_radix_enum default_radix = UVM_HEX
```

This is the default radix setting if [record_field](#) is called without a radix.

physical

```
bit physical = 1
```

This bit provides a filtering mechanism for fields.

The **abstract** and physical settings allow an object to distinguish between two different classes of fields.

It is up to you, in the [uvm_object::do_record](#) method, to test the setting of this field if you want to use the physical trait as a filter.

abstract

```
bit abstract = 1
```

This bit provides a filtering mechanism for fields.

The abstract and physical settings allow an object to distinguish between two different classes of fields.

It is up to you, in the [uvm_object::do_record](#) method, to test the setting of this field if you want to use the abstract trait as a filter.

identifier

```
bit identifier = 1
```

This bit is used to specify whether or not an object's reference should be recorded when the object is recorded.

recursion_policy

```
uvm_recursion_policy_enum policy = UVM_DEFAULT_POLICY
```

Sets the recursion policy for recording objects.

The default policy is deep (which means to recurse an object).

METHODS

record_field

```
virtual function void record_field (string name,
                                   uvm_bitstream_t value,
                                   int size,
                                   uvm_radix_enum radix = UVM_NORADIX)
```

Records an integral field (less than or equal to 4096 bits). *name* is the name of the field.

value is the value of the field to record. *size* is the number of bits of the field which apply. *radix* is the [uvm_radix_enum](#) to use.

record_field_real

```
virtual function void record_field_real (string name,  
                                         real    value)
```

Records an real field. *value* is the value of the field to record.

record_object

```
virtual function void record_object (string      name,  
                                    uvm_object value)
```

Records an object field. *name* is the name of the recorded field.

This method uses the recursion <policy> to determine whether or not to recurse into the object.

record_string

```
virtual function void record_string (string name,  
                                    string value)
```

Records a string field. *name* is the name of the recorded field.

record_time

```
virtual function void record_time (string name,  
                                   time   value)
```

Records a time value. *name* is the name to record to the database.

record_generic

```
virtual function void record_generic (string name,  
                                      string value)
```

Records the name-value pair, where value has been converted to a string, e.g. via
\$psprintf("%<format>",<some variable>);

uvm_packer

The uvm_packer class provides a policy object for packing and unpacking uvm_objects. The policies determine how packing and unpacking should be done. Packing an object causes the object to be placed into a bit (byte or int) array. If the `uvm_field_* macro are used to implement pack and unpack, by default no metadata information is stored for the packing of dynamic objects (strings, arrays, class objects).

Summary

uvm_packer

The uvm_packer class provides a policy object for packing and unpacking uvm_objects.

PACKING

pack_field	Packs an integral value (less than or equal to 4096 bits) into the packed array.
pack_field_int	Packs the integral value (less than or equal to 64 bits) into the pack array.
pack_string	Packs a string value into the pack array.
pack_time	Packs a time <i>value</i> as 64 bits into the pack array.
pack_real	Packs a real <i>value</i> as 64 bits into the pack array.
pack_object	Packs an object value into the pack array.

UNPACKING

is_null	This method is used during unpack operations to peek at the next 4-bit chunk of the pack data and determine if it is 0.
unpack_field_int	Unpacks bits from the pack array and returns the bit-stream that was unpacked.
unpack_field	Unpacks bits from the pack array and returns the bit-stream that was unpacked.
unpack_string	Unpacks a string.
unpack_time	Unpacks the next 64 bits of the pack array and places them into a time variable.
unpack_real	Unpacks the next 64 bits of the pack array and places them into a real variable.
unpack_object	Unpacks an object and stores the result into <i>value</i> .
get_packed_size	Returns the number of bits that were packed.

VARIABLES

physical	This bit provides a filtering mechanism for fields.
abstract	This bit provides a filtering mechanism for fields.
use_metadata	This flag indicates whether to encode metadata when packing dynamic data, or to decode metadata when unpacking.
big_endian	This bit determines the order that integral data is packed (using pack_field, pack_field_int, pack_time, or pack_real) and how the data is unpacked from the pack array (using unpack_field, unpack_field_int, unpack_time, or unpack_real).

PACKING

pack_field

```
virtual function void pack_field (uvm_bitstream_t value,
```

```
int size )
```

Packs an integral value (less than or equal to 4096 bits) into the packed array. *size* is the number of bits of *value* to pack.

[pack_field_int](#)

```
virtual function void pack_field_int (logic[63:0] value,  
int size )
```

Packs the integral value (less than or equal to 64 bits) into the pack array. The *size* is the number of bits to pack, usually obtained by \$bits. This optimized version of [pack_field](#) is useful for sizes up to 64 bits.

[pack_string](#)

```
virtual function void pack_string (string value)
```

Packs a string value into the pack array.

When the metadata flag is set, the packed string is terminated by a null character to mark the end of the string.

This is useful for mixed language communication where unpacking may occur outside of SystemVerilog UVM.

[pack_time](#)

```
virtual function void pack_time (time value)
```

Packs a time *value* as 64 bits into the pack array.

[pack_real](#)

```
virtual function void pack_real (real value)
```

Packs a real *value* as 64 bits into the pack array.

The real *value* is converted to a 6-bit scalar value using the function \$real2bits before it is packed into the array.

[pack_object](#)

```
virtual function void pack_object (uvm_object value)
```

Packs an object value into the pack array.

A 4-bit header is inserted ahead of the string to indicate the number of bits that was packed. If a null object was packed, then this header will be 0.

This is useful for mixed-language communication where unpacking may occur outside of SystemVerilog UVM.

UNPACKING

is_null

```
virtual function bit is_null ()
```

This method is used during unpack operations to peek at the next 4-bit chunk of the pack data and determine if it is 0.

If the next four bits are all 0, then the return value is a 1; otherwise it is 0.

This is useful when unpacking objects, to decide whether a new object needs to be allocated or not.

unpack_field_int

```
virtual function logic[63:0] unpack_field_int (int size)
```

Unpacks bits from the pack array and returns the bit-stream that was unpacked.

size is the number of bits to unpack; the maximum is 64 bits. This is a more efficient variant than `unpack_field` when unpacking into smaller vectors.

unpack_field

```
virtual function uvm_bitstream_t unpack_field (int size)
```

Unpacks bits from the pack array and returns the bit-stream that was unpacked. *size* is the number of bits to unpack; the maximum is 4096 bits.

unpack_string

```
virtual function string unpack_string (int num_chars = -1)
```

Unpacks a string.

num_chars bytes are unpacked into a string. If *num_chars* is -1 then unpacking stops on at the first null character that is encountered.

unpack_time

```
virtual function time unpack_time ()
```

Unpacks the next 64 bits of the pack array and places them into a time variable.

unpack_real

```
virtual function real unpack_real ()
```

Unpacks the next 64 bits of the pack array and places them into a real variable.

The 64 bits of packed data are converted to a real using the \$bits2real system function.

unpack_object

```
virtual function void unpack_object (uvm_object value)
```

Unpacks an object and stores the result into *value*.

value must be an allocated object that has enough space for the data being unpacked. The first four bits of packed data are used to determine if a null object was packed into the array.

The [is_null](#) function can be used to peek at the next four bits in the pack array before calling this method.

get_packed_size

```
virtual function int get_packed_size()
```

Returns the number of bits that were packed.

VARIABLES

physical

```
bit physical = 1
```

This bit provides a filtering mechanism for fields.

The [abstract](#) and physical settings allow an object to distinguish between two different classes of fields. It is up to you, in the [uvm_object::do_pack](#) and [uvm_object::do_unpack](#) methods, to test the setting of this field if you want to use it as a filter.

abstract

```
bit abstract = 0
```

This bit provides a filtering mechanism for fields.

The abstract and physical settings allow an object to distinguish between two different classes of fields. It is up to you, in the [uvm_object::do_pack](#) and [uvm_object::do_unpack](#) routines, to test the setting of this field if you want to use it as a filter.

use_metadata

```
bit use_metadata = 0
```

This flag indicates whether to encode metadata when packing dynamic data, or to decode metadata when unpacking. Implementations of <do_pack> and <do_unpack>

should regard this bit when performing their respective operation. When set, metadata should be encoded as follows:

- For strings, pack an additional null byte after the string is packed.
- For objects, pack 4 bits prior to packing the object itself. Use 4'b0000 to indicate the object being packed is null, otherwise pack 4'b0001 (the remaining 3 bits are reserved).
- For queues, dynamic arrays, and associative arrays, pack 32 bits indicating the size of the array prior to packing individual elements.

big_endian

```
bit big_endian = 1
```

This bit determines the order that integral data is packed (using [pack_field](#), [pack_field_int](#), [pack_time](#), or [pack_real](#)) and how the data is unpacked from the pack array (using [unpack_field](#), [unpack_field_int](#), [unpack_time](#), or [unpack_real](#)). When the bit is set, data is associated msb to lsb; otherwise, it is associated lsb to msb.

The following code illustrates how data can be associated msb to lsb and lsb to msb:

```
class mydata extends uvm_object;
    logic[15:0] value = 'h1234;

    function void do_pack (uvm_packer packer);
        packer.pack_field_int(value, 16);
    endfunction

    function void do_unpack (uvm_packer packer);
        value = packer.unpack_field_int(16);
    endfunction
endclass

mydata d = new;
bit bits[];

initial begin
    d.pack(bits); // 'b0001001000110100
    uvm_default_packer.big_endian = 0;
    d.pack(bits); // 'b0010110001001000
end
```

TLM Interfaces, Ports, and Exports

The UVM TLM library defines several abstract, transaction-level interfaces and the ports and exports that facilitate their use. Each TLM interface consists of one or more methods used to transport data, typically whole transactions (objects) at a time. Component designs that use TLM ports and exports to communicate are inherently more reusable, interoperable, and modular.

Interface Overview

The TLM standard specifies the required behavior (semantic) of each interface method. Classes (components) that implement a TLM interface must meet the specified semantic.

Each TLM interface is either blocking, non-blocking, or a combination of these two.

<i>blocking</i>	A blocking interface conveys transactions in blocking fashion; its methods do not return until the transaction has been successfully sent or retrieved. Because delivery may consume time to complete, the methods in such an interface are declared as tasks.
<i>non-blocking</i>	A non-blocking interface attempts to convey a transaction without consuming simulation time. Its methods are declared as functions. Because delivery may fail (e.g. the target component is busy and can not accept the request), the methods may return with failed status.
<i>combination</i>	A combination interface contains both the blocking and non-blocking variants. In SystemC, combination interfaces are defined through multiple inheritance. Because SystemVerilog does not support multiple inheritance, the UVM emulates hierarchical interfaces via a common base class and interface mask.

Like their SystemC counterparts, the UVM's TLM port and export implementations allow connections between ports whose interfaces are not an exact match. For example, an *uvm_blocking_get_port* can be connected to any port, export or imp port that provides *at the least* an implementation of the *blocking_get* interface, which includes the *uvm_get_** ports and exports, *uvm_blocking_get_peek_** ports and exports, and *uvm_get_peek_** ports and exports.

The sections below provide an overview of the unidirectional and bidirectional TLM interfaces, ports, and exports.

Summary

TLM Interfaces, Ports, and Exports

The UVM TLM library defines several abstract, transaction-level interfaces and the ports and exports that facilitate their use.

UNIDIRECTIONAL INTERFACES & PORTS

The unidirectional TLM interfaces consist of blocking, non-blocking, and combined blocking and non-blocking variants of the *put*, *get* and *peek* interfaces, plus a non-blocking *analysis* interface.

Put

The *put* interfaces are used to send, or *put*, transactions to other components.

Get and Peek

The *get* interfaces are used to retrieve transactions from other components.

Analysis

The *analysis* interface is used to perform non-blocking broadcasts of transactions to connected components.

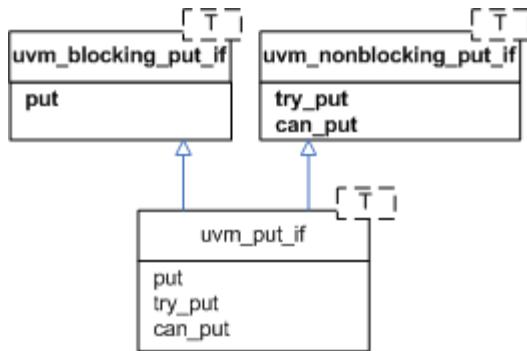
Ports, Exports, and Imps	The UVM provides unidirectional ports, exports, and implementation ports for connecting your components via the TLM interfaces.
BIDIRECTIONAL INTERFACES & PORTS	The bidirectional interfaces consist of blocking, non-blocking, and combined blocking and non-blocking variants of the <i>transport</i> , <i>master</i> , and <i>slave</i> interfaces.
Transport	The <i>transport</i> interface sends a request transaction and returns a response transaction in a single task call, thereby enforcing an in-order execution semantic.
Master and Slave	The primitive, unidirectional <i>put</i> , <i>get</i> , and <i>peek</i> interfaces are combined to form bidirectional master and slave interfaces.
Ports, Exports, and Imps	The UVM provides bidirectional ports, exports, and implementation ports for connecting your components via the TLM interfaces.
USAGE	This example illustrates basic TLM connectivity using the blocking put interface.

UNIDIRECTIONAL INTERFACES & PORTS

The unidirectional TLM interfaces consist of blocking, non-blocking, and combined blocking and non-blocking variants of the *put*, *get* and *peek* interfaces, plus a non-blocking *analysis* interface.

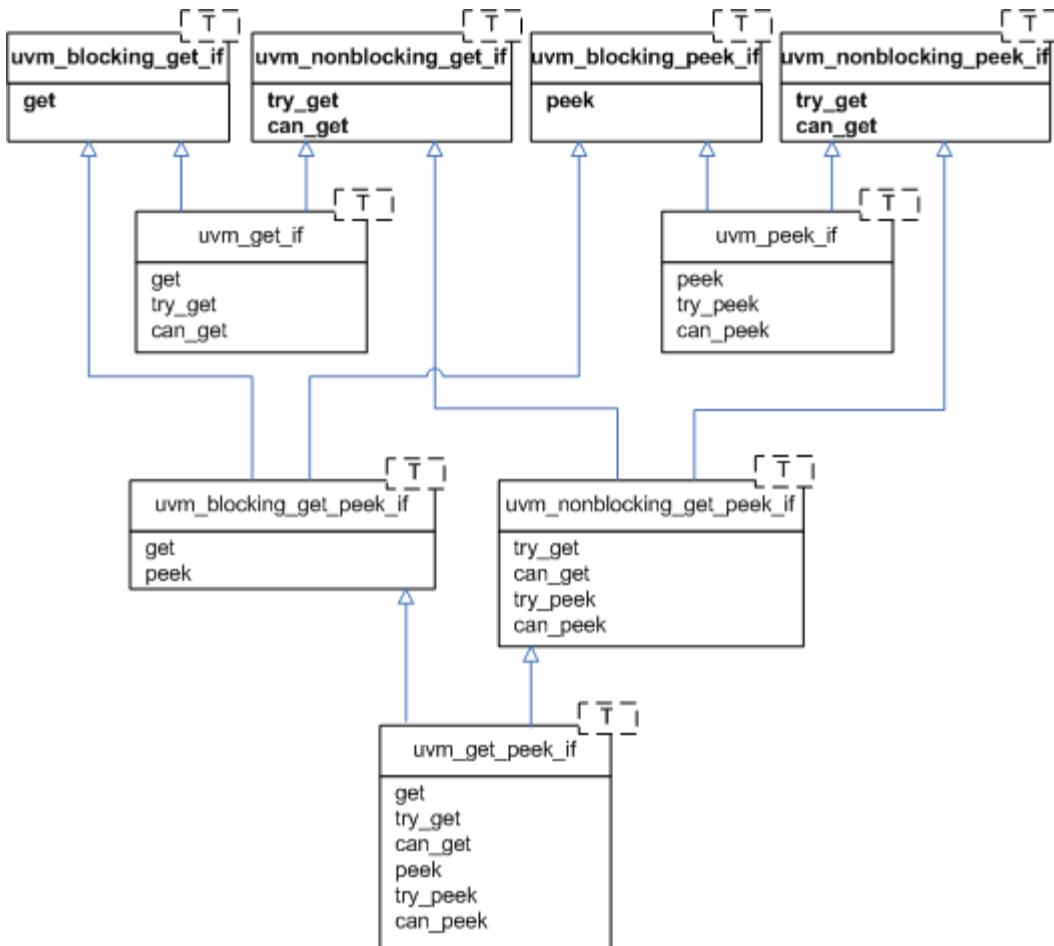
Put

The *put* interfaces are used to send, or *put*, transactions to other components. Successful completion of a *put* guarantees its delivery, not execution.



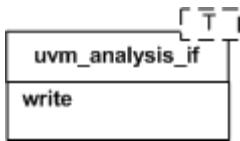
Get and Peek

The *get* interfaces are used to retrieve transactions from other components. The *peek* interfaces are used for the same purpose, except the retrieved transaction is not consumed; successive calls to *peek* will return the same object. Combined *get_peek* interfaces are also defined.



Analysis

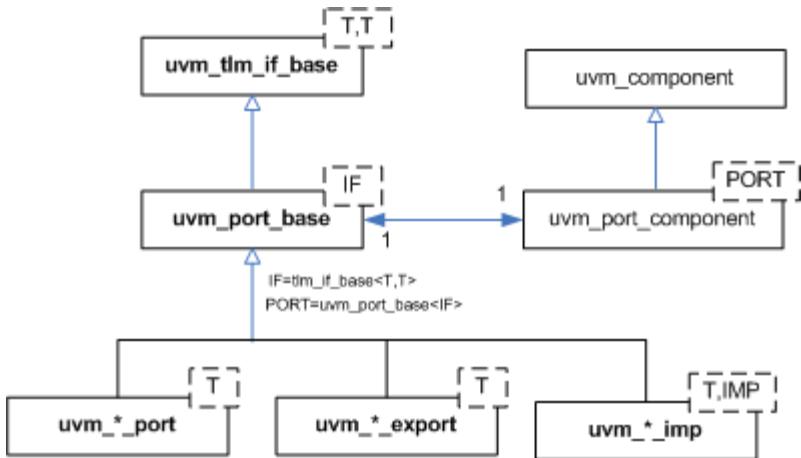
The *analysis* interface is used to perform non-blocking broadcasts of transactions to connected components. It is typically used by such components as monitors to publish transactions observed on a bus to its subscribers, which are typically scoreboards and response/coverage collectors.



Ports, Exports, and Imps

The UVM provides unidirectional ports, exports, and implementation ports for connecting your components via the TLM interfaces.

- | | |
|----------------|---|
| <i>Ports</i> | instantiated in components that <i>require</i> , or <i>use</i> , the associate interface to initiate transaction requests. |
| <i>Exports</i> | instantiated by components that <i>forward</i> an implementation of the methods defined in the associated interface. The implementation is typically provided by an <i>imp</i> port in a child component. |
| <i>Imps</i> | instantiated by components that <i>provide</i> or <i>implement</i> an implementation of the methods defined in the associated interface. |



A summary of port, export, and imp declarations are

```

class uvm_*_export #(type T=int)
  extends uvm_port_base #(tlm_if_base #(T,T));
class uvm_*_port #(type T=int)
  extends uvm_port_base #(tlm_if_base #(T,T));
class uvm_*_imp #(type T=int)
  extends uvm_port_base #(tlm_if_base #(T,T));

```

where the asterisk can be any of

```

blocking_put
nonblocking_put
put

blocking_get
nonblocking_get
get

blocking_peek
nonblocking_peek
peek

blocking_get_peek
nonblocking_get_peek
get_peek

analysis

```

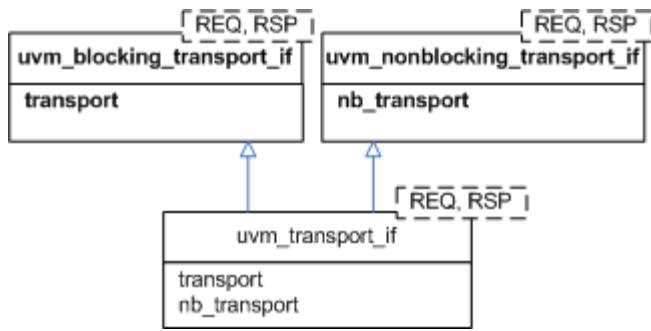
BIDIRECTIONAL INTERFACES & PORTS

The bidirectional interfaces consist of blocking, non-blocking, and combined blocking and non-blocking variants of the *transport*, *master*, and *slave* interfaces.

Bidirectional interfaces involve both a transaction request and response.

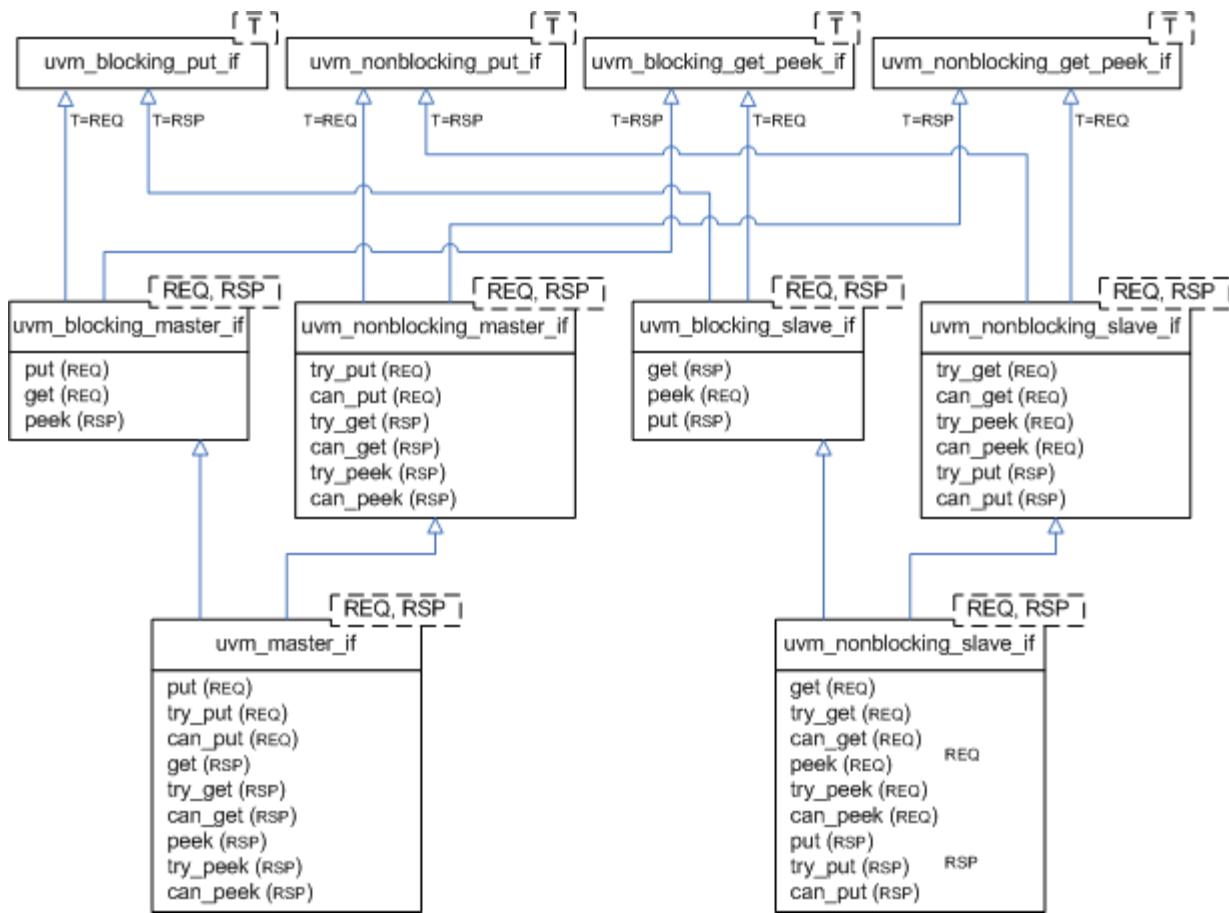
Transport

The *transport* interface sends a request transaction and returns a response transaction in a single task call, thereby enforcing an in-order execution semantic. The request and response transactions can be different types.



Master and Slave

The primitive, unidirectional *put*, *get*, and *peek* interfaces are combined to form bidirectional master and slave interfaces. The master puts requests and gets or peeks responses. The slave gets or peeks requests and puts responses. Because the *put* and the *get* come from different function interface methods, the requests and responses are not coupled as they are with the *transport* interface.



Ports, Exports, and Imps

The UVM provides bidirectional ports, exports, and implementation ports for connecting your components via the TLM interfaces.

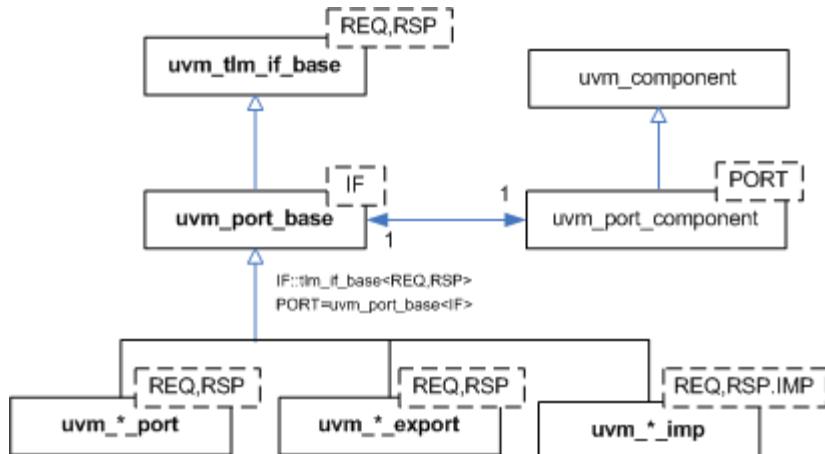
Ports instantiated in components that *require*, or *use*, the associate interface to initiate transaction requests.

Exports instantiated by components that *forward* an implementation of the methods defined in the associated interface. The implementation is

typically provided by an *imp* port in a child component.

Imps

instantiated by components that *provide* or *implement* an implementation of the methods defined in the associated interface.



A summary of port, export, and imp declarations are

```
class uvm_*_port #(type REQ=int, RSP=int)
    extends uvm_port_base #(tlm_if_base #(REQ, RSP));
class uvm_*_export #(type REQ=int, RSP=int)
    extends uvm_port_base #(tlm_if_base #(REQ, RSP));
class uvm_*_imp #(type REQ=int, RSP=int)
    extends uvm_port_base #(tlm_if_base #(REQ, RSP));
```

where the asterisk can be any of

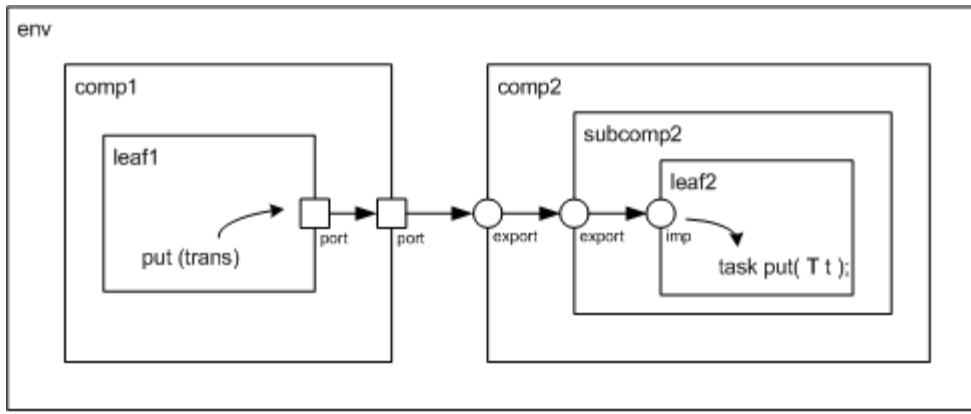
```
transport
blocking_transport
nonblocking_transport

blocking_master
nonblocking_master
master

blocking_slave
nonblocking_slave
slave
```

USAGE

This example illustrates basic TLM connectivity using the blocking put interface.



= port = export / imp = analysis port

- port-to-port* leaf1's *out* port is connected to its parent's (comp1) *out* port
- port-to-export* comp1's *out* port is connected to comp2's *in* export
- export-to-export* comp2's *in* export is connected to its child's (subcomp2) *in* export
- export-to-imp* subcomp2's *in* export is connected leaf2's *in* imp port.
- imp-to-implementation* leaf2's *in* imp port is connected to its implementation, leaf2

Hierarchical port connections are resolved and optimized just before the `uvm_component::end_of_elaboration` phase. After optimization, calling any port's interface method (e.g. `leaf1.out.put(trans)`) incurs a single hop to get to the implementation (e.g. `leaf2's put task`), no matter how far up and down the hierarchy the implementation resides.

```

`include "uvm_pkg.sv"
import uvm_pkg::*;

class trans extends uvm_transaction;
  rand int addr;
  rand int data;
  rand bit write;
endclass

class leaf1 extends uvm_component;
  `uvm_component_utils(leaf1)
  uvm_blocking_put_port #(trans) out;
  function new(string name, uvm_component parent=null);
    super.new(name,parent);
    out = new("out",this);
  endfunction
  virtual task run();
    trans t;
    t = new;
    t.randomize();
    out.put(t);
  endtask
endclass

class comp1 extends uvm_component;
  `uvm_component_utils(comp1)
  uvm_blocking_put_port #(trans) out;
  leaf1 leaf;
  function new(string name, uvm_component parent=null);
    super.new(name,parent);
  endfunction
endclass

```

```

virtual function void build();
    out = new("out",this);
    leaf = new("leaf1",this);
endfunction

// connect port to port
virtual function void connect();
    leaf.out.connect(out);
endfunction

endclass

class leaf2 extends uvm_component;
`uvm_component_utils(leaf2)
uvm_blocking_put_imp #(trans,leaf2) in;

function new(string name, uvm_component parent=null);
    super.new(name,parent);
    // connect imp to implementation (this)
    in = new("in",this);
endfunction

virtual task put(trans t);
    $display("Got trans: addr=%0d, data=%0d, write=%0d",
        t.addr, t.data, t.write);
endtask

endclass

class subcomp2 extends uvm_component;
`uvm_component_utils(subcomp2)
uvm_blocking_put_export #(trans) in;
leaf2 leaf;

function new(string name, uvm_component parent=null);
    super.new(name,parent);
endfunction

virtual function void build();
    in = new("in",this);
    leaf = new("leaf2",this);
endfunction

// connect export to imp
virtual function void connect();
    in.connect(leaf.in);
endfunction

endclass

class comp2 extends uvm_component;
`uvm_component_utils(comp2)
uvm_blocking_put_export #(trans) in;
subcomp2 subcomp;

function new(string name, uvm_component parent=null);
    super.new(name,parent);
endfunction

virtual function void build();
    in = new("in",this);
    subcomp = new("subcomp2",this);
endfunction

// connect export to export
virtual function void connect();
    in.connect(subcomp.in);
endfunction

endclass

class env extends uvm_component;
`uvm_component_utils(compl)
compl compl_i;
comp2 comp2_i;

function new(string name, uvm_component parent=null);
    super.new(name,parent);
endfunction

virtual function void build();
    compl_i = new("compl",this);
    comp2_i = new("comp2",this);

```

```
endfunction

// connect port to export
virtual function void connect();
    compl_i.out.connect(comp2_i.in);
endfunction

endclass

module top;
    env e = new("env");
    initial run_test();
    initial #10 uvm_top.stop_request();
endmodule
```

uvm_tlm_if_base #(T1,T2)

This class declares all of the methods of the TLM API.

Various subsets of these methods are combined to form primitive TLM interfaces, which are then paired in various ways to form more abstract “combination” TLM interfaces. Components that require a particular interface use ports to convey that requirement. Components that provide a particular interface use exports to convey its availability.

Communication between components is established by connecting ports to compatible exports, much like connecting module signal-level output ports to compatible input ports. The difference is that UVM ports and exports bind interfaces (groups of methods), not signals and wires. The methods of the interfaces so bound pass data as whole transactions (e.g. objects). The set of primitive and combination TLM interfaces afford many choices for designing components that communicate at the transaction level.

Summary

uvm_tlm_if_base #(T1,T2)

This class declares all of the methods of the TLM API.

CLASS DECLARATION

```
virtual class uvm_tlm_if_base #(type T1 = int,  
                           type T2 = int )
```

BLOCKING PUT

`put` Sends a user-defined transaction of type T.

BLOCKING GET

`get` Provides a new transaction of type T.

BLOCKING PEEK

`peek` Obtain a new transaction without consuming it.

NON-BLOCKING

PUT

`try_put` Sends a transaction of type T, if possible.

`can_put` Returns 1 if the component is ready to accept the transaction; 0 otherwise.

NON-BLOCKING

GET

`try_get` Provides a new transaction of type T.

`can_get` Returns 1 if a new transaction can be provided immediately upon request, 0 otherwise.

NON-BLOCKING

PEEK

`try_peek` Provides a new transaction without consuming it.

`can_peek` Returns 1 if a new transaction is available; 0 otherwise.

BLOCKING

TRANSPORT

`transport` Executes the given request and returns the response in the given output argument.

NON-BLOCKING

TRANSPORT

`nb_transport` Executes the given request and returns the response in the given output argument.

ANALYSIS

`write`

Broadcasts a user-defined transaction of type T to any number of listeners.

BLOCKING PUT

put

```
virtual task put(input T1 t)
```

Sends a user-defined transaction of type T.

Components implementing the put method will block the calling thread if it cannot immediately accept delivery of the transaction.

BLOCKING GET

get

```
virtual task get(output T2 t)
```

Provides a new transaction of type T.

The calling thread is blocked if the requested transaction cannot be provided immediately. The new transaction is returned in the provided output argument.

The implementation of get must regard the transaction as consumed. Subsequent calls to get must return a different transaction instance.

BLOCKING PEEK

peek

```
virtual task peek(output T2 t)
```

Obtain a new transaction without consuming it.

If a transaction is available, then it is written to the provided output argument. If a transaction is not available, then the calling thread is blocked until one is available.

The returned transaction is not consumed. A subsequent peek or get will return the same transaction.

NON-BLOCKING PUT

try_put

```
virtual function bit try_put(input T1 t)
```

Sends a transaction of type T, if possible.

If the component is ready to accept the transaction argument, then it does so and returns 1, otherwise it returns 0.

can_put

```
virtual function bit can_put()
```

Returns 1 if the component is ready to accept the transaction; 0 otherwise.

NON-BLOCKING GET

try_get

```
virtual function bit try_get(output T2 t)
```

Provides a new transaction of type T.

If a transaction is immediately available, then it is written to the output argument and 1 is returned. Otherwise, the output argument is not modified and 0 is returned.

can_get

```
virtual function bit can_get()
```

Returns 1 if a new transaction can be provided immediately upon request, 0 otherwise.

NON-BLOCKING PEEK

try_peek

```
virtual function bit try_peek(output T2 t)
```

Provides a new transaction without consuming it.

If available, a transaction is written to the output argument and 1 is returned. A subsequent peek or get will return the same transaction. If a transaction is not available, then the argument is unmodified and 0 is returned.

can_peek

```
virtual function bit can_peek()
```

Returns 1 if a new transaction is available; 0 otherwise.

BLOCKING TRANSPORT

transport

```
virtual task transport(input T1 req ,  
                      output T2 rsp)
```

Executes the given request and returns the response in the given output argument. The calling thread may block until the operation is complete.

NON-BLOCKING TRANSPORT

nb_transport

```
virtual function bit nb_transport( input T1 req,  
                                  output T2 rsp )
```

Executes the given request and returns the response in the given output argument. Completion of this operation must occur without blocking.

If for any reason the operation could not be executed immediately, then a 0 must be returned; otherwise 1.

ANALYSIS

write

```
virtual function void write(input T1 t)
```

Broadcasts a user-defined transaction of type T to any number of listeners. The operation must complete without blocking.

uvm_tlm_fifo_base #(T)

This class is the base for [uvm_tlm_fifo #\(T\)](#). It defines the TLM exports through which all transaction-based FIFO operations occur. It also defines default implementations for each interface method provided by these exports.

The interface methods provided by the [put_export](#) and the [get_peek_export](#) are defined and described by [uvm_tlm_if_base #\(T1,T2\)](#). See the TLM Overview section for a general discussion of TLM interface definition and usage.

Parameter type

T The type of transactions to be stored by this FIFO.

Summary

uvm_tlm_fifo_base #(T)

This class is the base for [uvm_tlm_fifo #\(T\)](#).

CLASS HIERARCHY



CLASS DECLARATION

```
virtual class uvm_tlm_fifo_base #(  
    type T = int  
) extends uvm_component
```

PORTS

put_export	The put_export provides both the blocking and non-blocking put interface methods to any attached port:
get_peek_export	The get_peek_export provides all the blocking and non-blocking get and peek interface methods:
put_ap	Transactions passed via put or try_put (via any port connected to the put_export) are sent out this port via its write method.
get_ap	Transactions passed via get , try_get , peek , or try_peek (via any port connected to the get_peek_export) are sent out this port via its write method.

METHODS

new	The <i>name</i> and <i>parent</i> are the normal uvm_component constructor arguments.
---------------------	---

PORTS

[put_export](#)

The [put_export](#) provides both the blocking and non-blocking put interface methods to

any attached port:

```
task put (input T t)
function bit can_put ()
function bit try_put (input T t)
```

Any *put* port variant can connect and send transactions to the FIFO via this export, provided the transaction types match. See [uvm_tlm_if_base #\(T1,T2\)](#) for more information on each of the above interface methods.

get_peek_export

The *get_peek_export* provides all the blocking and non-blocking get and peek interface methods:

```
task get (output T t)
function bit can_get ()
function bit try_get (output T t)
task peek (output T t)
function bit can_peek ()
function bit try_peek (output T t)
```

Any *get* or *peek* port variant can connect to and retrieve transactions from the FIFO via this export, provided the transaction types match. See [uvm_tlm_if_base #\(T1,T2\)](#) for more information on each of the above interface methods.

put_ap

Transactions passed via *put* or *try_put* (via any port connected to the *put_export*) are sent out this port via its *write* method.

