Universal Verification Methodology 1.2 API Specification

Feb 26, 2025

Generated by Specador™ 25.1.3. Last updated on Feb 26, 2025.

OVERVIEW DOCUMENTATION

CHAPTER

ONE

UVM 1.2 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.

See also

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.

Globals

This category defines a small list of types, variables, functions, and tasks defined in the *uvm_pkg* scope. These items are accessible from any scope that imports the *uvm_pkg* .

See also

See Types and Enumerations and *Globals* for details.

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 also

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 also

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 also

See *Factory Classes* for details.

Phasing

This section describes the phasing capability provided by UVM.

Tip

The details can be found in *Phasing Overview*.

Configuration and Resources
: The *Configuration and Resource Classes* are a set of classes which provide a configuration database. The configuration database is used to store and retrieve both configuration time and run time properties.

Synchronization
: The UVM provides event and barrier synchronization classes for process synchronization.

See also

See *Synchronization Classes* for more information.

Containers
: The *Container Classes* are type parameterized data structures which provide queue and pool services. The class based queue and pool types allow for efficient sharing of the data structures compared with their SystemVerilog built-in counterparts.

Policies
: Each of UVM's policy classes performs 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 also

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 also

See *TLM Interfaces* for details.

Components
: Components form the foundation of the UVM. They encapsulate behavior of drivers, scoreboards, and other objects in a testbench. The UVM library provides a set of predefined component types, all derived directly or indirectly from *uvm_component*.

See also

See Predefined Component Classes for more information.

Sequencers
: The sequencer serves as an arbiter for controlling transaction flow from multiple stimulus generators. More specifically, the sequencer controls the flow of *uvm_sequence_item*-based transactions generated by one or more *uvm_sequence*-based sequences.

See also

See *Sequencer Classes* for more information.

Sequences
: Sequences encapsulate user-defined procedures that generate multiple *uvm_sequence_item*-based transactions. Such sequences can be reused, extended, randomized, and combined sequentially and hierarchically in interesting ways to produce realistic stimulus to your DUT.

See also

See *Sequence Classes* for more information.

Macros
: The UVM provides several macros to help increase user productivity. See the set of macro categories in the main menu for a complete list of macros for Reporting, Components, Objects, Sequences, Callbacks, TLM and Registers.

Register Layer
: The Register abstraction classes, when properly extended, abstract the read/write operations to registers and memories in a design-under-verification.

See also

See *Register Layer* for more information.

Command Line Processor
: The command line processor provides a general interface to the command line arguments that were provided for the given simulation.

Tip

The capabilities are detailed in the *uvm_cmdline_processor* section.

CHAPTER
TWO

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.

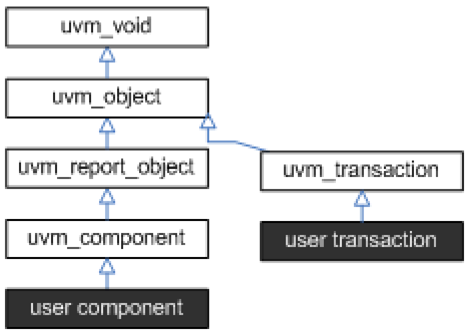


Fig. 1: Base Classes

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.

umv_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".

Tip

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.

2.1 Class uvm_pkg::uvm_void

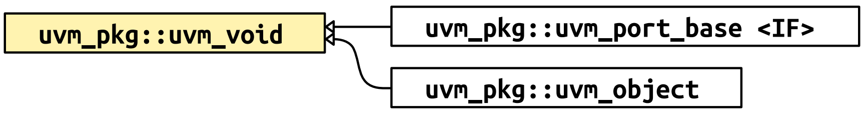


Fig. 2: Inheritance Diagram of uvm_void

Class

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.

2.2 Class uvm_pkg::uvm_object

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*

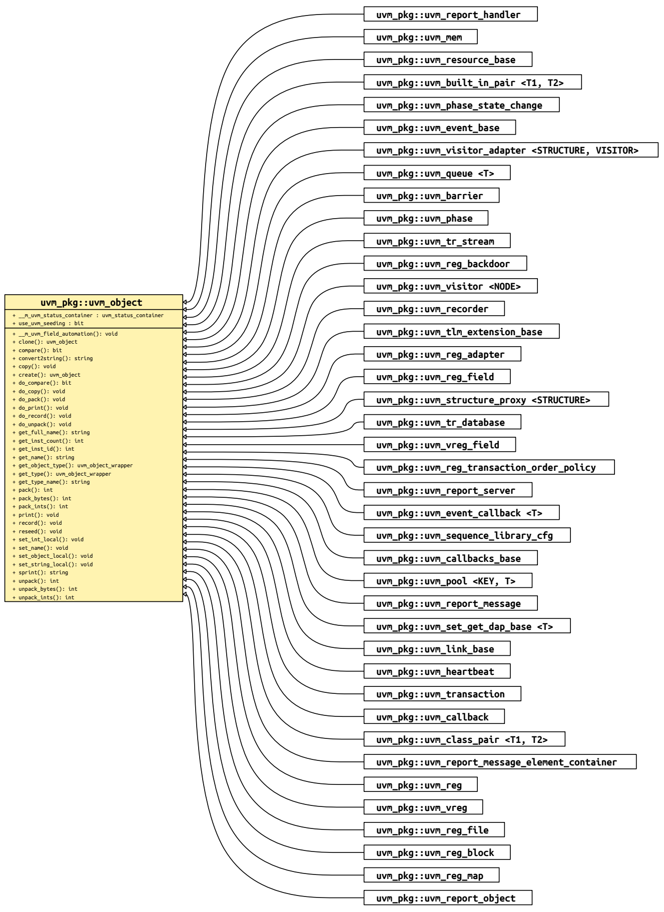

uvm_pkg::uvm_object
uvm_pkg::uvm_report_handler
uvm_pkg::uvm_mem
uvm_pkg::uvm_resource_base
uvm_pkg::uvm_built_in_pair <T1, T2>
uvm_pkg::uvm_phase_state_change
uvm_pkg::uvm_event_base
uvm_pkg::uvm_visitor_adapter <STRUCTURE, VISITOR>
uvm_pkg::uvm_queue <T>
uvm_pkg::uvm_barrier
uvm_pkg::uvm_phase
uvm_pkg::uvm_tr_stream
uvm_pkg::uvm_reg_backdoor
uvm_pkg::uvm_visitor <NODE>
uvm_pkg::uvm_recorder
uvm_pkg::uvm_tlm_extension_base
uvm_pkg::uvm_reg_adapter
uvm_pkg::uvm_reg_field
uvm_pkg::uvm_structure_proxy <STRUCTURE>
uvm_pkg::uvm_tr_database
uvm_pkg::uvm_vreg_field
uvm_pkg::uvm_reg_transaction_order_policy
uvm_pkg::uvm_report_server
uvm_pkg::uvm_event_callback <T>
uvm_pkg::uvm_sequence_library_cfg
uvm_pkg::uvm_callbacks_base
uvm_pkg::uvm_pool <KEY, T>
uvm_pkg::uvm_report_message
uvm_pkg::uvm_set_get_dap_base <T>
uvm_pkg::uvm_link_base
uvm_pkg::uvm_heartbeat
uvm_pkg::uvm_transaction
uvm_pkg::uvm_callback
uvm_pkg::uvm_class_pair <T1, T2>
uvm_pkg::uvm_report_message_element_container
uvm_pkg::uvm_reg
uvm_pkg::uvm_vreg
uvm_pkg::uvm_reg_file
uvm_pkg::uvm_reg_block
uvm_pkg::uvm_reg_map
uvm_pkg::uvm_report_object

Fig. 3: Inheritance Diagram of uvm_object

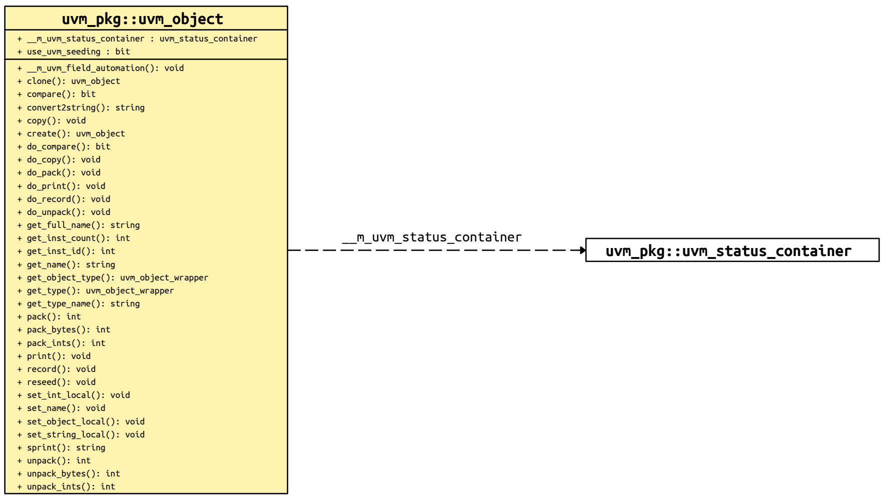


Fig. 4: Collaboration Diagram of uvm_object

CLASS

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*.

Table 1: Variables

Name	Type	Description
use_uvm_seeding	bit	***Variable*** use_uvm_seeding 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.

Constructors

`function  new(string name = "")`

Function

new

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

Parameters

name (*string*)

Functions

`function void reseed()`

Function

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. Reseed

`virtual  function void set_name(string name)`

Function

set_name

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

Parameters

name (*string*)

`virtual  function string get_name()`

Function

get_name

Returns the name of the object, as provided by the *name* argument in the *new* constructor or *set_name* method. Get_name

`virtual function string get_full_name()`

Function

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_full_name

`virtual function int get_inst_id()`

Function

get_inst_id

Returns the object's unique, numeric instance identifier. Get inst_id

`static function int get_inst_count()`

Function

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 inst_count

`static function uvm_object_wrapper get_type()`

Function

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 &96;uvm*utils macros, if employed. Get type

Return type
: *uvm_object_wrapper*

`virtual function uvm_object_wrapper get_object_type()`

Function

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 &96;uvm*utils macros, if employed. Get_object_type

Return type

uvm_object_wrapper

`virtual function string get_type_name()`

Function

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* .

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

Function

create

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
```

Parameters

name (`string`)

Return type

uvm_object

`virtual function uvm_object clone()`

Function

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. Clone

Return type

uvm_object

`function void print(uvm_printer printer = null)`

Function

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. 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. Print

Parameters

`printer` (*uvm_printer*)

`function string sprint(uvm_printer printer = null)`

Function

sprint

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. Sprint

Parameters

`printer` (*uvm_printer*)

`virtual  function void do_print(uvm_printer printer)`

Function

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 provided by the &96;uvm_field_* macros, <Utility and Field Macros for Components and Objects>.

The *printer* argument is the policy object that governs the format and content of the output. To ensure correct *print* and *sprint* operation, and to ensure a consistent output format, the *printer* must be used by all *do_print* implementations. That is, instead of using *$display* or string concatenations directly, a *do_print* implementation must call through the *printer's* API to add information to be printed or sprinted.

An example implementation of *do_print* is as follows:

```
class mytype extends uvm_object;
  data_obj data;
  int f1;
  virtual function void do_print (uvm_printer printer);
    super.do_print(printer);
    printer.print_field_int("f1", f1, $bits(f1), UVM_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. Do_print (virtual override)

Parameters

printer (*uvm_printer*)

virtual function string convert2string()

Convert2string (virtual)

function void record(uvm_recorder recorder = null)

Function

record

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. 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. Record

Parameters

recorder (*uvm_recorder*)

virtual function void do_record(uvm_recorder recorder)

Function

do_record

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("f1", f1, $bits(f1), UVM_DEC);
    recorder.record_object("data", data);
  endfunction. Do_record (virtual)
```

Parameters

recorder (*uvm_recorder*)

function void copy(uvm_object rhs)

Function

copy

The copy makes this object a copy of the specified 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. Copy

Parameters

rhs (*uvm_object*)

virtual function void do_copy(uvm_object rhs)

Function

do_copy

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. Do_copy

Parameters

rhs (*uvm_object*)

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

Function

compare

Deep compares members of this data object with those of the object provided in the *rhs* (right-hand side) argument, returning 1 on a match, 0 otherwise.

The *compare* method is not virtual and should not be overloaded in derived classes. To compare the fields of a derived class, that class should override the *do_compare* method.

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

Parameters

rhs (*uvm_object*)

comparer (*uvm_comparer*)

virtual function bit do_compare(uvm_object rhs, uvm_comparer comparer)

Function

do_compare

The *do_compare* method is the user-definable hook called by the *compare* method. A derived class should override this method to include its fields in a compare operation. It should return 1 if the comparison succeeds, 0 otherwise.

A typical implementation is as follows:

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

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

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

Parameters

`rhs` (*uvm_object*)

`comparer` (*uvm_comparer*)

`function int pack(bit bitstream, uvm_packer packer = null)`

Function

pack. Pack

Parameters

`bitstream` (`bit`)

`packer` (*uvm_packer*)

`function int pack_bytes(byte unsigned bytestream, uvm_packer packer = null)`

Function

pack_bytes. Pack_bytes

Parameters

`bytestream` (`byte unsigned`)

`packer` (*uvm_packer*)

`function int pack_ints(int unsigned intstream, uvm_packer packer = null)`

Function

pack_ints

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. Pack_ints

Parameters

`intstream` (`int unsigned`)

`packer` (*uvm_packer*)

`virtual  function void do_pack(uvm_packer packer)`

Function

do_pack

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 &96;uvm_field_* macros are used, <Utility and Field Macros for Components and Objects>, the above meta information is included provided the *uvm_packer::use_metadata* variable is set for the packer.

Packing order does not need to match declaration order. However, unpacking order must match packing order.
Do_pack
Parameters
packer (*uvm_packer*)

function int unpack(bit bitstream, uvm_packer packer = null)

Function

unpack. Unpack
Parameters
bitstream (`bit`)
packer (*uvm_packer*)

function int unpack_bytes(byte unsigned bytestream, uvm_packer packer = null)

Function

unpack_bytes. Unpack_bytes
Parameters
bytestream (`byte unsigned`)
packer (*uvm_packer*)

function int unpack_ints(int unsigned intstream, uvm_packer packer = null)

Function

unpack_ints

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. Unpack_ints
Parameters
intstream (`int unsigned`)
packer (*uvm_packer*)

virtual function void do_unpack(uvm_packer packer)

Function

do_unpack

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. Do_unpack

Parameters

packer (*uvm_packer*)

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

Function

set_int_local. Set_int_local

Parameters

field_name (*string*)
value (*uvm_bitstream_t*)
recurse (*bit*)

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

Function

set_string_local. Set_string_local

Parameters

field_name (*string*)
value (*string*)
recurse (*bit*)

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

Function

set_object_local

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;
```

(continues on next page)

(continued from previous page)

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

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

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

// provide access to sub-objects
function void set_object_local(string field_name, uvm_object value,
                               bit clone=1);
  if (uvm_is_match (field_name, "myobj")) begin
    if (value != null) begin
      obj_type tmp;
      // if provided value is not correct type, produce error
      if (!$cast(tmp, value) )
        /* error *
      else begin
        if(clone)
          $cast(myobj, tmp.clone());
        else
          myobj = tmp;
      end
    end
    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. Set_object_local

Parameters

- **field_name** (*string*)
- **value** (*uvm_object*)
- **clone** (*bit*)
- **recurse** (*bit*)

2.3 Class uvm_pkg::uvm_transaction

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*

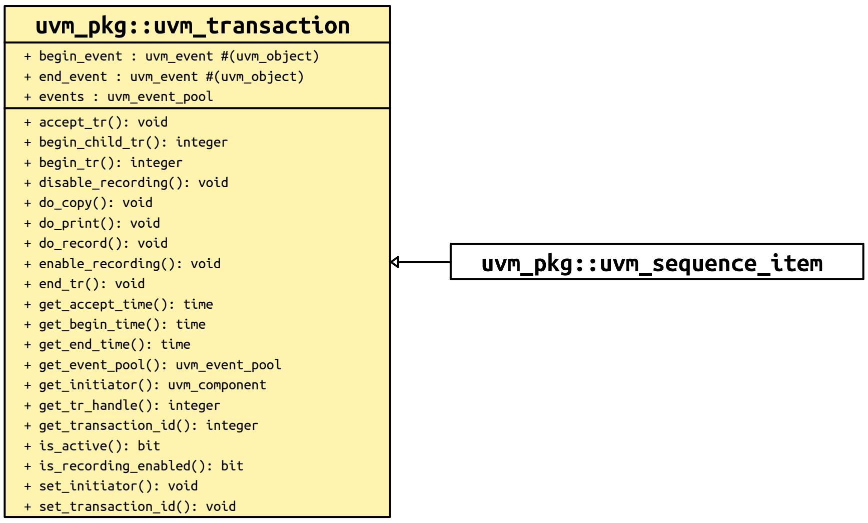

uvm_pkg::uvm_transaction
+ begin_event : uvm_event #(uvm_object)
+ end_event : uvm_event #(uvm_object)
+ events : uvm_event_pool
+ accept_tr(): void
+ begin_child_tr(): integer
+ begin_tr(): integer
+ disable_recording(): void
+ do_copy(): void
+ do_print(): void
+ do_record(): void
+ enable_recording(): void
+ end_tr(): void
+ get_accept_time(): time
+ get_begin_time(): time
+ get_end_time(): time
+ get_event_pool(): uvm_event_pool
+ get_initiator(): uvm_component
+ get_tr_handle(): integer
+ get_transaction_id(): integer
+ is_active(): bit
+ is_recording_enabled(): bit
+ set_initiator(): void
+ set_transaction_id(): void
uvm_pkg::uvm_sequence_item

Fig. 5: Inheritance Diagram of uvm_transaction

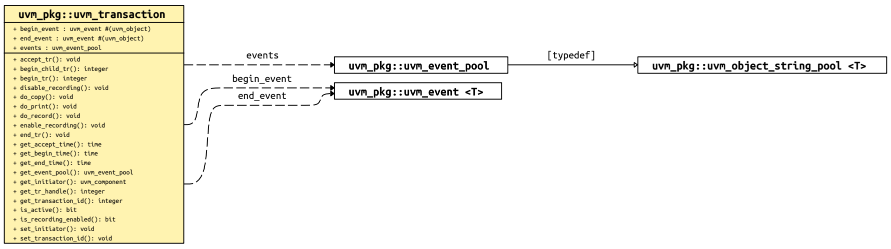


Fig. 6: Collaboration Diagram of uvm_transaction

CLASS

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.

This class provides timestamp properties, notification events, and transaction recording support.

Use of this class as a base for user-defined transactions is deprecated. Its subtype, *uvm_sequence_item*, shall be used as the base class for all user-defined transaction types.

The intended use of this API is via a <uvm_driver (REQ, RSP)> to call *uvm_component::accept_tr*, *uvm_component::begin_tr*, and *uvm_component::end_tr* during the course of sequence item execution. These methods in the component base class will call into the corresponding methods in this class to set the corresponding timestamps (*accept_time* , *begin_time* , and *end_time*), trigger the corresponding event (*begin_event* and *end_event*, and, if enabled, record the transaction contents to a vendor-specific transaction database.

Note that get_next_item/item_done when called on a uvm_seq_item_pull_port will automatically trigger the begin_event and end_events via calls to begin_tr and end_tr. While convenient, it is generally the responsibility of drivers to mark a transaction's progress during execution. To allow the driver or layering sequence to control sequence item timestamps, events, and recording, you must call <uvm_sqr_if_base(REQ, RSP)::disable_auto_item_recording> at the beginning of the driver's *run_phase* task.

Users may also use the transaction's event pool, *events*, to define custom events for the driver to trigger and the sequences to wait on. Any in-between events such as marking the beginning of the address and data phases of transaction execution could be implemented via the *events* pool.

In pipelined protocols, the driver may release a sequence (return from finish_item() or it's &96;uvm_do macro) before the item has been completed. If the driver uses the begin_tr/end_tr API in uvm_component, the sequence can wait on the item's *end_event* to block until the item was fully executed, as in the following example.

```
task uvm_execute(item, ...);
    // can use the `uvm_do macros as well
    start_item(item);
    item.randomize();
    finish_item(item);
    item.end_event.wait_on();
    // get_response(rsp, item.get_transaction_id()); //if needed
endtask
```

A simple two-stage pipeline driver that can execute address and data phases concurrently might be implemented as follows:

```
task run();
    // this driver supports a two-deep pipeline
    fork
      do_item();
      do_item();
    join
endtask


task do_item();

  forever begin
    mbus_item req;

    lock.get();

    seq_item_port.get(req); // Completes the sequencer-driver handshake

    accept_tr(req);

      // request bus, wait for grant, etc.

    begin_tr(req);

      // execute address phase

    // allows next transaction to begin address phase
    lock.put();

      // execute data phase
      // (may trigger custom "data_phase" event here)

    end_tr(req);

  end

endtask: do_item
```

Table 2: Variables

Name	Type	Description
events	*uvm_event_pool*	***Variable*** events The event pool instance for this transaction. This pool is used to track various milestones: by default, begin, accept, and end
begin_event	*uvm_event#(uvm_object)*	***Variable*** begin_event A *uvm_event(uvm_object)* that is triggered when this transaction's actual execution on the bus begins, typically as a result of a driver calling *uvm_component::begin_tr*. Processes that wait on this event will block until the transaction has begun. For more information, see the general discussion for *uvm_transaction*. See <uvm_event(T)> for details on the event API.

end_event	*uvm_event#(uvm_object)*	***Variable***
		
		end_event
		
		A *uvm_event(uvm_object)* that is triggered when this transaction's actual execution on the bus ends, typically as a result of a driver calling *uvm_component::end_tr*. Processes that wait on this event will block until the transaction has ended.
		
		For more information, see the general discussion for *uvm_transaction*. See <uvm_event(T)> for details on the event API.
		`virtual task my_sequence::body();`
		` ...`
		` start_item(item);    \`
		` item.randomize(); } ``uvm_do(item)`
		` finish_item(item);   /`
		` // return from finish item does not always mean item is completed`
		` item.end_event.wait_on();`
		` ...`

Constructors

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

Function

new

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

Parameters

name (*string*)
initiator (*uvm_component*)

Functions

`function void accept_tr(time accept_time = 0)`

Function

accept_tr

Calling *accept_tr* indicates that the transaction item has been received by a consumer component. Typically a <uvm_driver (REQ, RSP)> would call *uvm_component::accept_tr*, which calls this method-- upon return from a *get_next_item()* , *get()* , or *peek()* call on its sequencer port, <uvm_driver(REQ, RSP)::seq_item_port>.

With some protocols, the received item may not be started immediately after it is accepted. For example, a bus driver, having accepted a request transaction, may still have to wait for a bus grant before beginning to execute the request.

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. Accept_tr
Parameters
`accept_time` (`time`)

`function integer begin_tr(time begin_time = 0)`

Begin_tr
Parameters
`begin_time` (`time`)

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

Function

begin_child_tr

This function indicates that the transaction has been started as a child of a parent transaction given by *parent_handle* . Generally, a consumer component calls this method via *uvm_component::begin_child_tr* to indicate the actual start of execution of this transaction.

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 inherited *uvm_object::record* method 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. Use a parent handle of zero to link to the parent after begin
Parameters
`begin_time` (`time`)
`parent_handle` (`integer`)

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

Function

end_tr

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

You must have previously called *begin_tr* or *begin_child_tr* for this call to be successful.

Typically a <uvm_driver (REQ, RSP)> would call *uvm_component::end_tr*, which calls this method, upon completion of a sequence item transaction. Sequence items received by a driver are always a child of a parent sequence. In this case, begin_tr obtain the parent handle and delegate to *begin_child_tr*.

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. End_tr

Parameters

end_time (*time*)

free_handle (*bit*)

function integer get_tr_handle()

Function

get_tr_handle

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

function void disable_recording()

Function

disable_recording

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

function void enable_recording(uvm_tr_stream stream)

Function

enable_recording

Turns on recording to the *stream* specified.

If transaction recording is on, then a call to *record* is made when the transaction is ended. Enable_recording

Parameters

stream (*uvm_tr_stream*)

function bit is_recording_enabled()

Function

is_recording_enabled

Returns 1 if recording is currently on, 0 otherwise. Is_recording_enabled

function bit is_active()

Function

is_active

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

function uvm_event_pool get_event_pool()

Function

get_event_pool

Returns the event pool associated with this transaction.

By default, the event pool contains the events

begin, accept, and end.

Events can also be added by derivative objects. An event pool is a specialization of <uvm_pool(KEY, T)>, e.g. a *uvm_pool(uvm_event)* . Get_event_pool

Return type
uvm_event_pool

`function void set_initiator(uvm_component initiator)`

Function

set_initiator

Sets initiator as the initiator of this transaction.

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

Parameters
`initiator` (*uvm_component*)

`function uvm_component get_initiator()`

Function

get_initiator

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

Return type
uvm_component

`function time get_accept_time()`

Function

get_accept_time. Get_accept_time

`function time get_begin_time()`

Function

get_begin_time. Get_begin_time

`function time get_end_time()`

Function

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*. Get_end_time

`function void set_transaction_id(integer id)`

Function

set_transaction_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. Set_transaction_id

Parameters
`id` (*`integer`*)

`function integer get_transaction_id()`

Function

get_transaction_id

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

When using a <uvm_sequence (REQ, RSP)> to generate stimulus, the transaction ID is used along with the sequence ID to route responses in sequencers and to correlate responses to requests. Get_transaction_id

`virtual function void do_print(uvm_printer printer)`

Override data control methods for internal properties. Do_print

Parameters
`printer` (*uvm_printer*)

`virtual function void do_record(uvm_recorder recorder)`

Do_record

Parameters
`recorder` (*uvm_recorder*)

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

Parameters

- **`rhs`** (*uvm_object*)

2.4 Class uvm_pkg::uvm_root

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_root*

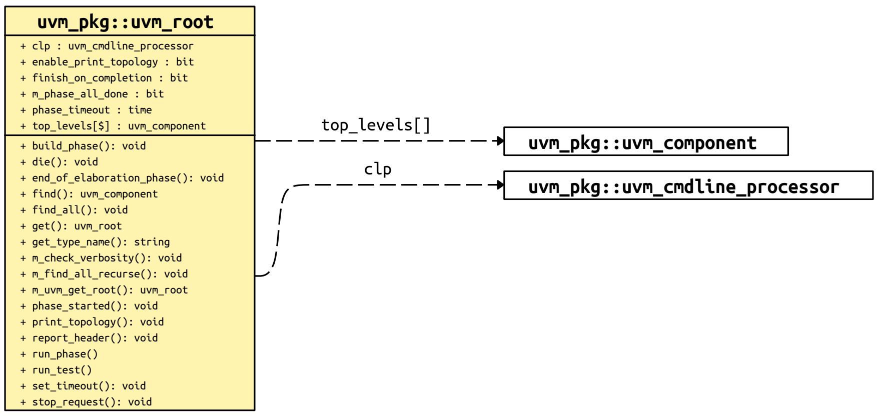


Fig. 7: Collaboration Diagram of uvm_root

Table 3: Variables

Name	Type	Description
clp	*uvm_cmdline_processor*	
finish_on_completion	bit	***Variable*** finish_on_completion If set, then run_test will call $finish after all phases are executed.
top_levels	*uvm_component*	***Variable*** top_levels This variable is a list of all of the top level components in UVM. It includes the uvm_test_top component that is created by *run_test* as well as any other top level components that have been instantiated anywhere in the hierarchy.
enable_print_topology	bit	***Variable*** enable_print_topology If set, then the entire testbench topology is printed just after completion of the end_of_elaboration phase.
phase_timeout	time	

Functions

`static function uvm_root get()`

Function

get()

Static accessor for *uvm_root*.

The static accessor is provided as a convenience wrapper around retrieving the root via the *uvm_coreservice_t::get_root* method.

```
// Using the uvm_coreservice_t:
uvm_coreservice_t cs;
uvm_root r;
cs = uvm_coreservice_t::get();
r = cs.get_root();

// Not using the uvm_coreservice_t:
uvm_root r;
r = uvm_root::get();. Get
```

Return type
uvm_root

virtual function string get_type_name()

virtual function void die()

Function

die

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

Calls the *uvm_component::pre_abort()* method on the entire *uvm_component* hierarchy in a bottom-up fashion. It then calls *uvm_report_server::report_summarize* and terminates the simulation with *$finish* .

function void set_timeout(time timeout, bit overridable = 1)

Function

set_timeout

Specifies the timeout for the simulation. Default is *`UVM_DEFAULT_TIMEOUT*

The timeout is simply the maximum absolute simulation time allowed before a *FATAL* occurs. If the timeout is set to 20ns, then the simulation must end before 20ns, or a *FATAL* timeout will occur.