```
function void write (T t)
```

All connected analysis exports and imps will receive put transactions. See [uvm_tlm_if_base #\(T1,T2\)](#) for more information on the *write* interface method.

get_ap

Transactions passed via *get*, *try_get*, *peek*, or *try_peek* (via any port connected to the *get_peek_export*) are sent out this port via its *write* method.

```
function void write (T t)
```

All connected analysis exports and imps will receive get transactions. See [uvm_tlm_if_base #\(T1,T2\)](#) for more information on the *write* method.

METHODS

new

```
function new(string      name,  
            uvm_component parent = null)
```

The *name* and *parent* are the normal uvm_component constructor arguments. The *parent* should be null if the uvm_tlm_fifo is going to be used in a statically elaborated construct (e.g., a module). The *size* indicates the maximum size of the FIFO. A value of zero indicates no upper bound.

uvm_tlm_fifo #(T)

This class provides storage of transactions between two independently running processes. Transactions are put into the FIFO via the *put_export*. transactions are fetched from the FIFO in the order they arrived via the *get_peek_export*. The *put_export* and *get_peek_export* are inherited from the [uvm_tlm_fifo_base #\(T\)](#) super class, and the interface methods provided by these exports are defined by the [uvm_tlm_if_base #\(T1,T2\)](#) class.

Summary

uvm_tlm_fifo #(T)

This class provides storage of transactions between two independently running processes.

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_tlm_fifo #(  
    type T = int  
) extends uvm_tlm_fifo_base #(T)
```

METHODS

new	The <i>name</i> and <i>parent</i> are the normal uvm_component constructor arguments.
size	Returns the capacity of the FIFO-- that is, the number of entries the FIFO is capable of holding.
used	Returns the number of entries put into the FIFO.
is_empty	Returns 1 when there are no entries in the FIFO, 0 otherwise.
is_full	Returns 1 when the number of entries in the FIFO is equal to its size , 0 otherwise.
flush	Removes all entries from the FIFO, after which used returns 0 and is_empty returns 1.

METHODS

new

```
function new(string name,  
            uvm_component parent = null,  
            int size = 1 )
```

The *name* and *parent* are the normal uvm_component constructor arguments. The *parent* should be null if the uvm_tlm_fifo is going to be used in a statically elaborated construct (e.g., a module). The *size* indicates the maximum size of the FIFO; a value of

zero indicates no upper bound.

size

```
virtual function int size()
```

Returns the capacity of the FIFO-- that is, the number of entries the FIFO is capable of holding. A return value of 0 indicates the FIFO capacity has no limit.

used

```
virtual function int used()
```

Returns the number of entries put into the FIFO.

is_empty

```
virtual function bit is_empty()
```

Returns 1 when there are no entries in the FIFO, 0 otherwise.

is_full

```
virtual function bit is_full()
```

Returns 1 when the number of entries in the FIFO is equal to its [size](#), 0 otherwise.

flush

```
virtual function void flush()
```

Removes all entries from the FIFO, after which [used](#) returns 0 and [is_empty](#) returns 1.

uvm_tlm_analysis_fifo #(T)

An analysis_fifo is a uvm_tlm_fifo with an unbounded size and a write interface. It can be used anywhere an <uvm_subscriber #(T)> is used. Typical usage is as a buffer between an analysis_port in a monitor and an analysis component (e.g., a component derived from uvm_subscriber).

Summary

uvm_tlm_analysis_fifo #(T)

An analysis_fifo is a uvm_tlm_fifo with an unbounded size and a write interface.

CLASS HIERARCHY

```
uvm_void
```

uvm_object
uvm_report_object
uvm_component
uvm_tlm_fifo_base #(T)
uvm_tlm_fifo #(T)
uvm_tlm_analysis_fifo #(T)

CLASS DECLARATION

```
class uvm_tlm_analysis_fifo #(  
    type T = int  
) extends uvm_tlm_fifo #(T)
```

PORTS

[analysis_port #\(T\)](#) The analysis_export provides the write method to all connected analysis ports and parent exports:

METHODS

[new](#) This is the standard uvm_component constructor.

PORTS

[analysis_port #\(T\)](#)

The analysis_export provides the write method to all connected analysis ports and parent exports:

```
function void write (T t)
```

Access via ports bound to this export is the normal mechanism for writing to an analysis FIFO. See write method of [uvm_tlm_if_base #\(T1,T2\)](#) for more information.

METHODS

[new](#)

```
function new(string name,  
           uvm_component parent = null)
```

This is the standard uvm_component constructor. *name* is the local name of this component. The *parent* should be left unspecified when this component is instantiated in statically elaborated constructs and must be specified when this component is a child of another UVM component.

uvm_tlm_req_rsp_channel #(REQ,RSP)

The uvm_tlm_req_rsp_channel contains a request FIFO of type *REQ* and a response FIFO of type *RSP*. These FIFOs can be of any size. This channel is particularly useful for dealing with pipelined protocols where the request and response are not tightly coupled.

Type parameters

REQ Type of the request transactions conveyed by this channel.

RSP Type of the reponse transactions conveyed by this channel.

Summary

uvm_tlm_req_rsp_channel #(REQ,RSP)

The uvm_tlm_req_rsp_channel contains a request FIFO of type *REQ* and a response FIFO of type *RSP*.

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_tlm_req_rsp_channel #(  
    type REQ = int,  
    type RSP = REQ  
) extends uvm_component
```

PORTS

put_request_export	The put_export provides both the blocking and non-blocking put interface methods to the request FIFO:
get_peek_response_export	The get_peek_response_export provides all the blocking and non-blocking get and peek interface methods to the response FIFO:
get_peek_request_export	The get_peek_export provides all the blocking and non-blocking get and peek interface methods to the response FIFO:
put_response_export	The put_export provides both the blocking and non-blocking put interface methods to the response FIFO:
request_ap	Transactions passed via put or try_put (via any port connected to the put_request_export) are sent out this port via its write method.
response_ap	Transactions passed via put or try_put (via any port connected to the put_response_export) are sent out this port via its write method.
master_export	Exports a single interface that allows a master to put requests and get or peek responses.
slave_export	Exports a single interface that allows a slave to get or peek requests and to put responses.

METHODS

new The *name* and *parent* are the standard **uvm_component** constructor arguments.

PORTS

put_request_export

The put_export provides both the blocking and non-blocking put interface methods to the request FIFO:

```
task put (input T t);
function bit can_put ();
function bit try_put (input T t);
```

Any put port variant can connect and send transactions to the request FIFO via this export, provided the transaction types match.

get_peek_response_export

The get_peek_response_export provides all the blocking and non-blocking get and peek interface methods to the response FIFO:

```
task get (output T t);
function bit can_get ();
function bit try_get (output T t);
task peek (output T t);
function bit can_peek ();
function bit try_peek (output T t);
```

Any get or peek port variant can connect to and retrieve transactions from the response FIFO via this export, provided the transaction types match.

get_peek_request_export

The get_peek_export provides all the blocking and non-blocking get and peek interface methods to the response FIFO:

```
task get (output T t);
function bit can_get ();
function bit try_get (output T t);
task peek (output T t);
function bit can_peek ();
function bit try_peek (output T t);
```

Any get or peek port variant can connect to and retrieve transactions from the response FIFO via this export, provided the transaction types match.

put_response_export

The put_export provides both the blocking and non-blocking put interface methods to the response FIFO:

```
task put (input T t);
function bit can_put ();
function bit try_put (input T t);
```

Any put port variant can connect and send transactions to the response FIFO via this export, provided the transaction types match.

[request_ap](#)

Transactions passed via put or try_put (via any port connected to the put_request_export) are sent out this port via its write method.

```
function void write (T t);
```

All connected analysis exports and imps will receive these transactions.

[response_ap](#)

Transactions passed via put or try_put (via any port connected to the put_response_export) are sent out this port via its write method.

```
function void write (T t);
```

All connected analysis exports and imps will receive these transactions.

[master_export](#)

Exports a single interface that allows a master to put requests and get or peek responses. It is a combination of the put_request_export and get_peek_response_export.

[slave_export](#)

Exports a single interface that allows a slave to get or peek requests and to put responses. It is a combination of the get_peek_request_export and put_response_export.

[METHODS](#)

[new](#)

```
function new (string name,
              uvm_component parent = null,
              int request_fifo_size = 1,
```

```
int response_fifo_size = 1 )
```

The *name* and *parent* are the standard [uvm_component](#) constructor arguments. The *parent* must be null if this component is defined within a static component such as a module, program block, or interface. The last two arguments specify the request and response FIFO sizes, which have default values of 1.

uvm_tlm_transport_channel #(REQ,RSP)

A uvm_tlm_transport_channel is a [uvm_tlm_req_rsp_channel #\(REQ,RSP\)](#) that implements the transport interface. It is useful when modeling a non-pipelined bus at the transaction level. Because the requests and responses have a tightly coupled one-to-one relationship, the request and response FIFO sizes are both set to one.

Summary

uvm_tlm_transport_channel #(REQ,RSP)

A uvm_tlm_transport_channel is a [uvm_tlm_req_rsp_channel #\(REQ,RSP\)](#) that implements the transport interface.

CLASS HIERARCHY

```
uvm_void
uvm_object
uvm_report_object
uvm_component
uvm_tlm_req_rsp_channel#(REQ,RSP)
uvm_tlm_transport_channel#(REQ,RSP)
```

CLASS DECLARATION

```
class uvm_tlm_transport_channel #( 
    type REQ = int,
    type RSP = REQ
) extends uvm_tlm_req_rsp_channel #(REQ, RSP)
```

PORTS

[transport_export](#) The `put_export` provides both the blocking and non-blocking transport interface methods to the response FIFO:

METHODS

[new](#) The *name* and *parent* are the standard [uvm_component](#) constructor arguments.

PORTS

[transport_export](#)

The `put_export` provides both the blocking and non-blocking transport interface methods to the response FIFO:

```
task transport(REQ request, output RSP response);  
function bit nb_transport(REQ request, output RSP response);
```

Any transport port variant can connect to and send requests and retrieve responses via this export, provided the transaction types match. Upon return, the response argument carries the response to the request.

METHODS

new

```
function new (string name,  
             uvm_component parent = null)
```

The *name* and *parent* are the standard [uvm_component](#) constructor arguments. The *parent* must be null if this component is defined within a statically elaborated construct such as a module, program block, or interface.

uvm_*_export #(T)

The unidirectional uvm_*_export is a port that *forwards* or *promotes* an interface implementation from a child component to its parent. An export can be connected to any compatible child export or imp port. It must ultimately be connected to at least one implementation of its associated interface.

The interface type represented by the asterisk is any of the following

```
blocking_put  
nonblocking_put  
put  
  
blocking_get  
nonblocking_get  
get  
  
blocking_peek  
nonblocking_peek  
peek  
  
blocking_get_peek  
nonblocking_get_peek  
get_peek  
  
analysis
```

Type parameters

T The type of transaction to be communicated by the export

Exports are connected to interface implementations directly via [uvm_*_imp #\(T,IMP\)](#) ports or indirectly via other [uvm_*_export #\(T\)](#) exports.

Summary

uvm_*_export #(T)

The unidirectional uvm_*_export is a port that *forwards* or *promotes* an interface implementation from a child component to its parent.

METHODS

[new](#) The *name* and *parent* are the standard [uvm_component](#) constructor arguments.

METHODS

[new](#)

The *name* and *parent* are the standard [uvm_component](#) constructor arguments. The *min_size* and *max_size* specify the minimum and maximum number of interfaces that must have been supplied to this port by the end of elaboration.

```
function new (string name,  
             uvm_component parent,  
             int min_size=1,  
             int max_size=1)
```

uvm_*_export #(REQ,RSP)

The bidirectional uvm_*_export is a port that *forwards* or *promotes* an interface implementation from a child component to its parent. An export can be connected to any compatible child export or imp port. It must ultimately be connected to at least one implementation of its associated interface.

The interface type represented by the asterisk is any of the following

```
blocking_transport  
nonblocking_transport  
transport  
  
blocking_master  
nonblocking_master  
master  
  
blocking_slave  
nonblocking_slave  
slave
```

Type parameters

- REQ* The type of request transaction to be communicated by the export
RSP The type of response transaction to be communicated by the export

Exports are connected to interface implementations directly via <uvm_*_imp #(REQ,RSP,IMP)> ports or indirectly via other [uvm_*_export #\(REQ,RSP\)](#) exports.

Summary

uvm_*_export #(REQ,RSP)

The bidirectional uvm_*_export is a port that *forwards* or *promotes* an interface implementation from a child component to its parent.

METHODS

- [new](#) The *name* and *parent* are the standard [uvm_component](#) constructor arguments.

METHODS

[new](#)

The *name* and *parent* are the standard [uvm_component](#) constructor arguments. The *min_size* and *max_size* specify the minimum and maximum number of interfaces that must have been supplied to this port by the end of elaboration.

```
function new (string name,  
            uvm_component parent,  
            int min_size=1,
```

```
int max_size=1)
```

uvm_*_imp ports

This page documents the following port classes

- `uvm_*_imp #(T,IMP)` - unidirectional implementation ports
- `uvm_*_imp #(REQ, RSP, IMP, REQ_IMP, RSP_IMP)` - bidirectional implementation ports

Summary

uvm_*_imp ports

This page documents the following port classes

uvm_*_imp #(T,IMP)

Unidirectional implementation (imp) port classes--An imp port provides access to an implementation of the associated interface to all connected *ports* and *exports*. Each imp port instance *must* be connected to the component instance that implements the associated interface, typically the imp port's parent. All other connections-- e.g. to other ports and exports-- are prohibited.

The asterisk in *uvm_*_imp* may be any of the following

```
blocking_put  
nonblocking_put  
put  
  
blocking_get  
nonblocking_get  
get  
  
blocking_peek  
nonblocking_peek  
peek  
  
blocking_get_peek  
nonblocking_get_peek  
get_peek  
  
analysis
```

Type parameters

- T* The type of transaction to be communicated by the imp
IMP The type of the component implementing the interface. That is, the class to which this imp will delegate.

The interface methods are implemented in a component of type *IMP*, a handle to which is passed in a constructor argument. The imp port delegates all interface calls to this component.

Summary

uvm_*_imp #(T,IMP)

Unidirectional implementation (imp) port classes--An imp port provides access to an implementation of the associated interface to all connected *ports* and *exports*.

METHODS

<code>new</code>	Creates a new unidirectional imp port with the given <i>name</i> and <i>parent</i> .
------------------	--

METHODS

new

Creates a new unidirectional imp port with the given *name* and *parent*. The *parent* must implement the interface associated with this port. Its type must be the type specified in the imp's type-parameter, *IMP*.

```
function new (string name, IMP parent);
```

uvm_*_imp #(REQ, RSP, IMP, REQ_IMP, RSP_IMP)

Bidirectional implementation (imp) port classes--An imp port provides access to an implementation of the associated interface to all connected *ports* and *exports*. Each imp port instance *must* be connected to the component instance that implements the associated interface, typically the imp port's parent. All other connections-- e.g. to other ports and exports-- are prohibited.

The interface represented by the asterisk is any of the following

```
blocking_transport  
nonblocking_transport  
transport  
  
blocking_master  
nonblocking_master  
master  
  
blocking_slave  
nonblocking_slave  
slave
```

Type parameters

<i>REQ</i>	Request transaction type
<i>RSP</i>	Response transaction type
<i>IMP</i>	Component type that implements the interface methods, typically the the parent of this imp port.
<i>REQ_IMP</i>	Component type that implements the request side of the interface. Defaults to IMP. For master and slave imps only.
<i>RSP_IMP</i>	Component type that implements the response side of the interface. Defaults to IMP. For master and slave imps only.

The interface methods are implemented in a component of type *IMP*, a handle to which is passed in a constructor argument. The *imp* port delegates all interface calls to this component.

The master and slave imps have two modes of operation.

- A single component of type *IMP* implements the entire interface for both requests and responses.
- Two sibling components of type *REQ_IMP* and *RSP_IMP* implement the request and response interfaces, respectively. In this case, the *IMP* parent instantiates this *imp* port *and* the *REQ_IMP* and *RSP_IMP* components.

The second mode is needed when a component instantiates more than one *imp* port, as in the [uvm_tlm_req_rsp_channel #\(REQ,RSP\)](#) channel.

Summary

uvm_*_imp #(REQ, RSP, IMP, REQ_IMP, RSP_IMP)

Bidirectional implementation (*imp*) port classes--An *imp* port provides access to an implementation of the associated interface to all connected *ports* and *exports*.

METHODS

new Creates a new bidirectional *imp* port with the given *name* and *parent*.

METHODS

new

Creates a new bidirectional *imp* port with the given *name* and *parent*. The *parent*, whose type is specified by *IMP* type parameter, must implement the interface associated with this port.

Transport *imp* constructor

```
function new(string name, IMP imp)
```

Master and slave *imp* constructor

The optional *req_imp* and *rsp_imp* arguments, available to master and slave *imp* ports, allow the requests and responses to be handled by different subcomponents. If they are specified, they must point to the underlying component that implements the request and response methods, respectively.

```
function new(string name, IMP imp,
            REQ_IMP req_imp=imp, RSP_IMP rsp_imp=imp)
```

uvm_*_port #(T)

These unidirectional ports are instantiated by components that *require*, or *use*, the associated interface to convey transactions. A port can be connected to any compatible port, export, or imp port. Unless its *min_size* is 0, a port *must* be connected to at least one implementation of its associated interface.

The asterisk in *uvm_*_port* is any of the following

```
blocking_put  
nonblocking_put  
put  
  
blocking_get  
nonblocking_get  
get  
  
blocking_peek  
nonblocking_peek  
peek  
  
blocking_get_peek  
nonblocking_get_peek  
get_peek  
  
analysis
```

Type parameters

T The type of transaction to be communicated by the export

Ports are connected to interface implementations directly via [uvm_*_imp #\(T,IMP\)](#) ports or indirectly via hierarchical connections to [uvm_*_port #\(T\)](#) and [uvm_*_export #\(T\)](#) ports.

Summary

uvm_*_port #(T)

These unidirectional ports are instantiated by components that *require*, or *use*, the associated interface to convey transactions.

METHODS

[new](#) The *name* and *parent* are the standard [uvm_component](#) constructor arguments.

METHODS

[new](#)

The *name* and *parent* are the standard [uvm_component](#) constructor arguments. The *min_size* and *max_size* specify the minimum and maximum number of interfaces that must have been connected to this port by the end of elaboration.

```
function new (string name,  
            uvm_component parent,  
            int min_size=1,  
            int max_size=1)
```

uvm_*_port #(REQ,RSP)

These bidirectional ports are instantiated by components that *require*, or *use*, the associated interface to convey transactions. A port can be connected to any compatible port, export, or imp port. Unless its *min_size* is 0, a port *must* be connected to at least one implementation of its associated interface.

The asterisk in *uvm_*_port* is any of the following

```
blocking_transport  
nonblocking_transport  
transport  
  
blocking_master  
nonblocking_master  
master  
  
blocking_slave  
nonblocking_slave  
slave
```

Ports are connected to interface implementations directly via [uvm_*_imp #\(REQ,RSP,IMP,REQ_IMP,RSP_IMP\)](#) ports or indirectly via hierarchical connections to [uvm_*_port #\(REQ,RSP\)](#) and [uvm_*_export #\(REQ,RSP\)](#) ports.

Type parameters

- REQ* The type of request transaction to be communicated by the export
RSP The type of response transaction to be communicated by the export

Summary

uvm_*_port #(REQ,RSP)

These bidirectional ports are instantiated by components that *require*, or *use*, the associated interface to convey transactions.

METHODS

- [new](#) The *name* and *parent* are the standard [uvm_component](#) constructor arguments.

METHODS

[new](#)

The *name* and *parent* are the standard [uvm_component](#) constructor arguments. The *min_size* and *max_size* specify the minimum and maximum number of interfaces that must have been supplied to this port by the end of elaboration.

```
function new (string name, uvm_component parent, int min_size=1, int max_size=1)
```

uvm_seq_item_pull_port #(REQ,RSP)

UVM provides a port, export, and imp connector for use in sequencer-driver communication. All have standard port connector constructors, except that uvm_seq_item_pull_port's default min_size argument is 0; it can be left unconnected.

Summary

uvm_seq_item_pull_port #(REQ,RSP)

UVM provides a port, export, and imp connector for use in sequencer-driver communication.

CLASS HIERARCHY

```
uvm_port_base#(uvm_sqr_if_base#(REQ,RSP))
```

```
uvm_seq_item_pull_port#(REQ,RSP)
```

CLASS DECLARATION

```
class uvm_seq_item_pull_port #(
    type REQ = int,
    type RSP = REQ
) extends uvm_port_base #(uvm_sqr_if_base #(REQ, RSP))
```

uvm_seq_item_pull_export #(REQ,RSP)

This export type is used in sequencer-driver communication. It has the standard constructor for exports.

Summary

uvm_seq_item_pull_export #(REQ,RSP)

This export type is used in sequencer-driver communication.

CLASS HIERARCHY

```
uvm_port_base#(uvm_sqr_if_base#(REQ,RSP))
```

```
uvm_seq_item_pull_export#(REQ,RSP)
```

CLASS DECLARATION

```
class uvm_seq_item_pull_export #(
    type REQ = int,
    type RSP = REQ
) extends uvm_port_base #(uvm_sqr_if_base #(REQ, RSP))
```

uvm_seq_item_pull_imp #(REQ,RSP,IMP)

This imp type is used in sequencer-driver communication. It has the standard

constructor for imp-type ports.

Summary

uvm_seq_item_pull_imp #(REQ,RSP,IMP)

This imp type is used in sequencer-driver communication.

CLASS HIERARCHY

```
uvm_port_base#(uvm_sqr_if_base #(REQ,RSP))
```

```
uvm_seq_item_pull_imp #(REQ,RSP,IMP)
```

CLASS DECLARATION

```
class uvm_seq_item_pull_imp #(
    type REQ = int,
    type RSP = REQ,
    type IMP = int
) extends uvm_port_base #(uvm_sqr_if_base #(REQ, RSP))
```

end

uvm_sqr_if_base #(REQ,RSP)

This class defines an interface for sequence drivers to communicate with sequencers. The driver requires the interface via a port, and the sequencer implements it and provides it via an export.

Summary

uvm_sqr_if_base #(REQ,RSP)

This class defines an interface for sequence drivers to communicate with sequencers.

CLASS DECLARATION

```
virtual class uvm_sqr_if_base #(type T1 = uvm_object,  
                           T2 = T1           )
```

METHODS

get_next_item	Retrieves the next available item from a sequence.
try_next_item	Retrieves the next available item from a sequence if one is available.
item_done	Indicates that the request is completed to the sequencer.
wait_for_sequences	Waits for a sequence to have a new item available.
has_do_available	Indicates whether a sequence item is available for immediate processing.
get	Retrieves the next available item from a sequence.
peek	Returns the current request item if one is in the sequencer fifo.
put	Sends a response back to the sequence that issued the request.

METHODS

get_next_item

```
virtual task get_next_item(output T1 t)
```

Retrieves the next available item from a sequence. The call will block until an item is available. The following steps occur on this call:

- 1 Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, wait for a requesting unlocked relevant sequence, then re-arbitrate.
- 2 The chosen sequence will return from wait_for_grant
- 3 The chosen sequence 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.

[try_next_item](#)

```
virtual task try_next_item(output T1 t)
```

Retrieves the next available item from a sequence if one is available. Otherwise, the function returns immediately with request set to null. The following steps occur on this call:

- 1 Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, return null.
- 2 The chosen sequence will return from wait_for_grant
- 3 The chosen sequence 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.

[item_done](#)

```
virtual function void item_done(input T2 t = null)
```

Indicates that the request is completed to the sequencer. Any wait_for_item_done calls made by a sequence for this item will return.

The current item is removed from the sequencer fifo.

If a response item is provided, then it will be sent back to the requesting sequence. The response item must have its sequence ID and transaction ID set correctly, using the set_id_info method:

```
rsp.set_id_info(req);
```

Before item_done is called, any calls to peek will retrieve the current item that was obtained by get_next_item. After item_done is called, peek will cause the sequencer to arbitrate for a new item.

[wait_for_sequences](#)

```
virtual task wait_for_sequences()
```

Waits for a sequence to have a new item available. The default implementation in the sequencer delays pound_zero_count delta cycles. (This variable is defined in uvm_sequencer_base.) User-derived sequencers may override its wait_for_sequences implementation to perform some other application-specific implementation.

[has_do_available](#)

```
virtual function bit has_do_available()
```

Indicates whether a sequence item is available for immediate processing.
Implementations should return 1 if an item is available, 0 otherwise.

get

```
virtual task get(output T1 t)
```

Retrieves the next available item from a sequence. The call blocks until an item is available. The following steps occur on this call:

- 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.

peek

```
virtual task peek(output T1 t)
```

Returns the current request item if one is in the sequencer fifo. If no item is in the fifo, then the call will block until the sequencer has a new request. The following steps will occur if the sequencer fifo is empty:

- 1 Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, wait for a requesting unlocked relevant sequence, then re-arbitrate.
- 2 The chosen sequence will return from 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

```
virtual task put(input T2 t)
```

Sends a response back to the sequence that issued the request. Before the response is put, it must have its sequence ID and transaction ID set to match the request. This can be done using the set_id_info call:

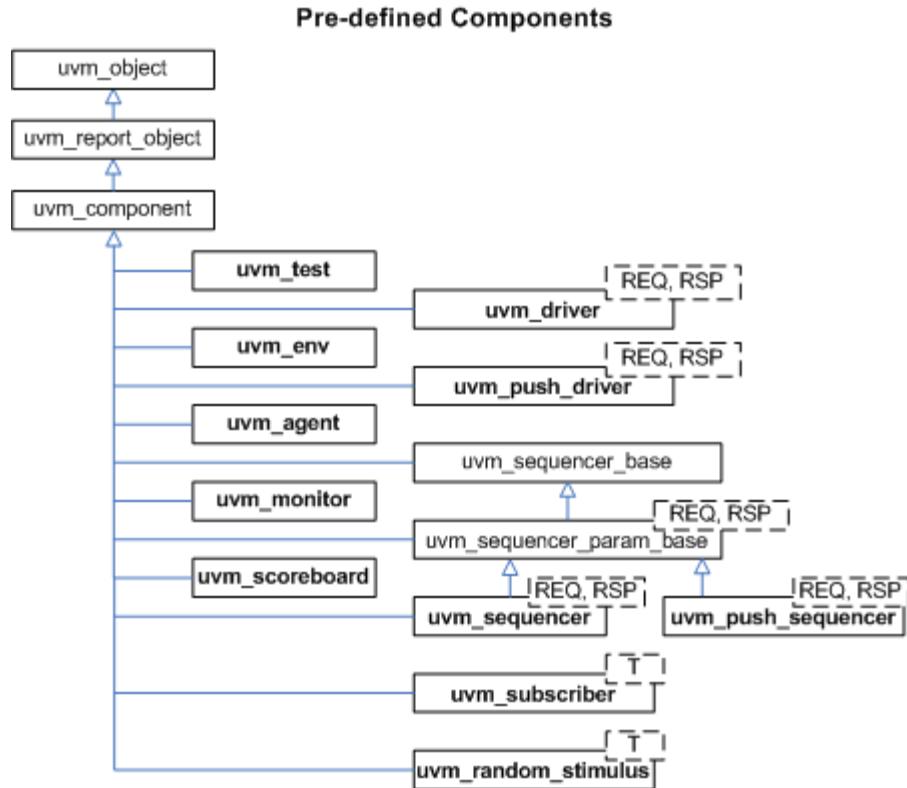
```
rsp.set_id_info(req);
```

This task will not block. The response will be put into the sequence response_queue or it will be sent to the sequence response handler.

PREDEFINED COMPONENT CLASSES

Components form the foundation of the UVM. They encapsulate behavior of drivers, scoreboards, and other objects in a testbench. The UVM library provides a set of predefined component types, all derived directly or indirectly from [uvm_component](#).

Predefined Components



uvm_test

This class is the virtual base class for the user-defined tests.

The uvm_test virtual class should be used as the base class for user-defined tests. Doing so provides the ability to select which test to execute using the UVM_TESTNAME command line or argument to the [uvm_root::run_test](#) task.

For example

```
prompt> SIM_COMMAND +UVM_TESTNAME=test_bus_retry
```

The global run_test() task should be specified inside an initial block such as

```
initial run_test();
```

Multiple tests, identified by their type name, are compiled in and then selected for execution from the command line without need for recompilation. Random seed selection is also available on the command line.

If +UVM_TESTNAME=test_name is specified, then an object of type 'test_name' is created by factory and phasing begins. Here, it is presumed that the test will instantiate the test environment, or the test environment will have already been instantiated before the call to run_test().

If the specified test_name cannot be created by the [uvm_factory](#), then a fatal error occurs. If run_test() is called without UVM_TESTNAME being specified, then all components constructed before the call to run_test will be cycled through their simulation phases.

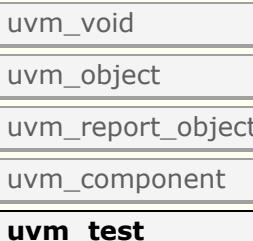
Deriving from uvm_test will allow you to distinguish tests from other component types that inherit from uvm_component directly. Such tests will automatically inherit features that may be added to uvm_test in the future.

Summary

uvm_test

This class is the virtual base class for the user-defined tests.

CLASS HIERARCHY



CLASS DECLARATION

```
virtual class uvm_test extends uvm_component
```

METHODS

new

Creates and initializes an instance of this class using the normal constructor arguments for `uvm_component`: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

METHODS

new

```
function new (string          name,  
             uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for `uvm_component`: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

The base class for hierarchical containers of other components that together comprise a complete environment. The environment may initially consist of the entire testbench. Later, it can be reused as a sub-environment in even larger system-level environments.

Summary

uvm_env

The base class for hierarchical containers of other components that together comprise a complete environment.

CLASS HIERARCHY



CLASS DECLARATION

```
virtual class uvm_env extends uvm_component
```

METHODS

new Creates and initializes an instance of this class using the normal constructor arguments for [uvm_component](#): *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

METHODS

new

```
function new (string name = "env",  
             uvm_component parent = null )
```

Creates and initializes an instance of this class using the normal constructor arguments for [uvm_component](#): *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

uvm_agent

The uvm_agent virtual class should be used as the base class for the user-defined agents. Deriving from uvm_agent will allow you to distinguish agents from other component types also using its inheritance. Such agents will automatically inherit features that may be added to uvm_agent in the future.

While an agent's build function, inherited from [uvm_component](#), can be implemented to define any agent topology, an agent typically contains three subcomponents: a driver, sequencer, and monitor. If the agent is active, subtypes should contain all three subcomponents. If the agent is passive, subtypes should contain only the monitor.

Summary

uvm_agent

The uvm_agent virtual class should be used as the base class for the user-defined agents.

CLASS HIERARCHY



CLASS DECLARATION

```
virtual class uvm_agent extends uvm_component
```

METHODS

new Creates and initializes an instance of this class using the normal constructor arguments for [uvm_component](#): *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

METHODS

[new](#)

```
function new (string name,  
             uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for [uvm_component](#): *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

uvm_monitor

This class should be used as the base class for user-defined monitors.

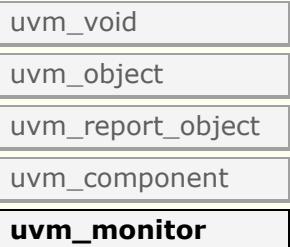
Deriving from uvm_monitor allows you to distinguish monitors from generic component types inheriting from uvm_component. Such monitors will automatically inherit features that may be added to uvm_monitor in the future.

Summary

uvm_monitor

This class should be used as the base class for user-defined monitors.

CLASS HIERARCHY



CLASS DECLARATION

```
virtual class uvm_monitor extends uvm_component
```

METHODS

new

Creates and initializes an instance of this class using the normal constructor arguments for [uvm_component](#): *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

METHODS

new

```
function new (string name,  
             uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for [uvm_component](#): *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

uvm_scoreboard

The uvm_scoreboard virtual class should be used as the base class for user-defined scoreboards.

Deriving from uvm_scoreboard will allow you to distinguish scoreboards from other component types inheriting directly from uvm_component. Such scoreboards will automatically inherit and benefit from features that may be added to uvm_scoreboard in the future.

Summary

uvm_scoreboard

The uvm_scoreboard virtual class should be used as the base class for user-defined scoreboards.

CLASS HIERARCHY



CLASS DECLARATION

```
virtual class uvm_scoreboard extends uvm_component
```

METHODS

- new** Creates and initializes an instance of this class using the normal constructor arguments for **uvm_component**: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

METHODS

new

```
function new (string name,  
             uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for **uvm_component**: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

uvm_driver #(REQ,RSP)

The base class for drivers that initiate requests for new transactions via a uvm_seq_item_pull_port. The ports are typically connected to the exports of an appropriate sequencer component.

This driver operates in pull mode. Its ports are typically connected to the corresponding exports in a pull sequencer as follows:

```
driver.seq_item_port.connect(sequencer.seq_item_export);
driver.rsp_port.connect(sequencer.rsp_export);
```

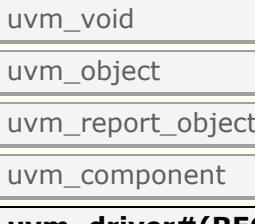
The *rsp_port* needs connecting only if the driver will use it to write responses to the analysis export in the sequencer.

Summary

uvm_driver #(REQ,RSP)

The base class for drivers that initiate requests for new transactions via a uvm_seq_item_pull_port.

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_driver #(
    type REQ = uvm_sequence_item,
    type RSP = REQ
) extends uvm_component
```

POR

seq_item_port Derived driver classes should use this port to request items from the sequencer.

rsp_port This port provides an alternate way of sending responses back to the originating sequencer.

METHODS

new Creates and initializes an instance of this class using the normal constructor arguments for **uvm_component**: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

PORTS

seq_item_port

Derived driver classes should use this port to request items from the sequencer. They may also use it to send responses back.

[rsp_port](#)

This port provides an alternate way of sending responses back to the originating sequencer. Which port to use depends on which export the sequencer provides for connection.

METHODS

[new](#)

```
function new (string name,  
             uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for [uvm_component](#): *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

uvm_push_driver #(REQ,RSP)

Base class for a driver that passively receives transactions, i.e. does not initiate requests transactions. Also known as *push* mode. Its ports are typically connected to the corresponding ports in a push sequencer as follows:

```
push_sequencer.req_port.connect(push_driver.req_export);  
push_driver.rsp_port.connect(push_sequencer.rsp_export);
```

The *rsp_port* needs connecting only if the driver will use it to write responses to the analysis export in the sequencer.

Summary

uvm_push_driver #(REQ,RSP)

Base class for a driver that passively receives transactions, i.e.

CLASS HIERARCHY

```
uvm_void  
uvm_object  
uvm_report_object  
uvm_component
```

```
uvm_push_driver#(REQ,RSP)
```

CLASS DECLARATION

```
class uvm_push_driver #(  
    type REQ = uvm_sequence_item,  
    type RSP = REQ  
) extends uvm_component
```

POR

req_export This export provides the blocking put interface whose default implementation produces an error.

rsp_port This analysis port is used to send response transactions back to the originating sequencer.

METHODS

new Creates and initializes an instance of this class using the normal constructor arguments for **uvm_component**: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

PORTS

req_export

This export provides the blocking put interface whose default implementation produces an error. Derived drivers must override *put* with an appropriate implementation (and not call super.put). Ports connected to this export will supply the driver with transactions.

[rsp_port](#)

This analysis port is used to send response transactions back to the originating sequencer.

METHODS

[new](#)

```
function new (string name,  
             uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for [uvm_component](#): *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

uvm_random_stimulus #(T)

A general purpose unidirectional random stimulus class.

The uvm_random_stimulus class generates streams of T transactions. These streams may be generated by the randomize method of T, or the randomize method of one of its subclasses. The stream may go indefinitely, until terminated by a call to stop_stimulus_generation, or we may specify the maximum number of transactions to be generated.

By using inheritance, we can add directed initialization or tidy up after random stimulus generation. Simply extend the class and define the run task, calling super.run() when you want to begin the random stimulus phase of simulation.

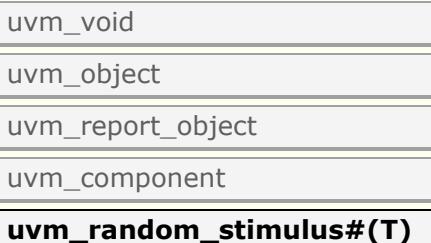
While very useful in its own right, this component can also be used as a template for defining other stimulus generators, or it can be extended to add additional stimulus generation methods and to simplify test writing.

Summary

uvm_random_stimulus #(T)

A general purpose unidirectional random stimulus class.

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_random_stimulus #(  
    type T = uvm_transaction  
) extends uvm_component
```

PORTS

[blocking_put_port](#)

The blocking_put_port is used to send the generated stimulus to the rest of the testbench.

METHODS

[new](#)

Creates a new instance of a specialization of this class.

[generate_stimulus](#)

Generate up to max_count transactions of type T.

[stop_stimulus_generation](#)

Stops the generation of stimulus.

PORTS

[blocking_put_port](#)

The blocking_put_port is used to send the generated stimulus to the rest of the testbench.

METHODS

new

```
function new(string name,  
           uvm_component parent)
```

Creates a new instance of a specialization of this class. Also, displays the random state obtained from a get_randstate call. In subsequent simulations, set_randstate can be called with the same value to reproduce the same sequence of transactions.

generate_stimulus

```
virtual task generate_stimulus(T t = null,  
                             int max_count = 0 )
```

Generate up to max_count transactions of type T. If t is not specified, a default instance of T is allocated and used. If t is specified, that transaction is used when randomizing. It must be a subclass of T.

max_count is the maximum number of transactions to be generated. A value of zero indicates no maximum - in this case, generate_stimulus will go on indefinitely unless stopped by some other process

The transactions are cloned before they are sent out over the blocking_put_port

stop_stimulus_generation

```
virtual function void stop_stimulus_generation
```

Stops the generation of stimulus. If a subclass of this method has forked additional processes, those processes will also need to be stopped in an overridden version of this method

uvm_subscriber

This class provides an analysis export for receiving transactions from a connected analysis export. Making such a connection “subscribes” this component to any transactions emitted by the connected analysis port.

Subtypes of this class must define the write method to process the incoming transactions. This class is particularly useful when designing a coverage collector that attaches to a monitor.

Summary

uvm_subscriber

This class provides an analysis export for receiving transactions from a connected analysis export.

CLASS HIERARCHY



CLASS DECLARATION

```
virtual class uvm_subscriber #(  
    type T = int  
) extends uvm_component
```

PORTS

[analysis_export](#) This export provides access to the write method, which derived subscribers must implement.

METHODS

[new](#) Creates and initializes an instance of this class using the normal constructor arguments for [uvm_component](#): *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

[write](#) A pure virtual method that must be defined in each subclass.

PORTS

[analysis_export](#)

This export provides access to the write method, which derived subscribers must implement.

METHODS

new

```
function new (string name,  
             uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for `uvm_component`: *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

write

```
pure virtual function void write(T t)
```

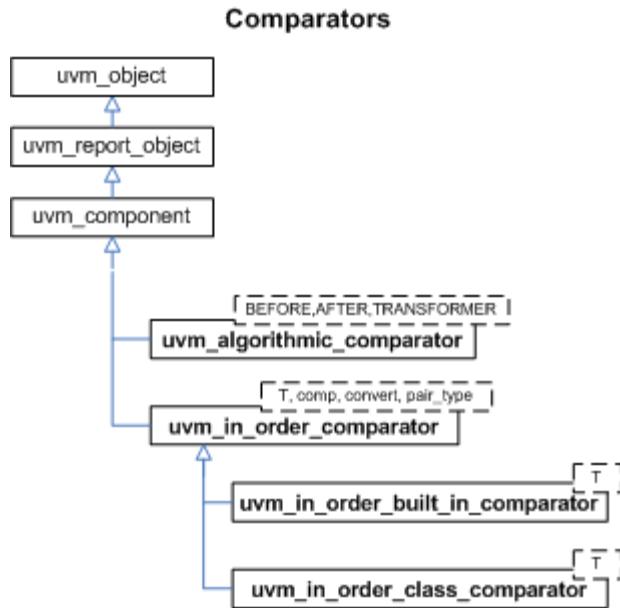
A pure virtual method that must be defined in each subclass. Access to this method by outside components should be done via the `analysis_export`.

COMPARATORS

A common function of testbenches is to compare streams of transactions for equivalence. For example, a testbench may compare a stream of transactions from a DUT with expected results.

The UVM library provides a base class called *uvm_in_order_comparator* and two derived classes: *uvm_in_order_builtin_comparator* for comparing streams of built-in types and *uvm_in_order_class_comparator* for comparing streams of class objects.

The *uvm_algorithmic_comparator* also compares two streams of transactions, but the transaction streams might be of different type objects. Thus, this comparator will employ a user-defined transformation function to convert one type to another before performing a comparison.



uvm_in_order_comparator #(T,comp_type,convert,pair_type)

Compares two streams of data objects of type T, a parameter to this class. These transactions may either be classes or built-in types. To be successfully compared, the two streams of data must be in the same order. Apart from that, there are no assumptions made about the relative timing of the two streams of data.

Type parameters

<i>T</i>	Specifies the type of transactions to be compared.
<i>comp</i>	The type of the comparator to be used to compare the two transaction streams.
<i>convert</i>	A policy class to allow convert2string() to be called on the transactions being compared. If T is an extension of uvm_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.
<i>pair_type</i>	A policy class to allow pairs of transactions to be handled as a single uvm_transaction type.

Built in types (such as ints, bits, logic, and structs) can be compared using the default values for comp_type, convert, and pair_type. For convenience, you can use the subtype, <uvm_in_order_builtin_comparator #(T)> for built-in types.

When T is a class, T must implement comp and convert2string, and you must specify class-based policy classes for comp_type, convert, and pair_type. In most cases, you can use the convenient subtype, uvm_in_order_class_comparator #(T).

Comparisons are commutative, meaning it does not matter which data stream is connected to which export, before_export or after_export.

Comparisons are done in order and as soon as a transaction is received from both streams. Internal fifos are used to buffer incoming transactions on one stream until a transaction to compare arrives on the other stream.

Summary

uvm_in_order_comparator #(T,comp_type,convert,pair_type)

Compares two streams of data objects of type T, a parameter to this class.

PORTS

<i>before_export</i>	The export to which one stream of data is written.
<i>after_export</i>	The export to which the other stream of data is written.
<i>pair_ap</i>	The comparator sends out pairs of transactions across this analysis port.

METHODS

<i>flush</i>	This method sets m_matches and m_mismatches back to zero.
--------------	---

PORTS

before_export

The export to which one stream of data is written. The port must be connected to an analysis port that will provide such data.

after_export

The export to which the other stream of data is written. The port must be connected to an analysis port that will provide such data.

pair_ap

The comparator sends out pairs of transactions across this analysis port. Both matched and unmatched pairs are published via a pair_type objects. Any connected analysis export(s) will receive these transaction pairs.

METHODS

flush

```
virtual function void flush()
```

This method sets m_matches and m_mismatches back to zero. The [uvm_tlm_fifo #\(T\)::flush](#) takes care of flushing the FIFOs.

in_order_builtin_comparator #(T)

This class uses the uvm_builtin_* comparison, converter, and pair classes. Use this class for built-in types (int, bit, string, etc.)

Summary

in_order_builtin_comparator #(T)

This class uses the uvm_builtin_* comparison, converter, and pair classes.

CLASS HIERARCHY

```
uvm_in_order_comparator#(T)
```

```
in_order_builtin_comparator#(T)
```

CLASS DECLARATION

```
class uvm_in_order_builtin_comparator #(  
    type T = int  
) extends uvm_in_order_comparator #(T)
```

in_order_class_comparator #(T)

This class uses the uvm_class_* comparison, converter, and pair classes. Use this class for comparing user-defined objects of type T, which must provide implementations of comp and convert2string.

uvm_algorithmic_comparator

Summary

uvm_algorithmic_comparator

COMPARATORS A common function of testbenches is to compare streams of transactions for equivalence.

COMPARATORS

A common function of testbenches is to compare streams of transactions for equivalence. For example, a testbench may compare a stream of transactions from a DUT with expected results.

The UVM library provides a base class called `uvm_in_order_comparator` and two derived classes, which are `uvm_in_order_builtin_comparator` for comparing streams of built-in types and `uvm_in_order_class_comparator` for comparing streams of class objects.

The `uvm_algorithmic_comparator` also compares two streams of transactions; however, the transaction streams might be of different type objects. This device will use a user-written transformation function to convert one type to another before performing a comparison.

uvm_algorithmic_comparator #(BEFORE,AFTER,TRANSFORMER)

Compares two streams of data objects of different types, BEFORE and AFTER.

The algorithmic comparator is a wrapper around `uvm_in_order_class_comparator`. Like the in-order comparator, the algorithmic comparator compares two streams of transactions, the BEFORE stream and the AFTER stream. It is often the case when two streams of transactions need to be compared that the two streams are in different forms. That is, the type of the BEFORE transaction stream is different than the type of the AFTER transaction stream.

The `uvm_algorithmic_comparator`'s TRANSFORMER type parameter specifies the class responsible for converting transactions of type BEFORE into those of type AFTER. This transformer class must provide a `transform()` method with the following prototype:

```
function AFTER transform (BEFORE b);
```

Matches and mismatches are reported in terms of the AFTER transactions. For more information, see the `uvm_in_order_comparator #(...)` class.

Summary

uvm_algorithmic_comparator #(BEFORE,AFTER,TRANSFORMER)

Compares two streams of data objects of different types, BEFORE and AFTER.

CLASS HIERARCHY

```
uvm_void
uvm_object
uvm_report_object
uvm_component
```

```
uvm_algorithmic_comparator#(BEFORE,AFTER,TRANSFORMER)
```

CLASS DECLARATION

```
class uvm_algorithmic_comparator #( 
    type BEFORE      = int,
    type AFTER       = int,
    type TRANSFORMER = int
) extends uvm_component
```

PORTS

before_export The export to which a data stream of type BEFORE is sent via a connected analysis port.

after_export The export to which a data stream of type AFTER is sent via a connected analysis port.

METHODS

new Creates an instance of a specialization of this class.

PORTS

[before_export](#)

The export to which a data stream of type BEFORE is sent via a connected analysis port. Publishers (monitors) can send in an ordered stream of transactions against which the transformed BEFORE transactions will be compared.

[after_export](#)

The export to which a data stream of type AFTER is sent via a connected analysis port. Publishers (monitors) can send in an ordered stream of transactions to be transformed and compared to the AFTER transactions.

METHODS

[new](#)

```
function new(
    string   TRANSFORMER transformer,
    name
    uvm_component parent      )
```

Creates an instance of a specialization of this class. In addition to the standard uvm_component constructor arguments, *name* and *parent*, the constructor takes a handle to a *transformer* object, which must already be allocated (no null handles) and

must implement the `transform()` method.

uvm_pair #(T1,T2)

Container holding handles to two objects whose types are specified by the type parameters, T1 and T2.

Summary

uvm_pair #(T1,T2)

Container holding handles to two objects whose types are specified by the type parameters, T1 and T2.

METHODS

- `new` Creates an instance of uvm_pair that holds a handle to two objects, as provided by the first two arguments.

METHODS

new

```
function new (T1      f      = null,  
             T2      s      = null,  
             string name = "")
```

Creates an instance of uvm_pair that holds a handle to two objects, as provided by the first two arguments. The optional *name* argument gives a name to the new pair object.

uvm_built_in_pair #(T1,T2)

Container holding two variables of built-in types (int, string, etc.). The types are specified by the type parameters, T1 and T2.

Summary

uvm_built_in_pair #(T1,T2)

Container holding two variables of built-in types (int, string, etc.)

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_built_in_pair #(  
    type T1 = int,
```

```
T2 = T1  
) extends uvm_transaction
```

METHODS

- new** Creates an instance of uvm_pair that holds a handle to two elements, as provided by the first two arguments.

METHODS

new

```
function new (T1      f,  
             T2      s,  
             string name = "")
```

Creates an instance of uvm_pair that holds a handle to two elements, as provided by the first two arguments. The optional name argument gives a name to the new pair object.

uvm_policies

Summary

uvm_policies

POLICY CLASSES Policy classes are used to implement polymorphic operations that differ between built-in types and class-based types.

POLICY CLASSES

Policy classes are used to implement polymorphic operations that differ between built-in types and class-based types. Generic components can then be built that work with either classes or built-in types, depending on what policy class is used.

uvm_built_in_comp #(T)

This policy class is used to compare built-in types.

Provides a comp method that compares the built-in type, T, for which the == operator is defined.

Summary

uvm_built_in_comp #(T)

This policy class is used to compare built-in types.

CLASS DECLARATION

```
class uvm_built_in_comp #(type T = int)
```

uvm_built_in_converter #(T)

This policy class is used to convert built-in types to strings.

Provides a convert2string method that converts the built-in type, T, to a string using the %op format specifier.

Summary

uvm_built_in_converter #(T)

This policy class is used to convert built-in types to strings.

CLASS DECLARATION

```
class uvm_built_in_converter #(type T = int)
```

uvm_built_in_clone #(T)

This policy class is used to clone built-in types via the = operator.

Provides a clone method that returns a copy of the built-in type, T.

Summary

uvm_built_in_clone #(T)

This policy class is used to clone built-in types via the = operator.

CLASS DECLARATION

```
class uvm_built_in_clone #(type T = int)
```

uvm_class_comp #(T)

This policy class is used to compare two objects of the same type.

Provides a comp method that compares two objects of type T. The class T must implement the comp method, to which this class delegates the operation.

Summary

uvm_class_comp #(T)

This policy class is used to compare two objects of the same type.

CLASS DECLARATION

```
class uvm_class_comp #(type T = int)
```

uvm_class_converter #(T)

This policy class is used to convert a class object to a string.

Provides a convert2string method that converts the built-in type, T, to a string. The class T must implement the convert2string method, to which this class delegates the operation.

Summary

uvm_class_converter #(T)

This policy class is used to convert a class object to a string.

CLASS DECLARATION

```
class uvm_class_converter #(type T = int)
```

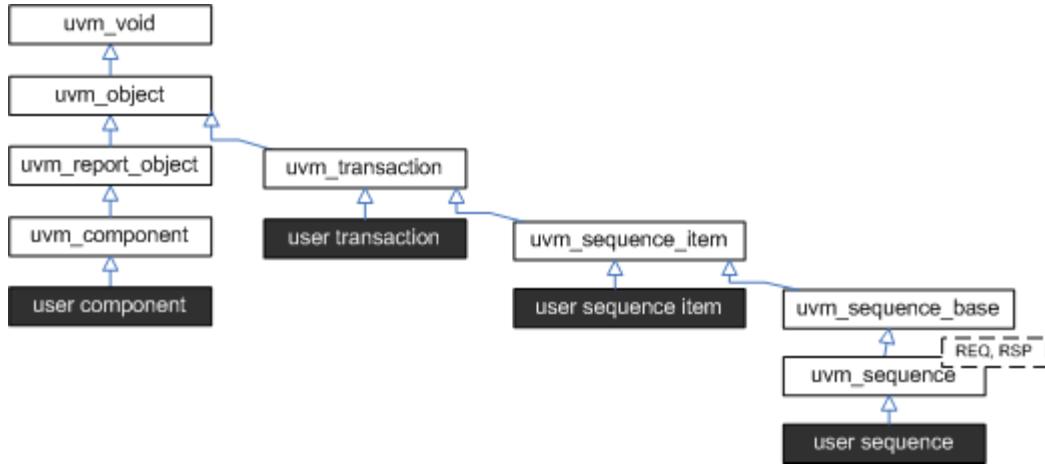
uvm_class_clone #(T)

This policy class is used to clone class objects.

Provides a clone method that returns a copy of the built-in type, T. The class T must implement the clone method, to which this class delegates the operation.

Sequencer Classes

The sequencer serves as an arbiter for controlling transaction flow from multiple stimulus generators. More specifically, the sequencer controls the flow of `uvm_sequence_item`-based transactions generated by one or more `uvm_sequence #(REQ,RSP)`-based sequences.



There are two sequencer variants available.

- `uvm_sequencer #(REQ,RSP)` - Requests for new sequence items are initiated by the driver. Upon such requests, the sequencer selects a sequence from a list of available sequences to produce and deliver the next item to execute. This sequencer is typically connected to a user-extension of `uvm_driver #(REQ,RSP)`.
- `uvm_push_sequencer #(REQ,RSP)` - Sequence items (from the currently running sequences) are pushed by the sequencer to the driver, which blocks item flow when it is not ready to accept new transactions. This sequencer is typically connected to a user-extension of `uvm_push_driver #(REQ,RSP)`.

Sequencer-driver communication follows a *pull* or *push* semantic, depending on which sequencer type is used. However, sequence-sequencer communication is *always* initiated by the user-defined sequence, i.e. follows a push semantic.

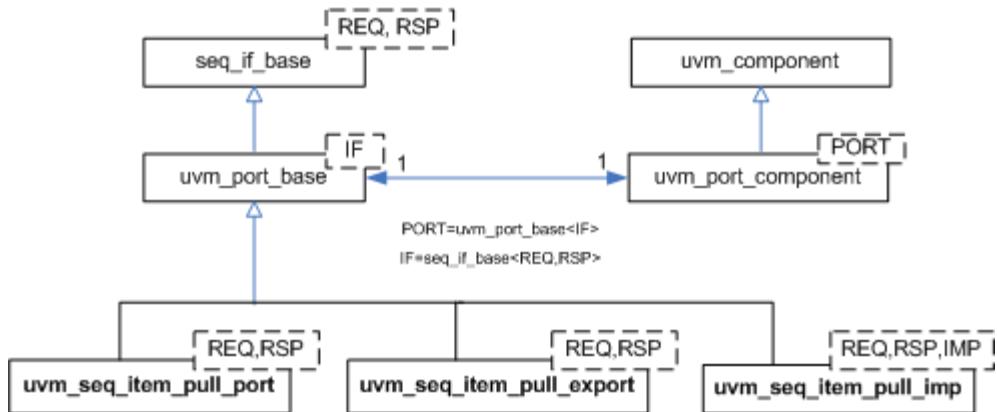
See [Sequence Classes](#) for an overview on sequences and sequence items.

Sequence Item Ports

As with all UVM components, the sequencers and drivers described above use [TLM Interfaces, Ports, and Exports](#) to communicate transactions.

The `uvm_sequencer #(REQ,RSP)` and `uvm_driver #(REQ,RSP)` pair also uses a *sequence item pull port* to achieve the special execution semantic needed by the sequencer-driver pair.

Sequence Item port, export, and imp



Sequencers and drivers use a `seq_item_port` specifically supports sequencer-driver communication. Connections to these ports are made in the same fashion as the TLM ports.

uvm_sequencer_base

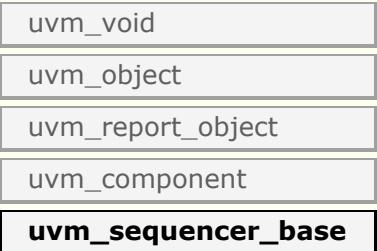
Controls the flow of sequences, which generate the stimulus (sequence item transactions) that is passed on to drivers for execution.

Summary

uvm_sequencer_base

Controls the flow of sequences, which generate the stimulus (sequence item transactions) that is passed on to drivers for execution.

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_sequencer_base extends uvm_component
```

VARIABLES

pound_zero_count	Set this variable via set_config_int to set the number of delta cycles to insert in the wait_for_sequences task.
count	Sets the number of items to execute.
max_random_count	Set this variable via set_config_int to set the number of sequence items to generate, at the discretion of the derived sequence.
max_random_depth	Used for setting the maximum depth inside random sequences.
default_sequence	This property defines the sequence type (by name) that will be auto-started.

METHODS

new	Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: name is the name of the instance, and parent is the handle to the hierarchical parent.
start_default_sequence	Sequencers provide the start_default_sequence task to execute the default sequence in the run phase.
user_priority_arbitration	If the sequencer arbitration mode is set to SEQ_ARB_USER (via the <i>set_arbitration</i> method), then the sequencer will call this function each time that it needs to arbitrate among sequences.
is_child	Returns 1 if the child sequence is a child of the parent sequence, 0 otherwise.
wait_for_grant	This task issues a request for the specified sequence.
wait_for_item_done	A sequence may optionally call wait_for_item_done.
is_blocked	Returns 1 if the sequence referred to by sequence_ptr is currently locked out of the sequencer.
has_lock	Returns 1 if the sequence referred to in the parameter currently has a lock on this sequencer, 0 otherwise.

<code>lock</code>	Requests a lock for the sequence specified by <code>sequence_ptr</code> .
<code>grab</code>	Requests a lock for the sequence specified by <code>sequence_ptr</code> .
<code>unlock</code>	Removes any locks and grabs obtained by the specified <code>sequence_ptr</code> .
<code>ungrab</code>	Removes any locks and grabs obtained by the specified <code>sequence_ptr</code> .
<code>stop_sequences</code>	Tells the sequencer to kill all sequences and child sequences currently operating on the sequencer, and remove all requests, locks and responses that are currently queued.
<code>is_grabbed</code>	Returns 1 if any sequence currently has a lock or grab on this sequencer, 0 otherwise.
<code>current_grabber</code>	Returns a reference to the sequence that currently has a lock or grab on the sequence.
<code>has_do_available</code>	Determines if a sequence is ready to supply a transaction.
<code>set_arbitration</code>	Specifies the arbitration mode for the sequencer.
<code>wait_for_sequences</code>	Waits for a sequence to have a new item available.
<code>add_sequence</code>	Adds a sequence of type specified in the <code>type_name</code> parameter to the sequencer's sequence library.
<code>get_seq_kind</code>	Returns an int <code>seq_kind</code> correlating to the sequence of type <code>type_name</code> in the sequencer's sequence library.
<code>get_sequence</code>	Returns a reference to a sequence specified by the <code>seq_kind</code> int.
<code>num_sequences</code>	Returns the number of sequences in the sequencer's sequence library.
<code>send_request</code>	Derived classes implement this function to send a request item to the sequencer, which will forward it to the driver.

VARIABLES

pound_zero_count

```
int unsigned pound_zero_count = 6
```

Set this variable via `set_config_int` to set the number of delta cycles to insert in the `wait_for_sequences` task. The delta cycles are used to ensure that a sequence with back-to-back items has an opportunity to fill the action queue when the driver uses the non-blocking `try_get` interface.

count

```
int count = -1
```

Sets the number of items to execute.

Supercedes the `max_random_count` variable for `uvm_random_sequence` class for backward compatibility.

max_random_count

```
int unsigned max_random_count = 10
```

Set this variable via `set_config_int` to set the number of sequence items to generate, at the discretion of the derived sequence. The predefined `uvm_random_sequence` uses `count` to determine the number of random items to generate.

max_random_depth

```
int unsigned max_random_depth = 4
```

Used for setting the maximum depth inside random sequences. (Beyond that depth, random creates only simple sequences.)

default_sequence

```
protected string default_sequence = "uvm_random_sequence"
```

This property defines the sequence type (by name) that will be auto-started. The default sequence is initially set to `uvm_random_sequence`. It can be configured through the `uvm_component`'s `set_config_string` method using the field name "default_sequence".

METHODS

new

```
function new (string name,  
             uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for `uvm_component`: `name` is the name of the instance, and `parent` is the handle to the hierarchical parent.

start_default_sequence

```
virtual task start_default_sequence()
```

Sequencers provide the `start_default_sequence` task to execute the default sequence in the run phase. This method is not intended to be called externally, but may be overridden in a derivative sequencer class if special behavior is needed when the default sequence is started. The user class `uvm_sequencer_param_base #(REQ,RSP)` implements this method.

user_priority_arbitration

```
virtual function integer user_priority_arbitration(integer avail_sequences[$]
```

If the sequencer arbitration mode is set to `SEQ_ARB_USER` (via the `set_arbitration` method), then the sequencer will call this function each time that it needs to arbitrate among sequences.

Derived sequencers may override this method to perform a custom arbitration policy. Such an override must return one of the entries from the avail_sequences queue, which are int indexes into an internal queue, arb_sequence_q.

The default implementation behaves like SEQ_ARB_FIFO, which returns the entry at avail_sequences[0].

If a user specifies that the sequencer is to use user_priority_arbitration through the call set_arbitration(SEQ_ARB_USER), then the sequencer will call this function each time that it needs to arbitrate among sequences.

This function must return an int that matches one of the available sequences that is passed into the call through the avail_sequences parameter

Each int in avail_sequences points to an entry in the arb_sequence_q, which is a protected queue that may be accessed from this function.

To modify the operation of user_priority_arbitration, the function may arbitrarily choose any sequence among the list of avail_sequences. It is important to choose only an available sequence.

is_child

```
function bit is_child (uvm_sequence_base parent,  
                      uvm_sequence_base child )
```

Returns 1 if the child sequence is a child of the parent sequence, 0 otherwise.

wait_for_grant

```
virtual task wait_for_grant(uvm_sequence_base sequence_ptr,  
                           int item_priority = -1,  
                           bit lock_request = 0 )
```

This task issues a request for the specified sequence. If item_priority is not specified, then the current sequence priority will be used by the arbiter. If a lock_request is made, then the sequencer will issue a lock immediately before granting the sequence. (Note that the lock may be granted without the sequence being granted if is_relevant is not asserted).

When this method returns, the sequencer has granted the sequence, and the sequence must call send_request without inserting any simulation delay other than delta cycles. The driver is currently waiting for the next item to be sent via the send_request call.

wait_for_item_done

```
virtual task wait_for_item_done(uvm_sequence_base sequence_ptr,  
                               int transaction_id)
```

A sequence may optionally call wait_for_item_done. This task will block until the driver calls item_done() or put() on a transaction issued by the specified sequence. If no transaction_id parameter is specified, then the call will return the next time that the driver calls item_done() or put(). If a specific transaction_id is specified, then the call will only return when the driver indicates that it has completed that specific item.

Note that if a specific transaction_id has been specified, and the driver has already issued an item_done or put for that transaction, then the call will hang waiting for that specific transaction_id.

is_blocked

```
function bit is_blocked(uvm_sequence_base sequence_ptr)
```

Returns 1 if the sequence referred to by sequence_ptr is currently locked out of the sequencer. It will return 0 if the sequence is currently allowed to issue operations.

Note that even when a sequence is not blocked, it is possible for another sequence to issue a lock before this sequence is able to issue a request or lock.

has_lock

```
function bit has_lock(uvm_sequence_base sequence_ptr)
```

Returns 1 if the sequence referred to in the parameter currently has a lock on this sequencer, 0 otherwise.

Note that even if this sequence has a lock, a child sequence may also have a lock, in which case the sequence is still blocked from issuing operations on the sequencer

lock

```
virtual task lock(uvm_sequence_base sequence_ptr)
```

Requests a lock for the sequence specified by sequence_ptr.

A lock request will be arbitrated the same as any other request. A lock is granted after all earlier requests are completed and no other locks or grabs are blocking this sequence.

The lock call will return when the lock has been granted.

grab

```
virtual task grab(uvm_sequence_base sequence_ptr)
```

Requests a lock for the sequence specified by sequence_ptr.

A grab request is put in front of the arbitration queue. It will be arbitrated before any other requests. A grab is granted when no other grabs or locks are blocking this sequence.

The grab call will return when the grab has been granted.

unlock

```
virtual function void unlock(uvm_sequence_base sequence_ptr)
```

Removes any locks and grabs obtained by the specified sequence_ptr.

ungrab

```
virtual function void ungrab(uvm_sequence_base sequence_ptr)
```

Removes any locks and grabs obtained by the specified sequence_ptr.

stop_sequences

```
virtual function void stop_sequences()
```

Tells the sequencer to kill all sequences and child sequences currently operating on the sequencer, and remove all requests, locks and responses that are currently queued. This essentially resets the sequencer to an idle state.

is_grabbed

```
virtual function bit is_grabbed()
```

Returns 1 if any sequence currently has a lock or grab on this sequencer, 0 otherwise.

current_grabber

```
virtual function uvm_sequence_base current_grabber()
```

Returns a reference to the sequence that currently has a lock or grab on the sequence. If multiple hierarchical sequences have a lock, it returns the child that is currently allowed to perform operations on the sequencer.

has_do_available

```
virtual function bit has_do_available()
```

Determines if a sequence is ready to supply a transaction. A sequence that obtains a transaction in pre-do must determine if the upstream object is ready to provide an item

Returns 1 if a sequence is ready to issue an operation. Returns 0 if no unblocked, relevant sequence is requesting.

set_arbitration

```
function void set_arbitration(SEQ_ARB_TYPE val)
```

Specifies the arbitration mode for the sequencer. It is one of

<i>SEQ_ARB_FIFO</i>	Requests are granted in FIFO order (default)
<i>SEQ_ARB_WEIGHTED</i>	Requests are granted randomly by weight
<i>SEQ_ARB_RANDOM</i>	Requests are granted randomly
<i>SEQ_ARB_STRICT_FIFO</i>	Requests at highest priority granted in fifo order
<i>SEQ_ARB_STRICT_RANDOM</i>	Requests at highest priority granted in randomly
<i>SEQ_ARB_USER</i>	Arbitration is delegated to the user-defined function, <i>user_priority_arbitration</i> . That function will specify the next sequence to grant.

The default user function specifies FIFO order.

[wait_for_sequences](#)

```
virtual task wait_for_sequences()
```

Waits for a sequence to have a new item available. The default implementation in the sequencer delays pound_zero_count delta cycles. (This variable is defined in uvm_sequencer_base.) User-derived sequencers may override its wait_for_sequences implementation to perform some other application-specific implementation.

[add_sequence](#)

```
function void add_sequence(string type_name)
```

Adds a sequence of type specified in the type_name parameter to the sequencer's sequence library.

[get_seq_kind](#)

```
function int get_seq_kind(string type_name)
```

Returns an int seq_kind correlating to the sequence of type type_name in the sequencer's sequence library. If the named sequence is not registered a SEQNF warning is issued and -1 is returned.

[get_sequence](#)

```
function uvm_sequence_base get_sequence(int req_kind)
```

Returns a reference to a sequence specified by the seq_kind int. The seq_kind int may be obtained using the get_seq_kind() method.

[num_sequences](#)

```
function int num_sequences()
```

Returns the number of sequences in the sequencer's sequence library.

[send_request](#)

```
virtual function void send_request(uvm_sequence_base sequence_ptr,  
                                  uvm_sequence_item t,  
                                  bit rerandomize = 0)
```

Derived classes implement this function to send a request item to the sequencer, which will forward it to the driver. If the rerandomize bit is set, the item will be randomized before being sent to the driver.

This function may only be called after a [wait_for_grant](#) call.

uvm_sequencer_param_base #(REQ,RSP)

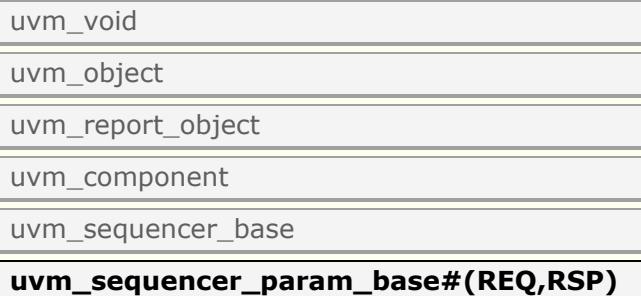
Provides base functionality used by the uvm_sequencer and uvm_push_sequencer. The implementation is dependent on REQ and RSP parameters.

Summary

uvm_sequencer_param_base #(REQ,RSP)

Provides base functionality used by the uvm_sequencer and uvm_push_sequencer.

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_sequencer_param_base #(  
    type REQ = uvm_sequence_item,  
    type RSP = REQ  
) extends uvm_sequencer_base
```

PORTS

rsp_export	This is the analysis export used by drivers or monitors to send responses to the sequencer.
------------	---

METHODS

new	Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: name is the name of the instance, and parent is the handle to the hierarchical parent, if any.
send_request	The send_request function may only be called after a wait_for_grant call.
get_current_item	Returns the request_item currently being executed by the sequencer.
start_default_sequence	Called when the run phase begins, this method starts the default sequence, as specified by the default_sequence member variable.
get_num_reqs_sent	Returns the number of requests that have been sent by this sequencer.
get_num_rsps_received	Returns the number of responses received thus far by this sequencer.
set_num_last_reqs	Sets the size of the last_requests buffer.
get_num_last_reqs	Returns the size of the last requests buffer, as set by set_num_last_reqs.
last_req	Returns the last request item by default.
set_num_last_rsps	Sets the size of the last_responses buffer.
get_num_last_rsps	Returns the max size of the last responses buffer, as set by set_num_last_rsps.
last_rsp	Returns the last response item by default.
execute_item	This task allows the user to supply an item or sequence to the sequencer and have it be executed procedurally.