This is provided so that the user can prevent the simulation from potentially consuming too many resources (Disk, Memory, CPU, etc) when the testbench is essentially hung. Set_timeout

Parameters
- **timeout** (*time*)
- **overridable** (*bit*)

function uvm_component find(string comp_match)

Function

find. Find

Parameters
- **comp_match** (*string*)

Return type
uvm_component

function void find_all(string comp_match, uvm_component comps, uvm_-component comp = null)

Function

find_all

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 the optional argument comp is provided, then search begins from that component down (default = all components). Find_all

Parameters
- **comp_match** (*string*)
- **comps** (*uvm_component*)
- **comp** (*uvm_component*)

`function void print_topology(uvm_printer printer = null)`

Function

print_topology

Print the verification environment's component topology. The *printer* is a *uvm_printer* object that controls the format of the topology printout; a *null* printer prints with the default output. Print_topology

Parameters

`printer` (*uvm_printer*)

`virtual  function void build_phase(uvm_phase phase)`

Build_phase

Parameters

`phase` (*uvm_phase*)

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

Parameters

`file` (*`UVM_FILE`*)

`virtual  function void phase_started(uvm_phase phase)`

phase_started

At end of elab phase we need to do tlm binding resolution.

Parameters

`phase` (*uvm_phase*)

`function void stop_request()`

backward compat only call global_stop_request() or uvm_test_done.stop_request() instead

`virtual  function void end_of_elaboration_phase(uvm_phase phase)`

Parameters

`phase` (*uvm_phase*)

Tasks

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

Task

run_test

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. Run_test

Parameters

`test_name` (*`string`*)

`virtual  function  run_phase(uvm_phase phase)`

For error checking. It is required that the run phase start at simulation time 0

TBD this looks wrong

taking advantage of uvm_root not doing anything else? TBD move to phase_started callback?

Parameters

`phase` (*uvm_phase*)

2.5 Class uvm_pkg::uvm_port_base

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_port_base*

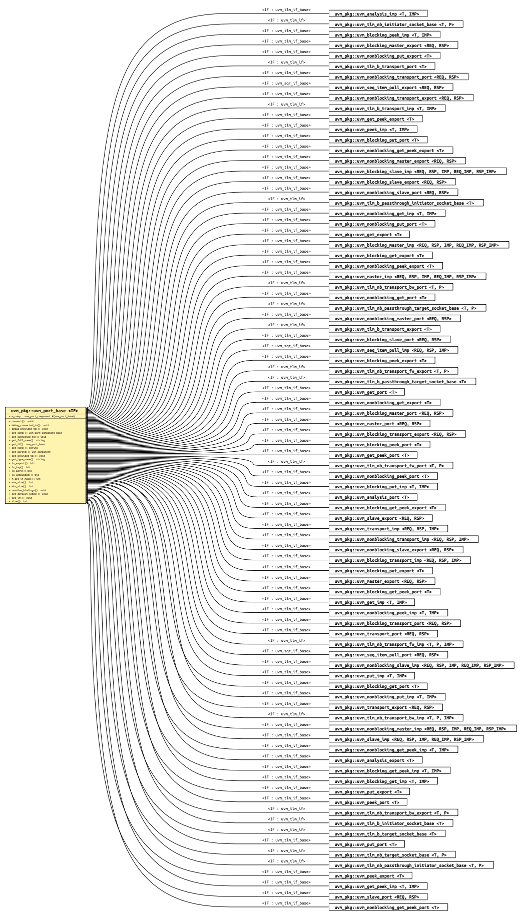


Fig. 8: Inheritance Diagram of uvm_port_base

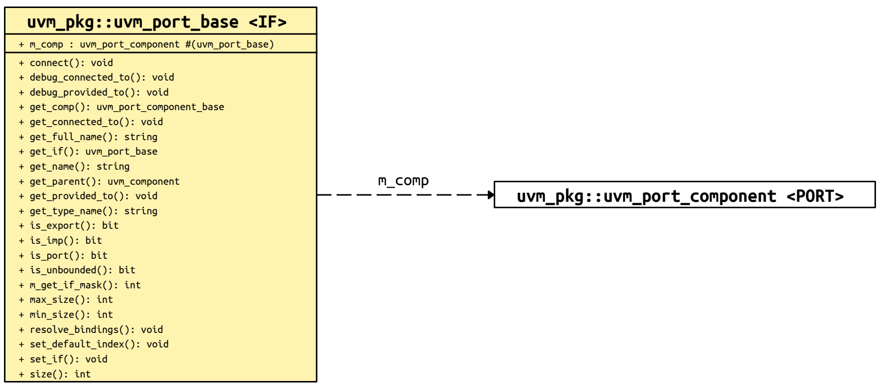

uvm_pkg::uvm_port_base <IF>
+ m_comp : uvm_port_component #(uvm_port_base)
+ connect(): void
+ debug_connected_to(): void
+ debug_provided_to(): void
+ get_comp(): uvm_port_component_base
+ get_connected_to(): void
+ get_full_name(): string
+ get_if(): uvm_port_base
+ get_name(): string
+ get_parent(): uvm_component
+ get_provided_to(): void
+ get_type_name(): string
+ is_export(): bit
+ is_imp(): bit
+ is_port(): bit
+ is_unbounded(): bit
+ m_get_if_mask(): int
+ max_size(): int
+ min_size(): int
+ resolve_bindings(): void
+ set_default_index(): void
+ set_if(): void
+ size(): int
m_comp
uvm_pkg::uvm_port_component <PORT>

Fig. 9: Collaboration Diagram of uvm_port_base

CLASS

uvm_port_base (IF)

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

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

IF

The interface type implemented by the subtype to this base port

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

Just before *uvm_component::end_of_elaboration_phase*, 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 cannot 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.

Table 4: Parameters

Name	Default value	Description
IF	uvm_void	

Table 5: Typedefs

Name	Actual Type	Description
this_type	*uvm_port_base#(IF)*	

Constructors

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

Function

new

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 *uvm_config_int::set()* . See *connect* for more information.

Parameters

name (`string`)

parent (*uvm_component*)

`port_type` (*uvm_port_type_e*)
`min_size` (*`int`*)
`max_size` (*`int`*)

Functions

`function string get_name()`

Function

get_name

Returns the leaf name of this port.

`virtual function string get_full_name()`

Function

get_full_name

Returns the full hierarchical name of this port.

`virtual function uvm_component get_parent()`

Function

get_parent

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

Return type

uvm_component

`virtual function uvm_port_component_base get_comp()`

Function

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.

Return type

uvm_port_component_base

`virtual function string get_type_name()`

Function

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.

`function int max_size()`

Function

min_size

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

`function int min_size()`

Function

max_size

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

`function bit is_unbounded()`

Function

is_unbounded

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

`function bit is_port()`

Function

is_port

`function bit is_export()`

Function

is_export

`function bit is_imp()`

Function

is_imp

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

`function int size()`

Function

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.

`function void set_if(int index = 0)`

Parameters

index (*int*)

`function void set_default_index(int index)`

Function

set_default_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.

Parameters

index (*int*)

`virtual  function void connect(this_type provider)`

Function

connect

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, a uvm_blocking_put_port (T) is compatible with a 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> cannot 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 a UVM_PORT type, the *provider* can be a parent port,

or a sibling export or implementation port.

- If this port is a <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_phase* 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.

Parameters
 `provider` (*this_type*)

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

Function

debug_connected_to

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.

Parameters
 `level` (*`int`*)
 `max_level` (*`int`*)

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

Function

debug_provided_to

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.

Parameters
 `level` (*`int`*)
 `max_level` (*`int`*)

`function void get_connected_to(uvm_port_list list)`

get_connected_to

Parameters
 `list` (*uvm_port_list*)

`function void get_provided_to(uvm_port_list list)`

get_provided_to

Parameters
 `list` (*uvm_port_list*)

`virtual  function void resolve_bindings()`

Function

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.

`function uvm_port_base get_if(int index = 0)`

Function

get_if

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.

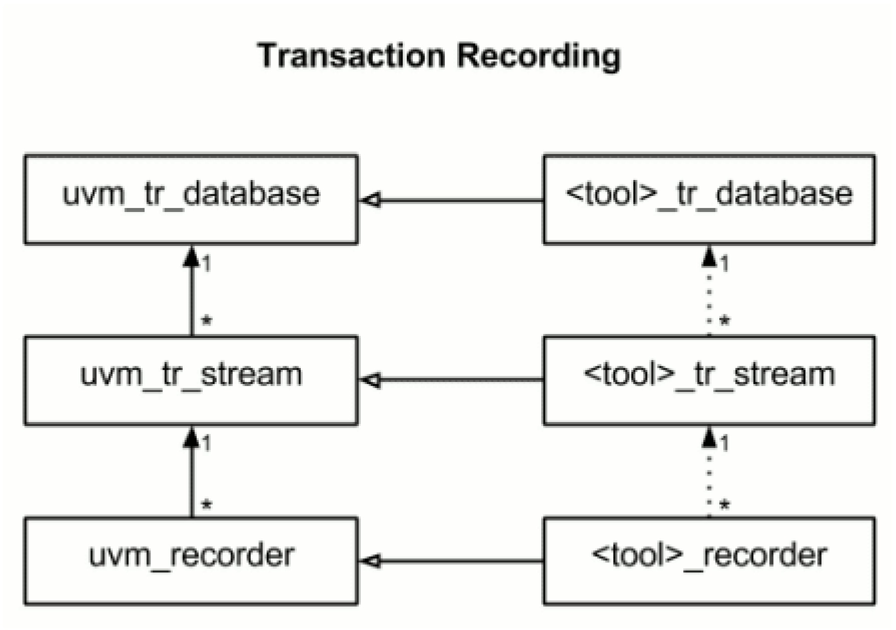
Parameters
 `index` (*`int`*)

Return type
 uvm_port_base

CHAPTER
THREE

REPORTING CLASSES

The reporting classes provide a facility for issuing reports with consistent formatting. Users can configure what actions to take and what files to send output to based on report severity, ID, or both severity and ID. Users can also filter messages based on their verbosity settings.
The primary interface to the UVM reporting facility is the *uvm_report_object* from which all *uvm_components* extend. The uvm_report_object delegates most tasks to its internal *uvm_report_handler* . If the report handler determines the report is not filtered based the configured verbosity setting, it sends the report to the central *uvm_report_server* for formatting and processing.



3.1 Class uvm_pkg::uvm_report_message

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_message*

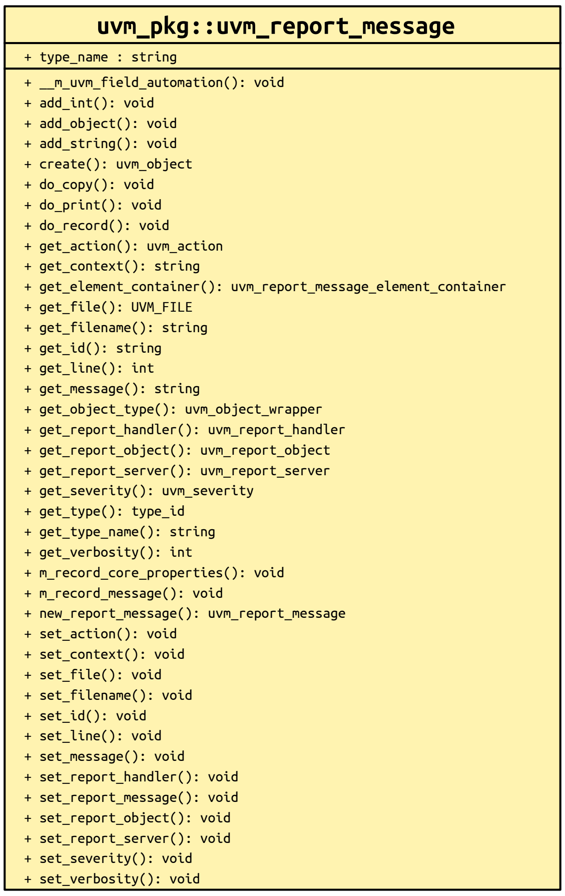

uvm_pkg::uvm_report_message
+ type_name : string
+ __m_uvm_field_automation(): void
+ add_int(): void
+ add_object(): void
+ add_string(): void
+ create(): uvm_object
+ do_copy(): void
+ do_print(): void
+ do_record(): void
+ get_action(): uvm_action
+ get_context(): string
+ get_element_container(): uvm_report_message_element_container
+ get_file(): UVM_FILE
+ get_filename(): string
+ get_id(): string
+ get_line(): int
+ get_message(): string
+ get_object_type(): uvm_object_wrapper
+ get_report_handler(): uvm_report_handler
+ get_report_object(): uvm_report_object
+ get_report_server(): uvm_report_server
+ get_severity(): uvm_severity
+ get_type(): type_id
+ get_type_name(): string
+ get_verbosity(): int
+ m_record_core_properties(): void
+ m_record_message(): void
+ new_report_message(): uvm_report_message
+ set_action(): void
+ set_context(): void
+ set_file(): void
+ set_filename(): void
+ set_id(): void
+ set_line(): void
+ set_message(): void
+ set_report_handler(): void
+ set_report_message(): void
+ set_report_object(): void
+ set_report_server(): void
+ set_severity(): void
+ set_verbosity(): void

Fig. 1: Collaboration Diagram of uvm_report_message

CLASS

uvm_report_message

The uvm_report_message is the basic UVM object message class. It provides the fields that are common to all messages. It also has a message element container and provides the APIs necessary to add integral types, strings and uvm_objects to the container. The report message object can be initialized with the common fields, and passes through the whole reporting system (i.e. report object, report handler, report server, report catcher, etc) as an object. The additional elements can be added/deleted to/from the message object anywhere in the reporting system, and can be printed or recorded along with the common fields.

Constructors

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

Function

new

Creates a new uvm_report_message object.

Parameters

`name` (*`string`*)

Functions

static function uvm_report_message new_report_message(string name = "uvm_report_-message")

Function

new_report_message

Creates a new uvm_report_message object. This function is the same as new(), but keeps the random stability.

Parameters
: **name** (*string*)

Return type
: *uvm_report_message*

virtual function void do_print(uvm_printer printer)

Parameters
: **printer** (*uvm_printer*)

virtual function void do_copy(uvm_object rhs)

Not documented.

Parameters
: **rhs** (*uvm_object*)

virtual function uvm_report_object get_report_object()

Function

get_report_object

Return type
: *uvm_report_object*

virtual function void set_report_object(uvm_report_object ro)

Function

set_report_object

Get or set the uvm_report_object that originated the message.

Parameters
: **ro** (*uvm_report_object*)

virtual function uvm_report_handler get_report_handler()

Function

get_report_handler

Return type
: *uvm_report_handler*

virtual function void set_report_handler(uvm_report_handler rh)

Function

set_report_handler

Get or set the uvm_report_handler that is responsible for checking whether the message is enabled, should be upgraded/downgraded, etc.

Parameters
: **rh** (*uvm_report_handler*)

virtual function uvm_report_server get_report_server()

Function

get_report_server

Return type
: *uvm_report_server*

virtual function void set_report_server(uvm_report_server rs)

Function

set_report_server

Get or set the uvm_report_server that is responsible for servicing the message's actions.

Parameters
: **rs** (*uvm_report_server*)

virtual function uvm_severity get_severity()

Function

get_severity

Return type

uvm_severity

virtual function void set_severity(uvm_severity sev)

Function

set_severity

Get or set the severity (UVM_INFO, UVM_WARNING, UVM_ERROR or UVM_FATAL) of the message. The value of this field is determined via the API used (&96;uvm_info(), &96;uvm_waring(), etc.) and populated for the user.

Parameters

sev (*uvm_severity*)

virtual function string get_id()

Function

get_id

virtual function void set_id(string id)

Function

set_id

Get or set the id of the message. The value of this field is completely under user discretion. Users are recommended to follow a consistent convention. Settings in the uvm_report_handler allow various messaging controls based on this field. See *uvm_report_handler*.

Parameters

id (*string*)

virtual function string get_message()

Function

get_message

virtual function void set_message(string msg)

Function

set_message

Get or set the user message content string.

Parameters

msg (*string*)

virtual function int get_verbosity()

Function

get_verbosity

virtual function void set_verbosity(int ver)

Function

set_verbosity

Get or set the message threshold value. This value is compared against settings in the *uvm_report_handler* to determine whether this message should be executed.

Parameters

ver (*int*)

virtual function string get_filename()

Function

get_filename

virtual function void set_filename(string fname)

Function

set_filename

Get or set the file from which the message originates. This value is automatically populated by the messaging macros.

Parameters
fname (*string*)

virtual function int get_line()

Function

get_line

virtual function void set_line(int ln)

Function

set_line

Get or set the line in the *file* from which the message originates. This value is automatically populate by the messaging macros.

Parameters
ln (*int*)

virtual function string get_context()

Function

get_context

virtual function void set_context(string cn)

Function

set_context

Get or set the optional user-supplied string that is meant to convey the context of the message. It can be useful in scopes that are not inherently UVM like modules, interfaces, etc.

Parameters
cn (*string*)

virtual function uvm_action get_action()

Function

get_action

Return type
uvm_action

virtual function void set_action(uvm_action act)

Function

set_action

Get or set the action(s) that the uvm_report_server should perform for this message. This field is populated by the uvm_report_handler during message execution flow.

Parameters
act (*uvm_action*)

virtual function UVM_FILE get_file()

Function

get_file

virtual function void set_file(UVM_FILE fl)

Function

set_file

Get or set the file that the message is to be written to when the message's action is UVM_LOG. This field is populated by the uvm_report_handler during message execution flow.

Parameters
fl (*UVM_FILE*)

virtual function uvm_report_message_element_container get_element_container()

Function

get_element_container

Get the element_container of the message

Return type
uvm_report_message_element_container

`virtual function void set_report_message(uvm_severity severity, string id, string message, int verbosity, string filename, int line, string context_name)`

Function

set_report_message

Set all the common fields of the report message in one shot.

Parameters

- **`severity`** (*uvm_severity*)
- **`id`** (*`string`*)
- **`message`** (*`string`*)
- **`verbosity`** (*`int`*)
- **`filename`** (*`string`*)
- **`line`** (*`int`*)
- **`context_name`** (*`string`*)

`virtual function void do_record(uvm_recorder recorder)`

Not documented.

Parameters

- **`recorder`** (*uvm_recorder*)

`virtual function void add_int(string name, uvm_bitstream_t value, int size, uvm_-radix_enum radix, uvm_action action = (UVM_LOG|UVM_RM_RECORD))`

Function

add_int

This method adds an integral type of the name *name* and value *value* to the message. The required *size* field indicates the size of *value* . The required *radix* field determines how to display and record the field. The optional print/record bit is to specify whether the element will be printed/recorded.

Parameters

- **`name`** (*`string`*)
- **`value`** (*uvm_bitstream_t*)
- **`size`** (*`int`*)
- **`radix`** (*uvm_radix_enum*)
- **`action`** (*uvm_action*)

`virtual function void add_string(string name, string value, uvm_-action action = (UVM_LOG|UVM_RM_RECORD))`

Function

add_string

This method adds a string of the name *name* and value *value* to the message. The optional print/record bit is to specify whether the element will be printed/recorded.

Parameters

- **`name`** (*`string`*)
- **`value`** (*`string`*)
- **`action`** (*uvm_action*)

`virtual function void add_object(string name, uvm_object obj, uvm_-action action = (UVM_LOG|UVM_RM_RECORD))`

Function

add_object

This method adds a uvm_object of the name *name* and reference *obj* to the message. The optional print/record bit is to specify whether the element will be printed/recorded.

Parameters

- **`name`** (*`string`*)
- **`obj`** (*uvm_object*)
- **`action`** (*uvm_action*)

3.2 Class uvm_pkg::uvm_report_object

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*

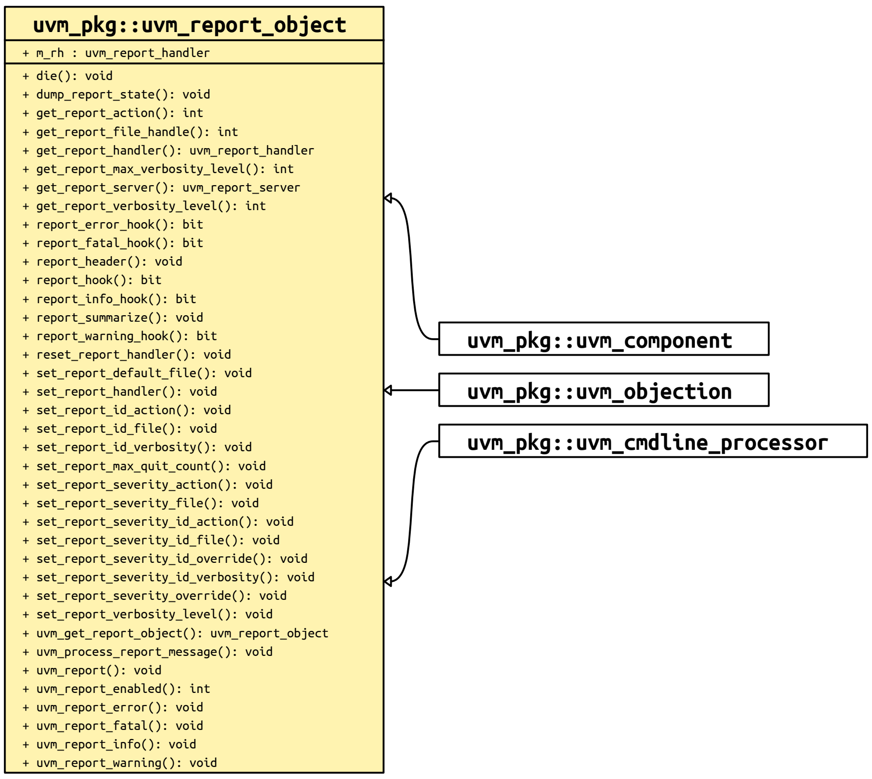

uvm_pkg::uvm_report_object
+ m_rh : uvm_report_handler
+ die(): void
+ dump_report_state(): void
+ get_report_action(): int
+ get_report_file_handle(): int
+ get_report_handler(): uvm_report_handler
+ get_report_max_verbosity_level(): int
+ get_report_server(): uvm_report_server
+ get_report_verbosity_level(): int
+ report_error_hook(): bit
+ report_fatal_hook(): bit
+ report_header(): void
+ report_hook(): bit
+ report_info_hook(): bit
+ report_summarize(): void
+ report_warning_hook(): bit
+ reset_report_handler(): void
+ set_report_default_file(): void
+ set_report_handler(): void
+ set_report_id_action(): void
+ set_report_id_file(): void
+ set_report_id_verbosity(): void
+ set_report_max_quit_count(): void
+ set_report_severity_action(): void
+ set_report_severity_file(): void
+ set_report_severity_id_action(): void
+ set_report_severity_id_file(): void
+ set_report_severity_id_override(): void
+ set_report_severity_id_verbosity(): void
+ set_report_severity_override(): void
+ set_report_verbosity_level(): void
+ uvm_get_report_object(): uvm_report_object
+ uvm_process_report_message(): void
+ uvm_report(): void
+ uvm_report_enabled(): int
+ uvm_report_error(): void
+ uvm_report_fatal(): void
+ uvm_report_info(): void
+ uvm_report_warning(): void
uvm_pkg::uvm_component
uvm_pkg::uvm_objection
uvm_pkg::uvm_cmdline_processor

Fig. 2: Inheritance Diagram of uvm_report_object

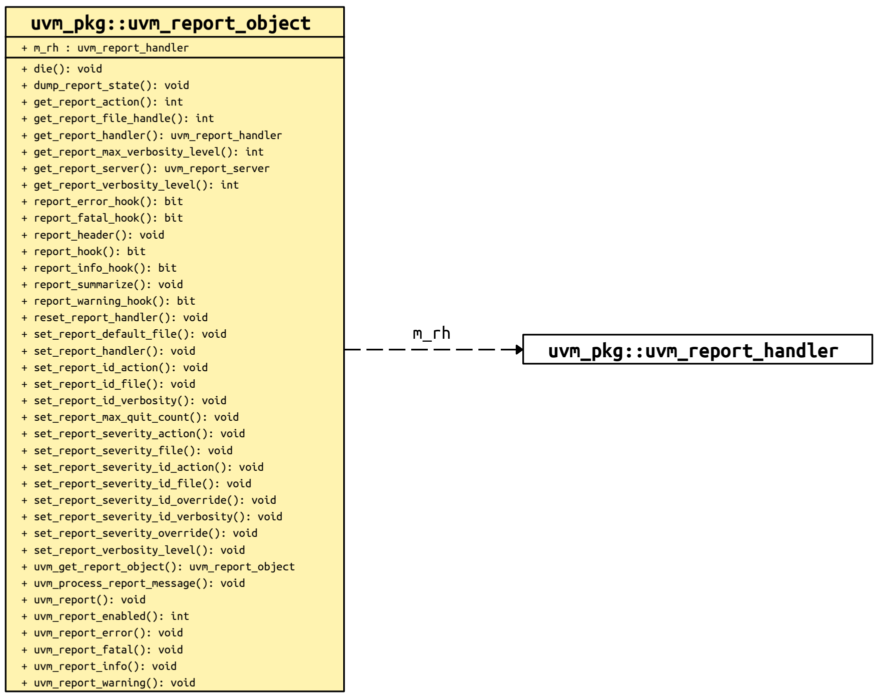

uvm_pkg::uvm_report_object
+ m_rh : uvm_report_handler
+ die(): void
+ dump_report_state(): void
+ get_report_action(): int
+ get_report_file_handle(): int
+ get_report_handler(): uvm_report_handler
+ get_report_max_verbosity_level(): int
+ get_report_server(): uvm_report_server
+ get_report_verbosity_level(): int
+ report_error_hook(): bit
+ report_fatal_hook(): bit
+ report_header(): void
+ report_hook(): bit
+ report_info_hook(): bit
+ report_summarize(): void
+ report_warning_hook(): bit
+ reset_report_handler(): void
+ set_report_default_file(): void
+ set_report_handler(): void
+ set_report_id_action(): void
+ set_report_id_file(): void
+ set_report_id_verbosity(): void
+ set_report_max_quit_count(): void
+ set_report_severity_action(): void
+ set_report_severity_file(): void
+ set_report_severity_id_action(): void
+ set_report_severity_id_file(): void
+ set_report_severity_id_override(): void
+ set_report_severity_id_verbosity(): void
+ set_report_severity_override(): void
+ set_report_verbosity_level(): void
+ uvm_get_report_object(): uvm_report_object
+ uvm_process_report_message(): void
+ uvm_report(): void
+ uvm_report_enabled(): int
+ uvm_report_error(): void
+ uvm_report_fatal(): void
+ uvm_report_info(): void
+ uvm_report_warning(): void
m_rh
uvm_pkg::uvm_report_handler

Fig. 3: Collaboration Diagram of uvm_report_object

CLASS

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 a *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 SystemVerilog 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 a UVM_LOG attribute is set in the action associated with the report. This can be overridden by any of the *set*__file* methods.

Constructors

`function  new(string name = "")`

Function

new

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.

Parameters

`name` (*`string`*)

Functions

`function uvm_report_object uvm_get_report_object()`

Function

uvm_get_report_object

Returns the nearest uvm_report_object when called. From inside a uvm_component, the method simply returns *this* .

See also the global version of *uvm_get_report_object*.

Return type

uvm_report_object

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

Function

uvm_report_enabled

Returns 1 if the configured verbosity for this severity/id is greater than or equal to *verbosity* else returns 0.

See also *get_report_verbosity_level* and the global version of *uvm_report_enabled*.

Parameters

`verbosity` (*`int`*)
`severity` (*uvm_severity*)
`id` (*`string`*)

`virtual  function void uvm_report(uvm_severity severity, string id, string message, int verbosity = (severity==uvm_severity'(UVM_ERROR))?UVM_LOW:(severity==uvm_-severity'(UVM_FATAL))?UVM_NONE:UVM_MEDIUM, string filename = "", int line = 0, string context_name = "", bit report_enabled_checked = 0)`

Function

uvm_report

Parameters

`severity` (*uvm_severity*)
`id` (*`string`*)
`message` (*`string`*)
`verbosity` (*`int`*)
`filename` (*`string`*)

line (*int*)
context_name (*string*)
report_enabled_checked (*bit*)

virtual function void uvm_report_info(string id, string message, int verbosity = UVM_MEDIUM, string filename = "", int line = 0, string context_- name = "", bit report_enabled_checked = 0)

Function

uvm_report_info

Parameters

id (*string*)
message (*string*)
verbosity (*int*)
filename (*string*)
line (*int*)
context_name (*string*)
report_enabled_checked (*bit*)

virtual function void uvm_report_warning(string id, string message, int verbosity = UVM_MEDIUM, string filename = "", int line = 0, string context_- name = "", bit report_enabled_checked = 0)

Function

uvm_report_warning

Parameters

id (*string*)
message (*string*)
verbosity (*int*)
filename (*string*)
line (*int*)
context_name (*string*)
report_enabled_checked (*bit*)

virtual function void uvm_report_error(string id, string message, int verbosity = UVM_LOW, string filename = "", int line = 0, string context_- name = "", bit report_enabled_checked = 0)

Function

uvm_report_error

Parameters

id (*string*)
message (*string*)
verbosity (*int*)
filename (*string*)
line (*int*)
context_name (*string*)
report_enabled_checked (*bit*)

virtual function void uvm_report_fatal(string id, string message, int verbosity = UVM_NONE, string filename = "", int line = 0, string context_- name = "", bit report_enabled_checked = 0)

Function

uvm_report_fatal

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:

id

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

message

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

verbosity

the verbosity of the message, indicating its relative importance. If this number is less than or equal to the effective verbosity level, see *set_report_verbosity_level*, then the report is issued, subject to the configured action and file descriptor settings. Verbosity is ignored for warnings, errors, and fatals. However, if a warning, error or fatal is demoted to an info message using the *uvm_report_catcher*, then the verbosity is taken into account.

filename/line

(Optional) The location from which the report was issued. Use the predefined macros, &96;*FILE* and &96;*LINE*. If specified, it is displayed in the output.

context_name

(Optional) The string context from where the message is originating. This can be the %m of a module, a specific method, etc.

report_enabled_checked

(Optional) This bit indicates whether the currently provided message has been checked as to whether the message should be processed. If it hasn't been checked, it will be checked inside the uvm_report function.