PORTS

rsp_export

This is the analysis export used by drivers or monitors to send responses to the sequencer. When a driver wishes to send a response, it may do so through exactly one of three methods:

```
seq_item_port.item_done(response)
seq_item_done.put(response)
rsp_port.write(response)
```

The `rsp_port` in the driver and/or monitor must be connected to the `rsp_export` in this sequencer in order to send responses through the response analysis port.

METHODS

new

```
function new (string name,
              uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for `uvm_component`: `name` is the name of the instance, and `parent` is the handle to the hierarchical parent, if any.

send_request

```
virtual function void send_request(uvm_sequence_base sequence_ptr,
                                   uvm_sequence_item t,
                                   bit rerandomize = 0)
```

The `send_request` function may only be called after a `wait_for_grant` call. This call will send the request item, `t`, to the sequencer pointed to by `sequence_ptr`. The sequencer will forward it to the driver. If `rerandomize` is set, the item will be randomized before being sent to the driver.

get_current_item

```
function REQ get_current_item()
```

Returns the `request_item` currently being executed by the sequencer. If the sequencer is not currently executing an item, this method will return null.

The sequencer is executing an item from the time that `get_next_item` or `peek` is called until the time that `get` or `item_done` is called.

Note that a driver that only calls `get()` will never show a current item, since the item is completed at the same time as it is requested.

[start_default_sequence](#)

```
virtual task start_default_sequence()
```

Called when the run phase begins, this method starts the default sequence, as specified by the default_sequence member variable.

[get_num_reqs_sent](#)

```
function int get_num_reqs_sent()
```

Returns the number of requests that have been sent by this sequencer.

[get_num_rsps_received](#)

```
function int get_num_rsps_received()
```

Returns the number of responses received thus far by this sequencer.

[set_num_last_reqs](#)

```
function void set_num_last_reqs(int unsigned max)
```

Sets the size of the last_requests buffer. Note that the maximum buffer size is 1024. If max is greater than 1024, a warning is issued, and the buffer is set to 1024. The default value is 1.

[get_num_last_reqs](#)

```
function int unsigned get_num_last_reqs()
```

Returns the size of the last requests buffer, as set by set_num_last_reqs.

[last_req](#)

```
function REQ last_req(int unsigned n = 0)
```

Returns the last request item by default. If n is not 0, then it will get the nth before last request item. If n is greater than the last request buffer size, the function will return null.

[set_num_last_rsps](#)

```
function void set_num_last_rsps(int unsigned max)
```

Sets the size of the last_responses buffer. The maximum buffer size is 1024. If max is greater than 1024, a warning is issued, and the buffer is set to 1024. The default value is 1.

[get_num_last_rsps](#)

```
function int unsigned get_num_last_rsps()
```

Returns the max size of the last responses buffer, as set by set_num_last_rsps.

[last_rsp](#)

```
function RSP last_rsp(int unsigned n = 0)
```

Returns the last response item by default. If n is not 0, then it will get the nth-before-last response item. If n is greater than the last response buffer size, the function will return null.

[execute_item](#)

```
virtual task execute_item(uvm_sequence_item item)
```

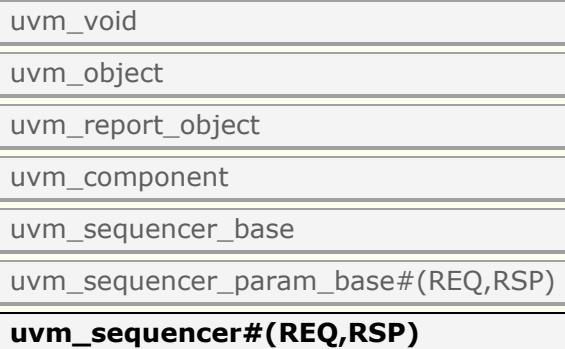
This task allows the user to supply an item or sequence to the sequencer and have it be executed procedurally. The parent sequence for the item or sequence is a temporary sequence that is automatically created. There is no capability to retrieve responses. The sequencer will drop responses to items done using this interface.

uvm_sequencer #(REQ,RSP)

Summary

uvm_sequencer #(REQ,RSP)

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_sequencer #(  
    type REQ = uvm_sequence_item,  
    type RSP = REQ  
) extends uvm_sequencer_param_base #(REQ, RSP)
```

VARIABLES

`seq_item_export` This export provides access to this sequencer's implementation of the sequencer interface, `<sqr_if_base #(REQ,RSP)>`, which defines the following methods:

METHODS

`new` Creates and initializes an instance of this class using the normal constructor arguments for `uvm_component`: name is the name of the instance, and parent is the handle to the hierarchical parent, if any.

`stop_sequences` Tells the sequencer to kill all sequences and child sequences currently operating on the sequencer, and remove all requests, locks and responses that are currently queued.

VARIABLES

seq_item_export

```
uvm_seq_item_pull_imp #(REQ,  
                      RSP,  
                      this_type) seq_item_export
```

This export provides access to this sequencer's implementation of the sequencer interface, `<sqr_if_base #(REQ,RSP)>`, which defines the following methods:

```
virtual task get_next_item (output REQ request);  
virtual task try_next_item (output REQ request);  
virtual function void item_done (input RSP response=null);  
virtual task wait_for_sequences();  
virtual function bit has do available();
```

```
virtual task get (output REQ request);
virtual task peek (output REQ request);
virtual task put (input RSP response);
```

See <sqr_if_base #(REQ,RSP)> for information about this interface.

METHODS

new

```
function new (string name,
              uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for uvm_component: name is the name of the instance, and parent is the handle to the hierarchical parent, if any.

stop_sequences

```
virtual function void stop_sequences()
```

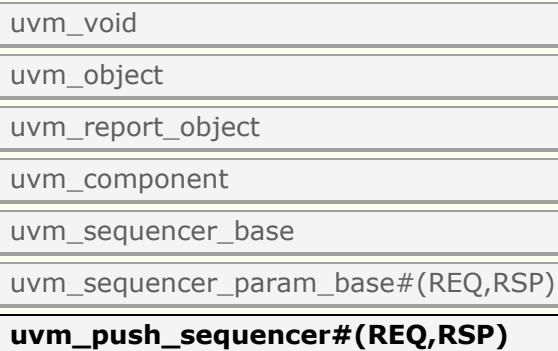
Tells the sequencer to kill all sequences and child sequences currently operating on the sequencer, and remove all requests, locks and responses that are currently queued. This essentially resets the sequencer to an idle state.

uvm_push_sequencer #(REQ,RSP)

Summary

uvm_push_sequencer #(REQ,RSP)

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_push_sequencer #(  
    type REQ = uvm_sequence_item,  
    type RSP = REQ  
) extends uvm_sequencer_param_base #(REQ, RSP)
```

POR

`req_port` The push sequencer requires access to a blocking put interface.

METHODS

`new` Creates and initializes an instance of this class using the normal constructor arguments for `uvm_component`: `name` is the name of the instance, and `parent` is the handle to the hierarchical parent, if any.

`run` The push sequencer continuously selects from its list of available sequences and sends the next item from the selected sequence out its `req_port` using `req_port.put(item)`.

PORTS

`req_port`

The push sequencer requires access to a blocking put interface. Continual sequence items, based on the list of available sequences loaded into this sequencer, are sent out this port.

METHODS

`new`

```
function new (string name,
```

```
uvm_component parent)
```

Creates and initializes an instance of this class using the normal constructor arguments for [uvm_component](#): *name* is the name of the instance, and *parent* is the handle to the hierarchical parent, if any.

run

```
task run()
```

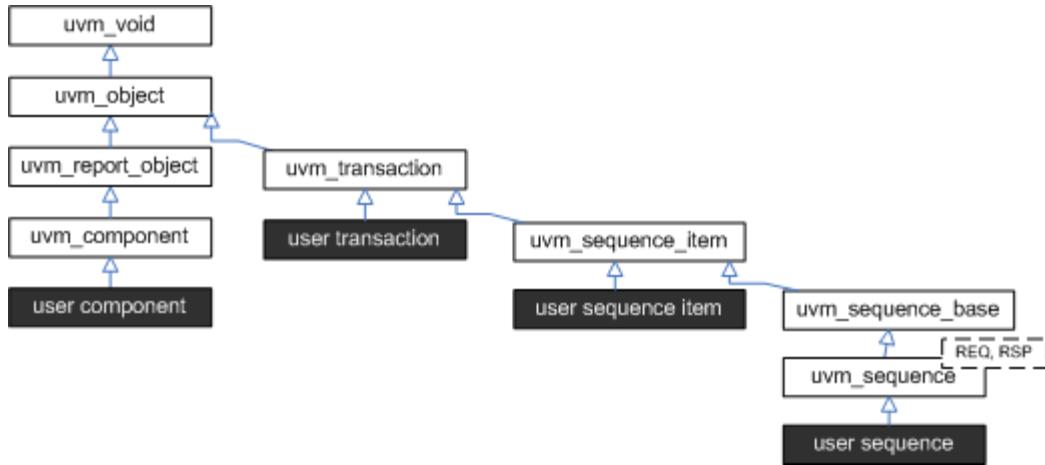
The push sequencer continuously selects from its list of available sequences and sends the next item from the selected sequence out its [req_port](#) using `req_port.put(item)`. Typically, the `req_port` would be connected to the `req_export` on an instance of an [uvm_push_driver #\(REQ,RSP\)](#), which would be responsible for executing the item.

Sequence Classes

Sequences encapsulate user-defined procedures that generate multiple `uvm_sequence_item`-based transactions. Such sequences can be reused, extended, randomized, and combined sequentially and hierarchically in interesting ways to produce realistic stimulus to your DUT.

With `uvm_sequence` objects, users can encapsulate DUT initializaton code, bus-based stress tests, network protocol stacks-- anything procedural-- then have them all execute in specific or random order to more quickly reach corner cases and coverage goals.

The UVM sequence item and sequence class hierarchy is shown below.



- `uvm_sequence_item` - The `uvm_sequence_item` is the base class for user-defined transactions that leverage the stimulus generation and control capabilities of the sequence-sequencer mechanism.
- `uvm_sequence #(REQ,RSP)` - The `uvm_sequence` extends `uvm_sequence_item` to add the ability to generate streams of `uvm_sequence_items`, either directly or by recursively executing other `uvm_sequences`.

uvm_sequence_item

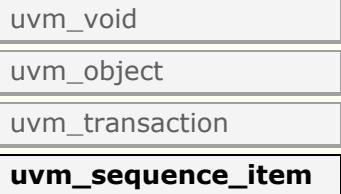
The base class for user-defined sequence items and also the base class for the uvm_sequence class. The uvm_sequence_item class provides the basic functionality for objects, both sequence items and sequences, to operate in the sequence mechanism.

Summary

uvm_sequence_item

The base class for user-defined sequence items and also the base class for the uvm_sequence class.

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_sequence_item extends uvm_transaction
```

METHODS

new	The constructor method for uvm_sequence_item.
get_sequence_id	private
set_use_sequence_info	These methods are used to set and get the status of the use_sequence_info bit.
get_use_sequence_info	Copies the sequence_id and transaction_id from the referenced item into the calling item.
set_id_info	
set_sequencer	These routines set and get the reference to the sequencer to which this sequence_item communicates.
get_sequencer	
set_parent_sequence	Sets the parent sequence of this sequence_item.
get_parent_sequence	Returns a reference to the parent sequence of any sequence on which this method was called.
set_depth	The depth of any sequence is calculated automatically.
get_depth	Returns the depth of a sequence from it's parent.
is_item	This function may be called on any sequence_item or sequence.
start_item	start_item and finish_item together will initiate operation of either a sequence_item or sequence object.
finish_item	finish_item, together with start_item together will initiate operation of either a sequence_item or sequence object.
get_root_sequence_name	Provides the name of the root sequence (the top-most parent sequence).
get_root_sequence	Provides a reference to the root sequence (the top-most parent sequence).
get_sequence_path	Provides a string of names of each sequence in the full hierarchical path.

METHODS

new

```
function new (string name = "uvm_sequence_item")
```

The constructor method for uvm_sequence_item.

get_sequence_id

```
function int get_sequence_id()
```

private

Get_sequence_id is an internal method that is not intended for user code. The sequence_id is not a simple integer. The get_transaction_id is meant for users to identify specific transactions.

These methods allow access to the sequence_item sequence and transaction IDs. get_transaction_id and set_transaction_id are methods on the uvm_transaction base_class. These IDs are used to identify sequences to the sequencer, to route responses back to the sequence that issued a request, and to uniquely identify transactions.

The sequence_id is assigned automatically by a sequencer when a sequence initiates communication through any sequencer calls (i.e. `uvm_do_xxx, wait_for_grant). A sequence_id will remain unique for this sequence until it ends or it is killed. However, a single sequence may have multiple valid sequence ids at any point in time. Should a sequence start again after it has ended, it will be given a new unique sequence_id.

The transaction_id is assigned automatically by the sequence each time a transaction is sent to the sequencer with the transaction_id in its default (-1) value. If the user sets the transaction_id to any non-default value, that value will be maintained.

Responses are routed back to this sequences based on sequence_id. The sequence may use the transaction_id to correlate responses with their requests.

set_use_sequence_info

```
function void set_use_sequence_info(bit value)
```

get_use_sequence_info

```
function bit get_use_sequence_info()
```

These methods are used to set and get the status of the use_sequence_info bit. Use_sequence_info controls whether the sequence information (sequencer, parent_sequence, sequence_id, etc.) is printed, copied, or recorded. When use_sequence_info is the default value of 0, then the sequence information is not used. When use_sequence_info is set to 1, the sequence information will be used in printing and copying.

set_id_info

```
function void set_id_info(uvm_sequence_item item)
```

Copies the sequence_id and transaction_id from the referenced item into the calling item. This routine should always be used by drivers to initialize responses for future compatibility.

set_sequencer

```
function void set_sequencer(uvm_sequencer_base sequencer)
```

get_sequencer

```
function uvm_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(uvm_sequence_base parent)
```

Sets the parent sequence of this sequence_item. This is used to identify the source sequence of a sequence_item.

get_parent_sequence

```
function uvm_sequence_base get_parent_sequence()
```

Returns a reference to the parent sequence of any sequence on which this method was called. If this is a parent sequence, the method returns null.

set_depth

```
function void set_depth(int value)
```

The depth of any sequence is calculated automatically. However, the user may use set_depth to specify the depth of a particular sequence. This method will override the automatically calculated depth, even if it is incorrect.

get_depth

```
function int get_depth()
```

Returns the depth of a sequence from its parent. A parent sequence will have a depth of 1, its child will have a depth of 2, and its grandchild will have a depth of 3.

is_item

```
virtual function bit is_item()
```

This function may be called on any sequence_item or sequence. It will return 1 for items and 0 for sequences (which derive from this class).

[start_item](#)

```
virtual task start_item(uvm_sequence_item item,  
int set_priority = -1)
```

start_item and finish_item together will initiate operation of either a sequence_item or sequence object. If the object has not been initiated using create_item, then start_item will be initialized in start_item to use the default sequencer specified by m_sequencer. Randomization may be done between start_item and finish_item to ensure late generation

[finish_item](#)

```
virtual task finish_item(uvm_sequence_item item,  
int set_priority = -1)
```

finish_item, together with start_item together will initiate operation of either a sequence_item or sequence object. Finish_item must be called after start_item with no delays or delta-cycles. Randomization, or other functions may be called between the start_item and finish_item calls.

[get_root_sequence_name](#)

```
function string get_root_sequence_name()
```

Provides the name of the root sequence (the top-most parent sequence).

[get_root_sequence](#)

```
function uvm_sequence_base get_root_sequence()
```

Provides a reference to the root sequence (the top-most parent sequence).

[get_sequence_path](#)

```
function string get_sequence_path()
```

Provides a string of names of each sequence in the full hierarchical path. A “.” is used as the separator between each sequence.

uvm_sequence_base

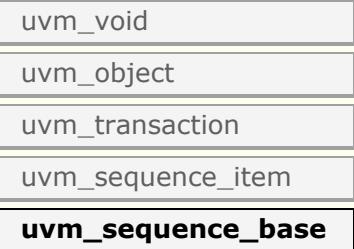
The uvm_sequence_base class provides the interfaces needed to create streams of sequence items and/or other sequences.

Summary

uvm_sequence_base

The uvm_sequence_base class provides the interfaces needed to create streams of sequence items and/or other sequences.

CLASS HIERARCHY



CLASS DECLARATION

```
class uvm_sequence_base extends uvm_sequence_item
```

VARIABLES

`seq_kind` Used as an identifier in constraints for a specific sequence type.

METHODS

`new` The constructor for uvm_sequence_base.

`get_sequence_state` Returns the sequence state as an enumerated value.

`wait_for_sequence_state` Waits until the sequence reaches the given state.

`start` The start task is called to begin execution of a sequence

`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`.

`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`.

`pre_do` 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.

`body` This is the user-defined task where the main sequence code resides.

`is_item` This function may be called on any sequence_item or sequence object.

`mid_do` This function is a user-definable callback function that is called after the sequence item has been randomized, and just before the item is sent to the driver.

`post_do` This function is a user-definable callback function that is called after the driver has indicated that it has completed the item, using either this `item_done` or `put` methods.

`num_sequences` Returns the number of sequences in the sequencer's sequence library.

<code>get_seq_kind</code>	This function returns an int representing the sequence kind that has been registered with the sequencer.
<code>get_sequence</code>	This function returns a reference to a sequence specified by req_kind, which can be obtained using the get_seq_kind method.
<code>get_sequence_by_name</code> <code>do_sequence_kind</code>	Internal method. This task will start a sequence of kind specified by req_kind, which can be obtained using the get_seq_kind method.
<code>set_priority</code>	The priority of a sequence may be changed at any point in time.
<code>get_priority</code>	This function returns the current priority of the sequence.
<code>wait_for_relevant</code>	This method is called by the sequencer when all available sequences are not relevant.
<code>is_relevant</code>	The default is_relevant implementation returns 1, indicating that the sequence is always relevant.
<code>is_blocked</code>	Returns a bit indicating whether this sequence is currently prevented from running due to another lock or grab.
<code>has_lock</code>	Returns 1 if this sequence has a lock, 0 otherwise.
<code>lock</code>	Requests a lock on the specified sequencer.
<code>grab</code>	Requests a lock on the specified sequencer.
<code>unlock</code>	Removes any locks or grabs obtained by this sequence on the specified sequencer.
<code>ungrab</code>	Removes any locks or grabs obtained by this sequence on the specified sequencer.
<code>wait_for_grant</code>	This task issues a request to the current sequencer.
<code>send_request</code>	The send_request function may only be called after a wait_for_grant call.
<code>wait_for_item_done</code>	A sequence may optionally call wait_for_item_done.
<code>set_sequencer</code>	Sets the default sequencer for the sequence to run on.
<code>get_sequencer</code>	Returns a reference to the current default sequencer of the sequence.
<code>kill</code>	This function will kill the sequence, and cause all current locks and requests in the sequence's default sequencer to be removed.
<code>use_response_handler</code>	When called with enable set to 1, responses will be sent to the response handler.
<code>get_use_response_handler</code>	Returns the state of the use_response_handler bit.
<code>response_handler</code>	When the use_response_handler bit is set to 1, this virtual task is called by the sequencer for each response that arrives for this sequence.
<code>create_item</code>	Create_item will create and initialize a sequence_item or sequence using the factory.
<code>start_item</code>	start_item and finish_item together will initiate operation of either a sequence_item or sequence object.
<code>finish_item</code>	finish_item, together with start_item together will initiate operation of either a sequence_item or sequence object.

VARIABLES

seq_kind

```
rand int unsigned seq_kind
```

Used as an identifier in constraints for a specific sequence type.

METHODS

new

```
function new (string name = "uvm_sequence")
```

The constructor for uvm_sequence_base.

get_sequence_state

```
function uvm_sequence_state_enum get_sequence_state()
```

Returns the sequence state as an enumerated value. Can use to wait on the sequence reaching or changing from one or more states.

```
wait(get_sequence_state() & (STOPPED|FINISHED));
```

wait_for_sequence_state

```
task wait_for_sequence_state(uvm_sequence_state_enum state)
```

Waits until the sequence reaches the given *state*. If the sequence is already in this state, this method returns immediately. Convenience for wait (get_sequence_state == *state*);

start

```
virtual task start (uvm_sequencer_base sequencer,
                    uvm_sequence_base parent_sequence = null,
                    integer           this_priority   = 100,
                    bit                call_pre_post    = 1      )
```

The start task is called to begin execution of a sequence

If parent_sequence is null, then the sequence is a parent, otherwise it is a child of the specified parent.

By default, the priority of a sequence is 100. A different priority may be specified by this_priority. Higher numbers indicate higher priority.

If call_pre_post is set to 1, then the pre_body and post_body tasks will be called before and after the sequence body is called.

pre_body

```
virtual task pre_body()
```

This task is a user-definable callback task that is called before the execution of the body, unless the sequence is started with call_pre_post=0. This method should not be called directly by the user.

post_body

```
virtual task post_body()
```

This task is a user-definable callback task that is called after the execution of the body, unless the sequence is started with call_pre_post=0. This method should not be called directly by the user.

pre_do

```
virtual task pre_do(bit is_item)
```

This task is a user-definable callback task that is called after the sequence has issued a wait_for_grant() call and after the sequencer has selected this sequence, and before the item is randomized. This method should not be called directly by the user.

Although pre_do is a task, consuming simulation cycles may result in unexpected behavior on the driver.

body

```
virtual task body()
```

This is the user-defined task where the main sequence code resides. This method should not be called directly by the user.

is_item

```
virtual function bit is_item()
```

This function may be called on any sequence_item or sequence object. It will return 1 on items and 0 on sequences.

mid_do

```
virtual function void mid_do(uvm_sequence_item this_item)
```

This function is a user-definable callback function that is called after the sequence item has been randomized, and just before the item is sent to the driver. This method should not be called directly by the user.

post_do

```
virtual function void post_do(uvm_sequence_item this_item)
```

This function is a user-definable callback function that is called after the driver has indicated that it has completed the item, using either this item_done or put methods. This method should not be called directly by the user.

[num_sequences](#)

```
function int num_sequences()
```

Returns the number of sequences in the sequencer's sequence library.

[get_seq_kind](#)

```
function int get_seq_kind(string type_name)
```

This function returns an int representing the sequence kind that has been registered with the sequencer. The seq_kind int may be used with the get_sequence or do_sequence_kind methods.

[get_sequence](#)

```
function uvm_sequence_base get_sequence(int unsigned req_kind)
```

This function returns a reference to a sequence specified by req_kind, which can be obtained using the get_seq_kind method.

[get_sequence_by_name](#)

```
function uvm_sequence_base get_sequence_by_name(string seq_name)
```

Internal method.

[do_sequence_kind](#)

```
task do_sequence_kind(int unsigned req_kind)
```

This task will start a sequence of kind specified by req_kind, which can be obtained using the get_seq_kind method.

[set_priority](#)

```
function void set_priority (int value)
```

The priority of a sequence may be changed at any point in time. When the priority of a sequence is changed, the new priority will be used by the sequencer the next time that it arbitrates between sequences.

The default priority value for a sequence is 100. Higher values result in higher priorities.

[get_priority](#)

```
function int get_priority()
```

This function returns the current priority of the sequence.

wait_for_relevant

```
virtual task wait_for_relevant()
```

This method is called by the sequencer when all available sequences are not relevant. When `wait_for_relevant` returns the sequencer attempt to re-arbitrate.

Returning from this call does not guarantee a sequence is relevant, although that would be the ideal. The method provide some delay to prevent an infinite loop.

If a sequence defines `is_relevant` so that it is not always relevant (by default, a sequence is always relevant), then the sequence must also supply a `wait_for_relevant` method.

is_relevant

```
virtual function bit is_relevant()
```

The default `is_relevant` implementation returns 1, indicating that the sequence is always relevant.

Users may choose to override with their own virtual function to indicate to the sequencer that the sequence is not currently relevant after a request has been made.

When the sequencer arbitrates, it will call `is_relevant` on each requesting, unblocked sequence to see if it is relevant. If a 0 is returned, then the sequence will not be chosen.

If all requesting sequences are not relevant, then the sequencer will call `wait_for_relevant` on all sequences and re-arbitrate upon its return.

Any sequence that implements `is_relevant` must also implement `wait_for_relevant` so that the sequencer has a way to wait for a sequence to become relevant.

is_blocked

```
function bit is_blocked()
```

Returns a bit indicating whether this sequence is currently prevented from running due to another lock or grab. A 1 is returned if the sequence is currently blocked. A 0 is returned if no lock or grab prevents this sequence from executing. Note that even if a sequence is not blocked, it is possible for another sequence to issue a lock or grab before this sequence can issue a request.

has_lock

```
function bit has_lock()
```

Returns 1 if this sequence has a lock, 0 otherwise.

Note that even if this sequence has a lock, a child sequence may also have a lock, in which case the sequence is still blocked from issuing operations on the sequencer>

lock

```
task lock(uvm_sequencer_base sequencer = null)
```

Requests a lock on the specified sequencer. If sequencer is null, the lock will be requested on the current default sequencer.

A lock request will be arbitrated the same as any other request. A lock is granted after all earlier requests are completed and no other locks or grabs are blocking this sequence.

The lock call will return when the lock has been granted.

grab

```
task grab(uvm_sequencer_base sequencer = null)
```

Requests a lock on the specified sequencer. If no argument is supplied, the lock will be requested on the current default sequencer.

A grab equest is put in front of the arbitration queue. It will be arbitrated before any other requests. A grab is granted when no other grabs or locks are blocking this sequence.

The grab call will return when the grab has been granted.

unlock

```
function void unlock(uvm_sequencer_base sequencer = null)
```

Removes any locks or grabs obtained by this sequence on the specified sequencer. If sequencer is null, then the unlock will be done on the current default sequencer.

ungrab

```
function void ungrab(uvm_sequencer_base sequencer = null)
```

Removes any locks or grabs obtained by this sequence on the specified sequencer. If sequencer is null, then the unlock will be done on the current default sequencer.

wait_for_grant

```
virtual task wait_for_grant(int item_priority = -1,  
                           bit lock_request = 0 )
```

This task issues a request to the current sequencer. If item_priority is not specified, then the current sequence priority will be used by the arbiter. If a lock_request is made, then the sequencer will issue a lock immediately before granting the sequence. (Note that the lock may be granted without the sequence being granted if is_relevant is not asserted).

When this method returns, the sequencer has granted the sequence, and the sequence must call send_request without inserting any simulation delay other than delta cycles. The driver is currently waiting for the next item to be sent via the send_request call.

send_request

```
virtual function void send_request(uvm_sequence_item request,  
                                bit rerandomize = 0)
```

The `send_request` function may only be called after a `wait_for_grant` call. This call will send the request item to the sequencer, which will forward it to the driver. If the `rerandomize` bit is set, the item will be randomized before being sent to the driver.

wait_for_item_done

```
virtual task wait_for_item_done(int transaction_id = -1)
```

A sequence may optionally call `wait_for_item_done`. This task will block until the driver calls `item_done` or `put`. If no `transaction_id` parameter is specified, then the call will return the next time that the driver calls `item_done` or `put`. If a specific `transaction_id` is specified, then the call will return when the driver indicates completion of that specific item.

Note that if a specific `transaction_id` has been specified, and the driver has already issued an `item_done` or `put` for that transaction, then the call will hang, having missed the earlier notification.

set_sequencer

```
virtual function void set_sequencer(uvm_sequencer_base sequencer)
```

Sets the default sequencer for the sequence to run on. It will take effect immediately, so it should not be called while the sequence is actively communicating with the sequencer.

get_sequencer

```
virtual function uvm_sequencer_base get_sequencer()
```

Returns a reference to the current default sequencer of the sequence.

kill

```
function void kill()
```

This function will kill the sequence, and cause all current locks and requests in the sequence's default sequencer to be removed. The sequence state will change to `STOPPED`, and its `post_body()` method, if will not be called.

If a sequence has issued locks, grabs, or requests on sequencers other than the default sequencer, then care must be taken to unregister the sequence with the other sequencer(s) using the sequencer `unregister_sequence()` method.

use_response_handler

```
function void use_response_handler(bit enable)
```

When called with enable set to 1, responses will be sent to the response handler. Otherwise, responses must be retrieved using get_response.

By default, responses from the driver are retrieved in the sequence by calling get_response.

An alternative method is for the sequencer to call the response_handler function with each response.

get_use_response_handler

```
function bit get_use_response_handler()
```

Returns the state of the use_response_handler bit.

response_handler

```
virtual function void response_handler(uvm_sequence_item response)
```

When the use_reponse_handler bit is set to 1, this virtual task is called by the sequencer for each response that arrives for this sequence.

create_item

```
protected function uvm_sequence_item create_item(  
    uvm_object_wrapper type_var,  
    uvm_sequencer_base l_sequencer,  
    string name  
)
```

Create_item will create and initialize a sequence_item or sequence using the factory. The sequence_item or sequence will be initialized to communicate with the specified sequencer.

start_item

start_item and finish_item together will initiate operation of either a sequence_item or sequence object. If the object has not been initiated using create_item, then start_item will be initialized in start_item to use the default sequencer specified by m_sequencer. Randomization may be done between start_item and finish_item to ensure late generation

```
virtual task start_item(uvm_sequence_item item, int set_priority = -1);
```

finish_item

finish_item, together with start_item together will initiate operation of either a sequence_item or sequence object. Finish_item must be called after start_item with no delays or delta-cycles. Randomization, or other functions may be called between the start_item and finish_item calls.

```
virtual task finish_item(uvm_sequence_item item, int set_priority = -1);
```

uvm_sequence #(REQ,RSP)

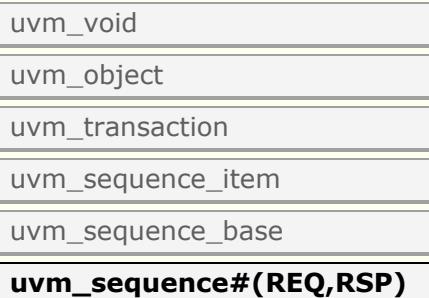
The uvm_sequence class provides the interfaces necessary in order to create streams of sequence items and/or other sequences.

Summary

uvm_sequence #(REQ,RSP)

The uvm_sequence class provides the interfaces necessary in order to create streams of sequence items and/or other sequences.

CLASS HIERARCHY



CLASS DECLARATION

```
virtual class uvm_sequence #(  
    type REQ = uvm_sequence_item,  
    type RSP = REQ  
) extends uvm_sequence_base
```

METHODS

<code>new</code>	Creates and initializes a new sequence object.
<code>start</code>	The start task is called to begin execution of a sequence.
<code>send_request</code>	This method will send the request item to the sequencer, which will forward it to the driver.
<code>get_current_item</code>	Returns the request item currently being executed by the sequencer.
<code>get_response</code>	By default, sequences must retrieve responses by calling <code>get_response</code> .
<code>set_sequencer</code>	Sets the default sequencer for the sequence to sequencer.
<code>set_response_queue_error_report_disabled</code>	By default, if the response_queue overflows, an error is reported.
<code>get_response_queue_error_report_disabled</code>	When this bit is 0 (default value), error reports are generated when the response queue overflows.
<code>set_response_queue_depth</code>	The default maximum depth of the response queue is 8.
<code>get_response_queue_depth</code>	Returns the current depth setting for the response queue.

METHODS

new

```
function new (string name = "uvm_sequence")
```

Creates and initializes a new sequence object.

start

```
virtual task start (uvm_sequencer_base sequencer,
                    uvm_sequence_base parent_sequence = null,
                    integer          this_priority   = 100,
                    bit              call_pre_post  = 1      )
```

The start task is called to begin execution of a sequence.

The *sequencer* argument specifies the sequencer on which to run this sequence. The sequencer must be compatible with the sequence.

If *parent_sequence* is null, then the sequence is a parent, otherwise it is a child of the specified parent.

By default, the *priority* of a sequence is 100. A different priority may be specified by *this_priority*. Higher numbers indicate higher priority.

If *call_pre_post* is set to 1, then the *pre_body* and *post_body* tasks will be called before and after the sequence body is called.

send_request

```
function void send_request(uvm_sequence_item request,
                           bit                      rerandomize = 0)
```

This method will send the request item to the sequencer, which will forward it to the driver. If the *rerandomize* bit is set, the item will be randomized before being sent to the driver. The *send_request* function may only be called after [uvm_sequence_base::wait_for_grant](#) returns.

get_current_item

```
function REQ get_current_item()
```

Returns the request item currently being executed by the sequencer. If the sequencer is not currently executing an item, this method will return null.

The sequencer is executing an item from the time that *get_next_item* or *peek* is called until the time that *get* or *item_done* is called.

Note that a driver that only calls *get* will never show a current item, since the item is completed at the same time as it is requested.

get_response

```
task get_response(output RSP response,  
                 input int transaction_id = -1)
```

By default, sequences must retrieve responses by calling `get_response`. If no `transaction_id` is specified, this task will return the next response sent to this sequence. If no response is available in the response queue, the method will block until a response is received.

If a `transaction_id` parameter is specified, the task will block until a response with that `transaction_id` is received in the response queue.

The default size of the response queue is 8. The `get_response` method must be called soon enough to avoid an overflow of the response queue to prevent responses from being dropped.

If a response is dropped in the response queue, an error will be reported unless the error reporting is disabled via `set_response_queue_error_report_disabled`.

[set_sequencer](#)

```
virtual function void set_sequencer(uvm_sequencer_base sequencer)
```

Sets the default sequencer for the sequence to sequencer. It will take effect immediately, so it should not be called while the sequence is actively communicating with the sequencer.

[set_response_queue_error_report_disabled](#)

```
function void set_response_queue_error_report_disabled(bit value)
```

By default, if the `response_queue` overflows, an error is reported. The `response_queue` will overflow if more responses are sent to this sequence from the driver than `get_response` calls are made. Setting `value` to 0 disables these errors, while setting it to 1 enables them.

[get_response_queue_error_report_disabled](#)

```
function bit get_response_queue_error_report_disabled()
```

When this bit is 0 (default value), error reports are generated when the response queue overflows. When this bit is 1, no such error reports are generated.

[set_response_queue_depth](#)

```
function void set_response_queue_depth(int value)
```

The default maximum depth of the response queue is 8. These method is used to examine or change the maximum depth of the response queue.

Setting the `response_queue_depth` to -1 indicates an arbitrarily deep response queue. No checking is done.

[get_response_queue_depth](#)

```
function int get_response_queue_depth()
```

Returns the current depth setting for the response queue.

uvm_random_sequence

This sequence randomly selects and executes a sequence from the sequencer's sequence library, excluding uvm_random_sequence itself, and uvm_exhaustive_sequence.

The uvm_random_sequence class is a built-in sequence that is preloaded into every sequencer's sequence library with the name "uvm_random_sequence".

The number of selections and executions is determined by the count property of the sequencer (or virtual sequencer) on which uvm_random_sequence is operating. See [uvm_sequencer_base](#) for more information.

Summary

uvm_random_sequence

This sequence randomly selects and executes a sequence from the sequencer's sequence library, excluding uvm_random_sequence itself, and uvm_exhaustive_sequence.

CLASS HIERARCHY

```
uvm_sequence#(uvm_sequence_item)  
|  
uvm_random_sequence
```

CLASS DECLARATION

```
class uvm_random_sequence extends uvm_sequence #(  
    uvm_sequence_item  
)
```

METHODS

[get_count](#) Returns the count of the number of sub-sequences which are randomly generated.

METHODS

[get_count](#)

```
function int unsigned get_count()
```

Returns the count of the number of sub-sequences which are randomly generated. By default, count is equal to the value from the sequencer's count variable. However, if the sequencer's count variable is -1, then a random value between 0 and sequencer.max_random_count (exclusive) is chosen. The sequencer's count variable is subsequently reset to the random value that was used. If get_count() is called before the sequence has started, the return value will be sequencer.count, which may be -1.

uvm_exhaustive_sequence

This sequence randomly selects and executes each sequence from the sequencer's sequence library once, excluding itself and uvm_random_sequence.

The uvm_exhaustive_sequence class is a built-in sequence that is preloaded into every sequencer's sequence library with the name "uvm_exhaustive_sequence".

Summary

uvm_exhaustive_sequence

This sequence randomly selects and executes each sequence from the sequencer's sequence library once, excluding itself and uvm_random_sequence.

CLASS HIERARCHY

```
uvm_sequence#(uvm_sequence_item)
```

```
uvm_exhaustive_sequence
```

CLASS DECLARATION

```
class uvm_exhaustive_sequence extends uvm_sequence #(  
    uvm_sequence_item  
)
```

uvm_simple_sequence

This sequence simply executes a single sequence item.

The item parameterization of the sequencer on which the uvm_simple_sequence is executed defines the actual type of the item executed.

The uvm_simple_sequence class is a built-in sequence that is preloaded into every sequencer's sequence library with the name "uvm_simple_sequence".

See [uvm_sequencer #\(REQ,RSP\)](#) for more information on running sequences.

Summary

uvm_simple_sequence

This sequence simply executes a single sequence item.

CLASS HIERARCHY

```
uvm_sequence#(uvm_sequence_item)
```

```
uvm_simple_sequence
```

CLASS DECLARATION

```
class uvm_simple_sequence extends uvm_sequence #(  
    uvm_sequence_item  
)
```

end

Report Macros

This set of macros provides wrappers around the `uvm_report_*` [Reporting](#) functions. The macros serve two essential purposes:

- To reduce the processing overhead associated with filtered out messages, a check is made against the report's verbosity setting and the action for the id/severity pair before any string formatting is performed. This affects only `'uvm_info` reports.
- The `'__FILE__` and `'__LINE__` information is automatically provided to the underlying `uvm_report_*` call. Having the file and line number from where a report was issued aides in debug. You can disable display of file and line information in reports by defining `UVM_DISABLE_REPORT_FILE_LINE` on the command line.

The macros also enforce a verbosity setting of `UVM_NONE` for warnings, errors and fatals so that they cannot be mistakenly turned off by setting the verbosity level too low (warning and errors can still be turned off by setting the actions appropriately).

To use the macros, replace the previous call to `uvm_report_*` with the corresponding macro.