Parameters

- **`id`** (*`string`*)
- **`message`** (*`string`*)
- **`verbosity`** (*`int`*)
- **`filename`** (*`string`*)
- **`line`** (*`int`*)
- **`context_name`** (*`string`*)
- **`report_enabled_checked`** (*`bit`*)

```
virtual  function void uvm_process_report_message(uvm_report_message report_message)
```

Function

uvm_process_report_message

This method takes a preformed uvm_report_message, populates it with the report object and passes it to the report handler for processing. It is expected to be checked for verbosity and populated.

Parameters

- **`report_message`** (*uvm_report_message*)

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

Function

get_report_verbosity_level

Gets the verbosity level in effect for this object. Reports issued with verbosity greater than this will be filtered out. The severity and tag arguments check if the verbosity level has been modified for specific severity/tag combinations.

Parameters

- **`severity`** (*uvm_severity*)
- **`id`** (*`string`*)

```
function int get_report_max_verbosity_level()
```

Function

get_report_max_verbosity_level

Gets the maximum verbosity level in effect for this report object. Any report from this component whose verbosity exceeds this maximum will be ignored.

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

Function

set_report_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.

Parameters
`verbosity_level` (*`int`*)

`function void set_report_id_verbosity(string id, int verbosity)`

Function

set_report_id_verbosity

Parameters
`id` (*`string`*)
`verbosity` (*`int`*)

`function void set_report_severity_id_verbosity(uvm_severity severity, string id, int verbosity)`

Function

set_report_severity_id_verbosity

These methods associate the specified verbosity threshold with reports of the given *severity* , *id* , or *severity-id* pair. This threshold is compared with the verbosity originally assigned to the report to decide whether it gets processed. A verbosity threshold associated with a particular *severity-id* pair takes precedence over a verbosity threshold associated with *id* , which takes precedence over a verbosity threshold associated with a *severity* .

The *verbosity* argument can be any integer, but is most commonly a predefined *uvm_verbosity* value, <UVM_NONE>, <UVM_LOW>, <UVM_MEDIUM>, <UVM_HIGH>, <UVM_FULL>.

Parameters
`severity` (*uvm_severity*)
`id` (*`string`*)
`verbosity` (*`int`*)

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

Function

get_report_action

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

Parameters
`severity` (*uvm_severity*)
`id` (*`string`*)

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

Function

set_report_severity_action

Parameters
`severity` (*uvm_severity*)
`action` (*uvm_action*)

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

Function

set_report_id_action

Parameters
`id` (*`string`*)
`action` (*uvm_action*)

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

Function

set_report_severity_id_action

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

Parameters
- **severity** (*uvm_severity*)
- **id** (*string*)
- **action** (*uvm_action*)

function int get_report_file_handle(uvm_severity severity, string id)

Function

get_report_file_handle

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

Parameters
- **severity** (*uvm_severity*)
- **id** (*string*)

function void set_report_default_file(UVM_FILE file)

Function

set_report_default_file

Parameters
- **file** (*UVM_FILE*)

function void set_report_id_file(string id, UVM_FILE file)

Function

set_report_id_file

Parameters
- **id** (*string*)
- **file** (*UVM_FILE*)

function void set_report_severity_file(uvm_severity severity, UVM_FILE file)

Function

set_report_severity_file

Parameters
- **severity** (*uvm_severity*)
- **file** (*UVM_FILE*)

function void set_report_severity_id_file(uvm_severity severity, string id, UVM_-FILE file)

Function

set_report_severity_id_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 a FILE associated with a *severity* , which takes precedence over the default FILE descriptor.

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

Parameters
- **severity** (*uvm_severity*)
- **id** (*string*)
- **file** (*UVM_FILE*)

function void set_report_severity_override(uvm_severity cur_severity, uvm_-severity new_severity)

Function

set_report_severity_override

Parameters
- **cur_severity** (*uvm_severity*)
- **new_severity** (*uvm_severity*)

function void set_report_severity_id_override(uvm_severity cur_severity, string id, uvm_severity new_severity)

Function

set_report_severity_id_override

These methods provide the ability to upgrade or downgrade a message in terms of severity given *severity* and *id* . An upgrade or downgrade for a specific *id* takes precedence over an upgrade or downgrade associated with a *severity* .

Parameters

- **cur_severity** (*uvm_severity*)
- **id** (*string*)
- **new_severity** (*uvm_severity*)

function void set_report_handler(uvm_report_handler handler)

Function

set_report_handler

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

Parameters

handler (*uvm_report_handler*)

function uvm_report_handler get_report_handler()

Function

get_report_handler

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

Return type

uvm_report_handler

function void reset_report_handler()

Function

reset_report_handler

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

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

Function- report_info_hook

Parameters

- **id** (*string*)
- **message** (*string*)
- **verbosity** (*int*)
- **filename** (*string*)
- **line** (*int*)

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

Function- report_error_hook

Parameters

- **id** (*string*)
- **message** (*string*)
- **verbosity** (*int*)
- **filename** (*string*)
- **line** (*int*)

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

Function- report_warning_hook

Parameters

- **id** (*string*)

message (*string*)
verbosity (*int*)
filename (*string*)
line (*int*)

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

Function- report_fatal_hook

Parameters

id (*string*)
message (*string*)
verbosity (*int*)
filename (*string*)
line (*int*)

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

Function- report_hook

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 *report_hook* method is called, followed by the severity-specific hook (*report_info_hook*, etc.). If either hook method returns 0 then the report is not processed further.

Parameters

id (*string*)
message (*string*)
verbosity (*int*)
filename (*string*)
line (*int*)

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

Function- report_header

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

Use *uvm_root::report_header()*

Parameters

file (*UVM_FILE*)

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

Function- report_summarize

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.

Use uvm_report_server rs =uvm_report_server::get_server(); rs.report_summarize();

Parameters

file (*UVM_FILE*)

`virtual function void die()`

Function- die

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

Calls the *uvm_component::pre_abort()* method on the entire *uvm_component* hierarchy in a bottom-up fashion. It then call calls *report_summarize* and terminates the simulation with *$finish* .

Use uvm_report_server rs =uvm_report_server::get_server(); rs.die()

`function void set_report_max_quit_count(int max_count)`

Function- set_report_max_quit_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.

Use uvm_report_server rs =uvm_report_server::get_server(); rs.set_max_quit_count()

Parameters

`max_count` (*`int`*)

`function uvm_report_server get_report_server()`

Function- get_report_server

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

Use *uvm_report_server::get_server()*

Return type

uvm_report_server

`function void dump_report_state()`

Function- 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.

Use uvm_report_handler rh =get_report_handler(); rh.print().

3.3 Class uvm_pkg::uvm_report_handler

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_handler*

CLASS

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.

Table 1: Variables

Name	Type	Description
id_verbosities	*uvm_id_verbosities_array*	***id verbosity settings*** default and severity
severity_id_verbosities	*uvm_id_verbosities_array*	
id_actions	*uvm_id_actions_array*	actions
severity_actions	*uvm_action*	
severity_id_actions	*uvm_id_actions_array*	
sev_overrides	*uvm_sev_override_array*	severity overrides
sev_id_overrides	*uvm_sev_override_array*	
default_file_handle	UVM_FILE	***file handles*** default, severity, action, (severity, id)
id_file_handles	*uvm_id_file_array*	
severity_file_handles	UVM_FILE	
severity_id_file_handles	*uvm_id_file_array*	

Constructors

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

Function

new

Creates and initializes a new uvm_report_handler object.

Parameters
name (*string*)

Functions

virtual function void do_print(uvm_printer printer)

Function

print

The uvm_report_handler implements the *uvm_object::do_print()* such that *print* method provides UVM printer formatted output of the current configuration. A snippet of example output is shown here:

```
uvm_test_top                 uvm_report_handler  -     @555
  max_verbosity_level        uvm_verbosity       32    UVM_FULL
  id_verbosities             uvm_pool            3     -
    [ID1]                    uvm_verbosity       32    UVM_LOW
  severity_id_verbosities    array               4     -
    [UVM_INFO:ID4]           int                 32    501
  id_actions                 uvm_pool            2     -
    [ACT_ID]                 uvm_action          32    DISPLAY LOG COUNT
  severity_actions           array               4     -
    [UVM_INFO]               uvm_action          32    DISPLAY
    [UVM_WARNING]            uvm_action          32    DISPLAY RM_RECORD COUNT
    [UVM_ERROR]              uvm_action          32    DISPLAY COUNT
    [UVM_FATAL]              uvm_action          32    DISPLAY EXIT
  default_file_handle        int                 32    'h1
```

Parameters
printer (*uvm_printer*)

virtual function void process_report_message(uvm_report_message report_message)

Function

process_report_message

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

Parameters
report_message (*uvm_report_message*)

static function string format_action(uvm_action action)

Function

format_action

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

Parameters
action (*uvm_action*)

function void initialize()

Function- initialize

Internal method for initializing report handler.

function void set_verbosity_level(int verbosity_level)

Function- set_verbosity_level

Internal method called by uvm_report_object.

Parameters
verbosity_level (*int*)

function int get_verbosity_level(uvm_severity severity = UVM_INFO, string id = "")

Function- get_verbosity_level

Returns the verbosity associated with the given *severity* and *id* .

First, if there is a verbosity associated with the *(severity, id)* pair, return that. Else, if there is a verbosity associated with the *id* , return that. Else, return the max verbosity setting.

Parameters
- **severity** (*uvm_severity*)
- **id** (*string*)

function uvm_action get_action(uvm_severity severity, string id)

Function- get_action

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* .

Parameters
- **severity** (*uvm_severity*)
- **id** (*string*)

Return type
uvm_action

function UVM_FILE get_file_handle(uvm_severity severity, string id)

Function- get_file_handle

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.

Parameters
- **severity** (*uvm_severity*)
- **id** (*string*)

function void set_severity_action(uvm_severity severity, uvm_action action)

Function- set_severity_action Function- set_id_action Function- set_severity_id_action Function- set_id_verbosity Function- set_severity_id_verbosity

Internal methods called by uvm_report_object.

Parameters
- **severity** (*uvm_severity*)
- **action** (*uvm_action*)

function void set_id_action(string id, uvm_action action)

Parameters
- **id** (*string*)
- **action** (*uvm_action*)

function void set_severity_id_action(uvm_severity severity, string id, uvm_-action action)

Parameters
- **severity** (*uvm_severity*)
- **id** (*string*)
- **action** (*uvm_action*)

function void set_id_verbosity(string id, int verbosity)

Parameters
- **id** (*string*)
- **verbosity** (*int*)

function void set_severity_id_verbosity(uvm_severity severity, string id, int verbosity)

Parameters
- **severity** (*uvm_severity*)
- **id** (*string*)
- **verbosity** (*int*)

function void set_default_file(UVM_FILE file)

Function- set_default_file Function- set_severity_file Function- set_id_file Function- set_severity_id_file

Internal methods called by uvm_report_object.

Parameters
`file` (*`UVM_FILE`*)

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

Parameters
`severity` (*uvm_severity*)
`file` (*`UVM_FILE`*)

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

Parameters
`id` (*`string`*)
`file` (*`UVM_FILE`*)

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

Parameters
`severity` (*uvm_severity*)
`id` (*`string`*)
`file` (*`UVM_FILE`*)

`function void set_severity_override(uvm_severity cur_severity, uvm_severity new_-severity)`

Parameters
`cur_severity` (*uvm_severity*)
`new_severity` (*uvm_severity*)

`function void set_severity_id_override(uvm_severity cur_severity, string id, uvm_-severity new_severity)`

Parameters
`cur_severity` (*uvm_severity*)
`id` (*`string`*)
`new_severity` (*uvm_severity*)

`virtual  function void report(uvm_severity severity, string name, string id, string message, int verbosity_level = UVM_MEDIUM, string filename = "", int line = 0, uvm_report_object client = null)`

Function- report

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

Parameters
`severity` (*uvm_severity*)
`name` (*`string`*)
`id` (*`string`*)
`message` (*`string`*)
`verbosity_level` (*`int`*)
`filename` (*`string`*)
`line` (*`int`*)
`client` (*uvm_report_object*)

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

Function- run_hooks

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

Parameters
`client` (*uvm_report_object*)
`severity` (*uvm_severity*)
`id` (*`string`*)
`message` (*`string`*)
`verbosity` (*`int`*)
`filename` (*`string`*)
`line` (*`int`*)

`function void dump_state()`

Function- dump_state

Internal method for debug.

3.4 Class uvm_pkg::uvm_report_server

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_server*

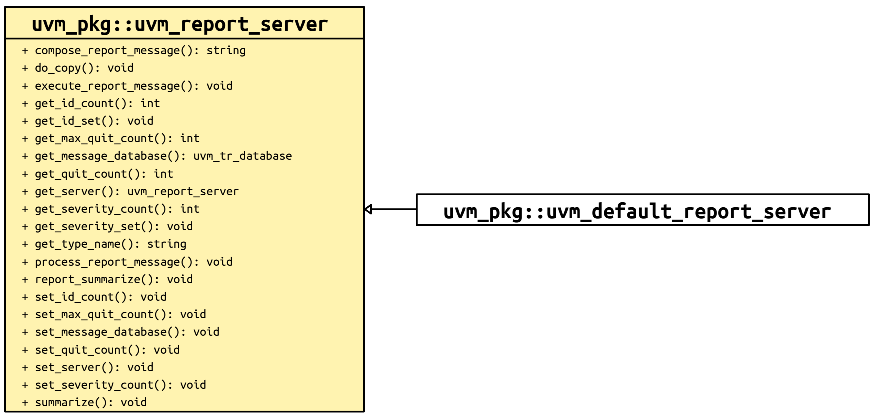

uvm_pkg::uvm_report_server
+ compose_report_message(): string
+ do_copy(): void
+ execute_report_message(): void
+ get_id_count(): int
+ get_id_set(): void
+ get_max_quit_count(): int
+ get_message_database(): uvm_tr_database
+ get_quit_count(): int
+ get_server(): uvm_report_server
+ get_severity_count(): int
+ get_severity_set(): void
+ get_type_name(): string
+ process_report_message(): void
+ report_summarize(): void
+ set_id_count(): void
+ set_max_quit_count(): void
+ set_message_database(): void
+ set_quit_count(): void
+ set_server(): void
+ set_severity_count(): void
+ summarize(): void
uvm_pkg::uvm_default_report_server

Fig. 4: Inheritance Diagram of uvm_report_server

Constructors

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

Parameters
name (*string*)

Functions

`virtual function string get_type_name()`

`virtual function void set_max_quit_count(int count, bit overridable = 1)`

Function

set_max_quit_count

count is the maximum number of *UVM_QUIT* actions the uvm_report_server will tolerate before invoking client.die(). when *overridable* = 0 is passed, the set quit count cannot be changed again

Parameters
count (*int*)
overridable (*bit*)

`virtual function int get_max_quit_count()`

Function

get_max_quit_count

returns the currently configured max quit count

`virtual function void set_quit_count(int quit_count)`

Function

set_quit_count

sets the current number of *UVM_QUIT* actions already passed through this uvm_report_server

Parameters
quit_count (*int*)

`virtual function int get_quit_count()`

Function

get_quit_count

returns the current number of *UVM_QUIT* actions already passed through this server

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

Function

set_severity_count

sets the count of already passed messages with severity *severity* to *count*

Parameters
- **`severity`** (*uvm_severity*)
- **`count`** (*`int`*)

`virtual function int get_severity_count(uvm_severity severity)`

Function

get_severity_count

returns the count of already passed messages with severity *severity*

Parameters
- **`severity`** (*uvm_severity*)

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

Function

set_id_count

sets the count of already passed messages with *id* to *count*

Parameters
- **`id`** (*`string`*)
- **`count`** (*`int`*)

`virtual function int get_id_count(string id)`

Function

get_id_count

returns the count of already passed messages with *id*

Parameters
- **`id`** (*`string`*)

`virtual function void get_id_set(string q)`

Function

get_id_set

returns the set of id's already used by this uvm_report_server

Parameters
- **`q`** (*`string`*)

`virtual function void get_severity_set(uvm_severity q)`

Function

get_severity_set

returns the set of severities already used by this uvm_report_server

Parameters
- **`q`** (*uvm_severity*)

`virtual function void set_message_database(uvm_tr_database database)`

Function

set_message_database

sets the *uvm_tr_database* used for recording messages

Parameters
- **`database`** (*uvm_tr_database*)

`virtual function uvm_tr_database get_message_database()`

Function

get_message_database

returns the *uvm_tr_database* used for recording messages

Return type
- *uvm_tr_database*

virtual function void do_copy(uvm_object rhs)

Function

do_copy

copies all message statistic severity, id counts to the destination uvm_report_server the copy is cummulative (only items from the source are transferred, already existing entries are not deleted, existing entries/counts are overridden when they exist in the source set)

Parameters

rhs (*uvm_object*)

virtual function void process_report_message(uvm_report_message report_message)

Function- process_report_message

Main entry for uvm_report_server, combines execute_report_message and compose_report_message

Parameters

report_message (*uvm_report_message*)

virtual function void execute_report_message(uvm_report_message report_message, string composed_message)

Function

execute_report_message

Processes the provided message per the actions contained within.

Expert users can overload this method to customize action processing.

Parameters

report_message (*uvm_report_message*)

composed_message (*string*)

virtual function string compose_report_message(uvm_report_message report_message, string report_object_name = "")

Function

compose_report_message

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.

Parameters

report_message (*uvm_report_message*)

report_object_name (*string*)

virtual function void report_summarize(UVM_FILE file = 0)

Function

report_summarize

Outputs statistical information on the reports issued by this 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.

Parameters

file (*UVM_FILE*)

virtual function void summarize(UVM_FILE file = 0)

Function- summarize

Parameters

file (*UVM_FILE*)

static function void set_server(uvm_report_server server)

Function

set_server

Sets the global report server to use for reporting.

This method is provided as a convenience wrapper around setting the report server via the *uvm_coreservice_t::set_report_server* method.

In addition to setting the server this also copies the severity/id counts from the current report_server to the new one

```
// Using the uvm_coreservice_t:
uvm_coreservice_t cs;
cs = uvm_coreservice_t::get();
your_server.copy(cs.get_report_server());
cs.set_report_server(your_server);

// Not using the uvm_coreservice_t:
uvm_report_server::set_server(your_server);
```

Parameters
server (*uvm_report_server*)

static function uvm_report_server get_server()

Function

get_server

Gets the global report server used for reporting.

This method is provided as a convenience wrapper around retrieving the report server via the *uvm_coreservice_t::get_report_server* method.

```
// Using the uvm_coreservice_t:
uvm_coreservice_t cs;
uvm_report_server rs;
cs = uvm_coreservice_t::get();
rs = cs.get_report_server();

// Not using the uvm_coreservice_t:
uvm_report_server rs;
rs = uvm_report_server::get_server();
```

Return type
uvm_report_server

3.5 Class uvm_pkg::uvm_report_catcher

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_callback*
↪*uvm_pkg* :: *uvm_report_catcher*

CLASS

uvm_report_catcher

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

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

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

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

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

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

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

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

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

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

Table 2: Variables

Name	Type	Description
DO_NOT_CATCH	int	Flag counts
DO_NOT_MODIFY	int	

Constructors

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

Function

new

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

Parameters

`name` (*`string`*)

Enums

`action_e`

Enum Items

UNKNOWN_ACTION
THROW
CAUGHT

Functions

`function uvm_report_object get_client()`

Function

get_client

Returns the *uvm_report_object* that has generated the message that is currently being processed.

Return type

uvm_report_object

`function uvm_severity get_severity()`

Function

get_severity

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

Return type

uvm_severity

`function string get_context()`

Function

get_context

Returns the context name of the message that is currently being processed. This is typically the full hierarchical name of the component that issued the message. However, if user-defined context is set from a uvm_report_message, the user-defined context will be returned.

`function int get_verbosity()`

Function

get_verbosity

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

`function string get_id()`

Function

get_id

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

`function string get_message()`

Function

get_message

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

`function uvm_action get_action()`

Function

get_action

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

Return type
uvm_action

`function string get_fname()`

Function

get_fname

Returns the file name of the message.

`function int get_line()`

Function

get_line

Returns the line number of the message.

`function uvm_report_message_element_container get_element_container()`

Function

get_element_container

Returns the element container of the message.

Return type
uvm_report_message_element_container

`static function uvm_report_catcher get_report_catcher(string name)`

Function

get_report_catcher

Returns the first report catcher that has *name* .

Parameters
name (*`string`*)

Return type
uvm_report_catcher

`static function void print_catcher(UVM_FILE file = 0)`

Function

print_catcher

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

Parameters
file (*UVM_FILE*)

`static function void debug_report_catcher(int what = 0)`

Funciton

debug_report_catcher

Turn on report catching debug information. *what* is a bitwise AND of DO_NOT_CATCH -- forces catch to be ignored so that all catchers see the the reports. DO_NOT_MODIFY -- forces the message to remain unchanged

Parameters
what (*int*)

`virtual function action_e catch()`

Function

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.

Return type
action_e

`static function int process_all_report_catchers(uvm_report_message rm)`

process_all_report_catchers method called by report_server.report to process catchers

Parameters
rm (*uvm_report_message*)

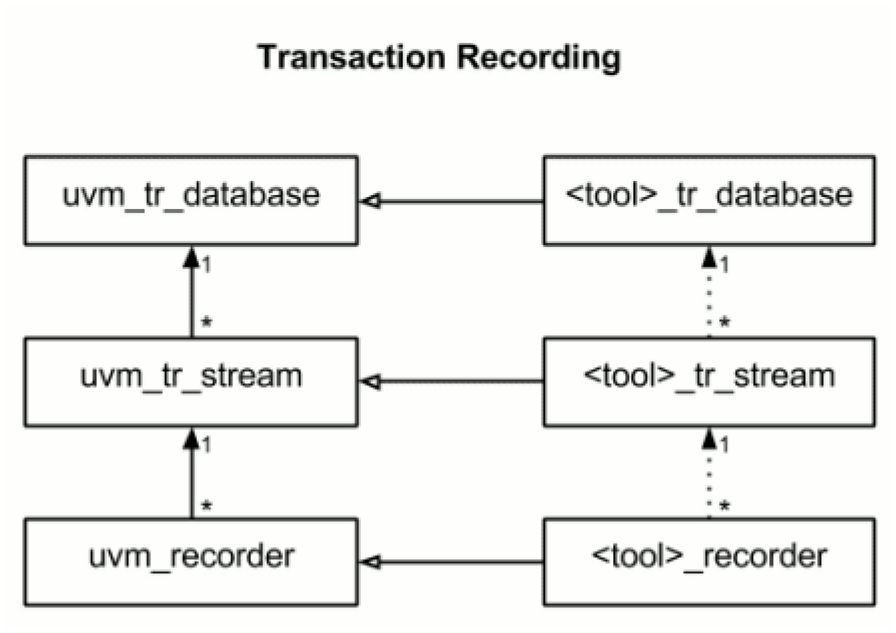
`static function void summarize()`

CHAPTER

FOUR

TRANSACTION RECORDING CLASSES

The recording classes provide a facility to record transactions into a database using a consistent API. Users can configure what gets sent to the backend database, without knowing exactly how the connection to that database is established.
The primary interface to the UVM recording facility is the *uvm_recorder* class, which serves as a reference to the transaction in the database, as well as the policy which is used to record information into the database.



The UVM provides a default implementation of the recording API, which creates textual logs. This is primarily intended to be used as an example of how to create a recording implementation without the user needing to have tool and/or vendor specific code in their testbench.

4.1 Class uvm_pkg::uvm_tr_database

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_tr_database*

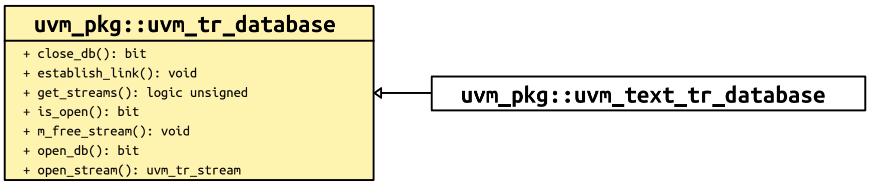


Fig. 1: Inheritance Diagram of uvm_tr_database

CLASS

uvm_tr_database

The *uvm_tr_database* class is intended to hide the underlying database implementation from the end user, as these details are often vendor or tool-specific.

The *uvm_tr_database* class is pure virtual, and must be extended with an implementation. A default text-based implementation is provided via the *uvm_text_tr_database* class.

Constructors

`function  new(string name = "unnamed-uvm_tr_database")`

Function

new

Constructor

Parameters

name

Instance name

Parameters

`name` (*`string`*)

Functions

`function bit open_db()`

Function

open_db

Open the backend connection to the database.

If the database is already open, then this method will return 1.

Otherwise, the method will call do_open_db, and return the result.

`function bit close_db()`

Function

close_db

Closes the backend connection to the database.

Closing a database implicitly closes and frees all *uvm_tr_streams* within the database.

If the database is already closed, then this method will return 1.

Otherwise, this method will trigger a do_close_db call, and return the result.

`function bit is_open()`

Function

is_open

Returns the open/closed status of the database.

This method returns 1 if the database has been successfully opened, but not yet closed.

`function uvm_tr_stream open_stream(string name, string scope = "", string type_-name = "")`

Function

open_stream

Provides a reference to a *stream* within the database.

Parameters

name

A string name for the stream. This is the name associated with the stream in the database.

scope

An optional scope for the stream.

type_name

An optional name describing the type of records which will be created in this stream.

The method returns a reference to a *uvm_tr_stream* object if successful, *null* otherwise.

This method will trigger a do_open_stream call, and if a non *null* stream is returned, then uvm_tr_stream::do_open will be called.

Streams can only be opened if the database is open (per *is_open*). Otherwise the request will be ignored, and *null* will be returned.

Parameters
- **`name`** (*`string`*)
- **`scope`** (*`string`*)
- **`type_name`** (*`string`*)

Return type
- *uvm_tr_stream*

`function logic unsigned get_streams(uvm_tr_stream q)`

Function

get_streams

Provides a queue of all streams within the database.

Parameters

q

A reference to a queue of *uvm_tr_stream*s

The *get_streams* method returns the size of the queue, such that the user can conditionally process the elements.

```
uvm_tr_stream stream_q[$];
if (my_db.get_streams(stream_q)) begin
  // Process the queue...
end
```

Parameters
- **`q`** (*uvm_tr_stream*)

`function void establish_link(uvm_link_base link)`

Function

establish_link

Establishes a *link* between two elements in the database

Links are only supported between *streams* and *records* within a single database.

This method will trigger a do_establish_link call.

Parameters
- **`link`** (*uvm_link_base*)

4.2 Class uvm_pkg::uvm_tr_stream

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_tr_stream*

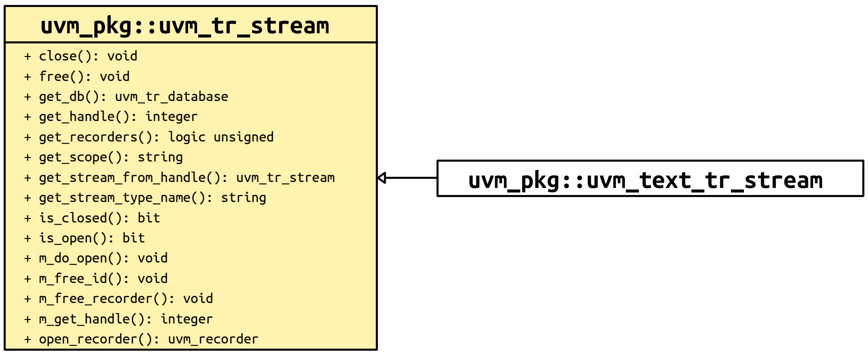

uvm_pkg::uvm_tr_stream
+ close(): void
+ free(): void
+ get_db(): uvm_tr_database
+ get_handle(): integer
+ get_recorders(): logic unsigned
+ get_scope(): string
+ get_stream_from_handle(): uvm_tr_stream
+ get_stream_type_name(): string
+ is_closed(): bit
+ is_open(): bit
+ m_do_open(): void
+ m_free_id(): void
+ m_free_recorder(): void
+ m_get_handle(): integer
+ open_recorder(): uvm_recorder
uvm_pkg::uvm_text_tr_stream

Fig. 2: Inheritance Diagram of uvm_tr_stream

CLASS

uvm_tr_stream

The *uvm_tr_stream* base class is a representation of a stream of records within a *uvm_tr_database*.

The record stream is intended to hide the underlying database implementation from the end user, as these details are often vendor or tool-specific.

The *uvm_tr_stream* class is pure virtual, and must be extended with an implementation. A default text-based implementation is provided via the *uvm_text_tr_stream* class.

Constructors

function new(string name = "unnamed-uvm_tr_stream")

Function

new

Constructor

Parameters

name

Stream instance name

Parameters

name (`string`)

Functions

function uvm_tr_database get_db()

Function

get_db

Returns a reference to the database which contains this stream.