```
//Previous calls to uvm_report_*
uvm_report_info("MYINFO1", $sformatf("val: %0d", val), UVM_LOW);
uvm_report_warning("MYWARN1", "This is a warning");
uvm_report_error("MYERR", "This is an error");
uvm_report_fatal("MYFATAL", "A fatal error has occurred");
```

The above code is replaced by

```
//New calls to `uvm_*
`uvm_info("MYINFO1", $sformatf("val: %0d", val), UVM_LOW)
`uvm_warning("MYWARN1", "This is a warning")
`uvm_error("MYERR", "This is an error")
`uvm_fatal("MYFATAL", "A fatal error has occurred")
```

Macros represent text substitutions, not statements, so they should not be terminated with semi-colons.

Summary

Report Macros

This set of macros provides wrappers around the `uvm_report_*` [Reporting](#) functions.

MACROS

<code>'uvm_info</code>	Calls <code>uvm_report_info</code> if <code>VERBOSITY</code> is lower than the configured verbosity of the associated reporter.
<code>'uvm_warning</code>	Calls <code>uvm_report_warning</code> with a verbosity of <code>UVM_NONE</code> .
<code>'uvm_error</code>	Calls <code>uvm_report_error</code> with a verbosity of <code>UVM_NONE</code> .
<code>'uvm_fatal</code>	Calls <code>uvm_report_fatal</code> with a verbosity of <code>UVM_NONE</code> .

MACROS

[`uvm_info](#)

Calls uvm_report_info if *VERBOSITY* is lower than the configured verbosity of the associated reporter. *ID* is given as the message tag and *MSG* is given as the message text. The file and line are also sent to the uvm_report_info call.

[`uvm_warning](#)

Calls uvm_report_warning with a verbosity of UVM_NONE. The message can not be turned off using the reporter's verbosity setting, but can be turned off by setting the action for the message. *ID* is given as the message tag and *MSG* is given as the message text. The file and line are also sent to the uvm_report_warning call.

[`uvm_error](#)

Calls uvm_report_error with a verbosity of UVM_NONE. The message can not be turned off using the reporter's verbosity setting, but can be turned off by setting the action for the message. *ID* is given as the message tag and *MSG* is given as the message text. The file and line are also sent to the uvm_report_error call.

[`uvm_fatal](#)

Calls uvm_report_fatal with a verbosity of UVM_NONE. The message can not be turned off using the reporter's verbosity setting, but can be turned off by setting the action for the message. *ID* is given as the message tag and *MSG* is given as the message text. The file and line are also sent to the uvm_report_fatal call.

Utility and Field Macros for Components and Objects

Summary

Utility and Field Macros for Components and Objects

UTILITY MACROS

The utility macros provide implementations of the `uvm_object::create` method, which is needed for cloning, and the `uvm_object::get_type_name` method, which is needed for a number of debugging features.

`'uvm_field_utils_begin`
`'uvm_field_utils_end`

These macros form a block in which `'uvm_field_*` macros can be placed.

`'uvm_object_utils`
`'uvm_object_param_utils`
`'uvm_object_utils_begin`
`'uvm_object_param_utils_begin`
`'uvm_object_utils_end`

`uvm_object`-based class declarations may contain one of the above forms of utility macros.

`'uvm_component_utils`
`'uvm_component_param_utils`
`'uvm_component_utils_begin`
`'uvm_component_param_utils_begin`
`'uvm_component_end`

`uvm_component`-based class declarations may contain one of the above forms of utility macros.

FIELD MACROS

The `'uvm_field_*` macros are invoked inside of the `'uvm_*_utils_begin` and `'uvm_*_utils_end` macro blocks to form “automatic” implementations of the core data methods: copy, compare, pack, unpack, record, print, and sprint.

`'UVM_FIELD_*` MACROS

Macros that implement data operations for scalar properties.

`'uvm_field_int`
`'uvm_field_object`
`'uvm_field_string`
`'uvm_field_enum`
`'uvm_field_real`
`'uvm_field_event`

Implements the data operations for any packed integral property.

Implements the data operations for an `uvm_object`-based property.

Implements the data operations for a string property.

Implements the data operations for an enumerated property.

Implements the data operations for any real property.

Implements the data operations for an event property.

`'UVM_FIELD_SARRAY_*` MACROS

Macros that implement data operations for one-dimensional static array properties.

`'uvm_field_sarray_int`
`'uvm_field_sarray_object`

Implements the data operations for a one-dimensional static array of integrals.

Implements the data operations for a one-dimensional static array of `uvm_object`-based objects.

<code>`uvm_field_sarray_string</code>	Implements the data operations for a one-dimensional static array of strings.
<code>`uvm_field_sarray_enum</code>	Implements the data operations for a one-dimensional static array of enums.
<code>* UVM_FIELD_ARRAY_* MACROS</code>	Macros that implement data operations for one-dimensional dynamic array properties.
<code>`uvm_field_array_int</code>	Implements the data operations for a one-dimensional dynamic array of integrals.
<code>`uvm_field_array_object</code>	Implements the data operations for a one-dimensional dynamic array of <code>uvm_object</code> -based objects.
<code>`uvm_field_array_string</code>	Implements the data operations for a one-dimensional dynamic array of strings.
<code>`uvm_field_array_enum</code>	Implements the data operations for a one-dimensional dynamic array of enums.
<code>* UVM_FIELD_QUEUE_* MACROS</code>	Macros that implement data operations for dynamic queues.
<code>`uvm_field_queue_int</code>	Implements the data operations for a queue of integrals.
<code>`uvm_field_queue_object</code>	Implements the data operations for a queue of <code>uvm_object</code> -based objects.
<code>`uvm_field_queue_string</code>	Implements the data operations for a queue of strings.
<code>`uvm_field_queue_enum</code>	Implements the data operations for a one-dimensional queue of enums.
<code>* UVM_FIELD_AA_*_STRING MACROS</code>	Macros that implement data operations for associative arrays indexed by <i>string</i> .
<code>`uvm_field_aa_int_string</code>	Implements the data operations for an associative array of integrals indexed by <i>string</i> .
<code>`uvm_field_aa_object_string</code>	Implements the data operations for an associative array of <code>uvm_object</code> -based objects indexed by <i>string</i> .
<code>`uvm_field_aa_string_string</code>	Implements the data operations for an associative array of strings indexed by <i>string</i> .
<code>* UVM_FIELD_AA_*_INT MACROS</code>	Macros that implement data operations for associative arrays indexed by an integral type.
<code>`uvm_field_aa_object_int</code>	Implements the data operations for an associative array of <code>uvm_object</code> -based objects indexed by the <i>int</i> data type.
<code>`uvm_field_aa_int_int</code>	Implements the data operations for an associative array of integral types indexed by the <i>int</i> data type.
<code>`uvm_field_aa_int_int_unsigned</code>	Implements the data operations for an associative array of integral types indexed by the <i>int unsigned</i> data type.
<code>`uvm_field_aa_int_integer</code>	Implements the data operations for an associative array of integral types indexed by the <i>integer</i> data type.
<code>`uvm_field_aa_int_integer_unsigned</code>	Implements the data operations for an associative array of integral types indexed by the <i>integer unsigned</i> data type.
<code>`uvm_field_aa_int_byte</code>	Implements the data operations for

<code>`uvm_field_aa_int_byte_unsigned</code>	an associative array of integral types indexed by the <i>byte</i> data type. Implements the data operations for an associative array of integral types indexed by the <i>byte unsigned</i> data type.
<code>`uvm_field_aa_int_shortint</code>	Implements the data operations for an associative array of integral types indexed by the <i>shortint</i> data type.
<code>`uvm_field_aa_int_shortint_unsigned</code>	Implements the data operations for an associative array of integral types indexed by the <i>shortint unsigned</i> data type.
<code>`uvm_field_aa_int_longint</code>	Implements the data operations for an associative array of integral types indexed by the <i>longint</i> data type.
<code>`uvm_field_aa_int_longint_unsigned</code>	Implements the data operations for an associative array of integral types indexed by the <i>longint unsigned</i> data type.
<code>`uvm_field_aa_int_key</code>	Implements the data operations for an associative array of integral types indexed by any integral key data type.
<code>`uvm_field_aa_int_enumkey</code>	Implements the data operations for an associative array of integral types indexed by any enumeration key data type.

UTILITY MACROS

The utility macros provide implementations of the `uvm_object::create` method, which is needed for cloning, and the `uvm_object::get_type_name` method, which is needed for a number of debugging features. They also register the type with the `uvm_factory`, and they implement a `get_type` method, which is used when configuring the factory. And they implement the virtual `uvm_object::get_object_type` method for accessing the factory proxy of an allocated object.

Below is an example usage of the utility and field macros. By using the macros, you do not have to implement any of the data methods to get all of the capabilities of an `uvm_object`.

```
class mydata extends uvm_object;
  string str;
  mydata subdata;
  int field;
  myenum e1;
  int queue[$];
`uvm_object_utils_begin(mydata) //requires ctor with default args
  `uvm_field_string(str, UVM_DEFAULT)
  `uvm_field_object(subdata, UVM_DEFAULT)
  `uvm_field_int(field, UVM_DEC) //use decimal radix
  `uvm_field_enum(myenum, e1, UVM_DEFAULT)
  `uvm_field_queue_int(queue, UVM_DEFAULT)
`uvm_object_utils_end
endclass
```

``uvm_field_utils_begin`

`uvm_field_utils_end

These macros form a block in which `uvm_field_* macros can be placed. Used as

```
`uvm_field_utils_begin(TYPE)
`uvm_field_* macros here
`uvm_field_utils_end
```

These macros do NOT perform factory registration, implement get_type_name, nor implement the create method. Use this form when you need custom implementations of these two methods, or when you are setting up field macros for an abstract class (i.e. virtual class).

`uvm_object_utils

`uvm_object_param_utils

`uvm_object_utils_begin

`uvm_object_param_utils_begin

`uvm_object_utils_end

uvm_object-based class declarations may contain one of the above forms of utility macros.

For simple objects with no field macros, use

```
`uvm_object_utils(TYPE)
```

For simple objects with field macros, use

```
`uvm_object_utils_begin(TYPE)
`uvm_field_* macro invocations here
`uvm_object_utils_end
```

For parameterized objects with no field macros, use

```
`uvm_object_param_utils(TYPE)
```

For parameterized objects, with field macros, use

```
`uvm_object_param_utils_begin(TYPE)
```

```
`uvm_field_* macro invocations here  
`uvm_object_utils_end
```

Simple (non-parameterized) objects use the `uvm_object_utils*` versions, which do the following:

- Implements `get_type_name`, which returns `TYPE` as a string
- Implements `create`, which allocates an object of type `TYPE` by calling its constructor with no arguments. `TYPE`'s constructor, if defined, must have default values on all its arguments.
- Registers the `TYPE` with the factory, using the string `TYPE` as the factory lookup string for the type.
- Implements the static `get_type()` method which returns a factory proxy object for the type.
- Implements the virtual `get_object_type()` method which works just like the static `get_type()` method, but operates on an already allocated object.

Parameterized classes must use the `uvm_object_param_utils*` versions. They differ from ``uvm_object_utils` only in that they do not supply a type name when registering the object with the factory. As such, name-based lookup with the factory for parameterized classes is not possible.

The macros with `_begin` suffixes are the same as the non-suffixed versions except that they also start a block in which ``uvm_field_*` macros can be placed. The block must be terminated by ``uvm_object_utils_end`.

Objects deriving from `uvm_sequence` must use the ``uvm_sequence_*` macros instead of these macros. See [`uvm_sequence_utils](#) for details.

[``uvm_component_utils`](#)

[``uvm_component_param_utils`](#)

[``uvm_component_utils_begin`](#)

[``uvm_component_param_utils_begin`](#)

[``uvm_component_end`](#)

`uvm_component`-based class declarations may contain one of the above forms of utility macros.

For simple components with no field macros, use

```
`uvm_component_utils(TYPE)
```

For simple components with field macros, use

```
`uvm_component_utils_begin(TYPE)
`uvm_field_* macro invocations here
`uvm_component_utils_end
```

For parameterized components with no field macros, use

```
`uvm_component_param_utils(TYPE)
```

For parameterized components with field macros, use

```
`uvm_component_param_utils_begin(TYPE)
`uvm_field_* macro invocations here
`uvm_component_utils_end
```

Simple (non-parameterized) components must use the uvm_components_utils* versions, which do the following:

- Implements get_type_name, which returns TYPE as a string.
- Implements create, which allocates a component of type TYPE using a two argument constructor. TYPE's constructor must have a name and a parent argument.
- Registers the TYPE with the factory, using the string TYPE as the factory lookup string for the type.
- Implements the static get_type() method which returns a factory proxy object for the type.
- Implements the virtual get_object_type() method which works just like the static get_type() method, but operates on an already allocated object.

Parameterized classes must use the uvm_object_param_utils* versions. They differ from `uvm_object_utils only in that they do not supply a type name when registering the object with the factory. As such, name-based lookup with the factory for parameterized classes is not possible.

The macros with _begin suffixes are the same as the non-suffixed versions except that they also start a block in which `uvm_field_* macros can be placed. The block must be terminated by `uvm_component_utils_end.

Components deriving from uvm_sequencer must use the `uvm_sequencer_* macros instead of these macros. See `uvm_sequencer_utils for details.

FIELD MACROS

The `uvm_field_* macros are invoked inside of the `uvm_*_utils_begin and `uvm_*_utils_end macro blocks to form "automatic" implementations of the core data methods: copy, compare, pack, unpack, record, print, and sprint. For example:

```
class my_trans extends uvm_transaction;
  string my_string;
  `uvm_object_utils_begin(my_trans)
    `uvm_field_string(my_string, UVM_ALL_ON)
  `uvm_object_utils_end
endclass
```

Each `uvm_field_* macro is named to correspond to a particular data type: integrals, strings, objects, queues, etc., and each has at least two arguments: *ARG* and *FLAG*.

ARG is the instance name of the variable, whose type must be compatible with the macro being invoked. In the example, class variable *my_string* is of type string, so we use the `uvm_field_string macro.

If *FLAG* is set to *UVM_ALL_ON*, as in the example, the *ARG* variable will be included in all data methods. The *FLAG*, if set to something other than *UVM_ALL_ON* or *UVM_DEFAULT*, specifies which data method implementations will NOT include the given variable. Thus, if *FLAG* is specified as *NO_COMPARE*, the *ARG* variable will not affect comparison operations, but it will be included in everything else.

All possible values for *FLAG* are listed and described below. Multiple flag values can be bitwise ORed together (in most cases they may be added together as well, but care must be taken when using the + operator to ensure that the same bit is not added more than once).

<i>UVM_ALL_ON</i>	Set all operations on (default).
<i>UVM_DEFAULT</i>	Use the default flag settings.
<i>UVM_NOCOPY</i>	Do not copy this field.
<i>UVM_NOCOMPARE</i>	Do not compare this field.
<i>UVM_NOPRINT</i>	Do not print this field.
<i>UVM_NODEFPRINT</i>	Do not print the field if it is the same as its
<i>UVM_NOPACK</i>	Do not pack or unpack this field.
<i>UVM_PHYSICAL</i>	Treat as a physical field. Use physical setting in policy class for this field.
<i>UVM_ABSTRACT</i>	Treat as an abstract field. Use the abstract setting in the policy class for this field.
<i>UVM_READONLY</i>	Do not allow setting of this field from the <i>set_*_local</i> methods.

A radix for printing and recording can be specified by OR'ing one of the following constants in the *FLAG* argument

<i>UVM_BIN</i>	Print / record the field in binary (base-2).
<i>UVM_DEC</i>	Print / record the field in decimal (base-10).
<i>UVM_UNSIGNED</i>	Print / record the field in unsigned decimal (base-10).
<i>UVM_OCT</i>	Print / record the field in octal (base-8).
<i>UVM_HEX</i>	Print / record the field in hexadecimal (base-16).
<i>UVM_STRING</i>	Print / record the field in string format.
<i>UVM_TIME</i>	Print / record the field in time format.

Radix settings for integral types. Hex is the default radix if none is specified.

`UVM_FIELD_* MACROS

Macros that implement data operations for scalar properties.

`uvm_field_int

Implements the data operations for any packed integral property.

```
`uvm_field_int(ARG,FLAG)
```

ARG is an integral property of the class, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

[**`uvm_field_object**](#)

Implements the data operations for an [uvm_object](#)-based property.

```
`uvm_field_object(ARG,FLAG)
```

ARG is an object property of the class, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

[**`uvm_field_string**](#)

Implements the data operations for a string property.

```
`uvm_field_string(ARG,FLAG)
```

ARG is a string property of the class, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

[**`uvm_field_enum**](#)

Implements the data operations for an enumerated property.

```
`uvm_field_enum(T,ARG,FLAG)
```

T is an enumerated [type](#), *ARG* is an instance of that type, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

[**`uvm_field_real**](#)

Implements the data operations for any real property.

```
`uvm_field_real(ARG,FLAG)
```

ARG is a real property of the class, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_event

Implements the data operations for an event property.

```
`uvm_field_event(ARG,FLAG)
```

ARG is an event property of the class, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`UVM_FIELD_SARRAY_* MACROS

Macros that implement data operations for one-dimensional static array properties.

`uvm_field_sarray_int

Implements the data operations for a one-dimensional static array of integrals.

```
`uvm_field_sarray_int(ARG,FLAG)
```

ARG is a one-dimensional static array of integrals, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_sarray_object

Implements the data operations for a one-dimensional static array of [uvm_object](#)-based objects.

```
`uvm_field_sarray_object(ARG,FLAG)
```

ARG is a one-dimensional static array of [uvm_object](#)-based objects, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_sarray_string

Implements the data operations for a one-dimensional static array of strings.

```
`uvm_field_sarray_string(ARG,FLAG)
```

ARG is a one-dimensional static array of strings, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_sarray_enum

Implements the data operations for a one-dimensional static array of enums.

```
`uvm_field_sarray_enum(T,ARG,FLAG)
```

T is a one-dimensional dynamic array of enums [type](#), *ARG* is an instance of that type, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

\`UVM_FIELD_ARRAY_* MACROS

Macros that implement data operations for one-dimensional dynamic array properties.

\`uvm_field_array_int

Implements the data operations for a one-dimensional dynamic array of integrals.

```
`uvm_field_array_int(ARG,FLAG)
```

ARG is a one-dimensional dynamic array of integrals, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

\`uvm_field_array_object

Implements the data operations for a one-dimensional dynamic array of [uvm_object](#)-based objects.

```
`uvm_field_array_object(ARG,FLAG)
```

ARG is a one-dimensional dynamic array of [uvm_object](#)-based objects, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

\`uvm_field_array_string

Implements the data operations for a one-dimensional dynamic array of strings.

```
`uvm_field_array_string(ARG,FLAG)
```

ARG is a one-dimensional dynamic array of strings, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

\`uvm_field_array_enum

Implements the data operations for a one-dimensional dynamic array of enums.

```
`uvm_field_array_enum(T,ARG,FLAG)
```

T is a one-dimensional dynamic array of enums [type](#), *ARG* is an instance of that type, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`UVM_FIELD_QUEUE_* MACROS

Macros that implement data operations for dynamic queues.

`uvm_field_queue_int

Implements the data operations for a queue of integrals.

```
`uvm_field_queue_int(ARG,FLAG)
```

ARG is a one-dimensional queue of integrals, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_queue_object

Implements the data operations for a queue of [uvm_object](#)-based objects.

```
`uvm_field_queue_object(ARG,FLAG)
```

ARG is a one-dimensional queue of [uvm_object](#)-based objects, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_queue_string

Implements the data operations for a queue of strings.

```
`uvm_field_queue_string(ARG,FLAG)
```

ARG is a one-dimensional queue of strings, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_queue_enum

Implements the data operations for a one-dimensional queue of enums.

```
`uvm_field_queue_enum(T,ARG,FLAG)
```

T is a queue of enums `type`, *ARG* is an instance of that type, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

UVM_FIELD_AA_*_STRING MACROS

Macros that implement data operations for associative arrays indexed by *string*.

uvm_field_aa_int_string

Implements the data operations for an associative array of integrals indexed by *string*.

```
`uvm_field_aa_int_string(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with `string` key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

uvm_field_aa_object_string

Implements the data operations for an associative array of [uvm_object](#)-based objects indexed by *string*.

```
`uvm_field_aa_object_string(ARG,FLAG)
```

ARG is the name of a property that is an associative array of objects with `string` key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

uvm_field_aa_string_string

Implements the data operations for an associative array of strings indexed by *string*.

```
`uvm_field_aa_string_string(ARG,FLAG)
```

ARG is the name of a property that is an associative array of strings with `string` key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

UVM_FIELD_AA_*_INT MACROS

Macros that implement data operations for associative arrays indexed by an integral type.

[`uvm_field_aa_object_int](#)

Implements the data operations for an associative array of [uvm_object](#)-based objects indexed by the *int* data type.

```
`uvm_field_aa_object_int(ARG,FLAG)
```

ARG is the name of a property that is an associative array of objects with *int* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

[`uvm_field_aa_int_int](#)

Implements the data operations for an associative array of integral types indexed by the *int* data type.

```
`uvm_field_aa_int_int(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *int* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

[`uvm_field_aa_int_int_unsigned](#)

Implements the data operations for an associative array of integral types indexed by the *int unsigned* data type.

```
`uvm_field_aa_int_int_unsigned(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *int unsigned* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

[`uvm_field_aa_int_integer](#)

Implements the data operations for an associative array of integral types indexed by the *integer* data type.

```
`uvm_field_aa_int_integer(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *integer* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

[`uvm_field_aa_int_integer_unsigned](#)

Implements the data operations for an associative array of integral types indexed by the

integer unsigned data type.

```
`uvm_field_aa_int_integer_unsigned(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *integer unsigned* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

[**`uvm_field_aa_int_byte**](#)

Implements the data operations for an associative array of integral types indexed by the *byte* data type.

```
`uvm_field_aa_int_byte(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *byte* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

[**`uvm_field_aa_int_byte_unsigned**](#)

Implements the data operations for an associative array of integral types indexed by the *byte unsigned* data type.

```
`uvm_field_aa_int_byte_unsigned(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *byte unsigned* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

[**`uvm_field_aa_int_shortint**](#)

Implements the data operations for an associative array of integral types indexed by the *shortint* data type.

```
`uvm_field_aa_int_shortint(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *shortint* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

[**`uvm_field_aa_int_shortint_unsigned**](#)

Implements the data operations for an associative array of integral types indexed by the *shortint unsigned* data type.

```
`uvm_field_aa_int_shortint_unsigned(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *shortint unsigned* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

[`uvm_field_aa_int_longint`](#)

Implements the data operations for an associative array of integral types indexed by the *longint* data type.

```
`uvm_field_aa_int_longint(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *longint* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

[`uvm_field_aa_int_longint_unsigned`](#)

Implements the data operations for an associative array of integral types indexed by the *longint unsigned* data type.

```
`uvm_field_aa_int_longint_unsigned(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *longint unsigned* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

[`uvm_field_aa_int_key`](#)

Implements the data operations for an associative array of integral types indexed by any integral key data type.

```
`uvm_field_aa_int_key(long unsigned,ARG,FLAG)
```

KEY is the data type of the integral key, *ARG* is the name of a property that is an associative array of integrals, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

[`uvm_field_aa_int_enumkey`](#)

Implements the data operations for an associative array of integral types indexed by any enumeration key data type.

```
`uvm_field_aa_int_longint_unsigned(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *longint unsigned* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

Sequence and Do Action Macros

Summary

Sequence and Do Action Macros

SEQUENCE REGISTRATION MACROS

```
`uvm_declare_p_sequencer
```

The sequence-specific macros perform the same function as the set of `uvm_object_*_utils macros, except they also set the default sequencer type the sequence will run on.

```
`uvm_sequence_utils_begin  
`uvm_sequence_utils_end  
`uvm_sequence_utils
```

This macro is used to set up a specific sequencer type with the sequence type the macro is placed in.

The sequence macros can be used in non-parameterized `<uvm_sequence>` extensions to pre-register the sequence with a given `<uvm_sequencer>` type.

SEQUENCER REGISTRATION MACROS

```
`uvm_update_sequence_lib
```

The sequencer-specific macros perform the same function as the set of `uvm_component_*_utils macros except that they also declare the plumbing necessary for creating the sequencer's sequence library.

```
`uvm_update_sequence_lib_and_item
```

This macro populates the instance-specific sequence library for a sequencer.

This macro populates the instance specific sequence library for a sequencer, and it registers the given `USER_ITEM` as an instance override for the simple sequence's item variable.

```
`uvm_sequencer_utils  
`uvm_sequencer_utils_begin  
`uvm_sequencer_param_utils  
`uvm_sequencer_param_utils_begin  
`uvm_sequencer_utils_end
```

The sequencer macros are used in `uvm_sequencer`-based class declarations in one of four ways.

SEQUENCE ACTION MACROS

```
`uvm_create
```

These macros are used to start sequences and sequence items that were either registered with a `<uvm_sequence_utils>` macro or whose associated sequencer was already set using the `<set_sequencer>` method.

This action creates the item or sequence using the factory.

This macro takes as an argument a `uvm_sequence_item` variable or object.

```
`uvm_do
```

This is the same as ``uvm_do` except that the sequence item or sequence is executed with the priority specified in the argument

```
`uvm_do_pri
```

This is the same as ``uvm_do` except that the constraint block in the 2nd argument is applied to the

<code>'uvm_do_pri_with</code>	item or sequence in a randomize with statement before execution. This is the same as <code>'uvm_do_pri</code> except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.
<code>'uvm_send</code>	This macro processes the item or sequence that has been created using <code>'uvm_create</code> .
<code>'uvm_send_pri</code>	This is the same as <code>'uvm_send</code> except that the sequence item or sequence is executed with the priority specified in the argument.
<code>'uvm_rand_send</code>	This macro processes the item or sequence that has been already been allocated (possibly with <code>'uvm_create</code>).
<code>'uvm_rand_send_pri</code>	This is the same as <code>'uvm_rand_send</code> except that the sequence item or sequence is executed with the priority specified in the argument.
<code>'uvm_rand_send_with</code>	This is the same as <code>'uvm_rand_send</code> except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.
<code>'uvm_rand_send_pri_with</code>	This is the same as <code>'uvm_rand_send_pri</code> except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.
SEQUENCE ON SEQUENCER ACTION MACROS	
<code>'uvm_create_on</code>	These macros are used to start sequences and sequence items on a specific sequencer, given in a macro argument.
<code>'uvm_do_on</code>	This is the same as <code>'uvm_create</code> except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified <i>SEQUENCER_REF</i> argument.
<code>'uvm_do_on_pri</code>	This is the same as <code>'uvm_create</code> except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified <i>SEQUENCER_REF</i> argument.
<code>'uvm_do_on_with</code>	This is the same as <code>'uvm_create</code> except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified <i>SEQUENCER_REF</i> argument.
<code>'uvm_do_on_pri_with</code>	This is the same as <code>'uvm_create</code> except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified <i>SEQUENCER_REF</i> argument.

to the specified *SEQUENCER_REF* argument.

SEQUENCE REGISTRATION MACROS

The sequence-specific macros perform the same function as the set of `uvm_object_*_utils macros, except they also set the default sequencer type the sequence will run on.

`uvm_declare_p_sequencer

This macro is used to set up a specific sequencer type with the sequence type the macro is placed in. This macro is implicit in the <uvm_sequence_utils> macro, but may be used directly in cases when the sequence is not to be registered in the sequencer's library.

The example below shows using the the uvm_declare_p_sequencer macro along with the uvm_object_utils macros to set up the sequence but not register the sequence in the sequencer's library.

```
class mysequence extends uvm_sequence#(mydata);
    `uvm_object_utils(mysequence)
    `uvm_declare_p_sequencer(some_seqr_type)
    task body;
        //Access some variable in the user's custom sequencer
        if(p_sequencer.some_variable) begin
            ...
        end
    endtask
endclass
```

`uvm_sequence_utils_begin

`uvm_sequence_utils_end

`uvm_sequence_utils

The sequence macros can be used in non-parameterized <uvm_sequence> extensions to pre-register the sequence with a given <uvm_sequencer> type.

For sequences that do not use any `uvm_field macros

```
`uvm_sequence_utils(TYPE_NAME,SQR_TYPE_NAME)
```

For sequences employing with field macros

```
`uvm_sequence_utils_begin(TYPE_NAME,SQR_TYPE_NAME)
`uvm_field_* macro invocations here
```

```
`uvm_sequence_utils_end
```

The sequence-specific macros perform the same function as the set of `uvm_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 `uvm_sequence_utils[_begin] for non-parameterized classes and `uvm_sequence_param_utils[_begin] for parameterized classes.

SEQUENCER REGISTRATION MACROS

The sequencer-specific macros perform the same function as the set of `uvm_componenent_*utils macros except that they also declare the plumbing necessary for creating the sequencer's sequence library.

[`uvm_update_sequence_lib](#)

This macro populates the instance-specific sequence library for a sequencer. It should be invoked inside the sequencer's constructor.

[`uvm_update_sequence_lib_and_item](#)

This macro populates the instance specific sequence library for a sequencer, and it registers the given *USER_ITEM* as an instance override for the simple sequence's item variable.

The macro should be invoked inside the sequencer's constructor.

[`uvm_sequencer_utils](#)

[`uvm_sequencer_utils_begin](#)

[`uvm_sequencer_param_utils](#)

[`uvm_sequencer_param_utils_begin](#)

[`uvm_sequencer_utils_end](#)

The sequencer macros are used in uvm_sequencer-based class declarations in one of four ways.

For simple sequencers, no field macros

```
`uvm_sequencer_utils(SQR_TYPE_NAME)
```

For simple sequencers, with field macros

```
`uvm_sequencer_utils_begin(SQR_TYPE_NAME) `uvm_field_* macros here  
`uvm_sequencer_utils_end
```

For parameterized sequencers, no field macros

```
`uvm_sequencer_param_utils(SQR_TYPE_NAME)
```

For parameterized sequencers, with field macros

```
`uvm_sequencer_param_utils_begin(SQR_TYPE_NAME) `uvm_field_* macros here  
`uvm_sequencer_utils_end
```

The sequencer-specific macros perform the same function as the set of

`'uvm_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 `'uvm_sequencer_utils[_begin]` for non-parameterized classes and

`'uvm_sequencer_param_utils[_begin]` for parameterized classes.

SEQUENCE ACTION MACROS

These macros are used to start sequences and sequence items that were either registered with a `<'uvm-sequence_utils>` macro or whose associated sequencer was already set using the `<set_sequencer>` method.

[`uvm_create](#)

This action creates the item or sequence using the factory. It intentionally does zero processing. After this action completes, the user can manually set values, manipulate `rand_mode` and `constraint_mode`, etc.

[`uvm_do](#)

This macro takes as an argument a `uvm_sequence_item` variable or object. `uvm_sequence_item`'s are randomized at the time the sequencer grants the do request. This is called late-randomization or late-generation. In the case of a sequence a sub-sequence is spawned. In the case of an item, the item is sent to the driver through the associated sequencer.

[`uvm_do_pri](#)

This is the same as ``uvm_do` except that the sequene item or sequence is executed with the priority specified in the argument

`uvm_do_with

This is the same as `uvm_do except that the constraint block in the 2nd argument is applied to the item or sequence in a randomize with statement before execution.

`uvm_do_pri_with

This is the same as `uvm_do_pri except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.

`uvm_send

This macro processes the item or sequence that has been created using `uvm_create. The processing is done without randomization. Essentially, an `uvm_do without the create or randomization.

`uvm_send_pri

This is the same as `uvm_send except that the sequene item or sequence is executed with the priority specified in the argument.

`uvm_rand_send

This macro processes the item or sequence that has been already been allocated (possibly with `uvm_create). The processing is done with randomization. Essentially, an `uvm_do without the create.

`uvm_rand_send_pri

This is the same as `uvm_rand_send except that the sequene item or sequence is executed with the priority specified in the argument.

`uvm_rand_send_with

This is the same as `uvm_rand_send except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.

`uvm_rand_send_pri_with

This is the same as `uvm_rand_send_pri except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.

SEQUENCE ON SEQUENCER ACTION MACROS

These macros are used to start sequences and sequence items on a specific sequencer, given in a macro argument.

``uvm_create_on`

This is the same as ``uvm_create` except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *SEQUENCER_REF* argument.

``uvm_do_on`

This is the same as ``uvm_do` except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *SEQUENCER_REF* argument.

``uvm_do_on_pri`

This is the same as ``uvm_do_pri` except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *SEQUENCER_REF* argument.

``uvm_do_on_with`

This is the same as ``uvm_do_with` except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *SEQUENCER_REF* argument. The user must supply brackets around the constraints.

``uvm_do_on_pri_with`

This is the same as ``uvm_do_pri_with` except that it also sets the parent sequence to the sequence in which the macro is invoked, and it sets the sequencer to the specified *SEQUENCER_REF* argument.

uvm_callbackDefines

Summary

uvm_callbackDefines

CALLBACK MACROS

`uvm_register_cb	Registers the given <i>CB</i> callback type with the given <i>T</i> object type.
`uvm_set_super_type	Defines the super type of <i>T</i> to be <i>ST</i> .
`uvm_do_callbacks	Calls the given <i>METHOD</i> of all callbacks of type <i>CB</i> registered with the calling object (i.e.
`uvm_do_obj_callbacks	Calls the given <i>METHOD</i> of all callbacks based on type <i>CB</i> registered with the given object, <i>OBJ</i> , which is or is based on type <i>T</i> .
`uvm_do_callbacks_exit_on	Calls the given <i>METHOD</i> of all callbacks of type <i>CB</i> registered with the calling object (i.e.
`uvm_do_obj_callbacks_exit_on	Calls the given <i>METHOD</i> of all callbacks of type <i>CB</i> registered with the given object <i>OBJ</i> , which must be or be based on type <i>T</i> , and returns upon the first callback that returns the bit value given by <i>VAL</i> .

CALLBACK MACROS

`uvm_register_cb

Registers the given *CB* callback type with the given *T* object type. If a type-callback pair is not registered then a warning is issued if an attempt is made to use the pair (add, delete, etc.).

The registration will typically occur in the component that executes the given type of callback. For instance:

```
virtual class mycb;
    virtual function void doit();
endclass

class my_comp extends uvm_component;
    `uvm_register_cb(my_comp,mycb)
    ...
    task run;
        ...
        `uvm_do_callbacks(my_comp, mycb, doit())
    endtask
endclass
```

`uvm_set_super_type

Defines the super type of *T* to be *ST*. This allows for derived class objects to inherit typewide callbacks that are registered with the base class.

The registration will typically occur in the component that executes the given type of callback. For instance:

```
virtual class mycb;
  virtual function void doit();
endclass

class my_comp extends uvm_component;
  `uvm_register_cb(my_comp,mycb)
  ...
  task run;
    ...
    `uvm_do_callbacks(my_comp, mycb, doit())
  endtask
endclass

class my_derived_comp extends my_comp;
  `uvm_set_super_type(my_derived_comp,my_comp)
  ...
  task run;
    ...
    `uvm_do_callbacks(my_comp, mycb, doit())
  endtask
endclass
```

`uvm_do_callbacks

Calls the given *METHOD* of all callbacks of type *CB* registered with the calling object (i.e. *this* object), which is or is based on type *T*.

This macro executes all of the callbacks associated with the calling object (i.e. *this* object). The macro takes three arguments:

- CB is the class type of the callback objects to execute. The class type must have a function signature that matches the *METHOD* argument.
- T is the type associated with the callback. Typically, an instance of type T is passed as one the arguments in the *METHOD* call.
- METHOD is the method call to invoke, with all required arguments as if they were invoked directly.

For example, given the following callback class definition

```
virtual class mycb extends uvm_cb;
  pure function void my_function (mycomp comp, int addr, int data);
endclass
```

A component would invoke the macro as

```
task mycomp::run();
  int curr_addr, curr_data;
  ...
  `uvm_do_callbacks(mycb, mycomp, my_function(this, curr_addr, curr_data))
  ...
endtask
```

`uvm_do_obj_callbacks

Calls the given *METHOD* of all callbacks based on type *CB* registered with the given object, *OBJ*, which is or is based on type *T*.

This macro is identical to `uvm_do_callbacks macro, but it has an additional *OBJ* argument to allow the specification of an external object to associate the callback with. For example, if the callbacks are being applied in a sequence, *OBJ* could be specified as the associated sequencer or parent sequence.