A warning will be asserted if get_db is called prior to the stream being initialized via do_open.

Return type

uvm_tr_database

`function string get_scope()`

Function

get_scope

Returns the *scope* supplied when opening this stream.

A warning will be asserted if get_scope is called prior to the stream being initialized via do_open.

`function string get_stream_type_name()`

Function

get_stream_type_name

Returns a reference to the database which contains this stream.

A warning will be asserted if get_stream_type_name is called prior to the stream being initialized via do_open.

`function void close()`

Function

close

Closes this stream.

Closing a stream closes all open recorders in the stream.

This method will trigger a do_close call, followed by *uvm_recorder::close* on all open recorders within the stream.

`function void free()`

Function

free

Frees this stream.

Freeing a stream indicates that the database can free any references to the stream (including references to records within the stream).

This method will trigger a do_free call, followed by *uvm_recorder::free* on all recorders within the stream.

`function bit is_open()`

Function

is_open

Returns true if this *uvm_tr_stream* was opened on the database, but has not yet been closed.

`function bit is_closed()`

Function

is_closed

Returns true if this *uvm_tr_stream* was closed on the database, but has not yet been freed.

`function uvm_recorder open_recorder(string name, time open_time = 0, string type_name = "")`

Function

open_recorder

Marks the opening of a new transaction recorder on the stream.

Parameters

name

A name for the new transaction

open_time

Optional time to record as the opening of this transaction

type_name

Optional type name for the transaction

If *open_time* is omitted (or set to 0), then the stream will use the current time.

This method will trigger a do_open_recorder call. If *do_open_recorder* returns a non- *null* value, then the uvm_recorder::do_open method will be called in the recorder.

Transaction recorders can only be opened if the stream is *open* on the database (per *is_open*). Otherwise the request will be ignored, and *null* will be returned.

Parameters

- **name** (*string*)
- **open_time** (*time*)
- **type_name** (*string*)

Return type

uvm_recorder

function logic unsigned get_recorders(uvm_recorder q)

Function

get_recorders

Provides a queue of all transactions within the stream.

Parameters

q

A reference to the queue of *uvm_recorder*s

The *get_recorders* method returns the size of the queue, such that the user can conditionally process the elements.

```
uvm_recorder tr_q[$];
if (my_stream.get_recorders(tr_q)) begin
  // Process the queue...
end
```

Parameters

- **q** (*uvm_recorder*)

function integer get_handle()

Function

get_handle

Returns a unique ID for this stream.

A value of *0* indicates that the recorder has been *freed* , and no longer has a valid ID.

static function uvm_tr_stream get_stream_from_handle(integer id)

Function

get_stream_from_handle

Static accessor, returns a stream reference for a given unique id.

If no stream exists with the given *id* , or if the stream with that *id* has been freed, then *null* is returned.

Parameters

- **id** (*integer*)

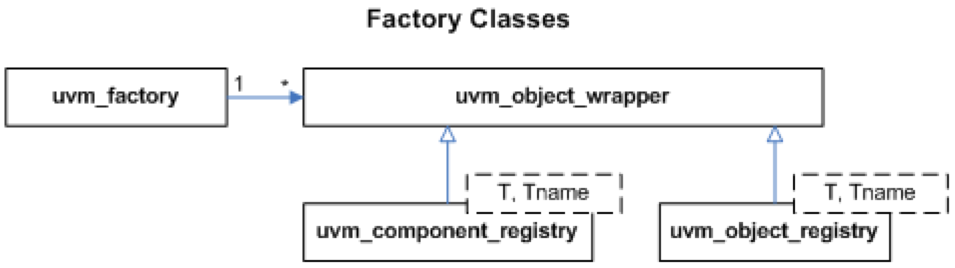
Return type

uvm_tr_stream

CHAPTER
FIVE

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_default_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.



CHAPTER

SIX

PHASING OVERVIEW

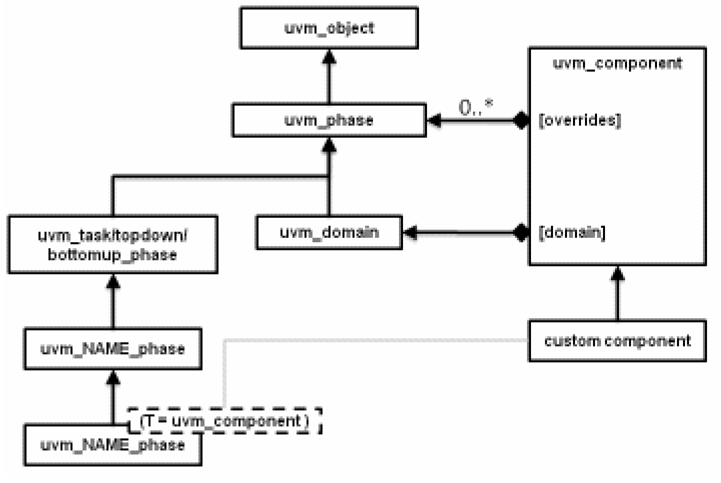
UVM implements an automated mechanism for phasing the execution of the various components in a testbench.

Phasing Implementation

The API described here provides a general purpose testbench phasing
solution, consisting of a phaser machine, traversing a master schedule
graph, which is built by the integrator from one or more instances of
template schedules provided by UVM or by 3rd-party VIP, and which supports
implicit or explicit synchronization, runtime control of threads and jumps.
Each schedule leaf node refers to a single phase that is compatible with
that VIP's components and which executes the required behavior via a
functor or delegate extending the phase into component context as required.
Execution threads are tracked on a per-component basis.

Class hierarchy

A single class represents both the definition, the state, and the context of a phase. It is instantiated once as a singleton IMP and one or more times as nodes in a graph which represents serial and parallel phase relationships and stores current state as the phaser progresses, and the phase implementation which specifies required component behavior (by extension into component context if non-default behavior required.)



The following classes related to phasing are defined herein

uvm_phase : The base class for defining a phase's behavior, state, context

uvm_domain : Phasing schedule node representing an independent branch of the schedule

uvm_bottomup_phase : A phase implementation for bottom up function phases.

uvm_topdown_phase : A phase implementation for topdown function phases.

uvm_task_phase : A phase implementation for task phases.

Common, Run-Time and User-Defined Phases

The common phases to all *uvm_component*s are described in <UVM Common Phases>.

The run-time phases are described in <UVM Run-Time Phases>.

The ability to create user-defined phases is described <User-Defined Phases>.

CHAPTER

SEVEN

CONFIGURATION AND RESOURCE CLASSES

The configuration and resources classes provide access to a centralized database where type specific information can be stored and received. The *uvm_resource_db* is the low level resource database which users can write to or read from. The *uvm_config_db* is layered on top of the resoure database and provides a typed interface for configuration setting that is consistent with the <uvm_component::Configuration Interface>.
Information can be read from or written to the database at any time during simulation. A resource may be associated with a specific hierarchical scope of a *uvm_component* or it may be visible to all components regardless of their hierarchical position.

CHAPTER
EIGHT

SYNCHRONIZATION CLASSES

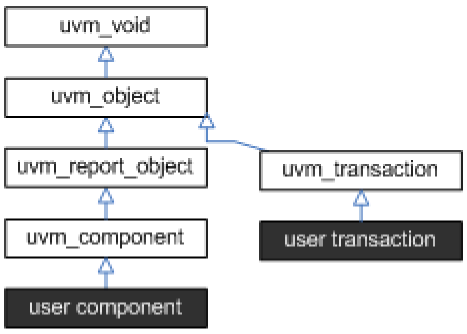


Fig. 1: Synchronization Classes

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

uvm_event#(T)
: *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 specializations of *uvm_object_string_pool #(T)* 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 #(T)*.

CHAPTER

NINE

CONTAINER CLASSES

The container classes are type parameterized data structures. The *uvm_queue #(T)* class implements a queue datastructure similar to the SystemVerilog queue construct. And the *uvm_pool #(KEY,T)* class implements a pool datastructure similar to the SystemVerilog associative array. The class based data structures allow the objects to be shared by reference; for example, a copy of a *uvm_pool #(KEY,T)* object will copy just the class handle instead of the entire associative array.

CHAPTER
TEN

TLM INTERFACES

The UVM TLM library defines several abstract, transaction-level interfaces and the ports and exports that facilitate their use. Each TLM interface consists of one or more methods used to transport data, typically whole transactions (objects) at a time. Component designs that use TLM ports and exports to communicate are inherently more reusable, interoperable, and modular.
The UVM TLM library specifies the required behavior (semantic) of each interface method. Classes (components) that implement a TLM interface must meet the specified semantic.

10.1 TLM1

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

10.2 TLM2

The TLM2 sockets provide blocking and non-blocking transaction-level interfaces with well-defined completion semantics.

10.3 Sequencer Port

A push or pull port, with well-defined completion semantics. It is used to connect sequencers with drivers and layering sequences.

10.4 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.

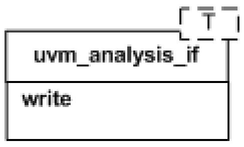
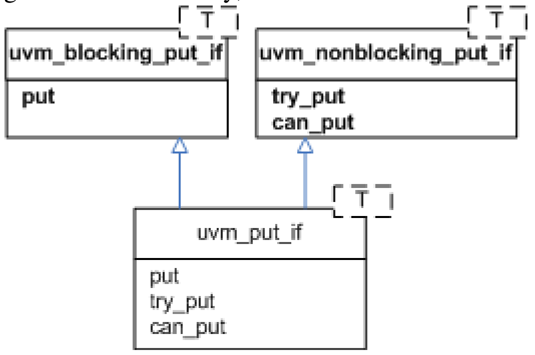


Fig. 1: Analysis

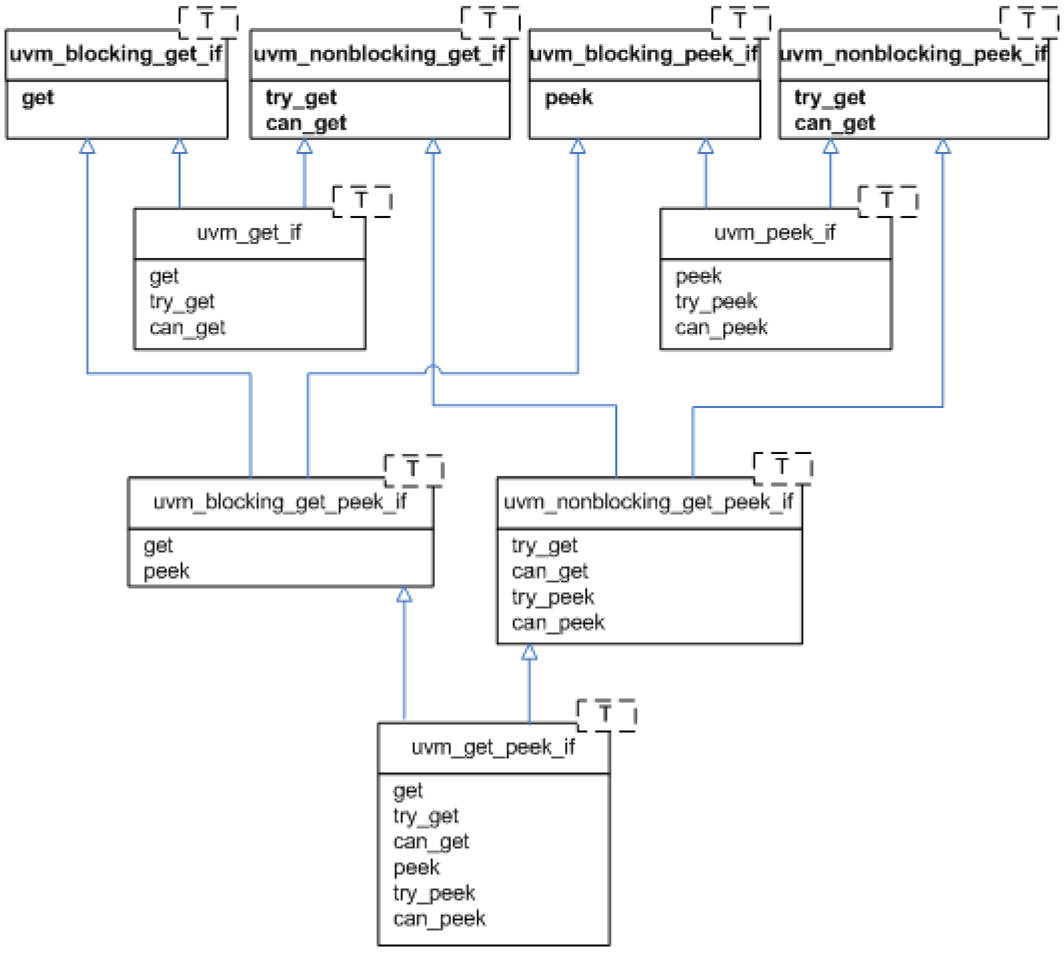
10.4.1 TLM1 Interfaces, Ports, Exports and Transport Interfaces

Each TLM1 interface is either blocking, non-blocking, or a combination of these two.
blocking
A blocking interface conveys transactions in blocking fashion; its methods do not return until the transaction has been successfully sent or retrieved. Because delivery may consume time to complete, the methods in such an interface are declared as tasks.
non-blocking

A non-blocking interface attempts to convey a transaction without consuming simulation time. Its methods are declared as functions. Because delivery may fail (e.g. the target component is busy and can not accept the request), the methods may return with failed status.
combination
A combination interface contains both the blocking and
non-blocking variants. In SystemC, combination interfaces are defined through multiple inheritance. Because SystemVerilog does not support multiple inheritance, the 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, a *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 and overview of the unidirectional and bidirectional TLM interfaces, ports, and exports.
Unidirectional Interfaces & Ports
The unidirectional TLM interfaces consist of blocking, non-blocking, and combined blocking and non-blocking variants of the *put* , *get* and *peek* interfaces, plus a non-blocking *analysis* interface.
Put
The *put* interfaces are used to send, or *put* , transactions to other components. Successful completion of a put guarantees its delivery, not execution.



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



Ports, Exports, and Imps

The UVM provides unidirectional ports, exports, and implementation ports for connecting your components via the TLM interfaces.

Ports

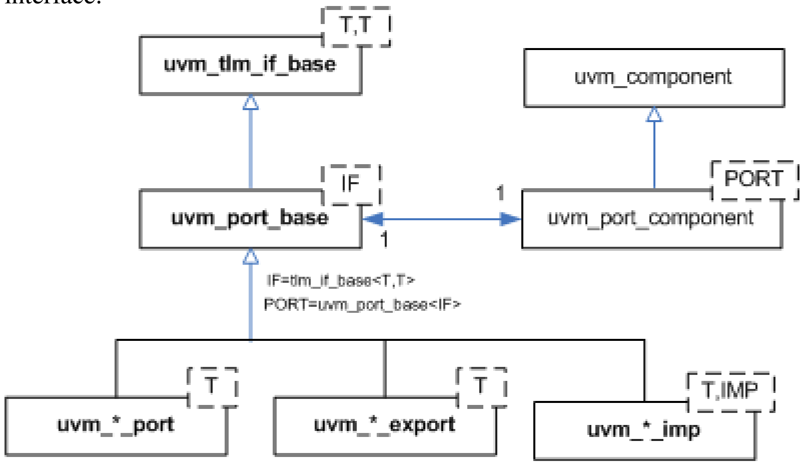
instantiated in components that *require* , or *use* , the associate interface to initiate transaction requests.

Exports

instantiated by components that *forward* an implementation of the methods defined in the associated interface. The implementation is typically provided by an
imp port in a child component.

Imps

instantiated by components that *provide* or *implement* an implementation of the methods defined in the associated interface.



A summary of port, export, and imp declarations are

```
class uvm_*_export #(type T=int)
  extends uvm_port_base #(tlm_if_base #(T,T));

class uvm_*_port #(type T=int)
  extends uvm_port_base #(tlm_if_base #(T,T));

class uvm_*_imp #(type T=int)
  extends uvm_port_base #(tlm_if_base #(T,T));
```

where the asterisk can be any of

```
blocking_put
nonblocking_put
put

blocking_get
nonblocking_get
get

blocking_peek
nonblocking_peek
peek

blocking_get_peek
nonblocking_get_peek
get_peek

analysis
```

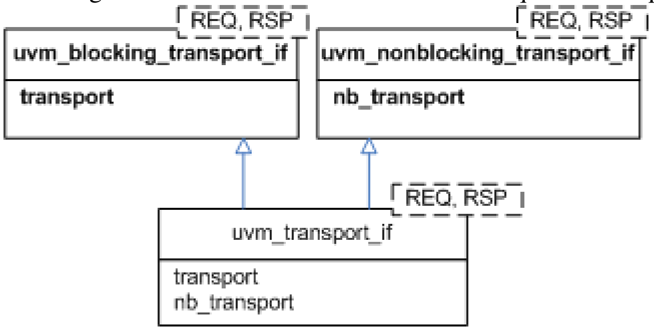
Group: Bidirectional Interfaces & Ports

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.

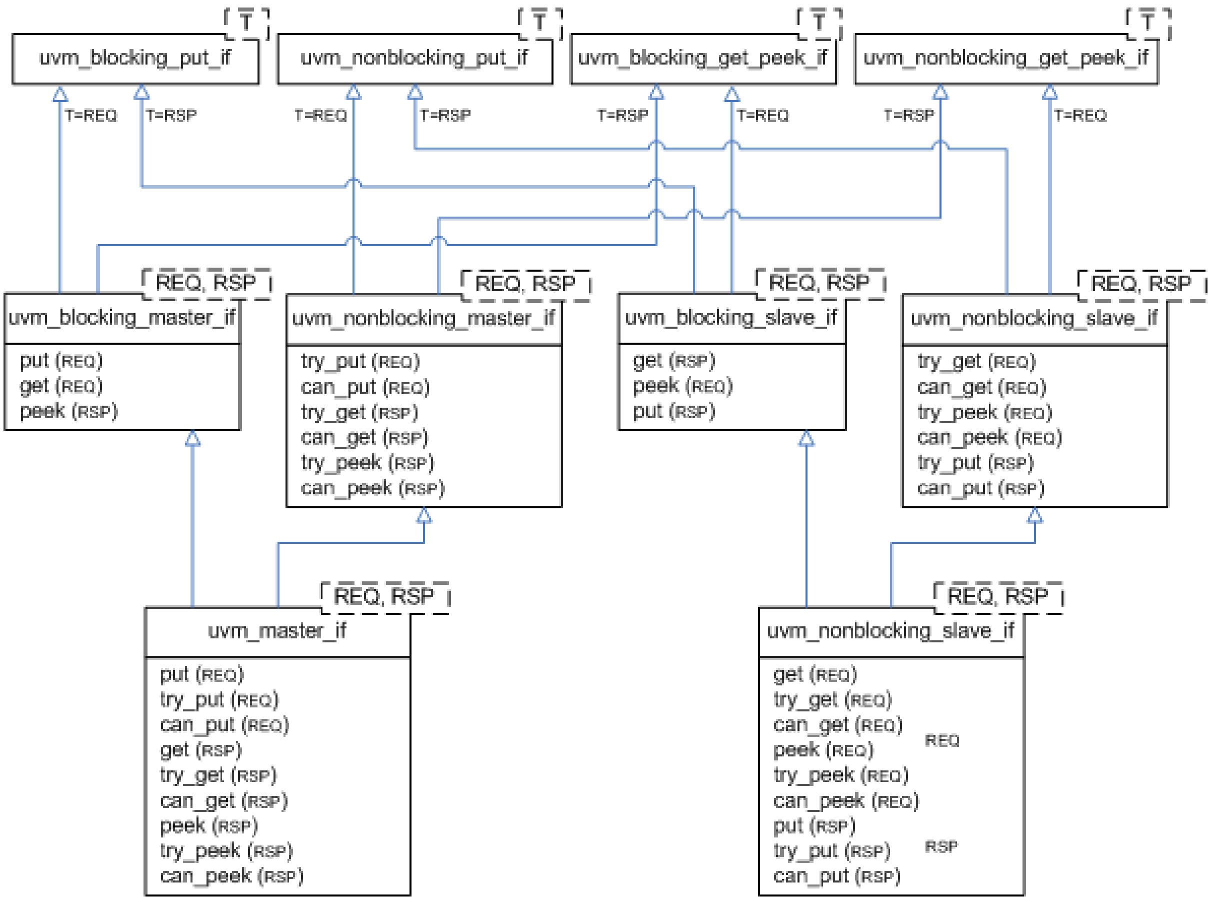
Transport

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

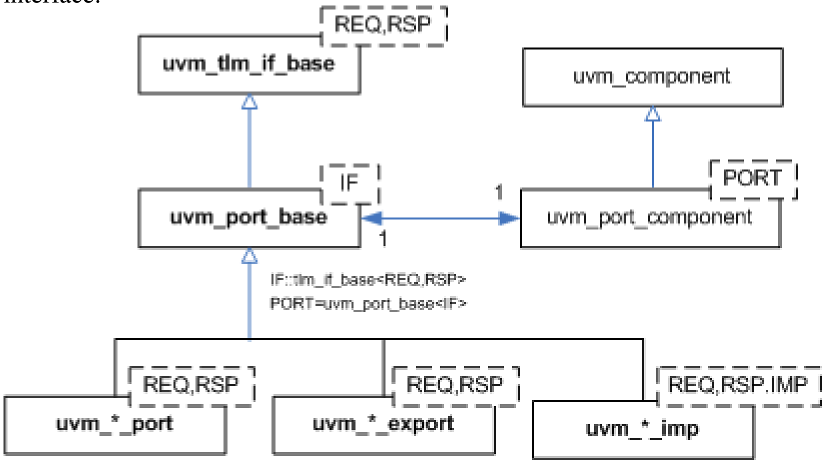
instantiated in components that *require* , or *use* , the associate interface to initiate transaction requests.

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)
```

(continues on next page)

(continued from previous page)

```
  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));
```

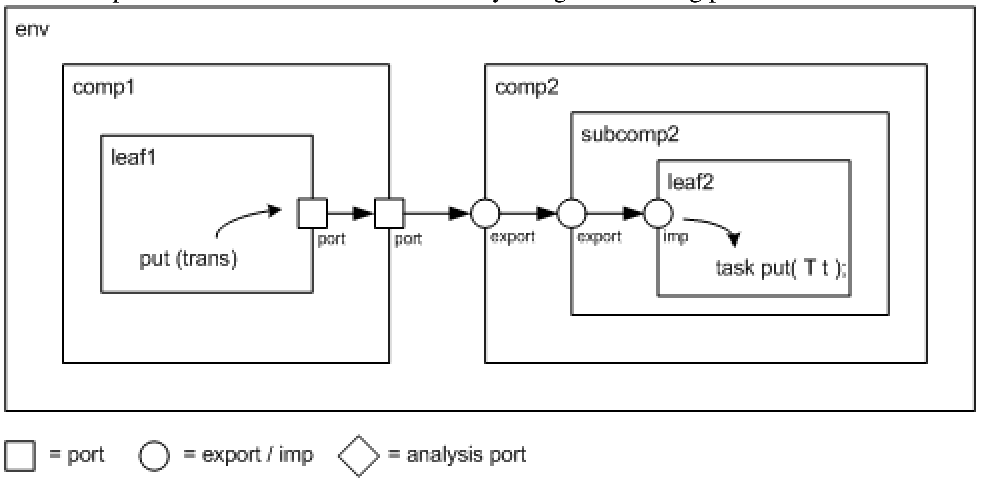
where the asterisk can be any of

```
transport
blocking_transport
nonblocking_transport

blocking_master
nonblocking_master
master

blocking_slave
nonblocking_slave
slave
```

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



port-to-port
leaf1's *out* port is connected to its parent's (comp1) *out* port
port-to-export
comp1's *out* port is connected to comp2's *in* export
export-to-export
comp2's *in* export is connected to its child's (subcomp2) *in* export
export-to-imp
subcomp2's *in* export is connected leaf2's *in* imp port.
imp-to-implementation
leaf2's *in* imp port is connected to its implementation, leaf2
Hierarchical port connections are resolved and optimized just before *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;
```

(continues on next page)

(continued from previous page)

```
  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_phase(uvm_phase phase);
    trans t;
    phase.raise_objection(this, "prolonging run_phase");
    t = new;
    t.randomize();
    out.put(t);
    phase.drop_objection(this, "prolonging run_phase");
  endtask

endclass


class comp1 extends uvm_component;

  `uvm_component_utils(comp1)

  uvm_blocking_put_port #(trans) out;

  leaf1 leaf;

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

  virtual function void build_phase(uvm_phase phase);
    out = new("out",this);
    leaf = new("leaf1",this);
  endfunction

  // connect port to port
  virtual function void connect_phase(uvm_phase phase);
    leaf.out.connect(out);
  endfunction

endclass


class leaf2 extends uvm_component;

  `uvm_component_utils(leaf2)
```

(continues on next page)

(continued from previous page)

```
  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_phase(uvm_phase phase);
    in = new("in",this);
    leaf = new("leaf2",this);
  endfunction

  // connect export to imp
  virtual function void connect_phase(uvm_phase phase);
    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_phase(uvm_phase phase);
    in = new("in",this);
```

(continues on next page)

(continued from previous page)

```
    subcomp = new("subcomp2",this);
  endfunction

  // connect export to export
  virtual function void connect_phase(uvm_phase phase);
    in.connect(subcomp.in);
  endfunction

endclass


class env extends uvm_component;

  `uvm_component_utils(comp1)

  comp1 comp1_i;
  comp2 comp2_i;

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

  virtual function void build_phase(uvm_phase phase);
    comp1_i = new("comp1",this);
    comp2_i = new("comp2",this);
  endfunction

  // connect port to export
  virtual function void connect_phase(uvm_phase phase);
    comp1_i.out.connect(comp2_i.in);
  endfunction

endclass


module top;
  env e = new("env");
  initial run_test();
  initial #10 uvm_top.stop_request();
endmodule
```

10.4.2 TLM2 Interfaces, Ports, Exports and Transport Interfaces Subset

Sockets group together all the necessary core interfaces for transportation and binding, allowing more generic usage models than just TLM core interfaces.
A socket is like a port or export; in fact it is derived from the same base class as ports and export, namely *uvm_-port_base #(IF)*. However, unlike a port or export a socket provides both a forward and backward path. Thus you can enable asynchronous (pipelined) bi-directional communication by connecting sockets together. To enable this, a socket contains both a port and an export. Components that initiate transactions are called initiators, and components that receive transactions sent by an initiator are called targets. Initiators have initiator sockets and targets have target sockets. Initiator sockets can connect to target sockets. You cannot connect initiator sockets to other initiator sockets and you cannot connect target sockets to target sockets.
The UVM TLM2 subset provides the following two transport interfaces:
Blocking (b_transport)
completes the entire transaction within a single method call

Non-blocking (nb_transport) - describes the progress of a transaction using multiple nb_transport() method calls going back-and-forth between initiator and target
In general, any component might modify a transaction object during its lifetime (subject to the rules of the protocol). Significant timing points during the lifetime of a transaction (for example: start of response- phase) are indicated by calling nb_transport() in either forward or backward direction, the specific timing point being given by the phase argument. Protocol-specific rules for reading or writing the attributes of a transaction can be expressed relative to the phase. The phase can be used for flow control, and for that reason might have a different value at each hop taken by a transaction; the phase is not an attribute of the transaction object.
A call to nb_transport() always represents a phase transition. However, the return from nb_transport() might or might not do so, the choice being indicated by the value returned from the function (<UVM_TLM_ACCEPTED> versus <UVM_TLM_UPDATED>). Generally, you indicate the completion of a transaction over a particular hop using the value of the phase argument. As a shortcut, a target might indicate the completion of the transaction by returning a special value of <UVM_TLM_COMPLETED>. However, this is an option, not a necessity.
The transaction object itself does not contain any timing information by design. Or even events and status information concerning the API. You can pass the delays as arguments to b_transport()/ nb_transport() and push the actual realization of any delay in the simulator kernel downstream and defer (for simulation speed).
Use Models
Since sockets are derived from *uvm_port_base #(IF)* they are created and connected in the same way as port, and exports. Create them in the build phase and connect them in the connect phase by calling connect(). Initiator and target termination sockets are on the ends of any connection. There can be an arbitrary number of pass-through sockets in the path between initiator and target. Some socket types must be bound to imps implementations of the transport tasks and functions. Blocking terminator sockets must be bound to an implementation of b_transport(), for example. Nonblocking initiator sockets must be bound to an implementation of nb_transport_bw() and nonblocking target sockets must be bound to an implementation of nb_transport_fw(). Typically, the task or function is implemented in the component in which the socket is instantiated and the component type and instance are provided to complete the binding.
Consider for example a consumer component with a blocking target socket.
Example:

```
class consumer extends uvm_component;
   tlm2_b_target_socket #(consumer, trans) target_socket;
   function new(string name, uvm_component parent);
     super.new(name, parent);
   endfunction
   function void build();
     target_socket = new("target_socket", this, this);
   endfunction
   task b_transport(trans t, uvm_tlm_time delay);
     #5;
     uvm_report_info("consumer", t.convert2string());
   endtask
endclass
```

The interface task b_transport() is implemented in the consumer component. The consumer component type is used in the declaration of the target socket. This informs the socket object the type of the object that contains the interface task, in this case b_transport(). When the socket is instantiated "this" is passed in twice, once as the parent just like any other component instantiation and again to identify the object that holds the implementation of b_transport(). Finally, in order to complete the binding, an implementation of b_transport() must be present in the consumer component. Any component that has either a blocking termination socket, a nonblocking initiator socket, or a nonblocking termination socket must provide implementations of the relevant components. This includes initiator and target components as well as interconnect components that have these kinds of sockets. Components with pass-through sockets do not need to provide implementations of any sort. Of course, they must ultimately be connected to sockets that do that the necessary implementations.
In summary
Call to b_transport()
start-of-life of transaction
Return from b_transport()
end-of-life of transaction

Phase argument to nb_transport()
timing point within lifetime of transaction
Return value of nb_transport()
whether return path is being used (also shortcut to final phase)
Response status within transaction object
protocol-specific status, success/failure of transaction

On top of this, TLM-2.0 defines a generic payload and base protocol to enhance interoperability for models with a memory-mapped bus interface.