```
...  
`uvm_do_callbacks(mycb, mycomp, seqr, my_function(seqr, curr_addr,  
curr_data))  
...
```

[`uvm_do_callbacks_exit_on](#)

Calls the given *METHOD* of all callbacks of type *CB* registered with the calling object (i.e. *this* object), which is or is based on type *T*, returning upon the first callback returning the bit value given by *VAL*.

This macro executes all of the callbacks associated with the calling object (i.e. *this* object). The macro takes three arguments:

- *CB* is the class type of the callback objects to execute. The class type must have a function signature that matches the *METHOD* argument.
- *T* is the type associated with the callback. Typically, an instance of type *T* is passed as one the arguments in the *METHOD* call.
- *METHOD* is the method call to invoke, with all required arguments as if they were invoked directly.
- *VAL*, if 1, says return upon the first callback invocation that returns 1. If 0, says return upon the first callback invocation that returns 0.

For example, given the following callback class definition

```
virtual class mycb extends uvm_cb;  
  pure function bit drop_trans (mycomp comp, my_trans trans);  
endclass
```

A component would invoke the macro as

```
task mycomp::run();  
  my_trans trans;  
  forever begin  
    get_port.get(trans);  
    if (`uvm_do_callbacks_exit_on(mycomp, mycb, extobj,  
drop_trans(this,trans), 1)  
      uvm_report_info("DROPPED", {"trans dropped:  
%s",trans.convert2string()});  
      // execute transaction  
    end  
  endtask
```

[`uvm_do_obj_callbacks_exit_on](#)

Calls the given *METHOD* of all callbacks of type *CB* registered with the given object *OBJ*, which must be or be based on type *T*, and returns upon the first callback that returns the bit value given by *VAL*. It is exactly the same as the `uvm_do_callbacks_exit_on but has a specific object instance (instead of the implicit this instance) as the third

argument.

```
...  
if (`uvm_do_callbacks_exit_on(mycb, mycomp, seqr, drop_trans(seqr,trans),  
1))  
...  
...
```

TLM Implementation Port Declaration Macros

The TLM implemenation declaration macros provide a way for an implementer to provide multiple implemenation ports of the same implementation interface. When an implementation port is defined using the built-in set of imps, there must be exactly one implementation of the interface.

For example, if a component needs to provide a put implemenation then it would have an implementation port defined like:

```
class mycomp extends uvm_component;
  uvm_put_imp#(data_type, mycomp) put_imp;
  ...
  virtual task put (data_type t);
  ...
endtask
endclass
```

There are times, however, when you need more than one implementation for for an interface. This set of declarations allow you to easily create a new implemenation class to allow for multiple implementations. Although the new implemenation class is a different class, it can be bound to the same types of exports and ports as the original class. Extending the put example above, lets say that mycomp needs to provide two put implementation ports. In that case, you would do something like:

```
//Define two new put interfaces which are compatible with uvm_put_ports
//and uvm_put_exports.

`uvm_put_imp_decl(_1)
`uvm_put_imp_decl(_2)

class my_put_imp#(type T=int) extends uvm_component;
  uvm_put_imp_1#(T) put_impl1;
  uvm_put_imp_2#(T) put_impl2;
  ...
  function void put_1 (input T t);
    //puts comming into put_impl1
  ...
  endfunction
  function void put_2(input T t);
    //puts comming into put_impl2
  ...
endfunction
endclass
```

The important thing to note is that each `uvm_<interface>_imp_decl creates a new class of type uvm_<interface>_imp<suffix>, where suffix is the input argument to the macro. For this reason, you will typically want to put these macros in a seperate package to avoid collisions and to allow sharing of the definitions.

Summary

TLM Implementation Port Declaration Macros

The TLM implemenation declaration macros provide a way for an implementer to provide multiple implemenation ports of the same implementation interface.

MACROS

`uvm_blocking_put_imp_decl

Define the class
uvm_blocking_put_impSFX for
providing blocking put
implementations.

`uvm_nonblocking_put_imp_decl

Define the class

	uvm_nonblocking_put_impSFX for providing non-blocking put implementations.
`uvm_put_imp_decl	Define the class uvm_put_impSFX for providing both blocking and non-blocking put implementations.
`uvm_blocking_get_imp_decl	Define the class uvm_blocking_get_impSFX for providing blocking get implementations.
`uvm_nonblocking_get_imp_decl	Define the class uvm_nonblocking_get_impSFX for providing non-blocking get implementations.
`uvm_get_imp_decl	Define the class uvm_get_impSFX for providing both blocking and non-blocking get implementations.
`uvm_blocking_peek_imp_decl	Define the class uvm_blocking_peek_impSFX for providing blocking peek implementations.
`uvm_nonblocking_peek_imp_decl	Define the class uvm_nonblocking_peek_impSFX for providing non-blocking peek implementations.
`uvm_peek_imp_decl	Define the class uvm_peek_impSFX for providing both blocking and non-blocking peek implementations.
`uvm_blocking_get_peek_imp_decl	Define the class uvm_blocking_get_peek_impSFX for providing the blocking get_peek implemenation.
`uvm_nonblocking_get_peek_imp_decl	Define the class uvm_nonblocking_get_peek_impSFX for providing non-blocking get_peek implementation.
`uvm_get_peek_imp_decl	Define the class uvm_get_peek_impSFX for providing both blocking and non-blocking get_peek implementations.
`uvm_blocking_master_imp_decl	Define the class uvm_blocking_master_impSFX for providing the blocking master implemenation.
`uvm_nonblocking_master_imp_decl	Define the class uvm_nonblocking_master_impSFX for providing the non-blocking master implemenation.
`uvm_master_imp_decl	Define the class uvm_master_impSFX for providing both blocking and non-blocking master implementations.
`uvm_blocking_slave_imp_decl	Define the class uvm_blocking_slave_impSFX for providing the blocking slave implemenation.
`uvm_nonblocking_slave_imp_decl	Define the class uvm_nonblocking_slave_impSFX for providing the non-blocking slave implemenation.
`uvm_slave_imp_decl	Define the class uvm_slave_impSFX for providing both blocking and non-blocking slave implementations.
`uvm_blocking_transport_imp_decl	Define the class uvm_blocking_transport_impSFX for providing the blocking transport implemenation.
`uvm_nonblocking_transport_imp_decl	Define the class uvm_nonblocking_transport_impSFX

``uvm_transport_imp_decl`

for providing the non-blocking transport implementation.

``uvm_analysis_imp_decl`

Define the class uvm_transport_impSFX for providing both blocking and non-blocking transport implementations.

Define the class uvm_analysis_impSFX for providing an analysis implementation.

MACROS

[`uvm_blocking_put_imp_decl](#)

Define the class uvm_blocking_put_impSFX for providing blocking put implementations. SFX is the suffix for the new class type.

[`uvm_nonblocking_put_imp_decl](#)

Define the class uvm_nonblocking_put_impSFX for providing non-blocking put implementations. SFX is the suffix for the new class type.

[`uvm_put_imp_decl](#)

Define the class uvm_put_impSFX for providing both blocking and non-blocking put implementations. SFX is the suffix for the new class type.

[`uvm_blocking_get_imp_decl](#)

Define the class uvm_blocking_get_impSFX for providing blocking get implementations. SFX is the suffix for the new class type.

[`uvm_nonblocking_get_imp_decl](#)

Define the class uvm_nonblocking_get_impSFX for providing non-blocking get implementations. SFX is the suffix for the new class type.

[`uvm_get_imp_decl](#)

Define the class uvm_get_impSFX for providing both blocking and non-blocking get implementations. SFX is the suffix for the new class type.

[`uvm_blocking_peek_imp_decl](#)

Define the class uvm_blocking_peek_impSFX for providing blocking peek implementations. SFX is the suffix for the new class type.

[**`uvm_nonblocking_peek_imp_decl**](#)

Define the class uvm_nonblocking_peek_impSFX for providing non-blocking peek implementations. *SFX* is the suffix for the new class type.

[**`uvm_peek_imp_decl**](#)

Define the class uvm_peek_impSFX for providing both blocking and non-blocking peek implementations. *SFX* is the suffix for the new class type.

[**`uvm_blocking_get_peek_imp_decl**](#)

Define the class uvm_blocking_get_peek_impSFX for providing the blocking get_peek implemenation.

[**`uvm_nonblocking_get_peek_imp_decl**](#)

Define the class uvm_nonblocking_get_peek_impSFX for providing non-blocking get_peek implemenation.

[**`uvm_get_peek_imp_decl**](#)

Define the class uvm_get_peek_impSFX for providing both blocking and non-blocking get_peek implementations. *SFX* is the suffix for the new class type.

[**`uvm_blocking_master_imp_decl**](#)

Define the class uvm_blocking_master_impSFX for providing the blocking master implemenation.

[**`uvm_nonblocking_master_imp_decl**](#)

Define the class uvm_nonblocking_master_impSFX for providing the non-blocking master implemenation.

[**`uvm_master_imp_decl**](#)

Define the class uvm_master_impSFX for providing both blocking and non-blocking master implementations. *SFX* is the suffix for the new class type.

[**`uvm_blocking_slave_imp_decl**](#)

Define the class uvm_blocking_slave_impSFX for providing the blocking slave implemenation.

[`uvm_nonblocking_slave_imp_decl](#)

Define the class uvm_nonblocking_slave_impSFX for providing the non-blocking slave implemenation.

[`uvm_slave_imp_decl](#)

Define the class uvm_slave_impSFX for providing both blocking and non-blocking slave implementations. SFX is the suffix for the new class type.

[`uvm_blocking_transport_imp_decl](#)

Define the class uvm_blocking_transport_impSFX for providing the blocking transport implemenation.

[`uvm_nonblocking_transport_imp_decl](#)

Define the class uvm_nonblocking_transport_impSFX for providing the non-blocking transport implemenation.

[`uvm_transport_imp_decl](#)

Define the class uvm_transport_impSFX for providing both blocking and non-blocking transport implementations. SFX is the suffix for the new class type.

[`uvm_analysis_imp_decl](#)

Define the class uvm_analysis_impSFX for providing an analysis implementation. SFX is the suffix for the new class type. The analysis implemenation is the write function. The `uvm_analysis_imp_decl allows for a scoreboard (or other analysis component) to support input from many places. For example:

```
`uvm_analysis_imp_decl(_ingress)
`uvm_analysis_imp_port(_egress)

class myscoreboard extends uvm_component;
  uvm_analysis_imp_ingress#(mydata, myscoreboard) ingress;
  uvm_analysis_imp_egress#(mydata, myscoreboard) egress;
  mydata ingress_list[$];
  ...

  function new(string name, uvm_component parent);
    super.new(name,parent);
    ingress = new("ingress", this);
    egress = new("egress", this);
  endfunction

  function void write_ingress(mydata t);
    ingress_list.push_back(t);
  endfunction

  function void write_egress(mydata t);
    find_match_in_ingress_list(t);
  endfunction

  function void find_match_in_ingress_list(mydata t);
    //implement scoreboarding for this particular dut
    ...
  endfunction
endclass
```

Types and Enumerations

Summary

Types and Enumerations

[uvm_bitstream_t](#)

The bitstream type is used as a argument type for passing integral values in such methods as `set_int_local`, `get_int_local`, `get_config_int`, `report`, `pack` and `unpack`.

[uvm_radix_enum](#)

[uvm_recursion_policy_enum](#)

REPORTING

[uvm_severity](#)

Defines all possible values for report severity.

[uvm_action](#)

Defines all possible values for report actions.

[uvm_verbosity](#)

Defines standard verbosity levels for reports.

PORT TYPE

[uvm_port_type_e](#)

SEQUENCES

[uvm_sequence_state_enum](#)

DEFAULT POLICY CLASSES

Policy classes for `uvm_object` basic functions, `uvm_object::copy`, `uvm_object::compare`, `uvm_object::pack`, `uvm_object::unpack`, and `uvm_object::record`.

[uvm_default_table_printer](#)

The table printer is a global object that can be used with `uvm_object::do_print` to get tabular style printing.

[uvm_default_tree_printer](#)

The tree printer is a global object that can be used with `uvm_object::do_print` to get multi-line tree style printing.

[uvm_default_line_printer](#)

The line printer is a global object that can be used with `uvm_object::do_print` to get single-line style printing.

[uvm_default_printer](#)

The default printer is a global object that is used by `uvm_object::print` or `uvm_object::sprint` when no specific printer is set.

[uvm_default_packer](#)

The default packer policy.

[uvm_default_comparer](#)

The default compare policy.

[uvm_default_recorder](#)

The default recording policy.

[uvm_bitstream_t](#)

The bitstream type is used as a argument type for passing integral values in such methods as `set_int_local`, `get_int_local`, `get_config_int`, `report`, `pack` and `unpack`.

[uvm_radix_enum](#)

`UVM_BIN` Selects binary (%b) format

`UVM_DEC` Selects decimal (%d) format

`UVM_UNSIGNED` Selects unsigned decimal (%u) format

`UVM_OCT` Selects octal (%o) format

`UVM_HEX` Selects hexadecimal (%h) format

<i>UVM_STRING</i>	Selects string (%s) format
<i>UVM_TIME</i>	Selects time (%t) format
<i>UVM_ENUM</i>	Selects enumeration value (name) format

[uvm_recursion_policy_enum](#)

<i>UVM_DEEP</i>	Objects are deep copied (object must implement copy method)
<i>UVM_SHALLOW</i>	Objects are shallow copied using default SV copy.
<i>UVM_REFERENCE</i>	Only object handles are copied.

[REPORTING](#)

[uvm_severity](#)

Defines all possible values for report severity.

<i>UVM_INFO</i>	Informative message.
<i>UVM_WARNING</i>	Indicates a potential problem.
<i>UVM_ERROR</i>	Indicates a real problem. Simulation continues subject to the configured message action.
<i>UVM_FATAL</i>	Indicates a problem from which simulation can not recover. Simulation exits via \$finish after a #0 delay.

[uvm_action](#)

Defines all possible values for report actions. Each report is configured to execute one or more actions, determined by the bitwise OR of any or all of the following enumeration constants.

<i>UVM_NO_ACTION</i>	No action is taken
<i>UVM_DISPLAY</i>	Sends the report to the standard output
<i>UVM_LOG</i>	Sends the report to the file(s) for this (severity,id) pair
<i>UVM_COUNT</i>	Counts the number of reports with the COUNT attribute. When this value reaches max_quit_count, the simulation terminates
<i>UVM_EXIT</i>	Terminates the simulation immediately.
<i>UVM_CALL_HOOK</i>	Callback the report hook methods

[uvm_verbosity](#)

Defines standard verbosity levels for reports.

<i>UVM_NONE</i>	Report is always printed. Verbosity level setting can not disable it.
<i>UVM_LOW</i>	Report is issued if configured verbosity is set to UVM_LOW or above.

<i>UVM_MEDIUM</i>	Report is issued if configured verbosity is set to UVM_MEDIUM or above.
<i>UVM_HIGH</i>	Report is issued if configured verbosity is set to UVM_HIGH or above.
<i>UVM_FULL</i>	Report is issued if configured verbosity is set to UVM_FULL or above.

PORT TYPE

[uvm_port_type_e](#)

<i>UVM_PORT</i>	The port requires the interface that is its type parameter.
<i>UVM_EXPORT</i>	The port provides the interface that is its type parameter via a connection to some other export or implementation.
<i>UVM_IMPLEMENTATION</i>	The port provides the interface that is its type parameter, and it is bound to the component that implements the interface.

SEQUENCES

[uvm_sequence_state_enum](#)

<i>CREATED</i>	The sequence has been allocated.
<i>PRE_BODY</i>	The sequence is started and the pre_body task is being executed.
<i>BODY</i>	The sequence is started and the body task is being executed.
<i>POST_BODY</i>	The sequence is started and the post_body task is being executed.
<i>ENDED</i>	The sequence has ended by the completion of the body task.
<i>STOPPED</i>	The sequence has been forcibly ended by issuing a kill() on the sequence.
<i>FINISHED</i>	The sequence is completely finished executing.

DEFAULT POLICY CLASSES

Policy classes for [uvm_object](#) basic functions, [uvm_object::copy](#), [uvm_object::compare](#), [uvm_object::pack](#), [uvm_object::unpack](#), and [uvm_object::record](#).

[uvm_default_table_printer](#)

```
uvm_table_printer uvm_default_table_printer = new()
```

The table printer is a global object that can be used with [uvm_object::do_print](#) to get

tabular style printing.

[uvm_default_tree_printer](#)

```
uvm_tree_printer uvm_default_tree_printer = new()
```

The tree printer is a global object that can be used with [uvm_object::do_print](#) to get multi-line tree style printing.

[uvm_default_line_printer](#)

```
uvm_line_printer uvm_default_line_printer = new()
```

The line printer is a global object that can be used with [uvm_object::do_print](#) to get single-line style printing.

[uvm_default_printer](#)

```
uvm_printer uvm_default_printer = uvm_default_table_printer
```

The default printer is a global object that is used by [uvm_object::print](#) or [uvm_object::sprint](#) when no specific printer is set.

The default printer may be set to any legal [uvm_printer](#) derived type, including the global line, tree, and table printers described above.

[uvm_default_packer](#)

```
uvm_packer uvm_default_packer = new()
```

The default packer policy. If a specific packer instance is not supplied in calls to [uvm_object::pack](#) and [uvm_object::unpack](#), this instance is selected.

[uvm_default_comparer](#)

```
uvm_comparer uvm_default_comparer = new()
```

The default compare policy. If a specific comparer instance is not supplied in calls to [uvm_object::compare](#), this instance is selected.

[uvm_default_recorder](#)

```
uvm_recorder uvm_default_recorder = new()
```

The default recording policy. If a specific recorder instance is not supplied in calls to [uvm_object::record](#).

Summary

Globals

SIMULATION CONTROL

<code>run_test</code>	Convenience function for <code>uvm_top.run_test()</code> .
<code>uvm_test_done</code>	An instance of the <code>uvm_test_done_objection</code> class, this object is used by components to coordinate when to end the currently running task-based phase.
<code>global_stop_request</code>	Convenience function for <code>uvm_top.stop_request()</code> .
<code>set_global_timeout</code>	Convenience function for <code>uvm_top.phase_timeout = timeout</code> .
<code>set_global_stop_timeout</code>	Convenience function for <code>uvm_top.stop_timeout = timeout</code> .

REPORTING

<code>uvm_report_enabled</code>	Returns 1 if the configured verbosity in <code><uvm_top></code> is greater than <code>verbosity</code> and the action associated with the given <code>severity</code> and <code>id</code> is not UVM_NO_ACTION, else returns 0.
---------------------------------	---

<code>uvm_report_info</code>
<code>uvm_report_warning</code>
<code>uvm_report_error</code>
<code>uvm_report_fatal</code>

Verbosity is ignored for warnings, errors, and fatals to ensure users do not inadvertently filter them out.

These methods, defined in package scope, are convenience functions that delegate to the corresponding component methods in `uvm_top`.

CONFIGURATION

<code>set_config_int</code>	This is the global version of <code>set_config_int</code> in <code>uvm_component</code> .
<code>set_config_object</code>	This is the global version of <code>set_config_object</code> in <code>uvm_component</code> .
<code>set_config_string</code>	This is the global version of <code>set_config_string</code> in <code>uvm_component</code> .

MISCELLANEOUS

<code>uvm_is_match</code>	Returns 1 if the two strings match, 0 otherwise.
<code>uvm_string_to_bits</code>	Converts an input string to its bit-vector equivalent.
<code>uvm_bits_to_string</code>	Converts an input bit-vector to its string equivalent.
<code>uvm_wait_for_nba_region</code>	Call this task to wait for a delta cycle.

SIMULATION CONTROL

run_test

```
task run_test (string test_name = "")
```

Convenience function for `uvm_top.run_test()`. See `uvm_root` for more information.

[uvm_test_done](#)

```
uvm_test_done_objection uvm_test_done = uvm_test_done_objection::get()
```

An instance of the [uvm_test_done_objection](#) class, this object is used by components to coordinate when to end the currently running task-based phase. When all participating components have dropped their raised objections, an implicit call to [global_stop_request](#) is issued to end the run phase (or any other task-based phase).

[global_stop_request](#)

```
function void global_stop_request()
```

Convenience function for `uvm_top.stop_request()`. See [uvm_root](#) for more information.

[set_global_timeout](#)

```
function void set_global_timeout(time timeout)
```

Convenience function for `uvm_top.phase_timeout = timeout`. See [uvm_root](#) for more information.

[set_global_stop_timeout](#)

```
function void set_global_stop_timeout(time timeout)
```

Convenience function for `uvm_top.stop_timeout = timeout`. See [uvm_root](#) for more information.

REPORTING

[uvm_report_enabled](#)

```
function bit uvm_report_enabled (int verbosity,
                                 uvm_severity severity = UVM_INFO,
                                 string id = "")
```

Returns 1 if the configured verbosity in `<uvm_top>` is greater than `verbosity` and the action associated with the given `severity` and `id` is not `UVM_NO_ACTION`, else returns 0.

See also [uvm_report_object::uvm_report_enabled](#).

Static methods of an extension of `uvm_report_object`, e.g. `uvm_compoent-based objects`, can not call `uvm_report_enabled` because the call will resolve to the [uvm_report_object::uvm_report_enabled](#), which is non-static. Static methods can not call non-static methods of the same class.

[uvm_report_info](#)

```
function void uvm_report_info(string id,
                             string message,
```

```
int      verbosity = UVM_MEDIUM,
string  filename  = "",
int      line      = 0
)
```

uvm_report_warning

```
function void uvm_report_warning(string id,
                                 string message,
                                 int   verbosity = UVM_MEDIUM,
                                 string filename = "",
                                 int   line      = 0
)
```

uvm_report_error

```
function void uvm_report_error(string id,
                               string message,
                               int   verbosity = UVM_LOW,
                               string filename = "",
                               int   line      = 0
)
```

uvm_report_fatal

These methods, defined in package scope, are convenience functions that delegate to the corresponding component methods in *uvm_top*. They can be used in module-based code to use the same reporting mechanism as class-based components. See [uvm_report_object](#) for details on the reporting mechanism.

Verbosity is ignored for warnings, errors, and fatals to ensure users

do not inadvertently filter them out. It remains in the methods for backward compatibility.

CONFIGURATION

set_config_int

```
function void set_config_int (string           inst_name,
                            string           field_name,
                            uvm_bitstream_t value
)
```

This is the global version of `set_config_int` in [uvm_component](#). This function places the configuration setting for an integral field in a global override table, which has highest precedence over any component-level setting. See [uvm_component::set_config_int](#) for details on setting configuration.

set_config_object

```
function void set_config_object (string           inst_name,
                                string           field_name,
                                uvm_object      value,
                                bit             clone
)
```

This is the global version of `set_config_object` in `uvm_component`. This function places the configuration setting for an object field in a global override table, which has highest precedence over any component-level setting. See `uvm_component::set_config_object` for details on setting configuration.

[set_config_string](#)

```
function void set_config_string (string inst_name,  
                                string field_name,  
                                string value )
```

This is the global version of `set_config_string` in `uvm_component`. This function places the configuration setting for an string field in a global override table, which has highest precedence over any component-level setting. See `uvm_component::set_config_string` for details on setting configuration.

MISCELLANEOUS

[uvm_is_match](#)

```
`ifdef UVM_DPI import "DPI" function bit uvm_is_match (string expr,  
                                                       string str )
```

Returns 1 if the two strings match, 0 otherwise.

The first string, `expr`, is a string that may contain '*' and '?' characters. A * matches zero or more characters, and ? matches any single character. The 2nd argument, `str`, is the string begin matched against. It must not contain any wildcards.

[uvm_string_to_bits](#)

```
function logic[UVM_LARGE_STRING:0] uvm_string_to_bits(string str)
```

Converts an input string to its bit-vector equivalent. Max bit-vector length is approximately 14000 characters.

[uvm_bits_to_string](#)

```
function string uvm_bits_to_string(logic [UVM_LARGE_STRING:0] str)
```

Converts an input bit-vector to its string equivalent. Max bit-vector length is approximately 14000 characters.

[uvm_wait_for_nba_region](#)

```
task uvm_wait_for_nba_region
```

Call this task to wait for a delta cycle. Program blocks don't have an nba so just delay for a #0 in a program block.

Index

[\\$#!](#) · [0-9](#) · [A](#) · [B](#) · [C](#) · [D](#) · [E](#) · [F](#) · [G](#) · [H](#) · [I](#) · [J](#) · [K](#) · [L](#) · [M](#) · [N](#) · [O](#) · [P](#) · [Q](#) · [R](#) · [S](#) · [T](#) · [U](#) · [V](#) · [W](#) · [X](#) · [Y](#) · [Z](#)

\$#!

- [`uvm_analysis_imp_decl](#)
- [`uvm_blocking_get_imp_decl](#)
- [`uvm_blocking_get_peek_imp_decl](#)
- [`uvm_blocking_master_imp_decl](#)
- [`uvm_blocking_peek_imp_decl](#)
- [`uvm_blocking_put_imp_decl](#)
- [`uvm_blocking_slave_imp_decl](#)
- [`uvm_blocking_transport_imp_decl](#)
- [`uvm_component_end](#)
- [`uvm_component_param_utils](#)
- [`uvm_component_param_utils_begin](#)
- [`uvm_component_utils](#)
- [`uvm_component_utils_begin](#)
- [`uvm_create](#)
- [`uvm_create_on](#)
- [`uvm_declare_p_sequencer](#)
- [`uvm_do](#)
- [`uvm_do_callbacks](#)
- [`uvm_do_callbacks_exit_on](#)
- [`uvm_do_obj_callbacks](#)
- [`uvm_do_obj_callbacks_exit_on](#)
- [`uvm_do_on](#)
- [`uvm_do_on_pri](#)
- [`uvm_do_on_pri_with](#)
- [`uvm_do_on_with](#)
- [`uvm_do_pri](#)
- [`uvm_do_pri_with](#)
- [`uvm_do_with](#)
- [`uvm_error](#)
- [`uvm_fatal](#)
- [`uvm_field_*macros](#)
- [`uvm_field_aa_*_int macros](#)
- [`uvm_field_aa_*_string macros](#)
- [`uvm_field_aa_int_byte](#)
- [`uvm_field_aa_int_byte_unsigned](#)
- [`uvm_field_aa_int_enumkey](#)
- [`uvm_field_aa_int_int](#)
- [`uvm_field_aa_int_int_unsigned](#)
- [`uvm_field_aa_int_integer](#)
- [`uvm_field_aa_int_integer_unsigned](#)
- [`uvm_field_aa_int_key](#)
- [`uvm_field_aa_int_longint](#)
- [`uvm_field_aa_int_longint_unsigned](#)
- [`uvm_field_aa_int_shortint](#)
- [`uvm_field_aa_int_shortint_unsigned](#)
- [`uvm_field_aa_int_string](#)
- [`uvm_field_aa_object_int](#)
- [`uvm_field_aa_object_string](#)
- [`uvm_field_aa_string_string](#)
- [`uvm_field_array_*macros](#)
- [`uvm_field_array_enum](#)
- [`uvm_field_array_int](#)

- ` uvm_field_array_object
- ` uvm_field_array_string
- ` uvm_field_enum
- ` uvm_field_event
- ` uvm_field_int
- ` uvm_field_object
- ` uvm_field_queue_*macros
- ` uvm_field_queue_enum
- ` uvm_field_queue_int
- ` uvm_field_queue_object
- ` uvm_field_queue_string
- ` uvm_field_real
- ` uvm_field_sarray_*macros
- ` uvm_field_sarray_enum
- ` uvm_field_sarray_int
- ` uvm_field_sarray_object
- ` uvm_field_sarray_string
- ` uvm_field_string
- ` uvm_field_utils_begin
- ` uvm_field_utils_end
- ` uvm_get_imp_decl
- ` uvm_get_peek_imp_decl
- ` uvm_info
- ` uvm_master_imp_decl
- ` uvm_nonblocking_get_imp_decl
- ` uvm_nonblocking_get_peek_imp_decl
- ` uvm_nonblocking_master_imp_decl
- ` uvm_nonblocking_peek_imp_decl
- ` uvm_nonblocking_put_imp_decl
- ` uvm_nonblocking_slave_imp_decl
- ` uvm_nonblocking_transport_imp_decl
- ` uvm_object_param_utils
- ` uvm_object_param_utils_begin
- ` uvm_object_utils
- ` uvm_object_utils_begin
- ` uvm_object_utils_end
- ` uvm_peek_imp_decl
- ` uvm_phase_func_bottomup_decl
- ` uvm_phase_func_decl
- ` uvm_phase_func_topdown_decl
- ` uvm_phase_task_bottomup_decl
- ` uvm_phase_task_decl
- ` uvm_phase_task_topdown_decl
- ` uvm_put_imp_decl
- ` uvm_rand_send
- ` uvm_rand_send_pri
- ` uvm_rand_send_pri_with
- ` uvm_rand_send_with
- ` uvm_register_cb
- ` uvm_send
- ` uvm_send_pri
- ` uvm_sequence_utils
- ` uvm_sequence_utils_begin
- ` uvm_sequence_utils_end
- ` uvm_sequencer_param_utils
- ` uvm_sequencer_param_utils_begin
- ` uvm_sequencer_utils
- ` uvm_sequencer_utils_begin
- ` uvm_sequencer_utils_end
- ` uvm_set_super_type

```
`uvm_slave_imp_decl  
`uvm_transport_imp_decl  
`uvm_update_sequence_lib  
`uvm_update_sequence_lib_and_item  
`uvm_warning
```

A

abstract

```
uvm_comparer  
uvm_packer  
uvm_recorder
```

accept_tr

```
uvm_component  
uvm_transaction
```

add

```
uvm_callbacks#(T,CB)  
uvm_heartbeat  
uvm_pool#(T)
```

Add/delete interface

```
uvm_callbacks#(T,CB)
```

add_by_name

```
uvm_callbacks#(T,CB)
```

add_callback

```
uvm_event
```

add_sequence

```
uvm_sequencer_base
```

after_export

```
uvm_algorithmic_comparator#(BEFORE,AFTER,TRANSFORMER)  
uvm_in_order_comparator#(T,comp_type,convert,pair_type)
```

all_dropped

```
uvm_component  
uvm_objection  
uvm_root  
uvm_test_done_objection
```

Analysis

```
Global  
uvm_tlm_if_base#(T1,T2)
```

analysis_export

```
uvm_subscriber
```

analysis_port#(T)

```
uvm_tlm_analysis_fifo#(T)
```

apply_config_settings

```
uvm_component
```

B

before_export

```
uvm_algorithmic_comparator#(BEFORE,AFTER,TRANSFORMER)  
uvm_in_order_comparator#(T,comp_type,convert,pair_type)
```

```

begin_child_tr
    uvm_component
    uvm_transaction

begin_elements
    uvm_printer_knobs

begin_tr
    uvm_component
    uvm_transaction

Bidirectional Interfaces&Ports
big_endian
    uvm_packer

bin_radix
    uvm_printer_knobs

Blocking get
    uvm_tlm_if_base#(T1,T2)

Blocking peek
    uvm_tlm_if_base#(T1,T2)

Blocking put
    uvm_tlm_if_base#(T1,T2)