It is possible to use the interfaces described above with user-defined transaction types and protocols for the sake of interoperability. However, TLM-2.0 strongly recommends either using the base protocol off-the-shelf or creating models of specific protocols on top of the base protocol.

The UVM 1.2 standard only defines and supports this TLM2 style interface for SystemVerilog to SystemVerilog communication. Mixed language TLM communication is saved for future extension.

CHAPTER
ELEVEN

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.

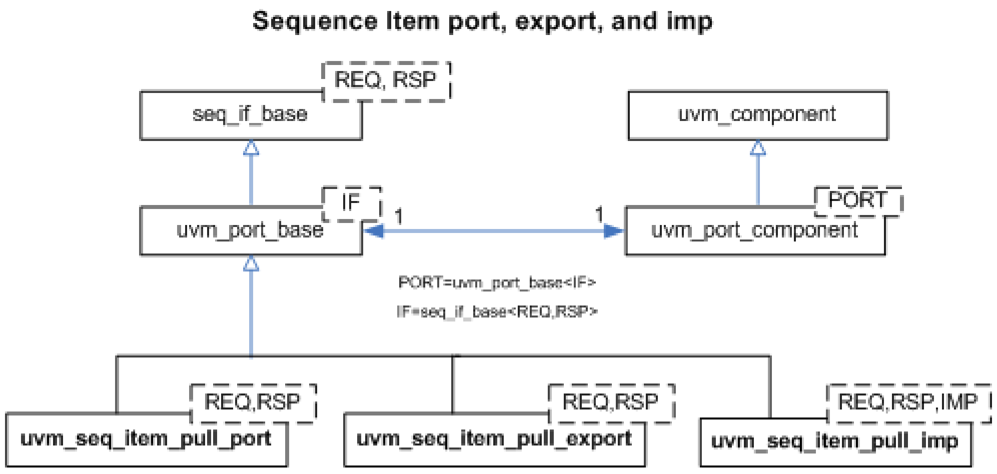


Fig. 1: Sequencer Classes

There are two sequencer variants available.

uvm_sequencer

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

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.

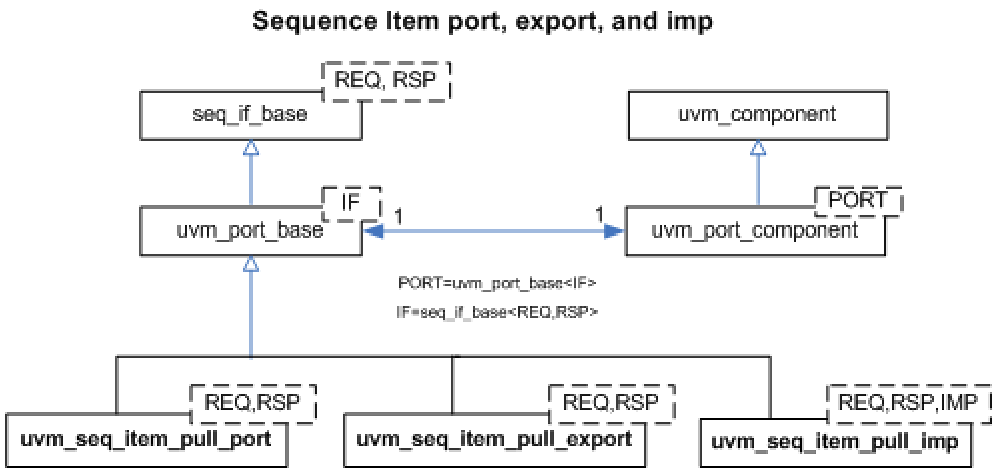
See also

See *Sequence Classes* for an overview on sequences and sequence items.

11.1 Sequence Item Ports

As with all UVM components, the sequencers and drivers described above use *TLM Interfaces* to communicate transactions.

The *uvm_sequencer #(REQ,RSP)* and *uvm_driver #(REQ,RSP)* pair also uses a *sequence item pull port* to achieve the special execution semantic needed by the sequencer-driver pair.



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.

CHAPTER

TWELVE

SEQUENCE CLASSES

Sequences encapsulate user-defined procedures that generate multiple *uvm_sequence_item* -based transactions. Such sequences can be reused, extended, randomized, and combined sequentially and hierarchically in interesting ways to produce realistic stimulus to your DUT.
With *uvm_sequence* objects, users can encapsulate DUT initialization code, bus-based stress tests, network protocol stacks-- anything procedural-- then have them all execute in specific or random order to more quickly reach corner cases and coverage goals.
The UVM sequence item and sequence class hierarchy is shown below.

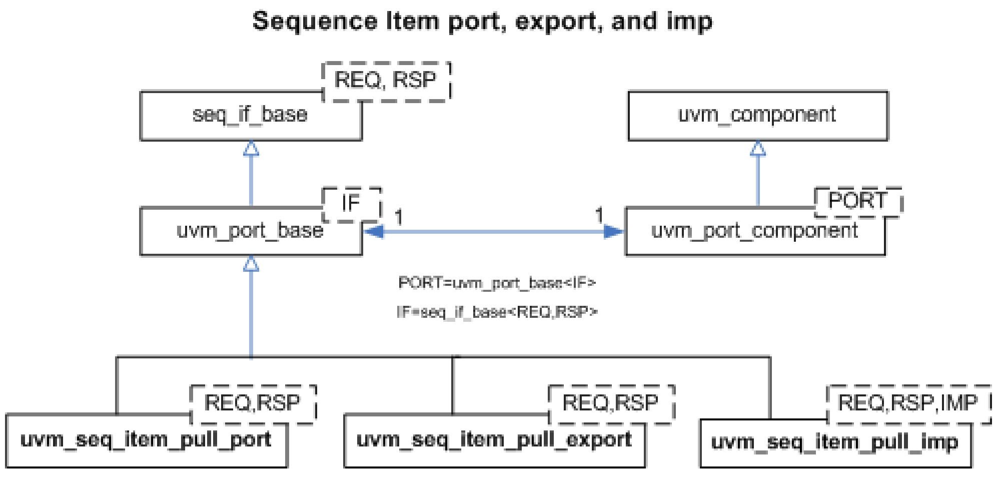


Fig. 1: Sequence Classes

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.

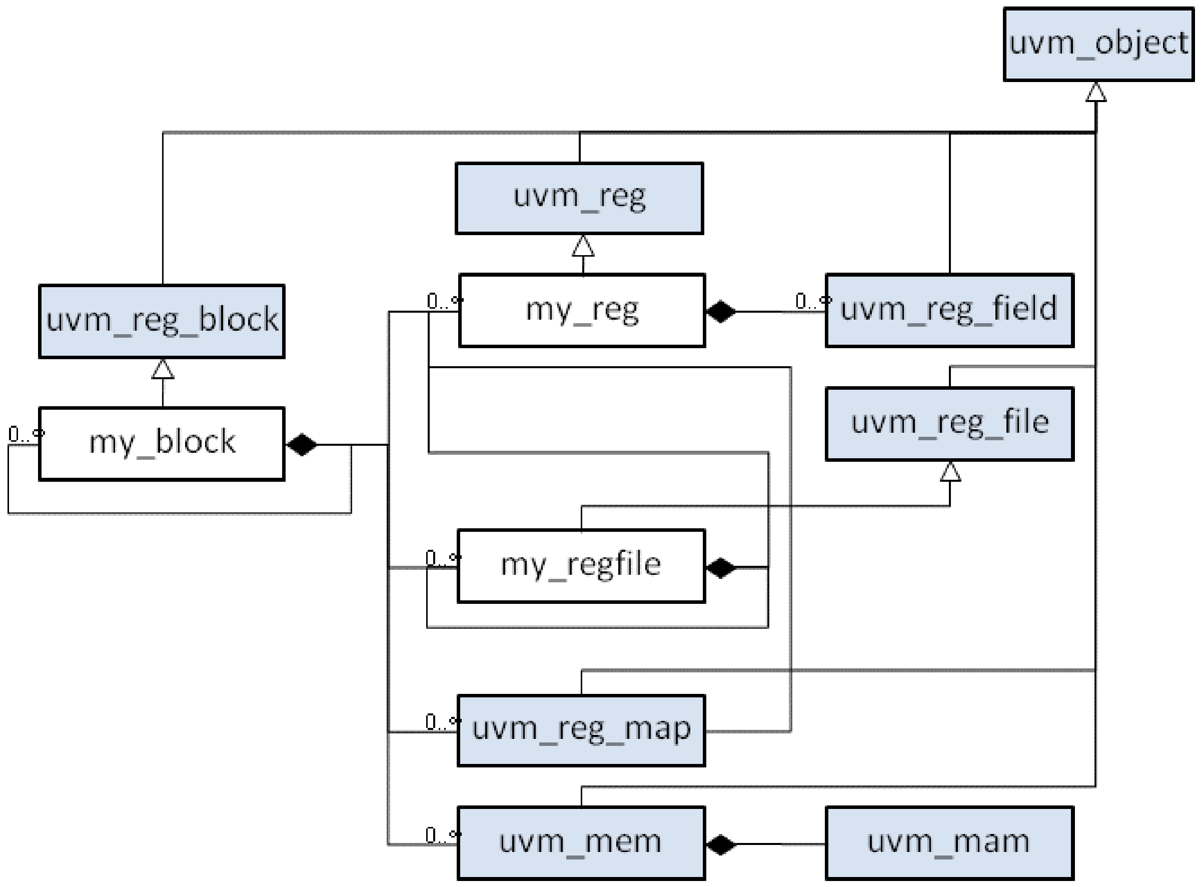
uvm_sequence
: The *uvm_sequence #(REQ,RSP)* extends *uvm_sequence_item* to add the ability to generate streams of *uvm_sequence_items* , either directly or by recursively executing other *uvm_sequences* .

CHAPTER

THIRTEEN

REGISTER LAYER

The UVM register layer defines several base classes that, when properly extended, abstract the read/write operations to registers and memories in a design-under-verification.
A register model is typically composed of a hierarchy of blocks that usually map to the design hierarchy. Blocks contain registers, register files and memories.
The UVM register layer classes are not usable as-is. They only provide generic and introspection capabilities. They must be specialized via extensions to provide an abstract view that corresponds to the actual registers and memories in a design. Due to the large number of registers in a design and the numerous small details involved in properly configuring the UVM register layer classes, this specialization is normally done by a model generator. Model generators work from a specification of the registers and memories in a design and are thus able to provide an up-to-date, correct-by-construction register model. Model generators are outside the scope of the UVM library.
The class diagram of a register layer model is shown below.



CHAPTER
FOURTEEN

COMMAND LINE PROCESSOR CLASS

This class provides a general interface to the command line arguments that were provided for the given simulation. Users can retrieve the complete arguments using methods such as *get_args()* and *get_arg_matches()* but also retrieve the suffixes of arguments using *get_arg_values()* .
The uvm_cmdline_processor class also provides support for setting various UVM variables from the command line such as components' verbosities and configuration settings for integral types and strings. Command line arguments that are in uppercase should only have one setting to invocation. Command line arguments that in lowercase can have multiple settings per invocation.
All of these capabilities are described in the *uvm_cmdline_processor* section.

CHAPTER

FIFTEEN

PACKAGES

15.1 Package uvm_pkg

15.1.1 Classes

15.1.1.1 Class uvm_pkg::get_t

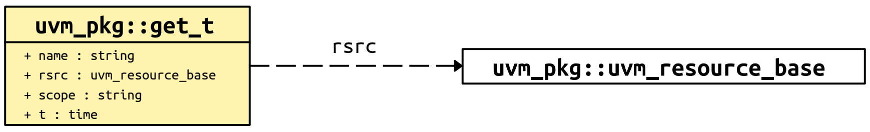


Fig. 1: Collaboration Diagram of get_t

Class

get_t

Instances of get_t are stored in the history list as a record of each get. Failed gets are indicated with rsrc set to *null* . This is part of the audit trail facility for resources.

Table 1: Variables

Name	Type	Description
name	string	
scope	string	
rsrc	*uvm_resource_base*	
t	time	

15.1.1.2 Class uvm_pkg::sev_id_struct

Table 2: Variables

Name	Type	Description
sev_specified	bit	
id_specified	bit	
sev	*uvm_severity*	
id	string	
is_on	bit	

15.1.1.3 Class uvm_pkg::uvm_agent

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_agent*

CLASS

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.

Table 3: Variables

Name	Type	Description
is_active	*uvm_active_passive_-enum*	
type_name	string	

Constructors

function new(string name, uvm_component parent)

Function

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.

The int configuration parameter is_active is used to identify whether this agent should be acting in active or passive mode. This parameter can be set by doing:

```
uvm_config_int::set(this, "<relative_path_to_agent>, "is_active", UVM_ACTIVE);
```

Parameters
name (`string`)
parent (*uvm_component*)

Functions

virtual function void build_phase(uvm_phase phase)

Parameters
phase (*uvm_phase*)

virtual function string get_type_name()

virtual function uvm_active_passive_enum get_is_active()

Function

get_is_active

Returns UVM_ACTIVE is the agent is acting as an active agent and UVM_PASSIVE if it is acting as a passive agent. The default implementation is to just return the is_active flag, but the component developer may override this behavior if a more complex algorithm is needed to determine the active/passive nature of the agent.

Return type
uvm_active_passive_enum

15.1.1.4 Class uvm_pkg::uvm_algorithmic_comparator

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_algorithmic_comparator*

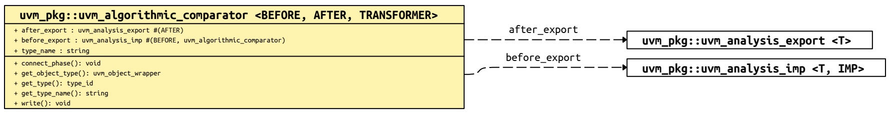

uvm_pkg::uvm_algorithmic_comparator <BEFORE, AFTER, TRANSFORMER>
+ after_export : uvm_analysis_export #(AFTER)
+ before_export : uvm_analysis_imp #(BEFORE, uvm_algorithmic_comparator)
+ type_name : string
+ connect_phase(): void
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string
+ write(): void
after_export
uvm_pkg::uvm_analysis_export <T>
before_export
uvm_pkg::uvm_analysis_imp <T, IMP>

Fig. 2: Collaboration Diagram of uvm_algorithmic_comparator

Table 4: Parameters

Name	Default value	Description
BEFORE	int	
AFTER	int	
TRANSFORMER	int	

Table 5: Variables

Name	Type	Description
type_name	string	
before_export	*uvm_analysis_imp#(int, uvm_algorithmic_comparator#(int, int, int))*	***Port*** 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	*uvm_analysis_export#(int)*	***Port*** 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.

Table 6: Typedefs

Name	Actual Type	Description
this_type	*uvm_algorithmic_comparator#(BEFORE, AFTER, TRANSFORMER)*	

Constructors

`function  new(string name, uvm_component parent = null, int transformer = null)`

Function

new

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 (handles can't be *null*) and must implement the transform() method.

Parameters

`name` (*`string`*)
`parent` (*uvm_component*)
`transformer` (*`int`*)

Functions

`virtual  function string get_type_name()`

`virtual  function void connect_phase(uvm_phase phase)`

Parameters

`phase` (*uvm_phase*)

`function void write(int b)`

Parameters

`b` (*`int`*)

15.1.1.5 Class uvm_pkg::uvm_analysis_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_analysis_export*

Class

uvm_analysis_export

Exports a lower-level *uvm_analysis_imp* to its parent.

Table 7: Parameters

Name	Default value	Description
T	int	

Constructors

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

Function

new

Instantiate the export.

Parameters

`name` (*`string`*)
`parent` (*uvm_component*)

Functions

`virtual function string get_type_name()`

`virtual function void write(int t)`

analysis port differs from other ports in that it broadcasts to all connected interfaces. Ports only send to the interface at the index specified in a call to set_if (0 by default).

Parameters

`t` (*`int`*)

15.1.1.6 Class uvm_pkg::uvm_analysis_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_analysis_imp*

Class

uvm_analysis_imp

Receives all transactions broadcasted by a *uvm_analysis_port*. It serves as the termination point of an analysis port/export/imp connection. The component attached to the *imp* class--called a *subscriber* -- implements the analysis interface.

Will invoke the *write(T)* method in the parent component. The implementation of the *write(T)* method must not modify the value passed to it.

```
class sb extends uvm_component;
  uvm_analysis_imp#(trans, sb) ap;

  function new(string name = "sb", uvm_component parent = null);
     super.new(name, parent);
     ap = new("ap", this);
  endfunction

  function void write(trans t);
      ...
  endfunction
endclass
```

Table 8: Parameters

Name	Default value	Description
T	int	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **name** (*string*)
- **imp** (*int*)

Functions

`virtual function void write(int t)`

Parameters
- **t** (*int*)

15.1.1.7 Class uvm_pkg::uvm_analysis_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_analysis_port*

Class

uvm_analysis_port

Broadcasts a value to all subscribers implementing a *uvm_analysis_imp*.

```
class mon extends uvm_component;
  uvm_analysis_port#(trans) ap;

  function new(string name = "sb", uvm_component parent = null);
     super.new(name, parent);
     ap = new("ap", this);
  endfunction

  task run_phase(uvm_phase phase);
      trans t;
      ...
      ap.write(t);
      ...
  endfunction
endclass
```

Table 9: Parameters

Name	Default value	Description
T	int	

Constructors

`function new(string name, uvm_component parent)`

Parameters
- **`name`** (`string`)
- **`parent`** (*uvm_component*)

Functions

`virtual function string get_type_name()`

`virtual function void write(int t)`

Method

write

Send specified value to all connected interface

Parameters
- **`t`** (`int`)

15.1.1.8 Class uvm_pkg::uvm_barrier

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_barrier*

CLASS

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.

Table 10: Variables

Name	Type	Description
type_name	string	

Constructors

`function new(string name = "", int threshold = 0)`

Function

new

Creates a new barrier object.

Parameters

`name` (*`string`*)
`threshold` (*`int`*)

Functions

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

Function

reset

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.

Parameters

`wakeup` (*`bit`*)

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

Function

set_auto_reset

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.

Parameters

`value` (*`bit`*)

`virtual function void set_threshold(int threshold)`

Function

set_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.

Parameters

`threshold` (*`int`*)

`virtual function int get_threshold()`

Function

get_threshold

Gets the current threshold setting for the barrier.

`virtual function int get_num_waiters()`

Function

get_num_waiters

Returns the number of processes currently waiting at the barrier.

`virtual function void cancel()`

Function

cancel

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

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

Parameters

`name` (*`string`*)

Return type

uvm_object

`virtual function string get_type_name()`

`virtual function void do_print(uvm_printer printer)`

Parameters

`printer` (*uvm_printer*)

`virtual function void do_copy(uvm_object rhs)`

Parameters

`rhs` (*uvm_object*)

Tasks

`virtual function wait_for()`

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.

15.1.1.9 Class uvm_pkg::uvm_bit_rsrc

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_resource_base*
↪*uvm_pkg* :: *uvm_resource*
↪*uvm_pkg* :: *uvm_bit_rsrc*

uvm_bit_rsrc

specialization of uvm_resource (T) for T = vector of bits

Table 11: Parameters

Name	Default value	Description
N	1	

Table 12: Typedefs

Name	Actual Type	Description
this_subtype	*uvm_bit_rsrc#(N)*	

Constructors

```
function  new(string name, string s = "*")
```

Parameters
- **name** (*string*)
- **s** (*string*)

Functions

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

15.1.1.10 Class uvm_pkg::uvm_blocking_get_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_blocking_get_export*

Table 13: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.11 Class uvm_pkg::uvm_blocking_get_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_blocking_get_imp*

Table 14: Parameters

Name	Default value	Description
T	int	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **`name`** (*`string`*)
- **`imp`** (*`int`*)

15.1.1.12 Class uvm_pkg::uvm_blocking_get_peek_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_blocking_get_peek_export*

Table 15: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.1.1.13 Class uvm_pkg::uvm_blocking_get_peek_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_blocking_get_peek_imp*

Table 16: Parameters

Name	Default value	Description
T	int	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **`name`** (*`string`*)
- **`imp`** (*`int`*)

15.1.1.14 Class uvm_pkg::uvm_blocking_get_peek_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_blocking_get_peek_port*

Table 17: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters

- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.15 Class uvm_pkg::uvm_blocking_get_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_blocking_get_port*

Table 18: Parameters

Name	Default value	Description
T	int	

Constructors

function new(string name, uvm_component parent, int min_size = 1, int max_size = 1)

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.16 Class uvm_pkg::uvm_blocking_master_export

uvm_pkg :: *uvm_tlm_if_base*
  ↪*uvm_pkg* :: *uvm_port_base*
    ↪*uvm_pkg* :: *uvm_blocking_master_export*

Table 19: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.1.1.17 Class uvm_pkg::uvm_blocking_master_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_blocking_master_imp*

Function

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)
```

Table 20: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	
IMP	int	
REQ_IMP	IMP	
RSP_IMP	IMP	

Table 21: Typedefs

Name	Actual Type	Description
this_imp_type	IMP	
this_req_type	REQ_IMP	
this_rsp_type	RSP_IMP	

Constructors

function new(string name, this_imp_type imp, this_req_type req_imp = null, this_-rsp_type rsp_imp = null)

Parameters

name (*string*)
imp (*this_imp_type*)

`req_imp` (*this_req_type*)
`rsp_imp` (*this_rsp_type*)

15.1.1.18 Class uvm_pkg::uvm_blocking_master_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_blocking_master_port*

Table 22: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.1.1.19 Class uvm_pkg::uvm_blocking_peek_export

uvm_pkg :: *uvm_tlm_if_base*
  ↪*uvm_pkg* :: *uvm_port_base*
    ↪*uvm_pkg* :: *uvm_blocking_peek_export*

Table 23: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters

- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.1.1.20 Class uvm_pkg::uvm_blocking_peek_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_blocking_peek_imp*

Table 24: Parameters

Name	Default value	Description
T	int	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **`name`** (*`string`*)
- **`imp`** (*`int`*)

15.1.1.21 Class uvm_pkg::uvm_blocking_peek_port

uvm_pkg :: *uvm_tlm_if_base*
  ↪*uvm_pkg* :: *uvm_port_base*
    ↪*uvm_pkg* :: *uvm_blocking_peek_port*

Table 25: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.22 Class uvm_pkg::uvm_blocking_put_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_blocking_put_export*

Table 26: Parameters

Name	Default value	Description
T	int	

Constructors

function new(string name, uvm_component parent, int min_size = 1, int max_size = 1)

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.23 Class uvm_pkg::uvm_blocking_put_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_blocking_put_imp*

Table 27: Parameters

Name	Default value	Description
T	int	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **`name`** (*`string`*)
- **`imp`** (*`int`*)

15.1.1.24 Class uvm_pkg::uvm_blocking_put_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_blocking_put_port*

Function

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)
```

Table 28: Parameters

Name	Default value	Description
T	int	

Constructors

function new(string name, uvm_component parent, int min_size = 1, int max_size = 1)

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.25 Class uvm_pkg::uvm_blocking_slave_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_blocking_slave_export*

Table 29: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters
- **name** (`string`)
- **parent** (*uvm_component*)
- **min_size** (`int`)
- **max_size** (`int`)

15.1.1.26 Class uvm_pkg::uvm_blocking_slave_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_blocking_slave_imp*

Table 30: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	
IMP	int	
REQ_IMP	IMP	
RSP_IMP	IMP	

Table 31: Typedefs

Name	Actual Type	Description
this_imp_type	IMP	
this_req_type	REQ_IMP	
this_rsp_type	RSP_IMP	

Constructors

`function new(string name, this_imp_type imp, this_req_type req_imp = null, this_rsp_type rsp_imp = null)`

Parameters
- **`name`** (`string`)
- **`imp`** (*this_imp_type*)
- **`req_imp`** (*this_req_type*)
- **`rsp_imp`** (*this_rsp_type*)

15.1.1.27 Class uvm_pkg::uvm_blocking_slave_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_blocking_slave_port*

Table 32: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.1.1.28 Class uvm_pkg::uvm_blocking_transport_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_blocking_transport_export*

Table 33: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.1.1.29 Class uvm_pkg::uvm_blocking_transport_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_blocking_transport_imp*

Table 34: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **name** (*string*)
- **imp** (*int*)

15.1.1.30 Class uvm_pkg::uvm_blocking_transport_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_blocking_transport_port*

Table 35: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters

- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.31 Class uvm_pkg::uvm_bottom_up_visitor_adapter

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_visitor_adapter*
↪*uvm_pkg* :: *uvm_bottom_up_visitor_adapter*

CLASS

uvm_bottom_up_visitor_adapter

This uvm_bottom_up_visitor_adapter traverses the STRUCTURE *s* (and will invoke the visitor) in a hierarchical fashion. During traversal all children of node *s* will be visited *s* will be visited.

Table 36: Parameters

Name	Default value	Description
STRUCTURE	uvm_component	
VISITOR	uvm_visitor	

Constructors

`function new(string name = "")`

Parameters
- **`name`** (`string`)

Functions

`virtual function void accept(uvm_component s, uvm_visitor#(uvm_component) v, uvm_-structure_proxy#(uvm_component) p, bit invoke_begin_end = 1)`

Parameters
- **`s`** (*uvm_component*)
- **`v`** (*uvm_visitor#(uvm_component)*)
- **`p`** (*uvm_structure_proxy#(uvm_component)*)
- **`invoke_begin_end`** (`bit`)

15.1.1.32 Class uvm_pkg::uvm_bottomup_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_bottomup_phase*

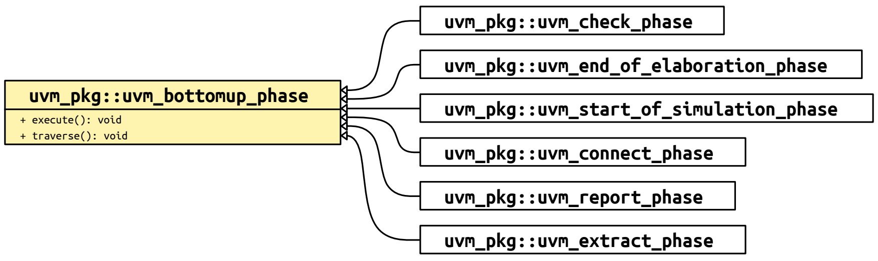

uvm_pkg::uvm_bottomup_phase
+ execute(): void
+ traverse(): void
uvm_pkg::uvm_check_phase
uvm_pkg::uvm_end_of_elaboration_phase
uvm_pkg::uvm_start_of_simulation_phase
uvm_pkg::uvm_connect_phase
uvm_pkg::uvm_report_phase
uvm_pkg::uvm_extract_phase

Fig. 3: Inheritance Diagram of uvm_bottomup_phase

Class

uvm_bottomup_phase

Virtual base class for function phases that operate bottom-up. The pure virtual function execute() is called for each component. This is the default traversal so is included only for naming.

A bottom-up function phase completes when the *execute()* method has been called and returned on all applicable components in the hierarchy.

Constructors

`function new(string name)`

Function

new

Create a new instance of a bottom-up phase.

Parameters

name (*string*)

Functions

`virtual function void traverse(uvm_component comp, uvm_phase phase, uvm_phase_-state state)`

Function

traverse

Traverses the component tree in bottom-up order, calling *execute* for each component.

Parameters

comp (*uvm_component*)
phase (*uvm_phase*)
state (*uvm_phase_state*)

`virtual function void execute(uvm_component comp, uvm_phase phase)`

Function

execute

Executes the bottom-up phase *phase* for the component *comp* .

Parameters

`comp` (*uvm_component*)
`phase` (*uvm_phase*)

15.1.1.33 Class uvm_pkg::uvm_build_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_topdown_phase*
↪*uvm_pkg* :: *uvm_build_phase*

Title

UVM Common Phases

The common phases are the set of function and task phases that all *uvm_component*s execute together. All *uvm_component*s are always synchronized with respect to the common phases.

The names of the UVM phases (which will be returned by get_name() for a phase instance) match the class names specified below with the "uvm"*and* "phase" removed. For example, the build phase corresponds to the uvm_build_phase class below and has the name "build", which means that the following can be used to call foo() at the end of the build phase (after all lower levels have finished build):

```
function void phase_ended(uvm_phase phase) ;
   if (phase.get_name()=="build") foo() ;
endfunction
```

The common phases are executed in the sequence they are specified below.

Class

uvm_build_phase

Create and configure of testbench structure

uvm_topdown_phase that calls the *uvm_component::build_phase* method.