Blocking transport
    uvm_tlm_if_base#(T1,T2)

blocking_put_port
    uvm_random_stimulus#(T)

body
    uvm_sequence_base

BODY
build
    uvm_component

```

C

```

call_func
    uvm_phase

call_task
    uvm_phase

Callback Hooks
    uvm_objection

Callback Interface
    uvm_report_catcher

Callback Macros
callback_mode
    uvm_callback

Callbacks
    uvm_report_object

can_get
    uvm_tlm_if_base#(T1,T2)

can_peek
    uvm_tlm_if_base#(T1,T2)

can_put
    uvm_tlm_if_base#(T1,T2)

```

cancel
 uvm_barrier
 uvm_event

catch
 uvm_report_catcher

CB
 uvm_callbacks#(T,CB)

Change Message State
 uvm_report_catcher

check
 uvm_component

check_config_usage
 uvm_component

check_type
 uvm_comparer

clone
 uvm_object

Comparators
 uvm_algorithmic_comparator
 comparators

compare
 uvm_object

compare_field
 uvm_comparer

compare_field_int
 uvm_comparer

compare_field_real
 uvm_comparer

compare_object
 uvm_comparer

compare_string
 uvm_comparer

Comparing
 uvm_object

compose_message
 uvm_report_server

Configuration
 Global
 uvm_object
 uvm_report_object

Configuration Interface
 uvm_component

connect
 uvm_component
 uvm_port_base#(IF)

convert2string
 uvm_object

copy
 uvm_object

Copying
 uvm_object

Core Base Classes

```

count
    uvm_sequencer_base

create
    uvm_component_registry#(T,Tname)
    uvm_object
    uvm_object_registry#(T,Tname)

create_component
    uvm_component
    uvm_component_registry#(T,Tname)
    uvm_object_wrapper

create_component_by_name
    uvm_factory

create_component_by_type
    uvm_factory

create_item
    uvm_sequence_base

create_object
    uvm_component
    uvm_object_registry#(T,Tname)
    uvm_object_wrapper

create_object_by_name
    uvm_factory

create_object_by_type
    uvm_factory

CREATED
Creation
    uvm_factory
    uvm_object

Current Message State
    uvm_report_catcher

current_grabber
    uvm_sequencer_base

```

D

```

Debug
    uvm_factory
    uvm_report_catcher

debug_connected_to
    uvm_port_base#(IF)

debug_create_by_name
    uvm_factory

debug_create_by_type
    uvm_factory

debug_provided_to
    uvm_port_base#(IF)

dec_radix
    uvm_printer_knobs

Default Policy Classes
default_radix
    uvm_printer_knobs

```

```
    uvm_recorder
default_sequence
    uvm_sequencer_base
delete
    uvm_callbacks#(T,CB)
    uvm_object_string_pool#(T)
    uvm_pool#(T)
    uvm_queue#(T)
delete_by_name
    uvm_callbacks#(T,CB)
delete_callback
    uvm_event
depth
    uvm_printer_knobs
die
    uvm_report_object
disable_recording
    uvm_transaction
display_objections
    uvm_objection
do_accept_tr
    uvm_component
    uvm_transaction
do_begin_tr
    uvm_component
    uvm_transaction
do_compare
    uvm_object
do_copy
    uvm_object
do_end_tr
    uvm_component
    uvm_transaction
do_kill_all
    uvm_component
do_pack
    uvm_object
do_print
    uvm_object
do_record
    uvm_object
do_sequence_kind
    uvm_sequence_base
do_unpack
    uvm_object
drop
    uvm_test_done_objection
drop_objection
    uvm_objection
dropped
    uvm_component
    uvm_objection
```

dump_report_state
 uvm_report_object
dump_server_state
 uvm_report_server

E

enable_print_topology
 uvm_root
enable_recording
 uvm_transaction
enable_stop_interrupt
 uvm_component
end
 uvm_sequence_builtin
 sqr_connections
end_elements
 uvm_printer_knobs
end_of_elaboration
 uvm_component
end_tr
 uvm_component
 uvm_transaction
ENDED
execute_item
 uvm_sequencer_param_base#(REQ,RSP)
exists
 uvm_pool#(T)
extract
 uvm_component

F

Factory Classes
Factory Interface
 uvm_component
Field Macros
Fields declared in `uvm_field_*> macros, if used, will not
 uvm_object
find
 uvm_root
find_all
 uvm_root
find_override_by_name
 uvm_factory
find_override_by_type
 uvm_factory
finish_item
 uvm_sequence_base

```
    uvm_sequence_item
finish_on_completion
    uvm_root

FINISHED
first
    uvm_callback_iter
    uvm_pool#(T)

flush
    uvm_in_order_comparator#(T,comp_type,convert,pair_type)
    uvm_tlm_fifo#(T)

footer
    uvm_printer_knobs

force_stop
    uvm_test_done_objection

format_action
    uvm_report_handler

full_name
    uvm_printer_knobs
```

G

```
generate_stimulus
    uvm_random_stimulus#(T)

get
    uvm_component_registry#(T,Tname)
    uvm_object_registry#(T,Tname)
    uvm_object_string_pool#(T)
    uvm_pool#(T)
    uvm_queue#(T)
    uvm_sqr_if_base#(REQ,RSP)
    uvm_tlm_if_base#(T1,T2)

Get and Peek
get_accept_time
    uvm_transaction

get_action
    uvm_report_catcher
    uvm_report_handler

get_ap
    uvm_tlm_fifo_base#(T)

get_begin_time
    uvm_transaction

get_cb
    uvm_callback_iter

get_child
    uvm_component

get_client
    uvm_report_catcher

get_comp
    uvm_port_base#(IF)

get_config_int
    uvm_component

get_config_object
    uvm_component

get_config_string
    uvm_component

get_count
    uvm_random_sequence

get_current_item
    uvm_sequence#(REQ,RSP)
    uvm_sequencer_param_base#(REQ,RSP)

get_current_phase
    uvm_root

get_depth
    uvm_sequence_item

get_drain_time
    uvm_objection
```

```
get_end_time
    uvm_transaction

get_event_pool
    uvm_transaction

get_file_handle
    uvm_report_handler

get_first
    uvm_callbacks#(T,CB)

get_first_child
    uvm_component

get_fname
    uvm_report_catcher

get_full_name
    uvm_component
    uvm_object
    uvm_port_base#(IF)

get_global
    uvm_pool#(T)
    uvm_queue#(T)

get_global_pool
    uvm_object_string_pool#(T)
    uvm_pool#(T)

get_global_queue
    uvm_queue#(T)

get_id
    uvm_report_catcher

get_id_count
    uvm_report_server

get_if
    uvm_port_base#(IF)

get_initiator
    uvm_transaction

get_inst_count
    uvm_object

get_inst_id
    uvm_object

get_line
    uvm_report_catcher

get_max_quit_count
    uvm_report_server

get_message
    uvm_report_catcher

get_name
    uvm_object
    uvm_phase
    uvm_port_base#(IF)

get_next
    uvm_callbacks#(T,CB)

get_next_child
    uvm_component

get_next_item
    uvm_sqr_if_base#(REQ,RSP)
```

get_num_children
 uvm_component

get_num_last_reqs
 uvm_sequencer_param_base#(REQ,RSP)

get_num_last_rsp
 uvm_sequencer_param_base#(REQ,RSP)

get_num_reqs_sent
 uvm_sequencer_param_base#(REQ,RSP)

get_num_rsp_received
 uvm_sequencer_param_base#(REQ,RSP)

get_num_waiters
 uvm_barrier
 uvm_event

get_object_type
 uvm_object

get_objection_count
 uvm_objection

get_objection_total
 uvm_objection

get_packed_size
 uvm_packer

get_parent
 uvm_component
 uvm_port_base#(IF)

get_parent_sequence
 uvm_sequence_item

get_peek_export
 uvm_tlm_fifo_base#(T)

get_peek_request_export
 uvm_tlm_req_rsp_channel#(REQ,RSP)

get_peek_response_export
 uvm_tlm_req_rsp_channel#(REQ,RSP)

get_phase_by_name
 uvm_root

get_priority
 uvm_sequence_base

get_quit_count
 uvm_report_server

get_radix_str
 uvm_printer_knobs

get_report_action
 uvm_report_object

get_report_catcher
 uvm_report_catcher

get_report_file_handle
 uvm_report_object

get_report_handler
 uvm_report_object

get_report_server
 uvm_report_object

get_report_verbosity_level

```
    uvm_report_object  
get_response  
        uvm_sequence#(REQ,RSP)  
get_response_queue_depth  
        uvm_sequence#(REQ,RSP)  
get_response_queue_error_report_disabled  
        uvm_sequence#(REQ,RSP)  
get_root_sequence  
        uvm_sequence_item  
get_root_sequence_name  
        uvm_sequence_item  
get_seq_kind  
        uvm_sequence_base  
        uvm_sequencer_base  
get_sequence  
        uvm_sequence_base  
        uvm_sequencer_base  
get_sequence_by_name  
        uvm_sequence_base  
get_sequence_id  
        uvm_sequence_item  
get_sequence_path  
        uvm_sequence_item  
get_sequence_state  
        uvm_sequence_base  
get_sequencer  
        uvm_sequence_base  
        uvm_sequence_item  
get_server  
        uvm_report_server  
get_severity  
        uvm_report_catcher  
get_severity_count  
        uvm_report_server  
get_threshold  
        uvm_barrier  
get_tr_handle  
        uvm_transaction  
get_transaction_id  
        uvm_transaction  
get_trigger_data  
        uvm_event  
get_trigger_time  
        uvm_event  
get_type  
        uvm_object  
get_type_name  
        uvm_callback  
        uvm_component_registry#(T,Tname)  
        uvm_object  
        uvm_object_registry#(T,Tname)  
        uvm_object_string_pool#(T)
```

uvm_object_wrapper
uvm_phase
uvm_port_base#(IF)
get_use_response_handler
 uvm_sequence_base
get_use_sequence_info
 uvm_sequence_item
get_verbosity
 uvm_report_catcher
get_verbosity_level
 uvm_report_handler
global_indent
 uvm_printer_knobs
global_stop_request
Globals
grab
 uvm_sequence_base
 uvm_sequencer_base

H

has_child
 uvm_component
has_do_available
 uvm_sequencer_base
 uvm_sqr_if_base#(REQ,RSP)
has_lock
 uvm_sequence_base
 uvm_sequencer_base
hb_mode
 uvm_heartbeat
header
 uvm_printer_knobs
hex_radix
 uvm_printer_knobs
Hierarchical Reporting Interface
 uvm_component
Hierarchy Interface
 uvm_component

I

id_count
 uvm_report_server
Identification
 uvm_object
identifier
 uvm_printer_knobs
 uvm_recorder

```
in_order_built_in_comparator#(T)
in_order_class_comparator#(T)
in_stop_request
    uvm_root

incr_id_count
    uvm_report_server

incr_quit_count
    uvm_report_server

incr_severity_count
    uvm_report_server

indent_str
    uvm_hier_printer_knobs

insert
    uvm_queue#(T)

insert_phase
    uvm_root

is_active
    uvm_transaction

is_blocked
    uvm_sequence_base
    uvm_sequencer_base

is_child
    uvm_sequencer_base

is_done
    uvm_phase

is_empty
    uvm_tlm_fifo#(T)

is_enabled
    uvm_callback

is_export
    uvm_port_base#(IF)

is_full
    uvm_tlm_fifo#(T)

is_grabbed
    uvm_sequencer_base

is_imp
    uvm_port_base#(IF)

is_in_progress
    uvm_phase

is_item
    uvm_sequence_base
    uvm_sequence_item

is_null
    uvm_packer

is_off
    uvm_event

is_on
    uvm_event

is_port
    uvm_port_base#(IF)

is_quit_count_reached
    uvm_report_server
```

is_recording_enabled
uvm_transaction

is_relevant
uvm_sequence_base

is_task
uvm_phase

is_top_down
uvm_phase

is_unbounded
uvm_port_base#(IF)

issue
uvm_report_catcher

item_done
uvm_sqr_if_base#(REQ,RSP)

Iterator interface
uvm_callbacks#(T,CB)

K

kill
uvm_component
uvm_sequence_base

knobs
uvm_printer
uvm_table_printer
uvm_tree_printer

L

last
uvm_pool#(T)

last_req
uvm_sequencer_param_base#(REQ,RSP)

last_rsp
uvm_sequencer_param_base#(REQ,RSP)

lock
uvm_sequence_base
uvm_sequencer_base

lookup
uvm_component

M

Macros
uvm_phases
uvm_messageDefines
uvm_tlmDefines

Master and Slave

master_export

 uvm_tlm_req_rsp_channel#(REQ,RSP)

max_random_count

 uvm_sequencer_base

max_random_depth

 uvm_sequencer_base

max_size

 uvm_port_base#(IF)

max_width

 uvm_printer_knobs

mcd

 uvm_printer_knobs

Methods

 uvm_*_export#(REQ,RSP)

 uvm_*_export#(T)

 uvm_*_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)

 uvm_*_imp#(T,IMP)

 uvm_*_port#(REQ,RSP)

 uvm_*_port#(T)

 uvm_agent

 uvm_algorithmic_comparator#(BEFORE,AFTER,TRANSFORMER)

 uvm_barrier

 uvm_built_in_pair#(T1,T2)

 uvm_callback

 uvm_callback_iter

 uvm_comparer

 uvm_component_registry#(T,Tname)

 uvm_driver#(REQ,RSP)

 uvm_env

 uvm_event

 uvm_event_callback

 uvm_heartbeat

 uvm_in_order_comparator#(T,comp_type,convert,pair_type)

 uvm_line_printer

 uvm_monitor

 uvm_object_string_pool#(T)

 uvm_object_wrapper

 uvm_pair#(T1,T2)

 uvm_phase

 uvm_pool#(T)

 uvm_port_base#(IF)

 uvm_printer_knobs

 uvm_push_driver#(REQ,RSP)

 uvm_push_sequencer#(REQ,RSP)

 uvm_queue#(T)

 uvm_random_sequence

 uvm_random_stimulus#(T)

 uvm_recorder

 uvm_report_handler

 uvm_report_server

 uvm_root

 uvm_scoreboard

 uvm_sequence#(REQ,RSP)

 uvm_sequence_base

 uvm_sequence_item

 uvm_sequencer#(REQ,RSP)

 uvm_sequencer_base

```
uvm_sequencer_param_base#(REQ,RSP)
uvm_sqr_if_base#(REQ,RSP)
uvm_subscriber
uvm_test
uvm_test_done_objection
uvm_tlm_analysis_fifo#(T)
uvm_tlm_fifo#(T)
uvm_tlm_fifo_base#(T)
uvm_tlm_req_rsp_channel#(REQ,RSP)
uvm_tlm_transport_channel#(REQ,RSP)
uvm_transaction
```

Methods for printer subtyping

```
uvm_printer
```

Methods for printer usage

```
uvm_printer
```

mid_do

```
uvm_sequence_base
```

min_size

```
uvm_port_base#(IF)
```

Miscellaneous

miscompares

```
uvm_comparer
```

N

name_width

```
uvm_table_printer_knobs
```

nb_transport

```
uvm_tlm_if_base#(T1,T2)
```

new

```
uvm_*_export#(REQ,RSP)
```

```
uvm_*_export#(T)
```

```
uvm_*_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
```

```
uvm_*_imp#(T,IMP)
```

```
uvm_*_port#(REQ,RSP)
```

```
uvm_*_port#(T)
```

```
uvm_agent
```

```
uvm_algorithmic_comparator#(BEFORE,AFTER,TRANSFORMER)
```

```
uvm_barrier
```

```
uvm_built_in_pair#(T1,T2)
```

```
uvm_callback
```

```
uvm_callback_iter
```

```
uvm_component
```

```
uvm_driver#(REQ,RSP)
```

```
uvm_env
```

```
uvm_event
```

```
uvm_event_callback
```

```
uvm_heartbeat
```

```
uvm_line_printer
```

```
uvm_monitor
```

```
uvm_object
```

```
uvm_object_string_pool#(T)
```

```
uvm_objection
```

```
uvm_pair#(T1,T2)
```

```
uvm_phase
```

```
uvm_pool#(T)
uvm_port_base#(IF)
uvm_push_driver#(REQ,RSP)
uvm_push_sequencer#(REQ,RSP)
uvm_queue#(T)
uvm_random_stimulus#(T)
uvm_report_catcher
uvm_report_handler
uvm_report_object
uvm_report_server
uvm_scoreboard
uvm_sequence#(REQ,RSP)
uvm_sequence_base
uvm_sequence_item
uvm_sequencer#(REQ,RSP)
uvm_sequencer_base
uvm_sequencer_param_base#(REQ,RSP)
uvm_subscriber
uvm_table_printer
uvm_test
uvm_tlm_analysis_fifo#(T)
uvm_tlm_fifo#(T)
uvm_tlm_fifo_base#(T)
uvm_tlm_req_rsp_channel#(REQ,RSP)
uvm_tlm_transport_channel#(REQ,RSP)
uvm_transaction
uvm_tree_printer
```

next

```
uvm_callback_iter
uvm_pool#(T)
```

Non-blocking get

```
uvm_tlm_if_base#(T1,T2)
```

Non-blocking peek

```
uvm_tlm_if_base#(T1,T2)
```

Non-blocking put

```
uvm_tlm_if_base#(T1,T2)
```

Non-blocking transport

```
uvm_tlm_if_base#(T1,T2)
```

num

```
uvm_pool#(T)
```

num_sequences

```
uvm_sequence_base
uvm_sequencer_base
```

O

Objection Control

```
uvm_objection
```

Objection Interface

```
uvm_component
```

Objection Status

```
uvm_objection
```

oct_radix

```
uvm_printer_knobs
```

P

pack
 uvm_object

pack_bytes
 uvm_object

pack_field
 uvm_packer

pack_field_int
 uvm_packer

pack_ints
 uvm_object

pack_object
 uvm_packer

pack_real
 uvm_packer

pack_string
 uvm_packer

pack_time
 uvm_packer

Packing
 uvm_object
 uvm_packer

pair_ap
 uvm_in_order_comparator#(T,comp_type,convert,pair_type)

peek
 uvm_sqr_if_base#(REQ,RSP)
 uvm_tlm_if_base#(T1,T2)

phase_timeout
 uvm_root

Phasing Interface
 uvm_component

physical
 uvm_comparer
 uvm_packer
 uvm_recorder

policy
 uvm_comparer

Policy Classes
 uvm_policies
 policies

pop_back
 uvm_queue#(T)

pop_front
 uvm_queue#(T)

Port Type

Ports
 uvm_algorithmic_comparator#(BEFORE,AFTER,TRANSFORMER)
 uvm_driver#(REQ,RSP)
 uvm_in_order_comparator#(T,comp_type,convert,pair_type)

```
uvm_push_driver#(REQ,RSP)
uvm_push_sequencer#(REQ,RSP)
uvm_random_stimulus#(T)
uvm_sequencer_param_base#(REQ,RSP)
uvm_subscriber
uvm_tlm_analysis_fifo#(T)
uvm_tlm_fifo_base#(T)
uvm_tlm_req_rsp_channel#(REQ,RSP)
uvm_tlm_transport_channel#(REQ,RSP)
```

Ports,Exports, and Imps

post_body

```
    uvm_sequence_base
```

POST_BODY

post_do

```
    uvm_sequence_base
```

post_trigger

```
    uvm_event_callback
```

pound_zero_count

```
    uvm_sequencer_base
```

pre_body

```
    uvm_sequence_base
```

PRE_BODY

pre_do

```
    uvm_sequence_base
```

pre_trigger

```
    uvm_event_callback
```

Predefined Component Classes

prefix

```
    uvm_printer_knobs
```

prev

```
    uvm_pool#(T)
```

print

```
    uvm_factory
    uvm_object
```

print_array_footer

```
    uvm_printer
```

print_array_header

```
    uvm_printer
```

print_array_range

```
    uvm_printer
```

print_catcher

```
    uvm_report_catcher
```

print_config_matches

```
    uvm_component
```

print_config_settings

```
    uvm_component
```

print_enabled

```
    uvm_component
```

print_field

```
    uvm_printer
```

print_footer

```
    uvm_printer
```

print_header

```
    uvm_printer
print_id
    uvm_printer
print_msg
    uvm_comparer
print_newline
    uvm_line_printer
    uvm_printer
print_object
    uvm_printer
print_object_header
    uvm_printer
print_override_info
    uvm_component
print_size
    uvm_printer
print_string
    uvm_printer
print_time
    uvm_printer
print_type_name
    uvm_printer
print_value
    uvm_printer
print_value_array
    uvm_printer
print_value_object
    uvm_printer
print_value_string
    uvm_printer
Printing
    uvm_object
process_report
    uvm_report_server
push_back
    uvm_queue#(T)
push_front
    uvm_queue#(T)
put
    uvm_sqr_if_base#(REQ,RSP)
    uvm_tlm_if_base#(T1,T2)
Put
put_ap
    uvm_tlm_fifo_base#(T)
put_export
    uvm_tlm_fifo_base#(T)
put_request_export
    uvm_tlm_req_rsp_channel#(REQ,RSP)
put_response_export
    uvm_tlm_req_rsp_channel#(REQ,RSP)
```

Q

qualify

 uvm_test_done_objection

R

raise_objection

 uvm_objection
 uvm_test_done_objection

raised

 uvm_component
 uvm_objection
 uvm_root

record

 uvm_object

record_error_tr

 uvm_component

record_event_tr

 uvm_component

record_field

 uvm_recorder

record_field_real

 uvm_recorder

record_generic

 uvm_recorder

record_object

 uvm_recorder

record_string

 uvm_recorder

record_time

 uvm_recorder

Recording

 uvm_object

Recording Interface

 uvm_component

recursion_policy

 uvm_recorder

reference

 uvm_printer_knobs

register

 uvm_factory

Registering Types

 uvm_factory

remove

uvm_heartbeat
report
 uvm_component
 uvm_report_handler

Report Macros

report_error_hook
 uvm_report_object
report_fatal_hook
 uvm_report_object
report_header
 uvm_report_object
report_hook
 uvm_report_object
report_info_hook
 uvm_report_object
report_summarize
 uvm_report_object
report_warning_hook
 uvm_report_object

Reporting

 Global
 uvm_globals
 uvm_object_globals
 uvm_report_catcher
 uvm_report_object

Reporting Classes

req_export
 uvm_push_driver#(REQ,RSP)
req_port
 uvm_push_sequencer#(REQ,RSP)
request_ap
 uvm_tlm_req_rsp_channel#(REQ,RSP)
reseed
 uvm_object
reset
 uvm_barrier
 uvm_event
 uvm_phase
reset_quit_count
 uvm_report_server
reset_report_handler
 uvm_report_object
reset_severity_counts
 uvm_report_server
resolve_bindings
 uvm_component
 uvm_port_base#(IF)
response_ap
 uvm_tlm_req_rsp_channel#(REQ,RSP)
response_handler
 uvm_sequence_base

result

```

    uvm_comparer
resume
    uvm_component

rsp_export
    uvm_sequencer_param_base#(REQ,RSP)

rsp_port
    uvm_driver#(REQ,RSP)
    uvm_push_driver#(REQ,RSP)

run
    uvm_component
    uvm_push_sequencer#(REQ,RSP)

run_hooks
    uvm_report_handler

run_test
    Global
    uvm_root

```

S

Seeding
 uvm_object

send_request
 uvm_sequence#(REQ,RSP)
 uvm_sequence_base
 uvm_sequencer_base
 uvm_sequencer_param_base#(REQ,RSP)

separator
 uvm_tree_printer_knobs

seq_item_export
 uvm_sequencer#(REQ,RSP)

seq_item_port
 uvm_driver#(REQ,RSP)

seq_kind
 uvm_sequence_base

Sequence Action Macros

Sequence and Do Action Macros

Sequence Classes

Sequence on Sequencer Action Macros

Sequence Registration Macros

Sequencer Classes

Sequencer Registration Macros

Sequences

set_action
 uvm_report_catcher

set_arbitration
 uvm_sequencer_base

set_auto_reset
 uvm_barrier

set_config_int
 Global
 uvm_component

set_config_object
Global
uvm_component

set_config_string
Global
uvm_component

set_default_index
uvm_port_base#(IF)

set_depth
uvm_sequence_item

set_drain_time
uvm_objection

set_global_stop_timeout

set_global_timeout

set_heartbeat
uvm_heartbeat

set_id
uvm_report_catcher

set_id_count
uvm_report_server

set_id_info
uvm_sequence_item

set_initiator
uvm_transaction

set_inst_override
uvm_component
uvm_component_registry#(T,Tname)
uvm_object_registry#(T,Tname)

set_inst_override_by_name
uvm_factory

set_inst_override_by_type
uvm_component
uvm_factory

set_int_local
uvm_object

set_max_quit_count
uvm_report_server

set_message
uvm_report_catcher

set_name
uvm_component
uvm_object

set_num_last_reqs
uvm_sequencer_param_base#(REQ,RSP)

set_num_last_rsps
uvm_sequencer_param_base#(REQ,RSP)

set_object_local
uvm_object

set_parent_sequence
uvm_sequence_item

set_priority
uvm_sequence_base

```
set_quit_count
    uvm_report_server

set_report_default_file
    uvm_report_object

set_report_default_file_hier
    uvm_component

set_report_handler
    uvm_report_object

set_report_id_action
    uvm_report_object

set_report_id_action_hier
    uvm_component

set_report_id_file
    uvm_report_object

set_report_id_file_hier
    uvm_component

set_report_max_quit_count
    uvm_report_object

set_report_severity_action
    uvm_report_object

set_report_severity_action_hier
    uvm_component

set_report_severity_file
    uvm_report_object

set_report_severity_file_hier
    uvm_component

set_report_severity_id_action
    uvm_report_object

set_report_severity_id_action_hier
    uvm_component

set_report_severity_id_file
    uvm_report_object

set_report_severity_id_file_hier
    uvm_component

set_report_verbosity_level
    uvm_report_object

set_report_verbosity_level_hier
    uvm_component

set_response_queue_depth
    uvm_sequence#(REQ,RSP)

set_response_queue_error_report_disabled
    uvm_sequence#(REQ,RSP)

set_sequencer
    uvm_sequence#(REQ,RSP)
    uvm_sequence_base
    uvm_sequence_item

set_severity_count
    uvm_report_server

set_string_local
    uvm_object

set_threshold
```

```
    uvm_barrier
set_transaction_id
    uvm_transaction

set_type_override
    uvm_component
    uvm_component_registry#(T,Tname)
    uvm_object_registry#(T,Tname)

set_type_override_by_name
    uvm_factory

set_type_override_by_type
    uvm_component
    uvm_factory

set_use_sequence_info
    uvm_sequence_item

set_verbosity
    uvm_report_catcher

Setup
    uvm_report_object

sev
    uvm_comparer

show_max
    uvm_comparer

show_radix
    uvm_printer_knobs

show_root
    uvm_hier_printer_knobs

Simulation Control

size
    uvm_port_base#(IF)
    uvm_printer_knobs
    uvm_queue#(T)
    uvm_tlm_fifo#(T)

size_width
    uvm_table_printer_knobs

slave_export
    uvm_tlm_req_rsp_channel#(REQ,RSP)

sprint
    uvm_object

start
    uvm_heartbeat
    uvm_sequence#(REQ,RSP)
    uvm_sequence_base

start_default_sequence
    uvm_sequencer_base
    uvm_sequencer_param_base#(REQ,RSP)

start_item
    uvm_sequence_base
    uvm_sequence_item

start_of_simulation
    uvm_component

status
    uvm_component
```

stop
 uvm_component
 uvm_heartbeat

stop_request
 uvm_root

stop_sequences
 uvm_sequencer#(REQ,RSP)
 uvm_sequencer_base

stop_stimulus_generation
 uvm_random_stimulus#(T)

stop_timeout
 uvm_root

STOPPED

summarize
 uvm_report_server

summarize_report_catcher
 uvm_report_catcher

suspend
 uvm_component

Synchronization Classes

T

T
 uvm_callbacks#(T,CB)

TLM Implementation Port Declaration Macros

TLM Interfaces, Ports, and Exports

tr_handle
 uvm_recorder

trace_mode
 uvm_objection

transport
 uvm_tlm_if_base#(T1,T2)

Transport

transport_export
 uvm_tlm_transport_channel#(REQ,RSP)

trigger
 uvm_event

truncation
 uvm_printer_knobs

try_get
 uvm_tlm_if_base#(T1,T2)

try_next_item
 uvm_sqr_if_base#(REQ,RSP)

try_peek
 uvm_tlm_if_base#(T1,T2)

try_put
 uvm_tlm_if_base#(T1,T2)

Type&Instance Overrides
 uvm_factory

type_name
 uvm_printer_knobs

type_width
 uvm_table_printer_knobs

Types and Enumerations

U

ungrab

 uvm_sequence_base
 uvm_sequencer_base

Unidirectional Interfaces&Ports

unlock

 uvm_sequence_base
 uvm_sequencer_base

unpack

 uvm_object

unpack_bytes

 uvm_object

unpack_field

 uvm_packer

unpack_field_int

 uvm_packer

unpack_ints

 uvm_object

unpack_object

 uvm_packer

unpack_real

 uvm_packer

unpack_string

 uvm_packer

unpack_time

 uvm_packer

Unpacking

 uvm_object
 uvm_packer

unsigned_radix

 uvm_printer_knobs

Usage

Global

 uvm_phases
 tlm_ifs_and_ports

 uvm_factory
 uvm_object_registry#(T,Tname)

use_metadata

 uvm_packer

use_response_handler

 uvm_sequence_base

use_uvm_seeding

 uvm_object

used

 uvm_tlm_fifo#(T)

user_priority_arbitration

uvm_sequencer_base
Utility and Field Macros for Components and Objects
Utility Macros
UVM Class Reference
UVM Factory
uvm_*_export#(REQ,RSP)
uvm_*_export#(T)
uvm_*_imp ports
uvm_*_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_*_imp#(T,IMP)
uvm_*_port#(REQ,RSP)
uvm_*_port#(T)
uvm_action
uvm_agent
uvm_algorithmic_comparator#(BEFORE,AFTER,TRANSFORMER)
uvm_algorithmic_comparator
uvm_barrier
UVM_BIN
uvm_bits_to_string
uvm_bitstream_t
uvm_built_in_clone#(T)
uvm_built_in_comp#(T)
uvm_built_in_converter#(T)
uvm_built_in_pair#(T1,T2)
UVM_CALL_HOOK
uvm_callback
uvm_callbackDefines
uvm_callback_iter
uvm_callbacks#(T,CB)
uvm_class_clone#(T)
uvm_class_comp#(T)
uvm_class_converter#(T)
uvm_comparer
uvm_component
uvm_component_registry#(T,Tname)
UVM_COUNT
UVM_DEC
UVM_DEEP
uvm_default_comparer
uvm_default_line_printer
uvm_default_packer
uvm_default_printer
uvm_default_recorder
uvm_default_table_printer
uvm_default_tree_printer
UVM_DISPLAY
uvm_driver#(REQ,RSP)
UVM_ENUM
uvm_env
UVM_ERROR
uvm_event
uvm_event_callback
uvm_exhaustive_sequence
UVM_EXIT
UVM_EXPORT
uvm_factory
UVM_FATAL
UVM_FULL
uvm_heartbeat
UVM_HEX

```
uvm_hier_printer_knobs
UVM_HIGH
UVM_IMPLEMENTATION
uvm_in_order_comparator#(T,comp_type,convert,pair_type)
UVM_INFO
uvm_is_match
uvm_line_printer
UVM_LOG
UVM_LOW
UVM_MEDIUM
uvm_monitor
UVM_NO_ACTION
UVM_NONE
uvm_object
uvm_object_registry#(T,Tname)
uvm_object_string_pool#(T)
uvm_object_wrapper
uvm_objection
uvm_objection_cb
UVM_OCT
uvm_packer
uvm_pair#(T1,T2)
uvm_phase
uvm_policies
uvm_pool#(T)
UVM_PORT
uvm_port_base#(IF)
uvm_port_type_e
uvm_printer
uvm_printer_knobs
uvm_push_driver#(REQ,RSP)
uvm_push_sequencer#(REQ,RSP)
uvm_queue#(T)
uvm_radix_enum
uvm_random_sequence
uvm_random_stimulus#(T)
uvm_recorder
uvm_recursion_policy_enum
UVM_REFERENCE
uvm_report_catcher
uvm_report_enabled
    Global
    uvm_report_object
uvm_report_error
    Global
    uvm_report_catcher
    uvm_report_object
uvm_report_fatal
    Global
    uvm_report_catcher
    uvm_report_object
uvm_report_handler
uvm_report_info
    Global
    uvm_report_catcher
    uvm_report_object
uvm_report_object
uvm_report_server
uvm_report_warning
```

```

Global
  uvm_report_catcher
  uvm_report_object

uvm_root
uvm_scoreboard
  uvm_seq_item_pull_export#(REQ,RSP)
  uvm_seq_item_pull_imp#(REQ,RSP,IMP)
  uvm_seq_item_pull_port#(REQ,RSP)
  uvm_sequence#(REQ,RSP)
uvm_sequence_base
uvm_sequence_item
uvm_sequence_state_enum
uvm_sequencer#(REQ,RSP)
uvm_sequencer_base
uvm_sequencer_param_base#(REQ,RSP)
uvm_severity
UVM_SHALLOW
uvm_simple_sequence
uvm_sqr_if_base#(REQ,RSP)
UVM_STRING
uvm_string_to_bits
uvm_subscriber
uvm_table_printer
uvm_table_printer_knobs
uvm_test
uvm_test_done
uvm_test_done_objection
UVM_TIME
uvm_tlm_analysis_fifo#(T)
uvm_tlm_fifo#(T)
uvm_tlm_fifo_base#(T)
uvm_tlm_if_base#(T1,T2)
uvm_tlm_req_rsp_channel#(REQ,RSP)
uvm_tlm_transport_channel#(REQ,RSP)
uvm_top
  uvm_root
uvm_transaction
uvm_tree_printer
uvm_tree_printer_knobs
UVM_UNSIGNED
uvm_verbosity
uvm_void
uvm_wait_for_nba_region
UVM_WARNING

```

V

value_width
 uvm_table_printer_knobs

Variables

uvm_comparer
 uvm_hier_printer_knobs
 uvm_line_printer
 uvm_packer
 uvm_printer_knobs
 uvm_recorder
 uvm_report_server

```
uvm_root  
uvm_sequence_base  
uvm_sequencer#(REQ,RSP)  
uvm_sequencer_base  
uvm_table_printer  
uvm_table_printer_knobs  
uvm_tree_printer  
uvm_tree_printer_knobs
```

verbosity

```
uvm_comparer
```

Verbosity is ignored for warnings,errors, and fats to ensure users

W

```
wait_done  
uvm_phase  
wait_for  
uvm_barrier  
wait_for_grant  
uvm_sequence_base  
uvm_sequencer_base  
wait_for_item_done  
uvm_sequence_base  
uvm_sequencer_base  
wait_for_relevant  
uvm_sequence_base  
wait_for_sequence_state  
uvm_sequence_base  
wait_for_sequences  
uvm_sequencer_base  
uvm_sqr_if_base#(REQ,RSP)  
wait_off  
uvm_event  
wait_on  
uvm_event  
wait_ptrigger  
uvm_event  
wait_ptrigger_data  
uvm_event  
wait_start  
uvm_phase  
wait_trigger  
uvm_event  
wait_trigger_data  
uvm_event  
write  
uvm_subscriber  
uvm_tlm_if_base#(T1,T2)
```

Class Index

\$#! · 0-9 · A · B · C · D · E · F · G · H · I · J · K · L · M · N · O · P · Q · R · S · T · U · V · W · X · Y · Z

I

in_order_built_in_comparator#(T)
in_order_class_comparator#(T)

U

uvm_*_export#(REQ,RSP)
uvm_*_export#(T)
uvm_*_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
uvm_*_imp#(T,IMP)
uvm_*_port#(REQ,RSP)
uvm_*_port#(T)
uvm_agent
uvm_algorithmic_comparator#(BEFORE,AFTER,TRANSFORMER)
uvm_barrier
uvm_built_in_clone#(T)
uvm_built_in_comp#(T)
uvm_built_in_converter#(T)
uvm_built_in_pair#(T1,T2)
uvm_callback
uvm_callback_iter
uvm_callbacks#(T,CB)
uvm_class_clone#(T)
uvm_class_comp#(T)
uvm_class_converter#(T)
uvm_comparer
uvm_component
uvm_component_registry#(T,Tname)
uvm_driver#(REQ,RSP)
uvm_env
uvm_event
uvm_event_callback
uvm_exhaustive_sequence
uvm_factory
uvm_heartbeat
uvm_hier_printer_knobs
uvm_in_order_comparator#(T,comp_type,convert,pair_type)
uvm_line_printer
uvm_monitor
uvm_object
uvm_object_registry#(T,Tname)
uvm_object_string_pool#(T)
uvm_object_wrapper
uvm_objection
uvm_objection_cb
uvm_packer
uvm_pair#(T1,T2)
uvm_phase
uvm_pool#(T)
uvm_port_base#(IF)