Upon entry

The top-level components have been instantiated under *uvm_root*.
Current simulation time is still equal to 0 but some "delta cycles" may have occurred

Typical Uses

Instantiate sub-components.
Instantiate register model.
Get configuration values for the component being built.
Set configuration values for sub-components.

Exit Criteria

- All *uvm_component*s have been instantiated.

Table 37: Variables

Name	Type	Description
type_name	string	

Functions

virtual function void exec_func(uvm_component comp, uvm_phase phase)

Parameters

comp (*uvm_component*)
phase (*uvm_phase*)

static function uvm_build_phase get()

Function

get

Returns the singleton phase handle

Return type

uvm_build_phase

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

15.1.1.34 Class uvm_pkg::uvm_built_in_clone

CLASS

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.

Table 38: Parameters

Name	Default value	Description
T	int	

Functions

`static function T clone(int from)`

Parameters

`from` (*`int`*)

15.1.1.35 Class uvm_pkg::uvm_built_in_comp

CLASS

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.

Table 39: Parameters

Name	Default value	Description
T	int	

Functions

`static function bit comp(int a, int b)`

Parameters
- **a** (*int*)
- **b** (*int*)

15.1.1.36 Class uvm_pkg::uvm_built_in_converter

CLASS

uvm_built_in_converter (T)

This policy class is used to convert built-in types to strings.

Provides a convert2string method that converts the built-in type, T, to a string using the %p format specifier.

Table 40: Parameters

Name	Default value	Description
T	int	

Functions

`static function string convert2string(int t)`

Parameters
- **`t`** (*`int`*)

15.1.1.37 Class uvm_pkg::uvm_built_in_pair

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_built_in_pair*

CLASS

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.

Table 41: Parameters

Name	Default value	Description
T1	int	
T2	T1	

Table 42: Variables

Name	Type	Description
type_name	string	
first	int	***Variable*** T1 first The first value in the pair
second	int	***Variable*** T2 second The second value in the pair

Table 43: Typedefs

Name	Actual Type	Description
this_type	*uvm_built_in_pair#(T1, T2)*	

Constructors

`function new(string name = "")`

Function

new

Creates an instance that holds two built-in type values. The optional name argument gives a name to the new pair object.

Parameters

name (*string*)

Functions

`virtual function string get_type_name()`

`virtual function string convert2string()`

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

Parameters

- **`rhs`** (*uvm_object*)
- **`comparer`** (*uvm_comparer*)

`virtual function void do_copy(uvm_object rhs)`

Parameters

- **`rhs`** (*uvm_object*)

15.1.1.38 Class uvm_pkg::uvm_by_level_visitor_adapter

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_visitor_adapter*
↪*uvm_pkg* :: *uvm_by_level_visitor_adapter*

CLASS

uvm_by_level_visitor_adapter

This uvm_by_level_visitor_adapter traverses the STRUCTURE *s* (and will invoke the visitor) in a hierarchical fashion. During traversal will visit all direct children of *s* before all grand-children are visited.

Table 44: Parameters

Name	Default value	Description
STRUCTURE	uvm_component	
VISITOR	uvm_visitor	

Constructors

`function new(string name = "")`

Parameters
- **`name`** (*`string`*)

Functions

`virtual function void accept(uvm_component s, uvm_visitor#(uvm_component) v, uvm_-structure_proxy#(uvm_component) p, bit invoke_begin_end = 1)`

Parameters
- **`s`** (*uvm_component*)
- **`v`** (*uvm_visitor#(uvm_component)*)
- **`p`** (*uvm_structure_proxy#(uvm_component)*)
- **`invoke_begin_end`** (*`bit`*)

15.1.1.39 Class uvm_pkg::uvm_byte_rsrc

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_resource_base*
↪*uvm_pkg* :: *uvm_resource*
↪*uvm_pkg* :: *uvm_byte_rsrc*

uvm_byte_rsrc

specialization of uvm_resource T() for T = vector of bytes

Table 45: Parameters

Name	Default value	Description
N	1	

Table 46: Typedefs

Name	Actual Type	Description
this_subtype	*uvm_byte_rsrc#(N)*	

Constructors

`function new(string name, string s = "*")`

Parameters
- **`name`** (*`string`*)
- **`s`** (*`string`*)

Functions

`virtual function string convert2string()`

15.1.1.40 Class uvm_pkg::uvm_callback

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_callback*

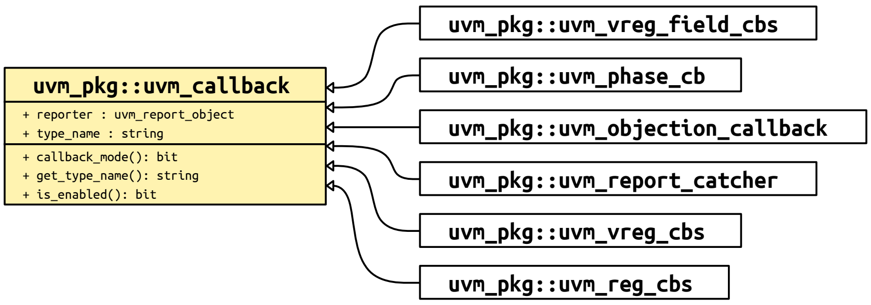


Fig. 4: Inheritance Diagram of uvm_callback

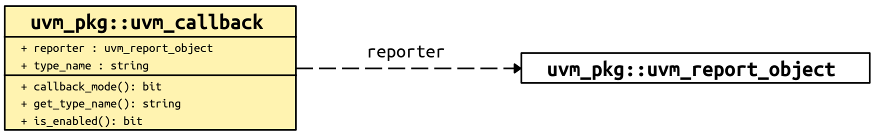


Fig. 5: Collaboration Diagram of uvm_callback

CLASS

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.

Table 47: Variables

Name	Type	Description
reporter	*uvm_report_object*	
type_name	string	

Constructors

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

Function

new

Creates a new uvm_callback object, giving it an optional *name* .

Parameters
name (*string*)

Functions

`function bit callback_mode(int on = -1)`

Function

callback_mode

Enable/disable callbacks (modeled like rand_mode and constraint_mode).

Parameters
on (*int*)

`function bit is_enabled()`

Function

is_enabled

Returns 1 if the callback is enabled, 0 otherwise.

`virtual  function string get_type_name()`

Function

get_type_name

Returns the type name of this callback object.

15.1.1.41 Class uvm_pkg::uvm_callback_iter

CLASS

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](macro-2a72a18d) and [`*uvm_do_callbacks_exit_on* provide a simple method for iterating callbacks and executing the callback methods.

Table 48: Parameters

Name	Default value	Description
T	uvm_object	
CB	uvm_callback	

Constructors

`function new(uvm_object obj)`

Function

new

Creates a new callback iterator object. It is required that the object context be provided.

Parameters

`obj` (*uvm_object*)

Functions

`function CB first()`

Function

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.

Return type

CB

`function CB last()`

Function

last

Returns the last valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object. If the queue is empty then *null* is returned.

Return type

CB

`function CB next()`

Function

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.

Return type
CB

`function CB prev()`

Function

prev

Returns the previous valid (enabled) callback of the callback type (or a derivative) that is in the queue of the context object. If there are no more valid callbacks in the queue, then *null* is returned.

Return type
CB

`function CB get_cb()`

Function

get_cb

Returns the last callback accessed via a first() or next() call.

Return type
CB

15.1.1.42 Class uvm_pkg::uvm_callbacks

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_callbacks_base*
↪*uvm_pkg* :: *uvm_typed_callbacks*
↪*uvm_pkg* :: *uvm_callbacks*

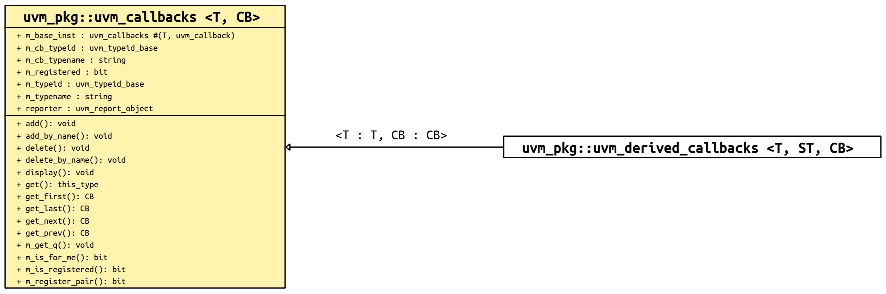


Fig. 6: Inheritance Diagram of uvm_callbacks

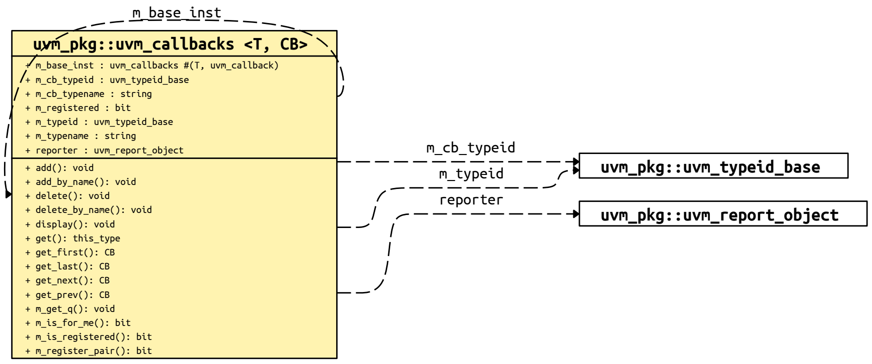


Fig. 7: Collaboration Diagram of uvm_callbacks

CLASS

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.

Table 49: Parameters

Name	Default value	Description
T	uvm_object	
CB	uvm_callback	

Table 50: Variables

Name	Type	Description
reporter	*uvm_report_object*	

Table 51: Typedefs

Name	Actual Type	Description
super_type	*uvm_typed_callbacks#(T)*	Parameter: 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>.
this_type	*uvm_callbacks#(T, CB)*	

Functions

`static function this_type get()`

get

Return type

this_type

`static function void add(uvm_object obj, uvm_callback cb, uvm_apprepend ordering = UVM_APPEND)`

Function

add

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 *ordering* 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);
```

Parameters

obj (*uvm_object*)
cb (*uvm_callback*)
ordering (*uvm_apprepend*)

`static function void add_by_name(string name, uvm_callback cb, uvm_component root, uvm_apprepend ordering = UVM_APPEND)`

Function

add_by_name

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.

Parameters

- **`name`** (`string`)
- **`cb`** (*uvm_callback*)
- **`root`** (*uvm_component*)
- **`ordering`** (*uvm_apprepend*)

`static function void delete(uvm_object obj, uvm_callback cb)`

Function

delete

Deletes the given callback object, *cb* , from the queue associated with the given *obj* handle. The *obj* handle can be *null* , which allows de-registration of callbacks without an object context. The *cb* is the callback handle; it must be non- *null* , and if the callback has already been removed from the object instance then a warning is issued. Note that the CB parameter is optional. For example, the following are equivalent:

```
uvm_callbacks#(my_comp)::delete(comp_a, cb);
uvm_callbacks#(my_comp, my_callback)::delete(comp_a,cb);
```

Parameters

- **`obj`** (*uvm_object*)
- **`cb`** (*uvm_callback*)

`static function void delete_by_name(string name, uvm_callback cb, uvm_component root)`

Function

delete_by_name

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.

Parameters

- **`name`** (`string`)
- **`cb`** (*uvm_callback*)
- **`root`** (*uvm_component*)

`static function CB get_first(int itr, uvm_object obj)`

Function

get_first

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.

Parameters

- **`itr`** (`int`)
- **`obj`** (*uvm_object*)

Return type

CB

`static function CB get_last(int itr, uvm_object obj)`

Function

get_last

Returns the last enabled callback of type CB which resides in the queue for *obj* . If *obj* is *null* then the typewide queue for T is searched. *itr* is the iterator; it will be updated with a value that can be supplied to *get_prev* to get the previous callback object.

If the queue is empty then *null* is returned.

The iterator class *uvm_callback_iter* may be used as an alternative, simplified, iterator interface.

Parameters
- **itr** (*int*)
- **obj** (*uvm_object*)

Return type
CB

`static function CB get_next(int itr, uvm_object obj)`

Function

get_next

Returns the next enabled callback of type CB which resides in the queue for *obj* , using *itr* as the starting point. If *obj* is *null* then the typewide queue for T is searched. *itr* is the iterator; it will be updated with a value that can be supplied to *get_next* to get the next callback object.

If no more callbacks exist in the queue, then *null* is returned. *get_next* will continue to return *null* in this case until *get_first* or *get_last* has been used to reset the iterator.

The iterator class *uvm_callback_iter* may be used as an alternative, simplified, iterator interface.

Parameters
- **itr** (*int*)
- **obj** (*uvm_object*)

Return type
CB

`static function CB get_prev(int itr, uvm_object obj)`

Function

get_prev

Returns the previous enabled callback of type CB which resides in the queue for *obj* , using *itr* as the starting point. If *obj* is *null* then the typewide queue for T is searched. *itr* is the iterator; it will be updated with a value that can be supplied to *get_prev* to get the previous callback object.

If no more callbacks exist in the queue, then *null* is returned. *get_prev* will continue to return *null* in this case until *get_first* or *get_last* has been used to reset the iterator.

The iterator class *uvm_callback_iter* may be used as an alternative, simplified, iterator interface.

Parameters
- **itr** (*int*)
- **obj** (*uvm_object*)

Return type
CB

`static function void display(uvm_object obj = null)`

Function

display

This function displays callback information for *obj* . If *obj* is *null* , then it displays callback information for all objects of type *T* , including typewide callbacks.

Parameters
- **obj** (*uvm_object*)

15.1.1.43 Class uvm_pkg::uvm_callbacks_base

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_callbacks_base*

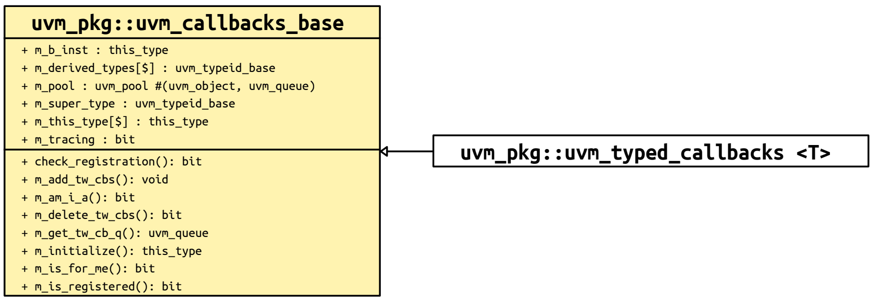

uvm_pkg::uvm_callbacks_base
+ m_b_inst : this_type
+ m_derived_types[$] : uvm_typeid_base
+ m_pool : uvm_pool #(uvm_object, uvm_queue)
+ m_super_type : uvm_typeid_base
+ m_this_type[$] : this_type
+ m_tracing : bit
+ check_registration(): bit
+ m_add_tw_cbs(): void
+ m_am_i_a(): bit
+ m_delete_tw_cbs(): bit
+ m_get_tw_cb_q(): uvm_queue
+ m_initialize(): this_type
+ m_is_for_me(): bit
+ m_is_registered(): bit
uvm_pkg::uvm_typed_callbacks <T>

Fig. 8: Inheritance Diagram of uvm_callbacks_base

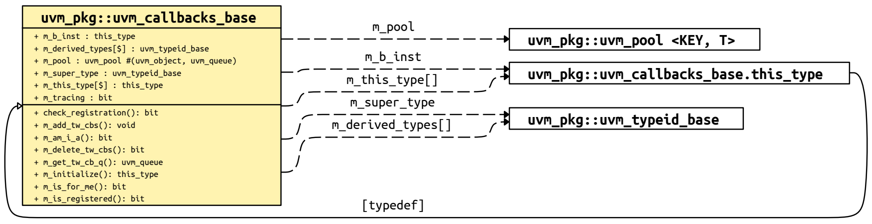

uvm_pkg::uvm_callbacks_base
m_pool
m_b_inst
m_this_type[]
m_super_type
m_derived_types[]
[typedef]
uvm_pkg::uvm_pool <KEY, T>
uvm_pkg::uvm_callbacks_base.this_type
uvm_pkg::uvm_typeid_base

Fig. 9: Collaboration Diagram of uvm_callbacks_base

Class

uvm_callbacks_base

Base class singleton that holds generic queues for all instance specific objects. This is an internal class. This class contains a global pool that has all of the instance specific callback queues in it. All of the typewide callback queues live in the derivative class uvm_typed_callbacks(T). This is not a user visible class.

This class holds the class inheritance hierarchy information (super types and derivative types).

Note, all derivative uvm_callbacks() class singletons access this global m_pool object in order to get access to their specific instance queue.

Table 52: Typedefs

Name	Actual Type	Description
this_type	*uvm_callbacks_base*	

Functions

`function bit check_registration(uvm_object obj, uvm_callback cb)`

Check registration. To test registration, start at this class and work down the class hierarchy. If any class returns true then the pair is legal.

Parameters

obj (*uvm_object*)

cb (*uvm_callback*)

15.1.1.44 Class uvm_pkg::uvm_cause_effect_link

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_link_base*
↪*uvm_pkg* :: *uvm_cause_effect_link*

CLASS

uvm_cause_effect_link

The *uvm_cause_effect_link* is used to represent a Cause/Effect relationship between two objects.

Constructors

`function new(string name = "unnamed-uvm_cause_effect_link")`

Function

new

Constructor

Parameters

name

Instance name

Parameters
`name` (*`string`*)

Functions

`static function uvm_cause_effect_link get_link(uvm_object lhs, uvm_object rhs, string name = "ce_link")`

Function

get_link

Constructs a pre-filled link

This allows for simple one-line link creations.

```
my_db.establish_link(uvm_cause_effect_link::get_link(record1, record2));
```

Parameters:

lhs

Left hand side reference

rhs

Right hand side reference

name

Optional name for the link object

Parameters
`lhs` (*uvm_object*)
`rhs` (*uvm_object*)
`name` (*`string`*)

Return type
uvm_cause_effect_link

`virtual  function void do_set_lhs(uvm_object lhs)`

Function

do_set_lhs

Sets the left-hand-side (Cause)

Parameters

`lhs` (*uvm_object*)

`virtual  function uvm_object do_get_lhs()`

Function

do_get_lhs

Retrieves the left-hand-side (Cause)

Return type

uvm_object

`virtual  function void do_set_rhs(uvm_object rhs)`

Function

do_set_rhs

Sets the right-hand-side (Effect)

Parameters

`rhs` (*uvm_object*)

`virtual  function uvm_object do_get_rhs()`

Function

do_get_rhs

Retrieves the right-hand-side (Effect)

Return type

uvm_object

15.1.1.45 Class uvm_pkg::uvm_check_phase

uvm_pkg :: *uvm_void*
 ↪*uvm_pkg* :: *uvm_object*
  ↪*uvm_pkg* :: *uvm_phase*
   ↪*uvm_pkg* :: *uvm_bottomup_phase*
    ↪*uvm_pkg* :: *uvm_check_phase*

Class

uvm_check_phase

Check for any unexpected conditions in the verification environment.

uvm_bottomup_phase that calls the *uvm_component::check_phase* method.

Upon Entry

- All data has been collected.

Typical Uses

- Check that no unaccounted-for data remain.

Exit Criteria

- Test is known to have passed or failed.

Table 53: Variables

Name	Type	Description
type_name	string	

Functions

`virtual function void exec_func(uvm_component comp, uvm_phase phase)`

Parameters
- **`comp`** (*uvm_component*)
- **`phase`** (*uvm_phase*)

`static function uvm_check_phase get()`

Function

get

Returns the singleton phase handle

Return type
uvm_check_phase

`virtual function string get_type_name()`

15.1.1.46 Class uvm_pkg::uvm_class_clone

CLASS

uvm_class_clone (T)

This policy class is used to clone class objects.

Provides a clone method that returns a copy of the built-in type, T. The class T must implement the clone method, to which this class delegates the operation. If T is derived from *uvm_object*, then T must instead implement *uvm_object::do_copy*, either directly or indirectly through use of the &96;uvm_field macros.

Table 54: Parameters

Name	Default value	Description
T	int	

Functions

`static function uvm_object clone(int from)`

Parameters
: **`from`** (*`int`*)

Return type
: *uvm_object*

15.1.1.47 Class uvm_pkg::uvm_class_comp

CLASS

uvm_class_comp (T)

This policy class is used to compare two objects of the same type.

Provides a comp method that compares two objects of type T. The class T must provide the method "function bit compare(T rhs)", similar to the *uvm_object::compare* method.

Table 55: Parameters

Name	Default value	Description
T	int	

Functions

`static function bit comp(int a, int b)`

Parameters
- **a** (*int*)
- **b** (*int*)

15.1.1.48 Class uvm_pkg::uvm_class_converter

CLASS

uvm_class_converter (T)

This policy class is used to convert a class object to a string.

Provides a convert2string method that converts an instance of type T to a string. The class T must provide the method "function string convert2string()", similar to the *uvm_object::convert2string* method.

Table 56: Parameters

Name	Default value	Description
T	int	

Functions

`static function string convert2string(int t)`

Parameters
- **`t`** (*`int`*)

15.1.1.49 Class uvm_pkg::uvm_class_pair

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_class_pair*

Class

uvm_class_pair (T1, T2)

Container holding handles to two objects whose types are specified by the type parameters, T1 and T2.

Table 57: Parameters

Name	Default value	Description
T1	int	
T2	T1	

Table 58: Variables

Name	Type	Description
type_name	string	
first	int	***Variable*** T1 first The handle to the first object in the pair
second	int	***Variable*** T2 second The handle to the second object in the pair

Table 59: Typedefs

Name	Actual Type	Description
this_type	*uvm_class_pair#(T1, T2)*	

Constructors

```
function  new(string name = "", int f = null, int s = null)
```

Function

new

Creates an instance that holds a handle to two objects. The optional name argument gives a name to the new pair object.

Parameters

- **name** (*string*)
- **f** (*int*)
- **s** (*int*)

Functions

`virtual function string get_type_name()`

`virtual function string convert2string()`

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

Parameters
- **`rhs`** (*uvm_object*)
- **`comparer`** (*uvm_comparer*)

`virtual function void do_copy(uvm_object rhs)`

Parameters
- **`rhs`** (*uvm_object*)

15.1.1.50 Class uvm_pkg::uvm_cmd_line_verb

Table 60: Variables

Name	Type	Description
comp_path	string	
id	string	
verb	*uvm_verbosity*	
exec_time	int	

15.1.1.51 Class uvm_pkg::uvm_cmdline_processor

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_cmdline_processor*

Class

uvm_cmdline_processor

This class provides an interface to the command line arguments that were provided for the given simulation. The class is intended to be used as a singleton, but that isn't required. The generation of the data structures which hold the command line argument information happens during construction of the class object. A global variable called *uvm_cmdline_proc* is created at initialization time and may be used to access command line information.

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

Constructors

`function new(string name = "")`

constructor

Parameters

`name` (*`string`*)

Functions

`static function uvm_cmdline_processor get_inst()`

Function

get_inst

Returns the singleton instance of the UVM command line processor.

Return type

uvm_cmdline_processor

`function void get_args(string args)`

Function

get_args

This function returns a queue with all of the command line arguments that were used to start the simulation. Note that element 0 of the array will always be the name of the executable which started the simulation.

Parameters

`args` (*`string`*)

`function void get_plusargs(string args)`

Function

get_plusargs

This function returns a queue with all of the plus arguments that were used to start the simulation. Plusarguments may be used by the simulator vendor, or may be specific to a company or individual user. Plusargs never have extra arguments (i.e. if there is a plusarg as the second argument on the command line, the third argument is unrelated); this is not necessarily the case with vendor specific dash arguments.

Parameters

`args` (*`string`*)

`function void get_uvm_args(string args)`

Function

get_uvmargs

This function returns a queue with all of the uvm arguments that were used to start the simulation. A UVM argument is

taken to be any argument that starts with a

or + and uses the keyword UVM (case insensitive) as the first three letters of the argument.

Parameters

args (*string*)

`function int get_arg_matches(string match, string args)`

Function

get_arg_matches

This function loads a queue with all of the arguments that match the input expression and returns the number of items that matched. If the input expression is bracketed with //, then it is taken as an extended regular expression otherwise, it is taken as the beginning of an argument to match. For example:

```
string myargs[$]
initial begin
   void'(uvm_cmdline_proc.get_arg_matches("+foo",myargs)); //matches +foo,␣
↪+foobar
                                                           //doesn't match␣
↪+barfoo
   void'(uvm_cmdline_proc.get_arg_matches("/foo/",myargs)); //matches +foo,␣
↪+foobar,
                                                            //foo.sv, barfoo,␣
↪etc.
   void'(uvm_cmdline_proc.get_arg_matches("/^foo.*\.sv",myargs)); //matches␣
↪foo.sv
                                                                  //and foo123.
↪sv,
                                                                  //not barfoo.
↪sv.
```

Parameters

match (*string*)

args (*string*)

`function int get_arg_value(string match, string value)`

Function

get_arg_value

This function finds the first argument which matches the *match* arg and returns the suffix of the argument. This is similar to the $value$plusargs system task, but does not take a formatting string. The return value is the number of command line arguments that match the *match* string, and *value* is the value of the first match.

Parameters

match (*string*)

value (*string*)

`function int get_arg_values(string match, string values)`

Function

get_arg_values

This function finds all the arguments which matches the *match* arg and returns the suffix of the arguments in a list of values. The return value is the number of matches that were found (it is the same as values.size()). For example if '+foo = 1, yes, on +foo = 5, no, off' was provided on the command line and the following code was executed:

```
string foo_values[$]
initial begin
   void'(uvm_cmdline_proc.get_arg_values("+foo=",foo_values));
```

The foo_values queue would contain two entries. These entries are shown here:

0

"1, yes, on"

1

"5, no, off"

Splitting the resultant string is left to user but using the uvm_split_string() function is recommended.

Parameters

`match` (*`string`*)

`values` (*`string`*)

`function string get_tool_name()`

Function

get_tool_name

Returns the simulation tool that is executing the simulation. This is a vendor specific string.

`function string get_tool_version()`

Function

get_tool_version

Returns the version of the simulation tool that is executing the simulation. This is a vendor specific string.

15.1.1.52 Class uvm_pkg::uvm_comparer

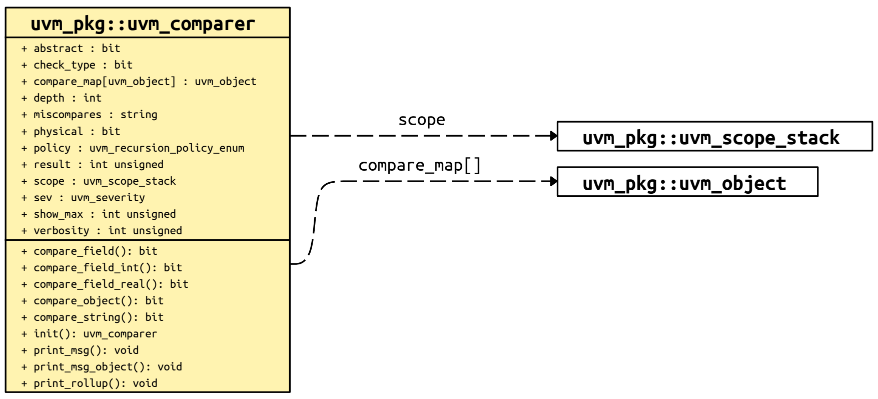


Fig. 10: Collaboration Diagram of uvm_comparer

CLASS

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 a uvm_comparer policy object.

Table 61: Variables

Name	Type	Description
policy	*uvm_recursion_policy_enum*	***Variable*** policy Determines whether comparison is UVM_DEEP, UVM_REFERENCE, or UVM_SHALLOW.
show_max	int unsigned	***Variable*** show_max Sets the maximum number of messages to send to the printer for miscompares of an object.
verbosity	int unsigned	***Variable*** verbosity 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*	***Variable*** sev Sets the severity for printed messages. The severity setting is used by the messaging mechanism for printing and filtering messages.

continues on next page

Table 61 – continued from previous page

Name	Type	Description
miscompares	string	***Variable*** 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	***Variable*** physical 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	***Variable*** abstract 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	***Variable*** 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. 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	***Variable*** result 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.
depth	int	Current depth of objects
compare_map	*uvm_object*	
scope	*uvm_scope_stack*	

Functions

`virtual function bit compare_field(string name, uvm_bitstream_t lhs, uvm_-bitstream_t rhs, int size, uvm_radix_enum radix = UVM_NORADIX)`

Function

compare_field

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.

Parameters

- **name** (`string`)
- **lhs** (*uvm_bitstream_t*)
- **rhs** (*uvm_bitstream_t*)
- **size** (`int`)
- **radix** (*uvm_radix_enum*)

`virtual function bit compare_field_int(string name, uvm_integral_t lhs, uvm_-integral_t rhs, int size, uvm_radix_enum radix = UVM_NORADIX)`

Function

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. It is automatically called by *compare_field* if the operand size is less than or equal to 64.

Parameters

- **name** (`string`)
- **lhs** (*uvm_integral_t*)
- **rhs** (*uvm_integral_t*)
- **size** (`int`)
- **radix** (*uvm_radix_enum*)

`virtual function bit compare_field_real(string name, real lhs, real rhs)`

Function

compare_field_real

This method is the same as *compare_field* except that the arguments are real numbers.

Parameters

- **name** (`string`)
- **lhs** (`real`)
- **rhs** (`real`)

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

Function

compare_object

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()*).

Parameters

- **name** (`string`)
- **lhs** (*uvm_object*)
- **rhs** (*uvm_object*)

`virtual function bit compare_string(string name, string lhs, string rhs)`

Function

compare_string

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.

Parameters
- **`name`** (*`string`*)
- **`lhs`** (*`string`*)
- **`rhs`** (*`string`*)

`function void print_msg(string msg)`

Function

print_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.

Parameters
- **`msg`** (*`string`*)

`function void print_rollup(uvm_object rhs, uvm_object lhs)`

Need this function because sformat doesn't support objects

Parameters
- **`rhs`** (*uvm_object*)
- **`lhs`** (*uvm_object*)

`function void print_msg_object(uvm_object lhs, uvm_object rhs)`

print_msg_object

Parameters
- **`lhs`** (*uvm_object*)
- **`rhs`** (*uvm_object*)

`static function uvm_comparer init()`

init ??

Return type

uvm_comparer

15.1.1.53 Class uvm_pkg::uvm_component

uvm_pkg :: *uvm_void*
  ↪*uvm_pkg* :: *uvm_object*
    ↪*uvm_pkg* :: *uvm_report_object*
      ↪*uvm_pkg* :: *uvm_component*

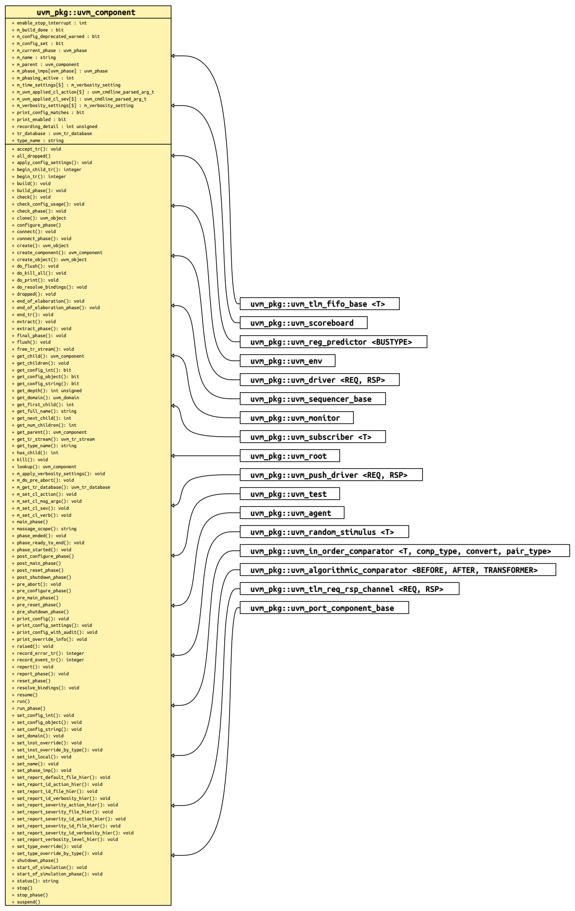

uvm_pkg::uvm_component
uvm_pkg::uvm_tlm_fifo_base <T>
uvm_pkg::uvm_scoreboard
uvm_pkg::uvm_reg_predictor <BUSTYPE>
uvm_pkg::uvm_env
uvm_pkg::uvm_driver <REQ, RSP>
uvm_pkg::uvm_sequencer_base
uvm_pkg::uvm_monitor
uvm_pkg::uvm_subscriber <T>
uvm_pkg::uvm_root
uvm_pkg::uvm_push_driver <REQ, RSP>
uvm_pkg::uvm_test
uvm_pkg::uvm_agent
uvm_pkg::uvm_random_stimulus <T>
uvm_pkg::uvm_in_order_comparator <T, comp_type, convert, pair_type>
uvm_pkg::uvm_algorithmic_comparator <BEFORE, AFTER, TRANSFORMER>
uvm_pkg::uvm_tlm_req_rsp_channel <REQ, RSP>
uvm_pkg::uvm_port_component_base

Fig. 11: Inheritance Diagram of uvm_component

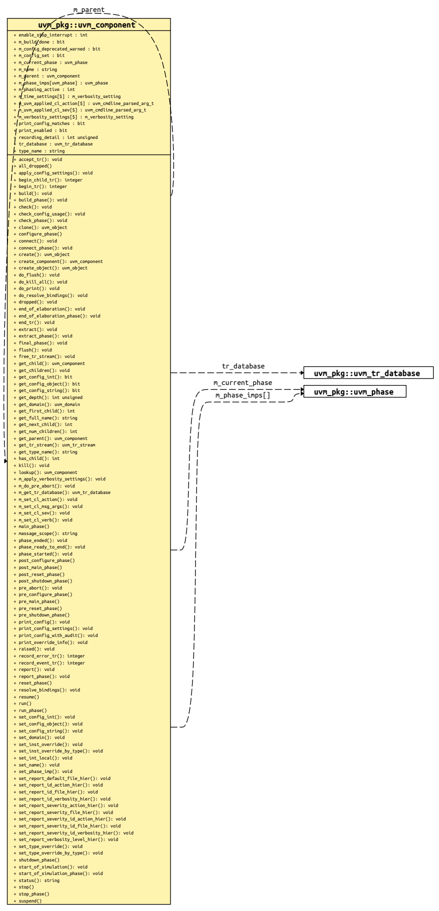

m_parent
uvm_pkg::uvm_component
+ enable_stop_interrupt : int
+ m_build_done : bit
+ m_config_deprecated_warned : bit
+ m_config_set : bit
+ m_current_phase : uvm_phase
+ m_name : string
+ m_parent : uvm_component
+ m_phase_imps[uvm_phase] : uvm_phase
+ m_phasing_active : int
+ m_time_settings[$] : m_verbosity_setting
+ m_uvm_applied_cl_action[$] : uvm_cmdline_parsed_arg_t
+ m_uvm_applied_cl_sev[$] : uvm_cmdline_parsed_arg_t
+ m_verbosity_settings[$] : m_verbosity_setting
+ print_config_matches : bit
+ print_enabled : bit
+ recording_detail : int unsigned
+ tr_database : uvm_tr_database
+ type_name : string
+ accept_tr(): void
+ all_dropped()
+ apply_config_settings(): void
+ begin_child_tr(): integer
+ begin_tr(): integer
+ build(): void
+ build_phase(): void
+ check(): void
+ check_config_usage(): void
+ check_phase(): void
+ clone(): uvm_object
+ configure_phase()
+ connect(): void
+ connect_phase(): void
+ create(): uvm_object
+ create_component(): uvm_component
+ create_object(): uvm_object
+ do_flush(): void
+ do_kill_all(): void
+ do_print(): void
+ do_resolve_bindings(): void
+ dropped(): void
+ end_of_elaboration(): void
+ end_of_elaboration_phase(): void
+ end_tr(): void
+ extract(): void
+ extract_phase(): void
+ final_phase(): void
+ flush(): void
+ free_tr_stream(): void
+ get_child(): uvm_component
+ get_children(): void
+ get_config_int(): bit
+ get_config_object(): bit
+ get_config_string(): bit
+ get_depth(): int unsigned
+ get_domain(): uvm_domain
+ get_first_child(): int
+ get_full_name(): string
+ get_next_child(): int
+ get_num_children(): int
+ get_parent(): uvm_component
+ get_tr_stream(): uvm_tr_stream
+ get_type_name(): string
+ has_child(): int
+ kill(): void
+ lookup(): uvm_component
+ m_apply_verbosity_settings(): void
+ m_do_pre_abort(): void
+ m_get_tr_database(): uvm_tr_database
+ m_set_cl_action(): void
+ m_set_cl_msg_args(): void
+ m_set_cl_sev(): void
+ m_set_cl_verb(): void
+ main_phase()
+ massage_scope(): string
+ phase_ended(): void
+ phase_ready_to_end(): void
+ phase_started(): void
+ post_configure_phase()
+ post_main_phase()
+ post_reset_phase()
+ post_shutdown_phase()
+ pre_abort(): void
+ pre_configure_phase()
+ pre_main_phase()
+ pre_reset_phase()
+ pre_shutdown_phase()
+ print_config(): void
+ print_config_settings(): void
+ print_config_with_audit(): void
+ print_override_info(): void
+ raised(): void
+ record_error_tr(): integer
+ record_event_tr(): integer
+ report(): void
+ report_phase(): void
+ reset_phase()
+ resolve_bindings(): void
+ resume()
+ run()
+ run_phase()
+ set_config_int(): void
+ set_config_object(): void
+ set_config_string(): void
+ set_domain(): void
+ set_inst_override(): void
+ set_inst_override_by_type(): void
+ set_int_local(): void
+ set_name(): void
+ set_phase_imp(): void
+ set_report_default_file_hier(): void
+ set_report_id_action_hier(): void
+ set_report_id_file_hier(): void
+ set_report_id_verbosity_hier(): void
+ set_report_severity_action_hier(): void
+ set_report_severity_file_hier(): void
+ set_report_severity_id_action_hier(): void
+ set_report_severity_id_file_hier(): void
+ set_report_severity_id_verbosity_hier(): void
+ set_report_verbosity_level_hier(): void
+ set_type_override(): void
+ set_type_override_by_type(): void
+ shutdown_phase()
+ start_of_simulation(): void
+ start_of_simulation_phase(): void
+ status(): string
+ stop()
+ stop_phase()
+ suspend()
tr_database
uvm_pkg::uvm_tr_database
m_current_phase
m_phase_imps[]
uvm_pkg::uvm_phase

Fig. 12: Collaboration Diagram of uvm_component

CLASS

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:

Hierarchy

provides methods for searching and traversing the component hierarchy.

Phasing

defines a phased test flow that all components follow, with a group of standard phase methods and an API for custom phases and multiple independent phasing domains to mirror DUT behavior e.g. power

Reporting

provides a convenience interface to the *uvm_report_handler*. All messages, warnings, and errors are processed through this interface.

Transaction recording

provides methods for recording the transactions produced or consumed by the component to a transaction database (vendor specific).

Factory

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.

Table 62: Variables

Name	Type	Description
enable_stop_interrupt	int	Variable- enable_stop_interrupt - **DEPRECATED** 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.
print_config_matches	bit	***Variable*** print_config_matches Setting this static variable causes uvm_config_db::get() to print info about matching configuration settings as they are being applied.
print_enabled	bit	***Variable*** print_enabled 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.
tr_database	*uvm_tr_database*	***Variable*** tr_database Specifies the *uvm_tr_database* object to use for *begin_tr* and other methods in the <Recording Interface>. Default is *uvm_coreservice_t::get_default_tr_database*.
type_name	string	
recording_detail	int unsigned	

Constructors

`function new(string name, uvm_component parent)`

Function

new

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). New
Parameters

name (*string*)
parent (*uvm_component*)

Structs

typedef struct uvm_cmdline_parsed_arg_t

Functions

virtual function uvm_component get_parent()

Function

get_parent

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

Return type
uvm_component

virtual function string get_full_name()

Function

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_full_name

function void get_children(uvm_component children)

Function

get_children

This function populates the end of the *children* array with the list of this component's children.

```
uvm_component array[$];
my_comp.get_children(array);
foreach(array[i])
  do_something(array[i]);. Get_children
```

Parameters
children (*uvm_component*)

function uvm_component get_child(string name)

Function

get_child. Get_child

Parameters
name (*string*)

Return type
uvm_component

function int get_next_child(string name)

Function

get_next_child. Get_next_child

Parameters
name (*string*)

function int get_first_child(string name)

Function

get_first_child

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_first_child
```

Parameters

name (*string*)

function int get_num_children()

Function

get_num_children

Returns the number of this component's children. Get_num_children

function int has_child(string name)

Function

has_child

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

Parameters

name (*string*)

virtual function void set_name(string name)

Function

set_name

Renames this component to *name* and recalculates all descendants' full names. This is an internal function for now. Set_name

Parameters

name (*string*)

function uvm_component lookup(string name)

Function

lookup

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. Lookup

Parameters

name (*string*)

Return type

uvm_component

function int unsigned get_depth()

Function

get_depth

Returns the component's depth from the root level. uvm_top has a depth of 0. The test and any other top level components have a depth of 1, and so on. Get_depth

virtual function void build_phase(uvm_phase phase)

Function

build_phase

The *uvm_build_phase* phase implementation method.

Any override should call super.build_phase(phase) to execute the automatic configuration of fields registered in the component by calling *apply_config_settings*. To turn off automatic configuration for a component, do not call super.build_phase(phase).

This method should never be called directly. Phase methods these are prototypes for the methods to be implemented in user components build_phase() has a default implementation, the others have an empty default

Parameters

phase (*uvm_phase*)

virtual function void build()

For backward compatibility the base *build_phase* method calls *build*. Backward compatibility build function

virtual function void connect_phase(uvm_phase phase)

Function

connect_phase

The *uvm_connect_phase* phase implementation method.

This method should never be called directly. These phase methods are common to all components in UVM. For backward compatibility, they call the old style name (without the _phse)

Parameters

phase (*uvm_phase*)

virtual function void connect()

For backward compatibility the base connect_phase method calls connect. These are the old style phase names. In order for runtime phase names to not conflict with user names, the _phase postfix was added.

virtual function void end_of_elaboration_phase(uvm_phase phase)

Function

end_of_elaboration_phase

The *uvm_end_of_elaboration_phase* phase implementation method.

This method should never be called directly.

Parameters

phase (*uvm_phase*)

virtual function void end_of_elaboration()

For backward compatibility the base *end_of_elaboration_phase* method calls *end_of_elaboration*.

virtual function void start_of_simulation_phase(uvm_phase phase)

Function

start_of_simulation_phase

The *uvm_start_of_simulation_phase* phase implementation method.

This method should never be called directly.

Parameters

phase (*uvm_phase*)

virtual function void start_of_simulation()

For backward compatibility the base *start_of_simulation_phase* method calls *start_of_simulation*.

virtual function void extract_phase(uvm_phase phase)

Function

extract_phase

The *uvm_extract_phase* phase implementation method.

This method should never be called directly.

Parameters

phase (*uvm_phase*)

virtual function void extract()

For backward compatibility the base extract_phase method calls extract.

virtual function void check_phase(uvm_phase phase)

Function

check_phase

The *uvm_check_phase* phase implementation method.

This method should never be called directly.

Parameters

phase (*uvm_phase*)

`virtual function void check()`

For backward compatibility the base check_phase method calls check.

`virtual function void report_phase(uvm_phase phase)`

Function

report_phase

The *uvm_report_phase* phase implementation method.

This method should never be called directly.

Parameters

`phase` (*uvm_phase*)

`virtual function void report()`

For backward compatibility the base report_phase method calls report.

`virtual function void final_phase(uvm_phase phase)`

Function

final_phase

The *uvm_final_phase* phase implementation method.

This method should never be called directly.

Parameters

`phase` (*uvm_phase*)

`virtual function void phase_started(uvm_phase phase)`

Function

phase_started

Invoked at the start of each phase. The *phase* argument specifies the phase being started. Any threads spawned in this callback are not affected when the phase ends. Phase_started

phase_started() and phase_ended() are extra callbacks called at the beginning and end of each phase, respectively. Since they are called for all phases the phase is passed in as an argument so the extender can decide what to do, if anything, for each phase.

Parameters

`phase` (*uvm_phase*)

`virtual function void phase_ready_to_end(uvm_phase phase)`

Function

phase_ready_to_end

Invoked when all objections to ending the given *phase* and all sibling phases have been dropped, thus indicating that *phase* is ready to begin a clean exit. Sibling phases are any phases that have a common successor phase in the schedule plus any phases that sync'd to the current phase. Components needing to consume delta cycles or advance time to perform a clean exit from the phase may raise the phase's objection.

```
phase.raise_objection(this,"Reason");
```

It is the responsibility of this component to drop the objection once it is ready for this phase to end (and processes killed). If no objection to the given *phase* or sibling phases are raised, then phase_ended() is called after a delta cycle. If any objection is raised, then when all objections to ending the given *phase* and siblings are dropped, another iteration of phase_ready_to_end is called. To prevent endless iterations due to coding error, after 20 iterations, phase_ended() is called regardless of whether previous iteration had any objections raised. Phase_ready_to_end

Parameters

`phase` (*uvm_phase*)

`virtual function void phase_ended(uvm_phase phase)`

Function

phase_ended

Invoked at the end of each phase. The *phase* argument specifies the phase that is ending. Any threads spawned in this callback are not affected when the phase ends. Phase_ended

Parameters

`phase` (*uvm_phase*)

`function void set_domain(uvm_domain domain, int hier = 1)`

Function

set_domain

Apply a phase domain to this component and, if *hier* is set, recursively to all its children.

Calls the virtual define_domain method, which derived components can override to augment or replace the domain definition of its base class. Set_domain

assigns this component [tree] to a domain. adds required schedules into graph If called from build, *hier* won't recurse into all chilren (which don't exist yet) If we have components inherit their parent's domain by default, then *hier* isn't needed and we need a way to prevent children from inheriting this component's domain

Parameters

- **`domain`** (*uvm_domain*)
- **`hier`** (`int`)

`function uvm_domain get_domain()`

Function

get_domain

Return handle to the phase domain set on this component. Get_domain

Return type

uvm_domain

`function void set_phase_imp(uvm_phase phase, uvm_phase imp, int hier = 1)`

Function

set_phase_imp

Override the default implementation for a phase on this component (tree) with a custom one, which must be created as a singleton object extending the default one and implementing required behavior in exec and traverse methods

The *hier* specifies whether to apply the custom functor to the whole tree or just this component. Set_phase_imp

Parameters

- **`phase`** (*uvm_phase*)
- **`imp`** (*uvm_phase*)
- **`hier`** (`int`)

`function string status()`

`virtual function void kill()`

Function- kill - **DEPRECATED**

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

`virtual function void do_kill_all()`

Function- do_kill_all - **DEPRECATED**

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

`virtual function void resolve_bindings()`

Function

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. Resolve_bindings

`function string massage_scope(string scope)`

Parameters

- **`scope`** (`string`)

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

Function- set_config_int. Set_config_int

Parameters

- **inst_name** (*string*)
- **field_name** (*string*)
- **value** (*uvm_bitstream_t*)

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

Function- set_config_string. Set_config_string

Parameters

- **inst_name** (*string*)
- **field_name** (*string*)
- **value** (*string*)

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

Set_config_object

Parameters

- **inst_name** (*string*)
- **field_name** (*string*)
- **value** (*uvm_object*)
- **clone** (*bit*)

virtual function bit get_config_int(string field_name, uvm_bitstream_t value)

Function- get_config_int. Get_config_int

Parameters

- **field_name** (*string*)
- **value** (*uvm_bitstream_t*)

virtual function bit get_config_string(string field_name, string value)

Function- get_config_string. Get_config_string

Parameters

- **field_name** (*string*)
- **value** (*string*)

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

Function- get_config_object

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 a 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;

```

(continues on next page)

(continued from previous page)

```
  function void build_phase(uvm_phase phase);
    uvm_object tmp;
    super.build_phase(phase);
    if(get_config_object("data", tmp))
      if (!$cast(data, tmp))
        `uvm_error("CFGERR","error! config setting for 'data' not of type␣
↪myobj_t")
      endfunction
    ...
```

The above example overrides the *build_phase* method. If you want to retain any base functionality, you must call super.build_phase(uvm_phase phase).

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

See Members for information on setting the global configuration table. Get_config_object

Note that this does not honor the set_config_object clone bit

Parameters

field_name (*string*)

value (*uvm_object*)

clone (*bit*)

function void check_config_usage(bit recurse = 1)

Function

check_config_usage

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.

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

```
function void start_of_simulation_phase(uvm_phase phase);
  check_config_usage();
endfunction. Check_config_usage
```

Parameters

recurse (*bit*)

virtual function void apply_config_settings(bit verbose = 0)

Function

apply_config_settings

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

This function is called by *uvm_component::build_phase*.

The apply_config_settings method determines all the configuration settings targeting this component and calls the appropriate set*local method to set each one. To work, you must override one or more set*local methods to accommodate setting of your component's specific properties. Any properties registered with the optional &96;uvm*field macros do not require special handling by the set*local methods; the macros provide the set*local functionality for you.

If you do not want apply_config_settings to be called for a component, then the build_phase() method should be overloaded and you should not call super.build_phase(phase). Likewise, apply_config_settings can be overloaded to customize automated configuration.

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

Parameters

`verbose` (*`bit`*)

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

Function

print_config_settings

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

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

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

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

This function has been deprecated. Use print_config instead. Print_config_settings

Parameters

`field` (*`string`*)

`comp` (*uvm_component*)

`recurse` (*`bit`*)

`function void print_config(bit recurse = 0, bit audit = 0)`

Function

print_config

Print_config_settings prints all configuration information for this component, as set by previous calls to *uvm_config_db::set()* and exports to the resources pool. The settings are printing in the order of their precedence.

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

if *audit* is set then the audit trail for each resource is printed along with the resource name and value. Print_config

Parameters

`recurse` (*`bit`*)

`audit` (*`bit`*)

`function void print_config_with_audit(bit recurse = 0)`

Function

print_config_with_audit

Operates the same as print_config except that the audit bit is forced to 1. This interface makes user code a bit more readable as it avoids multiple arbitrary bit settings in the argument list.

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

Parameters

`recurse` (*`bit`*)

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

Function

raised

The *raised* callback is called when this or a descendant of this component instance raises the specified *objection* . The *source_obj* is the object that originally raised the objection. The *description* is optionally provided by the *source_obj* to give a reason for raising the objection. The *count* indicates the number of objections raised by the *source_obj* .

Parameters

`objection` (*uvm_objection*)

`source_obj` (*uvm_object*)

`description` (*`string`*)

`count` (*`int`*)

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

Function

dropped

The *dropped* callback is called when this or a descendant of this component instance drops the specified *objection* . The *source_obj* is the object that originally dropped the objection. The *description* is optionally provided by the *source_obj* to give a reason for dropping the objection. The *count* indicates the number of objections dropped by the *source_obj* .

Parameters
- **objection** (*uvm_objection*)
- **source_obj** (*uvm_object*)
- **description** (*string*)
- **count** (*int*)

function uvm_component create_component(string requested_type_name, string name)

Function

create_component

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

Parameters
- **requested_type_name** (*string*)
- **name** (*string*)

Return type
- *uvm_component*

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

Function

create_object

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

Parameters
- **requested_type_name** (*string*)
- **name** (*string*)

Return type
- *uvm_object*

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

Function

set_type_override_by_type

A convenience function for *uvm_factory::set_type_override_by_type*, this method registers a factory override for components and objects created at this level of hierarchy or below. 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_type_override_by_type (static)

Parameters

original_type (*uvm_object_wrapper*)
override_type (*uvm_object_wrapper*)
replace (`bit`)

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

Function

set_inst_override_by_type

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

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

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

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

If you are employing the &96;uvm*utils macros, the typedef and the get_type method will be implemented for you. For details on the utils macros refer to <Utility and Field Macros for Components and Objects>.

The following example shows &96;uvm*utils usage:

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

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

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

Parameters

relative_inst_path (*string*)
original_type (*uvm_object_wrapper*)
override_type (*uvm_object_wrapper*)

static function void set_type_override(string original_type_name, string override_-type_name, bit replace = 1)

Function

set_type_override

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

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

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

Parameters

original_type_name (*string*)
override_type_name (*string*)
replace (*bit*)

function void set_inst_override(string relative_inst_path, string original_type_-name, string override_type_name)

Function

set_inst_override

A convenience function for *uvm_factory::set_inst_override_by_name*, 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(original_type_name,
                                  override_type_name,
                                  {get_full_name(),".",
                                   relative_inst_path}
                                   );
```

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

Parameters

relative_inst_path (*string*)
original_type_name (*string*)
override_type_name (*string*)

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

Function

print_override_info

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

Parameters

requested_type_name (*string*)
name (*string*)

function void set_report_id_verbosity_hier(string id, int verbosity)

Function

set_report_id_verbosity_hier. Set_report_id_verbosity_hier

Parameters

id (*string*)
verbosity (*int*)

function void set_report_severity_id_verbosity_hier(uvm_severity severity, string id, int verbosity)

Function

set_report_severity_id_verbosity_hier

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

For a list of severities and their default verbosities, refer to *uvm_report_handler*. Set_report_severity_id_verbosity_hier

Parameters

severity (*uvm_severity*)
id (*string*)
verbosity (*int*)

function void set_report_severity_action_hier(uvm_severity severity, uvm_action action)

Function

set_report_severity_action_hier. Set_report_severity_action_hier

Parameters

severity (*uvm_severity*)
action (*uvm_action*)

function void set_report_id_action_hier(string id, uvm_action action)

Function

set_report_id_action_hier. Set_report_id_action_hier

Parameters

id (*string*)
action (*uvm_action*)

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

Function

set_report_severity_id_action_hier

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_severity_id_action_hier

Parameters

severity (*uvm_severity*)
id (*string*)
action (*uvm_action*)

function void set_report_default_file_hier(UVM_FILE file)

Function

set_report_default_file_hier. Set_report_default_file_hier

Parameters

file (*UVM_FILE*)

function void set_report_severity_file_hier(uvm_severity severity, UVM_FILE file)

Function

set_report_severity_file_hier. Set_report_severity_file_hier

Parameters

severity (*uvm_severity*)
file (*UVM_FILE*)

function void set_report_id_file_hier(string id, UVM_FILE file)

Function

set_report_id_file_hier. Set_report_id_file_hier
Parameters
id (*string*)
file (*UVM_FILE*)

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

Function

set_report_severity_id_file_hier

These methods recursively associate the specified FILE descriptor with reports of the given *severity* , *id* , or *severity-id* pair. 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_severity_id_file_hier
Parameters
severity (*uvm_severity*)
id (*string*)
file (*UVM_FILE*)

function void set_report_verbosity_level_hier(int verbosity)

Function

set_report_verbosity_level_hier

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. Set_report_verbosity_level_hier
Parameters
verbosity (*int*)

virtual function void pre_abort()

Function

pre_abort

This callback is executed when the message system is executing a <UVM_EXIT> action. The exit action causes an immediate termination of the simulation, but the pre_abort callback hook gives components an opportunity to provide additional information to the user before the termination happens. For example, a test may want to executed the report function of a particular component even when an error condition has happened to force a premature termination you would write a function like:

```
function void mycomponent::pre_abort();
  report();
endfunction
```

The pre_abort() callback hooks are called in a bottom-up fashion.

function void accept_tr(uvm_transaction tr, time accept_time = 0)

Function

accept_tr

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. Accept_tr

Parameters

`tr` (*uvm_transaction*)

`accept_time` (`time`)

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

Function

begin_tr

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_tr

Parameters

`tr` (*uvm_transaction*)

`stream_name` (`string`)

`label` (`string`)

`desc` (`string`)

`begin_time` (`time`)

`parent_handle` (`integer`)

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

Function

begin_child_tr

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. Begin_child_tr

Parameters

`tr` (*uvm_transaction*)

`parent_handle` (`integer`)

`stream_name` (`string`)

`label` (`string`)

`desc` (`string`)

`begin_time` (`time`)

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

Function

end_tr

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 &96;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. End_tr

Parameters

tr (*uvm_transaction*)
end_time (*time*)
free_handle (*bit*)

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)

Function

record_error_tr

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

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

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

Parameters

stream_name (*string*)
info (*uvm_object*)
label (*string*)
desc (*string*)
error_time (*time*)
keep_active (*bit*)

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)

Function

record_event_tr

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. Record_event_tr

Parameters

stream_name (*string*)
info (*uvm_object*)
label (*string*)
desc (*string*)
event_time (*time*)
keep_active (*bit*)

virtual function uvm_tr_stream get_tr_stream(string name, string stream_type_-name = "")

Function

get_tr_stream

Returns a tr stream with *this* component's full name as a scope.

Streams which are retrieved via this method will be stored internally, such that later calls to *get_tr_stream* will return the same stream reference.

The stream can be removed from the internal storage via a call to *free_tr_stream*.

Parameters

name

Name for the stream

stream_type_name

Type name for the stream (Default = ""). Get_tr_stream

Parameters

name (*string*)

stream_type_name (*string*)

Return type

uvm_tr_stream

virtual function void free_tr_stream(uvm_tr_stream stream)

Function

free_tr_stream

Frees the internal references associated with *stream* .

The next call to *get_tr_stream* will result in a newly created *uvm_tr_stream*. If the current stream is open (or closed), then it will be freed. Free_tr_stream

Parameters

stream (*uvm_tr_stream*)

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

Set_int_local (override)

Parameters

field_name (*string*)

value (*uvm_bitstream_t*)

recurse (*bit*)

function void do_resolve_bindings()

Do_resolve_bindings

function void do_flush()

Do_flush (flush_hier?)

virtual function void flush()

Flush

virtual function uvm_object create(string name = "")

Overridden to disable. Create

Parameters

name (*string*)

Return type

uvm_object

virtual function uvm_object clone()

Clone

Return type

uvm_object

virtual function string get_type_name()

virtual function void do_print(uvm_printer printer)

Do_print (override)

Parameters

printer (*uvm_printer*)

Tasks

`virtual function run_phase(uvm_phase phase)`

Task

run_phase

The *uvm_run_phase* phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. Thus the phase will automatically end once all objections are dropped using *phase.drop_objection()* .