```
uvm_printer
uvm_printer_knobs
uvm_push_driver#(REQ,RSP)
uvm_push_sequencer#(REQ,RSP)
uvm_queue#(T)
uvm_random_sequence
uvm_random_stimulus#(T)
uvm_recorder
uvm_report_catcher
uvm_report_handler
uvm_report_object
uvm_report_server
uvm_root
uvm_scoreboard
uvm_seq_item_pull_export#(REQ,RSP)
uvm_seq_item_pull_imp#(REQ,RSP,IMP)
uvm_seq_item_pull_port#(REQ,RSP)
uvm_sequence#(REQ,RSP)
uvm_sequence_base
uvm_sequence_item
uvm_sequencer#(REQ,RSP)
uvm_sequencer_base
uvm_sequencer_param_base#(REQ,RSP)
uvm_simple_sequence
uvm_sqr_if_base#(REQ,RSP)
uvm_subscriber
uvm_table_printer
uvm_table_printer_knobs
uvm_test
uvm_test_done_objection
uvm_tlm_analysis_fifo#(T)
uvm_tlm_fifo#(T)
uvm_tlm_fifo_base#(T)
uvm_tlm_if_base#(T1,T2)
uvm_tlm_req_rsp_channel#(REQ,RSP)
uvm_tlm_transport_channel#(REQ,RSP)
uvm_transaction
uvm_tree_printer
uvm_tree_printer_knobs
```

File Index

[\\$#! · 0-9 · A · B · C · D · E · F · G · H · I · J · K · L · M · N · O · P · Q · R · S · T · U · V · W · X · Y · Z](#)

U

 [uvm_algorithmic_comparator](#)

 [uvm_callbackDefines](#)

 [uvm_policies](#)

Macro Index

[\\$#!](#) · [0-9](#) · [A](#) · [B](#) · [C](#) · [D](#) · [E](#) · [F](#) · [G](#) · [H](#) · [I](#) · [J](#) · [K](#) · [L](#) · [M](#) · [N](#) · [O](#) · [P](#) · [Q](#) · [R](#) · [S](#) · [T](#) · [U](#) · [V](#) · [W](#) · [X](#) · [Y](#) · [Z](#)

\$#!

- [`uvm_analysis_imp_decl](#)
- [`uvm_blocking_get_imp_decl](#)
- [`uvm_blocking_get_peek_imp_decl](#)
- [`uvm_blocking_master_imp_decl](#)
- [`uvm_blocking_peek_imp_decl](#)
- [`uvm_blocking_put_imp_decl](#)
- [`uvm_blocking_slave_imp_decl](#)
- [`uvm_blocking_transport_imp_decl](#)
- [`uvm_component_end](#)
- [`uvm_component_param_utils](#)
- [`uvm_component_param_utils_begin](#)
- [`uvm_component_utils](#)
- [`uvm_component_utils_begin](#)
- [`uvm_create](#)
- [`uvm_create_on](#)
- [`uvm_declare_p_sequencer](#)
- [`uvm_do](#)
- [`uvm_do_callbacks](#)
- [`uvm_do_callbacks_exit_on](#)
- [`uvm_do_obj_callbacks](#)
- [`uvm_do_obj_callbacks_exit_on](#)
- [`uvm_do_on](#)
- [`uvm_do_on_pri](#)
- [`uvm_do_on_pri_with](#)
- [`uvm_do_on_with](#)
- [`uvm_do_pri](#)
- [`uvm_do_pri_with](#)
- [`uvm_do_with](#)
- [`uvm_error](#)
- [`uvm_fatal](#)
- [`uvm_field_aa_int_byte](#)
- [`uvm_field_aa_int_byte_unsigned](#)
- [`uvm_field_aa_int_enumkey](#)
- [`uvm_field_aa_int_int](#)
- [`uvm_field_aa_int_int_unsigned](#)
- [`uvm_field_aa_int_integer](#)
- [`uvm_field_aa_int_integer_unsigned](#)
- [`uvm_field_aa_int_key](#)
- [`uvm_field_aa_int_longint](#)
- [`uvm_field_aa_int_longint_unsigned](#)
- [`uvm_field_aa_int_shortint](#)
- [`uvm_field_aa_int_shortint_unsigned](#)
- [`uvm_field_aa_int_string](#)
- [`uvm_field_aa_object_int](#)
- [`uvm_field_aa_object_string](#)
- [`uvm_field_aa_string_string](#)
- [`uvm_field_array_enum](#)
- [`uvm_field_array_int](#)
- [`uvm_field_array_object](#)
- [`uvm_field_array_string](#)
- [`uvm_field_enum](#)
- [`uvm_field_event](#)

- `uvm_field_int
- `uvm_field_object
- `uvm_field_queue_enum
- `uvm_field_queue_int
- `uvm_field_queue_object
- `uvm_field_queue_string
- `uvm_field_real
- `uvm_field_sarray_enum
- `uvm_field_sarray_int
- `uvm_field_sarray_object
- `uvm_field_sarray_string
- `uvm_field_string
- `uvm_field_utils_begin
- `uvm_field_utils_end
- `uvm_get_imp_decl
- `uvm_get_peek_imp_decl
- `uvm_info
- `uvm_master_imp_decl
- `uvm_nonblocking_get_imp_decl
- `uvm_nonblocking_get_peek_imp_decl
- `uvm_nonblocking_master_imp_decl
- `uvm_nonblocking_peek_imp_decl
- `uvm_nonblocking_put_imp_decl
- `uvm_nonblocking_slave_imp_decl
- `uvm_nonblocking_transport_imp_decl
- `uvm_object_param_utils
- `uvm_object_param_utils_begin
- `uvm_object_utils
- `uvm_object_utils_begin
- `uvm_object_utils_end
- `uvm_peek_imp_decl
- `uvm_phase_func_bottomup_decl
- `uvm_phase_func_decl
- `uvm_phase_func_topdown_decl
- `uvm_phase_task_bottomup_decl
- `uvm_phase_task_decl
- `uvm_phase_task_topdown_decl
- `uvm_put_imp_decl
- `uvm_rand_send
- `uvm_rand_send_pri
- `uvm_rand_send_pri_with
- `uvm_rand_send_with
- `uvm_register_cb
- `uvm_send
- `uvm_send_pri
- `uvm_sequence_utils
- `uvm_sequence_utils_begin
- `uvm_sequence_utils_end
- `uvm_sequencer_param_utils
- `uvm_sequencer_param_utils_begin
- `uvm_sequencer_utils
- `uvm_sequencer_utils_begin
- `uvm_sequencer_utils_end
- `uvm_set_super_type
- `uvm_slave_imp_decl
- `uvm_transport_imp_decl
- `uvm_update_sequence_lib
- `uvm_update_sequence_lib_and_item
- `uvm_warning

Method Index

\$#! · 0-9 · A · B · C · D · E · F · G · H · I · J · K · L · M · N · O · P · Q · R · S · T · U · V · W · X · Y · Z

A

accept_tr
uvm_component
uvm_transaction

add
uvm_callbacks#(T,CB)
uvm_heartbeat
uvm_pool#(T)

add_by_name
uvm_callbacks#(T,CB)

add_callback
uvm_event

add_sequence
uvm_sequencer_base

all_dropped
uvm_component
uvm_objection
uvm_root
uvm_test_done_objection

apply_config_settings
uvm_component

B

begin_child_tr
uvm_component
uvm_transaction

begin_tr
uvm_component
uvm_transaction

body
uvm_sequence_base

build
uvm_component

C

call_func
uvm_phase

call_task
uvm_phase

callback_mode
uvm_callback

```
can_get
    uvm_tlm_if_base#(T1,T2)

can_peek
    uvm_tlm_if_base#(T1,T2)

can_put
    uvm_tlm_if_base#(T1,T2)

cancel
    uvm_barrier
    uvm_event

catch
    uvm_report_catcher

check
    uvm_component

check_config_usage
    uvm_component

clone
    uvm_object

compare
    uvm_object

compare_field
    uvm_comparer

compare_field_int
    uvm_comparer

compare_field_real
    uvm_comparer

compare_object
    uvm_comparer

compare_string
    uvm_comparer

compose_message
    uvm_report_server

connect
    uvm_component
    uvm_port_base#(IF)

convert2string
    uvm_object

copy
    uvm_object

create
    uvm_component_registry#(T,Tname)
    uvm_object
    uvm_object_registry#(T,Tname)

create_component
    uvm_component
    uvm_component_registry#(T,Tname)
    uvm_object_wrapper

create_component_by_name
    uvm_factory

create_component_by_type
    uvm_factory

create_item
    uvm_sequence_base
```

```

create_object
  uvm_component
  uvm_object_registry#(T,Tname)
  uvm_object_wrapper

create_object_by_name
  uvm_factory

create_object_by_type
  uvm_factory

current_grabber
  uvm_sequencer_base

```

D

```

debug_connected_to
  uvm_port_base#(IF)

debug_create_by_name
  uvm_factory

debug_create_by_type
  uvm_factory

debug_provided_to
  uvm_port_base#(IF)

delete
  uvm_callbacks#(T,CB)
  uvm_object_string_pool#(T)
  uvm_pool#(T)
  uvm_queue#(T)

delete_by_name
  uvm_callbacks#(T,CB)

delete_callback
  uvm_event

die
  uvm_report_object

disable_recording
  uvm_transaction

display_objections
  uvm_objection

do_accept_tr
  uvm_component
  uvm_transaction

do_begin_tr
  uvm_component
  uvm_transaction

do_compare
  uvm_object

do_copy
  uvm_object

do_end_tr
  uvm_component
  uvm_transaction

do_kill_all
  uvm_component

```

do_pack
 uvm_object
do_print
 uvm_object
do_record
 uvm_object
do_sequence_kind
 uvm_sequence_base
do_unpack
 uvm_object
drop
 uvm_test_done_objection
drop_objection
 uvm_objection
dropped
 uvm_component
 uvm_objection
dump_report_state
 uvm_report_object
dump_server_state
 uvm_report_server

E

enable_recording
 uvm_transaction
end_of_elaboration
 uvm_component
end_tr
 uvm_component
 uvm_transaction
execute_item
 uvm_sequencer_param_base#(REQ,RSP)
exists
 uvm_pool#(T)
extract
 uvm_component

F

find
 uvm_root
find_all
 uvm_root
find_override_by_name
 uvm_factory
find_override_by_type
 uvm_factory

```

finish_item
    uvm_sequence_base
    uvm_sequence_item

first
    uvm_callback_iter
    uvm_pool#(T)

flush
    uvm_in_order_comparator#(T,comp_type,convert,pair_type)
    uvm_tlm_fifo#(T)

force_stop
    uvm_test_done_objection

format_action
    uvm_report_handler

```

G

```

generate_stimulus
    uvm_random_stimulus#(T)

get
    uvm_component_registry#(T,Tname)
    uvm_object_registry#(T,Tname)
    uvm_object_string_pool#(T)
    uvm_pool#(T)
    uvm_queue#(T)
    uvm_sqr_if_base#(REQ,RSP)
    uvm_tlm_if_base#(T1,T2)

get_accept_time
    uvm_transaction

get_action
    uvm_report_catcher
    uvm_report_handler

get_begin_time
    uvm_transaction

get_cb
    uvm_callback_iter

get_child
    uvm_component

get_client
    uvm_report_catcher

get_comp
    uvm_port_base#(IF)

get_config_int
    uvm_component

get_config_object
    uvm_component

get_config_string
    uvm_component

get_count
    uvm_random_sequence

get_current_item
    uvm_sequence#(REQ,RSP)

```

```
    uvm_sequencer_param_base#(REQ,RSP)
get_current_phase
    uvm_root
get_depth
    uvm_sequence_item
get_drain_time
    uvm_objection
get_end_time
    uvm_transaction
get_event_pool
    uvm_transaction
get_file_handle
    uvm_report_handler
get_first
    uvm_callbacks#(T,CB)
get_first_child
    uvm_component
get_fname
    uvm_report_catcher
get_full_name
    uvm_component
    uvm_object
    uvm_port_base#(IF)
get_global
    uvm_pool#(T)
    uvm_queue#(T)
get_global_pool
    uvm_object_string_pool#(T)
    uvm_pool#(T)
get_global_queue
    uvm_queue#(T)
get_id
    uvm_report_catcher
get_id_count
    uvm_report_server
get_if
    uvm_port_base#(IF)
get_initiator
    uvm_transaction
get_inst_count
    uvm_object
get_inst_id
    uvm_object
get_line
    uvm_report_catcher
get_max_quit_count
    uvm_report_server
get_message
    uvm_report_catcher
get_name
    uvm_object
    uvm_phase
```

```
    uvm_port_base#(IF)
get_next
    uvm_callbacks#(T,CB)
get_next_child
    uvm_component
get_next_item
    uvm_sqr_if_base#(REQ,RSP)
get_num_children
    uvm_component
get_num_last_reqs
    uvm_sequencer_param_base#(REQ,RSP)
get_num_last_rsps
    uvm_sequencer_param_base#(REQ,RSP)
get_num_reqs_sent
    uvm_sequencer_param_base#(REQ,RSP)
get_num_rsps_received
    uvm_sequencer_param_base#(REQ,RSP)
get_num_waiters
    uvm_barrier
    uvm_event
get_object_type
    uvm_object
get_objection_count
    uvm_objection
get_objection_total
    uvm_objection
get_packed_size
    uvm_packer
get_parent
    uvm_component
    uvm_port_base#(IF)
get_parent_sequence
    uvm_sequence_item
get_phase_by_name
    uvm_root
get_priority
    uvm_sequence_base
get_quit_count
    uvm_report_server
get_radix_str
    uvm_printer_knobs
get_report_action
    uvm_report_object
get_report_catcher
    uvm_report_catcher
get_report_file_handle
    uvm_report_object
get_report_handler
    uvm_report_object
get_report_server
    uvm_report_object
```

```
get_report_verbosity_level
    uvm_report_object

get_response
    uvm_sequence#(REQ,RSP)

get_response_queue_depth
    uvm_sequence#(REQ,RSP)

get_response_queue_error_report_disabled
    uvm_sequence#(REQ,RSP)

get_root_sequence
    uvm_sequence_item

get_root_sequence_name
    uvm_sequence_item

get_seq_kind
    uvm_sequence_base
    uvm_sequencer_base

get_sequence
    uvm_sequence_base
    uvm_sequencer_base

get_sequence_by_name
    uvm_sequence_base

get_sequence_id
    uvm_sequence_item

get_sequence_path
    uvm_sequence_item

get_sequence_state
    uvm_sequence_base

get_sequencer
    uvm_sequence_base
    uvm_sequence_item

get_server
    uvm_report_server

get_severity
    uvm_report_catcher

get_severity_count
    uvm_report_server

get_threshold
    uvm_barrier

get_tr_handle
    uvm_transaction

get_transaction_id
    uvm_transaction

get_trigger_data
    uvm_event

get_trigger_time
    uvm_event

get_type
    uvm_object

get_type_name
    uvm_callback
    uvm_component_registry#(T,Tname)
    uvm_object
    uvm_object_registry#(T,Tname)
```

uvm_object_string_pool#(T)
uvm_object_wrapper
uvm_phase
uvm_port_base#(IF)

get_use_response_handler
uvm_sequence_base

get_use_sequence_info
uvm_sequence_item

get_verbosity
uvm_report_catcher

get_verbosity_level
uvm_report_handler

global_stop_request

grab
uvm_sequence_base
uvm_sequencer_base

H

has_child
uvm_component

has_do_available
uvm_sequencer_base
uvm_sqr_if_base#(REQ,RSP)

has_lock
uvm_sequence_base
uvm_sequencer_base

hb_mode
uvm_heartbeat

I

in_stop_request
uvm_root

incr_id_count
uvm_report_server

incr_quit_count
uvm_report_server

incr_severity_count
uvm_report_server

insert
uvm_queue#(T)

insert_phase
uvm_root

is_active
uvm_transaction

is_blocked
uvm_sequence_base
uvm_sequencer_base

```
is_child
    uvm_sequencer_base

is_done
    uvm_phase

is_empty
    uvm_tlm_fifo#(T)

is_enabled
    uvm_callback

is_export
    uvm_port_base#(IF)

is_full
    uvm_tlm_fifo#(T)

is_grabbed
    uvm_sequencer_base

is_imp
    uvm_port_base#(IF)

is_in_progress
    uvm_phase

is_item
    uvm_sequence_base
    uvm_sequence_item

is_null
    uvm_packer

is_off
    uvm_event

is_on
    uvm_event

is_port
    uvm_port_base#(IF)

is_quit_count_reached
    uvm_report_server

is_recording_enabled
    uvm_transaction

is_relevant
    uvm_sequence_base

is_task
    uvm_phase

is_top_down
    uvm_phase

is_unbounded
    uvm_port_base#(IF)

issue
    uvm_report_catcher

item_done
    uvm_sqr_if_base#(REQ,RSP)
```

K

```
kill
    uvm_component
```

L

last
 uvm_pool#(T)
last_req
 uvm_sequencer_param_base#(REQ,RSP)
last_rsp
 uvm_sequencer_param_base#(REQ,RSP)
lock
 uvm_sequence_base
 uvm_sequencer_base
lookup
 uvm_component

M

max_size
 uvm_port_base#(IF)
mid_do
 uvm_sequence_base
min_size
 uvm_port_base#(IF)

nb_transport

```
  uvm_tlm_if_base#(T1,T2)

new
  uvm_*_export#(REQ,RSP)
  uvm_*_export#(T)
  uvm_*_imp#(REQ,RSP,IMP,REQ_IMP,RSP_IMP)
  uvm_*_imp#(T,IMP)
  uvm_*_port#(REQ,RSP)
  uvm_*_port#(T)
  uvm_agent
  uvm_algorithmic_comparator#(BEFORE,AFTER,TRANSFORMER)
  uvm_barrier
  uvm_built_in_pair#(T1,T2)
  uvm_callback
  uvm_callback_iter
  uvm_component
  uvm_driver#(REQ,RSP)
  uvm_env
  uvm_event
  uvm_event_callback
  uvm_heartbeat
  uvm_monitor
  uvm_object
  uvm_object_string_pool#(T)
  uvm_objection
  uvm_pair#(T1,T2)
  uvm_phase
  uvm_pool#(T)
  uvm_port_base#(IF)
  uvm_push_driver#(REQ,RSP)
  uvm_push_sequencer#(REQ,RSP)
  uvm_queue#(T)
  uvm_random_stimulus#(T)
  uvm_report_catcher
  uvm_report_handler
  uvm_report_object
  uvm_report_server
  uvm_scoreboard
  uvm_sequence#(REQ,RSP)
  uvm_sequence_base
  uvm_sequence_item
  uvm_sequencer#(REQ,RSP)
  uvm_sequencer_base
  uvm_sequencer_param_base#(REQ,RSP)
  uvm_subscriber
  uvm_test
  uvm_tlm_analysis_fifo#(T)
  uvm_tlm_fifo#(T)
  uvm_tlm_fifo_base#(T)
  uvm_tlm_req_rsp_channel#(REQ,RSP)
  uvm_tlm_transport_channel#(REQ,RSP)
  uvm_transaction
```

```
next
    uvm_callback_iter
    uvm_pool#(T)

num
    uvm_pool#(T)

num_sequences
    uvm_sequence_base
    uvm_sequencer_base
```

P

```
pack
    uvm_object

pack_bytes
    uvm_object

pack_field
    uvm_packer

pack_field_int
    uvm_packer

pack_ints
    uvm_object

pack_object
    uvm_packer

pack_real
    uvm_packer

pack_string
    uvm_packer

pack_time
    uvm_packer

peek
    uvm_sqr_if_base#(REQ,RSP)
    uvm_tlm_if_base#(T1,T2)

pop_back
    uvm_queue#(T)

pop_front
    uvm_queue#(T)

post_body
    uvm_sequence_base

post_do
    uvm_sequence_base

post_trigger
    uvm_event_callback

pre_body
    uvm_sequence_base

pre_do
    uvm_sequence_base

pre_trigger
    uvm_event_callback

prev
    uvm_pool#(T)
```

```
print
    uvm_factory
    uvm_object

print_array_footer
    uvm_printer

print_array_header
    uvm_printer

print_array_range
    uvm_printer

print_catcher
    uvm_report_catcher

print_config_settings
    uvm_component

print_field
    uvm_printer

print_footer
    uvm_printer

print_header
    uvm_printer

print_id
    uvm_printer

print_msg
    uvm_comparer

print_newline
    uvm_line_printer
    uvm_printer

print_object
    uvm_printer

print_object_header
    uvm_printer

print_override_info
    uvm_component

print_size
    uvm_printer

print_string
    uvm_printer

print_time
    uvm_printer

print_type_name
    uvm_printer

print_value
    uvm_printer

print_value_array
    uvm_printer

print_value_object
    uvm_printer

print_value_string
    uvm_printer

process_report
    uvm_report_server

push_back
```

uvm_queue#(T)
push_front
 uvm_queue#(T)
put
 uvm_sqr_if_base#(REQ,RSP)
 uvm_tlm_if_base#(T1,T2)

Q

qualify
 uvm_test_done_objection

R

raise_objection
 uvm_objection
 uvm_test_done_objection
raised
 uvm_component
 uvm_objection
 uvm_root

record
 uvm_object

record_error_tr
 uvm_component

record_event_tr
 uvm_component

record_field
 uvm_recorder

record_field_real
 uvm_recorder

record_generic
 uvm_recorder

record_object
 uvm_recorder

record_string
 uvm_recorder

record_time
 uvm_recorder

register
 uvm_factory

remove
 uvm_heartbeat

report
 uvm_component
 uvm_report_handler

report_error_hook
 uvm_report_object

```
report_fatal_hook
    uvm_report_object

report_header
    uvm_report_object

report_hook
    uvm_report_object

report_info_hook
    uvm_report_object

report_summarize
    uvm_report_object

report_warning_hook
    uvm_report_object

reseed
    uvm_object

reset
    uvm_barrier
    uvm_event
    uvm_phase

reset_quit_count
    uvm_report_server

reset_report_handler
    uvm_report_object

reset_severity_counts
    uvm_report_server

resolve_bindings
    uvm_component
    uvm_port_base#(IF)

response_handler
    uvm_sequence_base

resume
    uvm_component

run
    uvm_component
    uvm_push_sequencer#(REQ,RSP)

run_hooks
    uvm_report_handler

run_test
    Global
    uvm_root
```

S

```
send_request
    uvm_sequence#(REQ,RSP)
    uvm_sequence_base
    uvm_sequencer_base
    uvm_sequencer_param_base#(REQ,RSP)

set_action
    uvm_report_catcher

set_arbitration
    uvm_sequencer_base
```

```
set_auto_reset
    uvm_barrier

set_config_int
    Global
    uvm_component

set_config_object
    Global
    uvm_component

set_config_string
    Global
    uvm_component

set_default_index
    uvm_port_base#(IF)

set_depth
    uvm_sequence_item

set_drain_time
    uvm_objection

set_global_stop_timeout
set_global_timeout
set_heartbeat
    uvm_heartbeat

set_id
    uvm_report_catcher

set_id_count
    uvm_report_server

set_id_info
    uvm_sequence_item

set_initiator
    uvm_transaction

set_inst_override
    uvm_component
    uvm_component_registry#(T,Tname)
    uvm_object_registry#(T,Tname)

set_inst_override_by_name
    uvm_factory

set_inst_override_by_type
    uvm_component
    uvm_factory

set_int_local
    uvm_object

set_max_quit_count
    uvm_report_server

set_message
    uvm_report_catcher

set_name
    uvm_component
    uvm_object

set_num_last_reqs
    uvm_sequencer_param_base#(REQ,RSP)

set_num_last_rsp
    uvm_sequencer_param_base#(REQ,RSP)

set_object_local
```

```
    uvm_object
set_parent_sequence
    uvm_sequence_item
set_priority
    uvm_sequence_base
set_quit_count
    uvm_report_server
set_report_default_file
    uvm_report_object
set_report_default_file_hier
    uvm_component
set_report_handler
    uvm_report_object
set_report_id_action
    uvm_report_object
set_report_id_action_hier
    uvm_component
set_report_id_file
    uvm_report_object
set_report_id_file_hier
    uvm_component
set_report_max_quit_count
    uvm_report_object
set_report_severity_action
    uvm_report_object
set_report_severity_action_hier
    uvm_component
set_report_severity_file
    uvm_report_object
set_report_severity_file_hier
    uvm_component
set_report_severity_id_action
    uvm_report_object
set_report_severity_id_action_hier
    uvm_component
set_report_severity_id_file
    uvm_report_object
set_report_severity_id_file_hier
    uvm_component
set_report_verbosity_level
    uvm_report_object
set_report_verbosity_level_hier
    uvm_component
set_response_queue_depth
    uvm_sequence#(REQ,RSP)
set_response_queue_error_report_disabled
    uvm_sequence#(REQ,RSP)
set_sequencer
    uvm_sequence#(REQ,RSP)
    uvm_sequence_base
    uvm_sequence_item
```

set_severity_count
 uvm_report_server

set_string_local
 uvm_object

set_threshold
 uvm_barrier

set_transaction_id
 uvm_transaction

set_type_override
 uvm_component
 uvm_component_registry#(T,Tname)
 uvm_object_registry#(T,Tname)

set_type_override_by_name
 uvm_factory

set_type_override_by_type
 uvm_component
 uvm_factory

set_use_sequence_info
 uvm_sequence_item

set_verbosity
 uvm_report_catcher

size
 uvm_port_base#(IF)
 uvm_queue#(T)
 uvm_tlm_fifo#(T)

sprint
 uvm_object

start
 uvm_heartbeat
 uvm_sequence#(REQ,RSP)
 uvm_sequence_base

start_default_sequence
 uvm_sequencer_base
 uvm_sequencer_param_base#(REQ,RSP)

start_item
 uvm_sequence_base
 uvm_sequence_item

start_of_simulation
 uvm_component

status
 uvm_component

stop
 uvm_component
 uvm_heartbeat

stop_request
 uvm_root

stop_sequences
 uvm_sequencer#(REQ,RSP)
 uvm_sequencer_base

stop_stimulus_generation
 uvm_random_stimulus#(T)

summarize
 uvm_report_server

summarize_report_catcher
 uvm_report_catcher

suspend
 uvm_component

T

trace_mode
 uvm_objection

transport
 uvm_tlm_if_base#(T1,T2)

trigger
 uvm_event

try_get
 uvm_tlm_if_base#(T1,T2)

try_next_item
 uvm_sqr_if_base#(REQ,RSP)

try_peek
 uvm_tlm_if_base#(T1,T2)

try_put
 uvm_tlm_if_base#(T1,T2)

U

ungrab
 uvm_sequence_base
 uvm_sequencer_base

unlock
 uvm_sequence_base
 uvm_sequencer_base

unpack
 uvm_object

unpack_bytes
 uvm_object

unpack_field
 uvm_packer

unpack_field_int
 uvm_packer

unpack_ints
 uvm_object

unpack_object
 uvm_packer

unpack_real
 uvm_packer

unpack_string
 uvm_packer

unpack_time
 uvm_packer

use_response_handler
 uvm_sequence_base

used
 uvm_tlm_fifo#(T)

user_priority_arbitration
 uvm_sequencer_base

uvm_bits_to_string

uvm_is_match

uvm_report_enabled
 Global
 uvm_report_object

uvm_report_error
 Global
 uvm_report_catcher
 uvm_report_object

uvm_report_fatal
 Global
 uvm_report_catcher
 uvm_report_object

uvm_report_info
 Global
 uvm_report_catcher

uvm_report_object
uvm_report_warning
 Global
 uvm_report_catcher
 uvm_report_object
uvm_string_to_bits
uvm_wait_for_nba_region

W

wait_done
 uvm_phase
wait_for
 uvm_barrier
wait_for_grant
 uvm_sequence_base
 uvm_sequencer_base
wait_for_item_done
 uvm_sequence_base
 uvm_sequencer_base
wait_for_relevant
 uvm_sequence_base
wait_for_sequence_state
 uvm_sequence_base
wait_for_sequences
 uvm_sequencer_base
 uvm_sqr_if_base#(REQ,RSP)
wait_off
 uvm_event
wait_on
 uvm_event
wait_ptrigger
 uvm_event
wait_ptrigger_data
 uvm_event
wait_start
 uvm_phase
wait_trigger
 uvm_event
wait_trigger_data
 uvm_event
write
 uvm_subscriber
 uvm_tlm_if_base#(T1,T2)

Type Index

\$#! · 0-9 · A · B · C · D · E · F · G · H · I · J · K · L · M · N · O · P · Q · R · S · T · **U** · V · W · X · Y · Z

U

uvm_action
uvm_bitstream_t
uvm_port_type_e
uvm_radix_enum
uvm_recursion_policy_enum
uvm_sequence_state_enum
uvm_severity
uvm_verbosity

Variable Index

\$#! · 0-9 · A · B · C · D · E · F · G · H · I · J · K · L · M · N · O · P · Q · R · S · T · U · V · W · X · Y · Z

A

abstract

uvm_comparer
uvm_packer
uvm_recorder

B

begin_elements

uvm_printer_knobs

big_endian

uvm_packer

bin_radix

uvm_printer_knobs

C

check_type

uvm_comparer

count

uvm_sequencer_base

D

dec_radix

uvm_printer_knobs

default_radix

uvm_printer_knobs
uvm_recorder

default_sequence

uvm_sequencer_base

depth

uvm_printer_knobs

E

enable_print_topology

uvm_root

enable_stop_interrupt

uvm_component
end_elements

F

finish_on_completion

footer

full_name

G

global_indent

H

header

hex_radix

I

id_count

identifier

indent_str

K

knobs

M

max_random_count
 uvm_sequencer_base
max_random_depth
 uvm_sequencer_base
max_width
 uvm_printer_knobs
mcd
 uvm_printer_knobs
miscompares
 uvm_comparer

N

name_width
 uvm_table_printer_knobs
new
 uvm_line_printer
 uvm_table_printer
 uvm_tree_printer

O

oct_radix
 uvm_printer_knobs

P

phase_timeout
 uvm_root
physical
 uvm_comparer
 uvm_packer
 uvm_recorder
policy
 uvm_comparer
pound_zero_count
 uvm_sequencer_base
prefix
 uvm_printer_knobs
print_config_matches
 uvm_component
print_enabled
 uvm_component

R

recursion_policy
 uvm_recorder
reference
 uvm_printer_knobs
result
 uvm_comparer

S

separator
 uvm_tree_printer_knobs
seq_item_export
 uvm_sequencer#(REQ,RSP)
seq_kind
 uvm_sequence_base
sev
 uvm_comparer
show_max
 uvm_comparer
show_radix
 uvm_printer_knobs
show_root
 uvm_hier_printer_knobs
size
 uvm_printer_knobs
size_width
 uvm_table_printer_knobs
stop_timeout
 uvm_root

T

tr_handle
 uvm_recorder
truncation
 uvm_printer_knobs
type_name
 uvm_printer_knobs
type_width
 uvm_table_printer_knobs

U

unsigned_radix
 uvm_printer_knobs
use_metadata
 uvm_packer
use_uvm_seeding
 uvm_object
uvm_default_comparer
uvm_default_line_printer
uvm_default_packer
uvm_default_printer
uvm_default_recorder
uvm_default_table_printer
uvm_default_tree_printer
uvm_test_done
uvm_top
 uvm_root

V

value_width
 uvm_table_printer_knobs
verbosity
 uvm_comparer

Constant Index

\$#! · 0-9 · A · **B** · **C** · D · **E** · **F** · G · H · I · J · K · L · M · N · O · **P** · Q · R · **S** · T · **U** · V · W · X · Y · Z

B

 **BODY**

C

 **CREATED**

E

 **ENDED**

F

 **FINISHED**

P

 **POST_BODY**
 **PRE_BODY**

S

 **STOPPED**

U

 **UVM_BIN**
 **UVM_CALL_HOOK**
 **UVM_COUNT**
 **UVM_DEC**
 **UVM_DEEP**
 **UVM_DISPLAY**
 **UVM_ENUM**
 **UVM_ERROR**
 **UVM_EXIT**
 **UVM_EXPORT**
 **UVM_FATAL**

UVM_FULL
UVM_HEX
UVM_HIGH
UVM_IMPLEMENTATION
UVM_INFO
UVM_LOG
UVM_LOW
UVM_MEDIUM
UVM_NO_ACTION
UVM_NONE
UVM_OCT
UVM_PORT
UVM_REFERENCE
UVM_SHALLOW
UVM_STRING
UVM_TIME
UVM_UNSIGNED
UVM_WARNING

Port Index

\$#! · 0-9 · [A](#) · [B](#) · [C](#) · [D](#) · [E](#) · [F](#) · [G](#) · [H](#) · [I](#) · [J](#) · [K](#) · [L](#) · [M](#) · [N](#) · [O](#) · [P](#) · [Q](#) · [R](#) · [S](#) · [T](#) · [U](#) · [V](#) · [W](#) · [X](#) · [Y](#) · [Z](#)

A

after_export

uvm_algorithmic_comparator#(BEFORE,AFTER,TRANSFORMER)
uvm_in_order_comparator#(T,comp_type,convert,pair_type)

analysis_export

uvm_subscriber

analysis_port#(T)

uvm_tlm_analysis_fifo#(T)

B

before_export

uvm_algorithmic_comparator#(BEFORE,AFTER,TRANSFORMER)
uvm_in_order_comparator#(T,comp_type,convert,pair_type)

blocking_put_port

uvm_random_stimulus#(T)

G

get_ap

uvm_tlm_fifo_base#(T)

get_peek_export

uvm_tlm_fifo_base#(T)

get_peek_request_export

uvm_tlm_req_rsp_channel#(REQ,RSP)

get_peek_response_export

uvm_tlm_req_rsp_channel#(REQ,RSP)

M

master_export

uvm_tlm_req_rsp_channel#(REQ,RSP)

P

pair_ap

uvm_in_order_comparator#(T,comp_type,convert,pair_type)

put_ap

uvm_tlm_fifo_base#(T)

put_export
 uvm_tlm_fifo_base#(T)
put_request_export
 uvm_tlm_req_rsp_channel#(REQ,RSP)
put_response_export
 uvm_tlm_req_rsp_channel#(REQ,RSP)

R

req_export
 uvm_push_driver#(REQ,RSP)
req_port
 uvm_push_sequencer#(REQ,RSP)
request_ap
 uvm_tlm_req_rsp_channel#(REQ,RSP)
response_ap
 uvm_tlm_req_rsp_channel#(REQ,RSP)
rsp_export
 uvm_sequencer_param_base#(REQ,RSP)
rsp_port
 uvm_driver#(REQ,RSP)
 uvm_push_driver#(REQ,RSP)

S

seq_item_port
 uvm_driver#(REQ,RSP)
slave_export
 uvm_tlm_req_rsp_channel#(REQ,RSP)

T

transport_export
 uvm_tlm_transport_channel#(REQ,RSP)