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

The run_phase task should never be called directly.

Parameters

`phase` (*uvm_phase*)

`virtual function run()`

For backward compatibility the base *run_phase* method calls *run*.

`virtual function pre_reset_phase(uvm_phase phase)`

Task

pre_reset_phase

The *uvm_pre_reset_phase* phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()* , or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly. These runtime phase methods are only called if a set_domain() is done

Parameters

`phase` (*uvm_phase*)

`virtual function reset_phase(uvm_phase phase)`

Task

reset_phase

The *uvm_reset_phase* phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()* , or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

Parameters

`phase` (*uvm_phase*)

`virtual function post_reset_phase(uvm_phase phase)`

Task

post_reset_phase

The *uvm_post_reset_phase* phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()* , or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

Parameters
phase (*uvm_phase*)

`virtual function pre_configure_phase(uvm_phase phase)`

Task

pre_configure_phase

The *uvm_pre_configure_phase* phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()* , or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.
Parameters
phase (*uvm_phase*)

`virtual function configure_phase(uvm_phase phase)`

Task

configure_phase

The *uvm_configure_phase* phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()* , or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.
Parameters
phase (*uvm_phase*)

`virtual function post_configure_phase(uvm_phase phase)`

Task

post_configure_phase

The *uvm_post_configure_phase* phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()* , or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.
Parameters
phase (*uvm_phase*)

`virtual function pre_main_phase(uvm_phase phase)`

Task

pre_main_phase

The *uvm_pre_main_phase* phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()* , or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.
Parameters
phase (*uvm_phase*)

`virtual function main_phase(uvm_phase phase)`

Task

main_phase

The *uvm_main_phase* phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()* , or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

Parameters

`phase` (*uvm_phase*)

`virtual function post_main_phase(uvm_phase phase)`

Task

post_main_phase

The *uvm_post_main_phase* phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()* , or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

Parameters

`phase` (*uvm_phase*)

`virtual function pre_shutdown_phase(uvm_phase phase)`

Task

pre_shutdown_phase

The *uvm_pre_shutdown_phase* phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()* , or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

Parameters

`phase` (*uvm_phase*)

`virtual function shutdown_phase(uvm_phase phase)`

Task

shutdown_phase

The *uvm_shutdown_phase* phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()* , or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

Parameters

`phase` (*uvm_phase*)

`virtual function post_shutdown_phase(uvm_phase phase)`

Task

post_shutdown_phase

The *uvm_post_shutdown_phase* phase implementation method.

This task returning or not does not indicate the end or persistence of this phase. It is necessary to raise an objection using *phase.raise_objection()* to cause the phase to persist. Once all components have dropped their respective objection using *phase.drop_objection()* , or if no components raises an objection, the phase is ended.

Any processes forked by this task continue to run after the task returns, but they will be killed once the phase ends.

This method should not be called directly.

Parameters

`phase` (*uvm_phase*)

`virtual function suspend()`

Task

suspend

Suspend this component.

This method must be implemented by the user to suspend the component according to the protocol and functionality it implements. A suspended component can be subsequently resumed using *resume()*. Suspend

`virtual function resume()`

Task

resume

Resume this component.

This method must be implemented by the user to resume a component that was previously suspended using *suspend()*. Some component may start in the suspended state and may need to be explicitly resumed. Resume

`virtual function stop_phase(uvm_phase phase)`

Task- stop_phase -- **DEPRECATED**

The stop_phase 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_phase to finish the currently executing transaction, flush the queue, or perform other cleanup. Upon return from stop_phase, a component signals it is ready to be stopped.

The *stop_phase* method will not be called if *enable_stop_interrupt* is 0.

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

This method should never be called directly. Stop_phase

Parameters

`phase` (*uvm_phase*)

`virtual function stop(string ph_name)`

Backward compat. Stop

Parameters

`ph_name` (`string`)

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

Task

all_dropped

The *all_droppped* callback is called when all objections have been dropped by this component and all its descendants. The *source_obj* is the object that dropped the last objection. The *description* is optionally provided by the *source_obj* to give a reason for raising the objection. The *count* indicates the number of objections dropped by the *source_obj* .

Parameters

- **`objection`** (*uvm_objection*)
- **`source_obj`** (*uvm_object*)
- **`description`** (*`string`*)
- **`count`** (*`int`*)

15.1.1.54 Class uvm_pkg::uvm_component_name_check_visitor

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_visitor*
↪*uvm_pkg* :: *uvm_component_name_check_visitor*

Table 63: Assertions

Name	Kind	Description
uvm_pkg::uvm_component_name_check_visitor.[anonymous]	immediate assert	`(compiled_regex != null)`

Constructors

`function new(string name = "")`

Parameters
`name` (*`string`*)

Functions

`virtual function string get_name_constraint()`

Function

get_name_constraint

This method should return a regex for what is being considered a valid/good component name. The visitor will check all component names using this regex and report failing names

`virtual function void visit(uvm_component node)`

Parameters
`node` (*uvm_component*)

`virtual function void begin_v()`

`virtual function void end_v()`

15.1.1.55 Class uvm_pkg::uvm_component_proxy

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_structure_proxy*
↪*uvm_pkg* :: *uvm_component_proxy*

CLASS

uvm_component_proxy

The class is providing the proxy to extract the direct subcomponents of *s*

Constructors

`function new(string name = "")`

Parameters
- **`name`** (*`string`*)

Functions

`virtual function void get_immediate_children(uvm_component s, uvm_-component children)`

Parameters
- **`s`** (*uvm_component*)
- **`children`** (*uvm_component*)

15.1.1.56 Class uvm_pkg::uvm_component_registry

uvm_pkg :: *uvm_object_wrapper*
↪*uvm_pkg* :: *uvm_component_registry*

CLASS

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.

Table 64: Parameters

Name	Default value	Description
T	uvm_component	
Tname	"<unknown>"	

Table 65: Variables

Name	Type	Description
type_name	string	

Table 66: Typedefs

Name	Actual Type	Description
this_type	*uvm_component_registry#(T, Tname)*	

Functions

virtual function uvm_component create_component(string name, uvm_component parent)

Function

create_component

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.

Parameters
- **name** (`string`)
- **parent** (*uvm_component*)

Return type
uvm_component

virtual function string get_type_name()

Function

get_type_name

Returns the value given by the string parameter, *Tname* . This method overrides the method in *uvm_object_wrapper*.

`static function this_type get()`

Function

get

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

Return type
this_type

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

Function

create

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

Parameters
`name` (`string`)
`parent` (*uvm_component*)
`contxt` (`string`)

Return type
T

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

Function

set_type_override

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

Parameters
`override_type` (*uvm_object_wrapper*)
`replace` (`bit`)

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

Function

set_inst_override

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

Parameters
`override_type` (*uvm_object_wrapper*)
`inst_path` (`string`)
`parent` (*uvm_component*)

15.1.1.57 Class uvm_pkg::uvm_config_db

uvm_pkg :: *uvm_resource_db*
↪*uvm_pkg* :: *uvm_config_db*

class

uvm_config_db

All of the functions in uvm_config_db(T) are static, so they must be called using the :: operator. For example:

```
uvm_config_db#(int)::set(this, "*", "A");
```

The parameter value "int" identifies the configuration type as an int property.

The *set* and *get* methods provide the same API and semantics as the set/get_config_* functions in *uvm_component*.

Table 67: Parameters

Name	Default value	Description
T	int	

Functions

static function bit get(uvm_component cntxt, string inst_name, string field_name, int value)

function

get

Get the value for *field_name* in *inst_name* , using component *cntxt* as the starting search point. *inst_name* is an explicit instance name relative to *cntxt* and may be an empty string if the *cntxt* is the instance that the configuration object applies to. *field_name* is the specific field in the scope that is being searched for.

The basic *get_config_** methods from *uvm_component* are mapped to this function as:

```
get_config_int(...) => uvm_config_db#(uvm_bitstream_t)::get(cntxt,...)
get_config_string(...) => uvm_config_db#(string)::get(cntxt,...)
get_config_object(...) => uvm_config_db#(uvm_object)::get(cntxt,...)
```

Parameters
- **cntxt** (*uvm_component*)
- **inst_name** (*string*)
- **field_name** (*string*)
- **value** (*int*)

static function void set(uvm_component cntxt, string inst_name, string field_name, int value)

function

set

Create a new or update an existing configuration setting for *field_name* in *inst_name* from *cntxt* . The setting is made at *cntxt* , with the full scope of the set being { *cntxt* ,".", *inst_name* }. If *cntxt* is *null* then *inst_name* provides the complete scope information of the setting. *field_name* is the target field. Both *inst_name* and *field_name* may be glob style or regular expression style expressions.

If a setting is made at build time, the *cntxt* hierarchy is used to determine the setting's precedence in the database. Settings from hierarchically higher levels have higher precedence. Settings from the same level of hierarchy have a last setting wins semantic. A precedence setting of *uvm_resource_base::default_precedence* is used for uvm_top, and each hierarchical level below the top is decremented by 1.

After build time, all settings use the default precedence and thus have a last wins semantic. So, if at run time, a low level component makes a runtime setting of some field, that setting will have precedence over a setting from the test level that was made earlier in the simulation.

The basic *set_config_** methods from *uvm_component* are mapped to this function as:

```
set_config_int(...) => uvm_config_db#(uvm_bitstream_t)::set(cntxt,...)
set_config_string(...) => uvm_config_db#(string)::set(cntxt,...)
set_config_object(...) => uvm_config_db#(uvm_object)::set(cntxt,...)
```

Parameters

- **cntxt** (*uvm_component*)
- **inst_name** (*string*)
- **field_name** (*string*)
- **value** (*int*)

static function bit exists(uvm_component cntxt, string inst_name, string field_-name, bit spell_chk = 0)

function

exists

Check if a value for *field_name* is available in *inst_name* , using component *cntxt* as the starting search point. *inst_name* is an explicit instance name relative to *cntxt* and may be an empty string if the *cntxt* is the instance that the configuration object applies to. *field_name* is the specific field in the scope that is being searched for. The *spell_chk* arg can be set to 1 to turn spell checking on if it is expected that the field should exist in the database. The function returns 1 if a config parameter exists and 0 if it doesn't exist.

Parameters

- **cntxt** (*uvm_component*)
- **inst_name** (*string*)
- **field_name** (*string*)
- **spell_chk** (*bit*)

Tasks

static function wait_modified(uvm_component cntxt, string inst_name, string field_name)

Function

wait_modified

Wait for a configuration setting to be set for *field_name* in *cntxt* and *inst_name* . The task blocks until a new configuration setting is applied that effects the specified field.

Parameters

- **cntxt** (*uvm_component*)
- **inst_name** (*string*)
- **field_name** (*string*)

15.1.1.58 Class uvm_pkg::uvm_config_db_options

Options include:

tracing

on/off

The default for tracing is off.

Functions

`static function void turn_on_tracing()`

Function

turn_on_tracing

Turn tracing on for the configuration database. This causes all reads and writes to the database to display information about the accesses. Tracing is off by default.

This method is implicitly called by the +*UVM_CONFIG_DB_TRACE* .

`static function void turn_off_tracing()`

Function

turn_off_tracing

Turn tracing off for the configuration database.

`static function bit is_tracing()`

Function

is_tracing

Returns 1 if the tracing facility is on and 0 if it is off.

15.1.1.59 Class uvm_pkg::uvm_config_object_wrapper

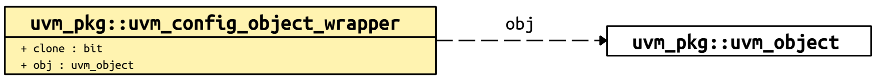


Fig. 13: Collaboration Diagram of uvm_config_object_wrapper

Undocumented struct for storing clone bit along w/ object on set_config_object(...) calls

Table 68: Variables

Name	Type	Description
obj	*uvm_object*	
clone	bit	

15.1.1.60 Class uvm_pkg::uvm_configure_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_task_phase*
↪*uvm_pkg* :: *uvm_configure_phase*

Class

uvm_configure_phase

The SW configures the DUT.

uvm_task_phase that calls the *uvm_component::configure_phase* method.

Upon Entry

- Indicates that the DUT is ready to be configured.

Typical Uses

Components required for DUT configuration execute transactions normally.
Set signals and program the DUT and memories (e.g. read/write operations and sequences) to match the desired configuration for the test and environment.

Exit Criteria

- The DUT has been configured and is ready to operate normally.

Table 69: Variables

Name	Type	Description
type_name	string	

Functions

static function uvm_configure_phase get()

Function

get

Returns the singleton phase handle
Return type
uvm_configure_phase

virtual function string get_type_name()

Tasks

virtual function exec_task(uvm_component comp, uvm_phase phase)

Parameters
comp (*uvm_component*)
phase (*uvm_phase*)

15.1.1.61 Class uvm_pkg::uvm_connect_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_bottomup_phase*
↪*uvm_pkg* :: *uvm_connect_phase*

Class

uvm_connect_phase

Establish cross-component connections.

uvm_bottomup_phase that calls the *uvm_component::connect_phase* method.

Upon Entry

All components have been instantiated.
Current simulation time is still equal to 0 but some "delta cycles" may have occurred.

Typical Uses

Connect TLM ports and exports.
Connect TLM initiator sockets and target sockets.
Connect register model to adapter components.
Setup explicit phase domains.

Exit Criteria

All cross-component connections have been established.
All independent phase domains are set.

Table 70: Variables

Name	Type	Description
type_name	string	

Functions

virtual function void exec_func(uvm_component comp, uvm_phase phase)

Parameters
comp (*uvm_component*)
phase (*uvm_phase*)

static function uvm_connect_phase get()

Function

get

Returns the singleton phase handle
Return type
uvm_connect_phase

virtual function string get_type_name()

15.1.1.62 Class uvm_pkg::uvm_coreservice_t

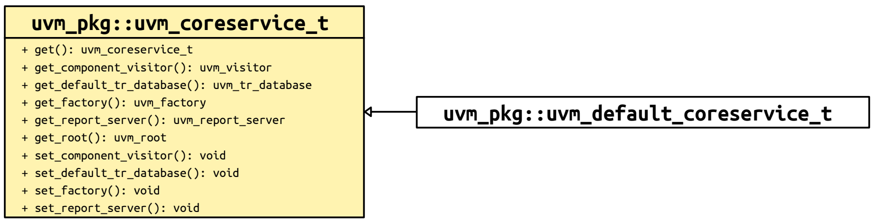

uvm_pkg::uvm_coreservice_t
+ get(): uvm_coreservice_t
+ get_component_visitor(): uvm_visitor
+ get_default_tr_database(): uvm_tr_database
+ get_factory(): uvm_factory
+ get_report_server(): uvm_report_server
+ get_root(): uvm_root
+ set_component_visitor(): void
+ set_default_tr_database(): void
+ set_factory(): void
+ set_report_server(): void
uvm_pkg::uvm_default_coreservice_t

Fig. 14: Inheritance Diagram of uvm_coreservice_t

Class

uvm_coreservice_t

The singleton instance of uvm_coreservice_t provides a common point for all central uvm services such as uvm_factory, uvm_report_server, ... The service class provides a static <::get> which returns an instance adhering to uvm_coreservice_t the rest of the set<*facility*> *get*<facility> pairs provide access to the internal uvm services

Custom implementations of uvm_coreservice_t can be included in uvm_pkg::* and can selected via the define UVM_CORESERVICE_TYPE. They cannot reside in another package.

Functions

`virtual function uvm_factory get_factory()`

Function

get_factory

intended to return the currently enabled uvm factory,

Return type
uvm_factory

`virtual function void set_factory(uvm_factory f)`

Function

set_factory

intended to set the current uvm factory

Parameters
`f` (*uvm_factory*)

`virtual function uvm_report_server get_report_server()`

Function

get_report_server

intended to return the current global report_server

Return type
uvm_report_server

`virtual function void set_report_server(uvm_report_server server)`

Function

set_report_server

intended to set the central report server to *server*

Parameters
`server` (*uvm_report_server*)

virtual function uvm_tr_database get_default_tr_database()

Function

get_default_tr_database

intended to return the current default record database

Return type
uvm_tr_database

virtual function void set_default_tr_database(uvm_tr_database db)

Function

set_default_tr_database

intended to set the current default record database to *db*

Parameters
db (*uvm_tr_database*)

virtual function void set_component_visitor(uvm_visitor#(uvm_component) v)

Function

set_component_visitor

intended to set the component visitor to *v* (this visitor is being used for the traversal at end_of_elaboration_phase for instance for name checking)

Parameters
v (*uvm_visitor#(uvm_component)*)

virtual function uvm_visitor get_component_visitor()

Function

get_component_visitor

intended to retrieve the current component visitor see *set_component_visitor*

Return type
uvm_visitor

virtual function uvm_root get_root()

Function

get_root

returns the uvm_root instance

Return type
uvm_root

static function uvm_coreservice_t get()

Function

get

Returns an instance providing the uvm_coreservice_t interface. The actual type of the instance is determined by the define &96;UVM_CORESERVICE_TYPE.

```
`define UVM_CORESERVICE_TYPE uvm_blocking_coreservice
class uvm_blocking_coreservice extends uvm_default_coreservice_t;
   virtual function void set_factory(uvm_factory f);
      `uvm_error("FACTORY","you are not allowed to override the factory")
   endfunction
endclass
```

Return type
uvm_coreservice_t

15.1.1.63 Class uvm_pkg::uvm_default_coreservice_t

uvm_pkg :: *uvm_coreservice_t*
↪*uvm_pkg* :: *uvm_default_coreservice_t*

Class

uvm_default_coreservice_t

uvm_default_coreservice_t provides a default implementation of the uvm_coreservice_t API. It instantiates uvm_default_factory, uvm_default_report_server, uvm_root.

Functions

`virtual function uvm_factory get_factory()`

Function

get_factory

Returns the currently enabled uvm factory. When no factory has been set before, instantiates a uvm_default_factory

Return type
uvm_factory

`virtual function void set_factory(uvm_factory f)`

Function

set_factory

Sets the current uvm factory. Please note: it is up to the user to preserve the contents of the original factory or delegate calls to the original factory

Parameters
`f` (*uvm_factory*)

`virtual function uvm_tr_database get_default_tr_database()`

Function

get_default_tr_database

returns the current default record database

If no default record database has been set before this method is called, returns an instance of *uvm_text_tr_database*

Return type
uvm_tr_database

`virtual function void set_default_tr_database(uvm_tr_database db)`

Function

set_default_tr_database

Sets the current default record database to *db*

Parameters
`db` (*uvm_tr_database*)

`virtual function uvm_report_server get_report_server()`

Function

get_report_server

returns the current global report_server if no report server has been set before, returns an instance of uvm_default_report_server

Return type
uvm_report_server

`virtual function void set_report_server(uvm_report_server server)`

Function

set_report_server

sets the central report server to *server*

Parameters

`server` (*uvm_report_server*)

`virtual function uvm_root get_root()`

Return type

uvm_root

`virtual function void set_component_visitor(uvm_visitor#(uvm_component) v)`

Function

set_component_visitor

sets the component visitor to *v* (this visitor is being used for the traversal at end_of_elaboration_phase for instance for name checking)

Parameters

`v` (*uvm_visitor#(uvm_component)*)

`virtual function uvm_visitor get_component_visitor()`

Function

get_component_visitor

retrieves the current component visitor if unset(or *null*) returns a *uvm_component_name_check_visitor* instance

Return type

uvm_visitor

15.1.1.64 Class uvm_pkg::uvm_default_factory

uvm_pkg :: *uvm_factory*
↪*uvm_pkg* :: *uvm_default_factory*

CLASS

uvm_default_factory

Default implementation of the UVM factory.

Functions

`virtual function void register(uvm_object_wrapper obj)`

Function

register

Registers the given proxy object, *obj* , with the factory. Register
Parameters
`obj` (*uvm_object_wrapper*)

`virtual function void set_inst_override_by_type(uvm_object_wrapper original_type, uvm_object_wrapper override_type, string full_inst_path)`

Function

set_inst_override_by_type. Set_inst_override_by_type
Parameters
`original_type` (*uvm_object_wrapper*)
`override_type` (*uvm_object_wrapper*)
`full_inst_path` (*string*)

`virtual function void set_inst_override_by_name(string original_type_name, string override_type_name, string full_inst_path)`

Function

set_inst_override_by_name

Configures the factory to create an object of the override's type whenever a request is made to create an object of the original type using a context that matches *full_inst_path* . Set_inst_override_by_name
Parameters
`original_type_name` (*string*)
`override_type_name` (*string*)
`full_inst_path` (*string*)

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

Function

set_type_override_by_type. Set_type_override_by_type
Parameters
`original_type` (*uvm_object_wrapper*)
`override_type` (*uvm_object_wrapper*)
`replace` (*bit*)

`virtual function void set_type_override_by_name(string original_type_name, string override_type_name, bit replace = 1)`

Function

set_type_override_by_name

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

original_type_name (*string*)
override_type_name (*string*)
replace (*bit*)

virtual function uvm_object create_object_by_type(uvm_object_wrapper requested_-type, string parent_inst_path = "", string name = "")

Function

create_object_by_type. Create_object_by_type

Parameters
- **requested_type** (*uvm_object_wrapper*)
- **parent_inst_path** (*string*)
- **name** (*string*)

Return type
uvm_object

virtual function uvm_component create_component_by_type(uvm_object_-wrapper requested_type, string parent_inst_path = "", string name, uvm_-component parent)

Function

create_component_by_type. Create_component_by_type

Parameters
- **requested_type** (*uvm_object_wrapper*)
- **parent_inst_path** (*string*)
- **name** (*string*)
- **parent** (*uvm_component*)

Return type
uvm_component

virtual function uvm_object create_object_by_name(string requested_type_name, string parent_inst_path = "", string name = "")

Function

create_object_by_name. Create_object_by_name

Parameters
- **requested_type_name** (*string*)
- **parent_inst_path** (*string*)
- **name** (*string*)

Return type
uvm_object

virtual function uvm_component create_component_by_name(string requested_type_name, string parent_inst_path = "", string name, uvm_component parent)

Function

create_component_by_name

Creates and returns a component or object of the requested type, which may be specified by type or by name. Create_component_by_name

Parameters
- **requested_type_name** (*string*)
- **parent_inst_path** (*string*)
- **name** (*string*)
- **parent** (*uvm_component*)

Return type
uvm_component

virtual function void debug_create_by_type(uvm_object_wrapper requested_type, string parent_inst_path = "", string name = "")

Function

debug_create_by_type. Debug_create_by_type

Parameters
- **requested_type** (*uvm_object_wrapper*)

`parent_inst_path` (*`string`*)
`name` (*`string`*)

`virtual function void debug_create_by_name(string requested_type_name, string parent_inst_path = "", string name = "")`

Function

debug_create_by_name

These methods perform the same search algorithm as the *create_** methods, but they do not create new objects. Debug_create_by_name

Parameters

`requested_type_name` (*`string`*)
`parent_inst_path` (*`string`*)
`name` (*`string`*)

`virtual function uvm_object_wrapper find_override_by_type(uvm_object_-wrapper requested_type, string full_inst_path)`

Function

find_override_by_type. Find_override_by_type

Parameters

`requested_type` (*uvm_object_wrapper*)
`full_inst_path` (*`string`*)

Return type

uvm_object_wrapper

`virtual function uvm_object_wrapper find_override_by_name(string requested_type_-name, string full_inst_path)`

Function

find_override_by_name

These methods return the proxy to the object that would be created given the arguments. Find_override_by_name

Parameters

`requested_type_name` (*`string`*)
`full_inst_path` (*`string`*)

Return type

uvm_object_wrapper

`virtual function uvm_object_wrapper find_wrapper_by_name(string type_name)`

Find_wrapper_by_name

Parameters

`type_name` (*`string`*)

Return type

uvm_object_wrapper

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

Function

print

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

Parameters

`all_types` (*`int`*)

`function bit check_inst_override_exists(uvm_object_wrapper original_type, uvm_-object_wrapper override_type, string full_inst_path)`

Check_inst_override_exists

Parameters

`original_type` (*uvm_object_wrapper*)
`override_type` (*uvm_object_wrapper*)
`full_inst_path` (*`string`*)

15.1.1.65 Class uvm_pkg::uvm_default_report_server

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_server*
↪*uvm_pkg* :: *uvm_default_report_server*

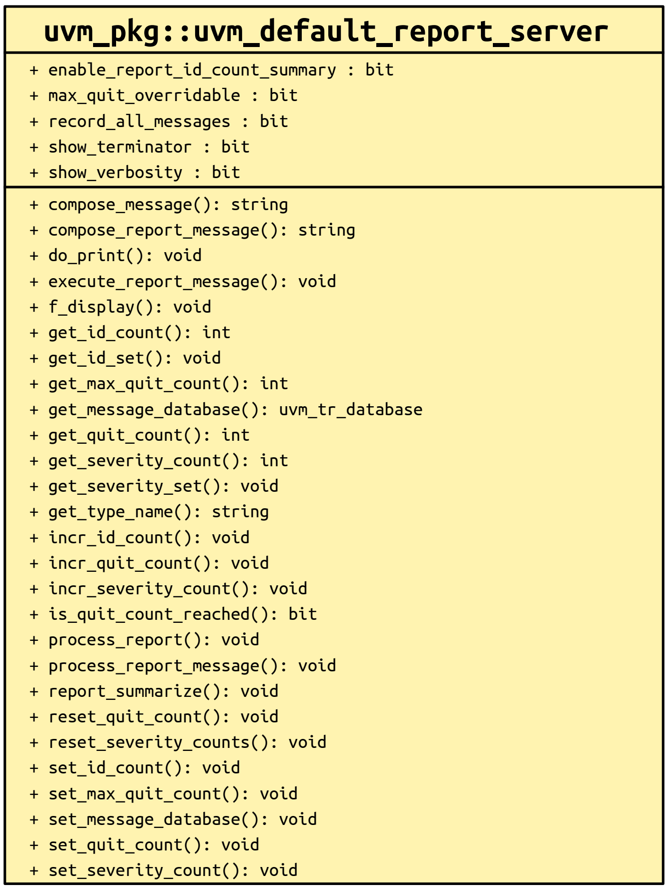

uvm_pkg::uvm_default_report_server
+ enable_report_id_count_summary : bit
+ max_quit_overridable : bit
+ record_all_messages : bit
+ show_terminator : bit
+ show_verbosity : bit
+ compose_message(): string
+ compose_report_message(): string
+ do_print(): void
+ execute_report_message(): void
+ f_display(): void
+ get_id_count(): int
+ get_id_set(): void
+ get_max_quit_count(): int
+ get_message_database(): uvm_tr_database
+ get_quit_count(): int
+ get_severity_count(): int
+ get_severity_set(): void
+ get_type_name(): string
+ incr_id_count(): void
+ incr_quit_count(): void
+ incr_severity_count(): void
+ is_quit_count_reached(): bit
+ process_report(): void
+ process_report_message(): void
+ report_summarize(): void
+ reset_quit_count(): void
+ reset_severity_counts(): void
+ set_id_count(): void
+ set_max_quit_count(): void
+ set_message_database(): void
+ set_quit_count(): void
+ set_severity_count(): void

Fig. 15: Collaboration Diagram of uvm_default_report_server

CLASS

uvm_default_report_server

Default implementation of the UVM report server.

Table 71: Variables

Name	Type	Description
max_quit_overridable	bit	
enable_report_id_-count_summary	bit	***Variable*** enable_report_id_count_summary A flag to enable report count summary for each ID

continues on next page

Table 71 – continued from previous page

Name	Type	Description
record_all_messages	bit	***Variable*** record_all_messages A flag to force recording of all messages (add UVM_RM_RECORD action)
show_verbosity	bit	***Variable*** show_verbosity A flag to include verbosity in the messages, e.g. ***"UVM_INFO(UVM_MEDIUM) file.v(3) at 60*** reporter [ID0] Message 0"
show_terminator	bit	***Variable*** show_terminator A flag to add a terminator in the messages, e.g. ***"UVM_INFO file.v(3) at 60*** reporter [ID0] Message 0 -UVM_INFO"

Constructors

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

Function

new

Creates an instance of the class.

Parameters

`name` (*`string`*)

Functions

`virtual function string get_type_name()`

Needed for callbacks

`virtual function void do_print(uvm_printer printer)`

Print to show report server state

Parameters

`printer` (*uvm_printer*)

`virtual function int get_max_quit_count()`

Function

get_max_quit_count

`virtual function void set_max_quit_count(int count, bit overridable = 1)`

Function

set_max_quit_count

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

Parameters

`count` (*`int`*)

`overridable` (*`bit`*)

`virtual function int get_quit_count()`

Function

get_quit_count

`virtual function void set_quit_count(int quit_count)`

Function

set_quit_count

Parameters

`quit_count` (*`int`*)

`function void incr_quit_count()`

Function

incr_quit_count

`function void reset_quit_count()`

Function

reset_quit_count

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

`function bit is_quit_count_reached()`

Function

is_quit_count_reached

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

`virtual function int get_severity_count(uvm_severity severity)`

Function

get_severity_count

Parameters

`severity` (*uvm_severity*)

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

Function

set_severity_count

Parameters

`severity` (*uvm_severity*)

`count` (*`int`*)

`function void incr_severity_count(uvm_severity severity)`

Function

incr_severity_count

Parameters

`severity` (*uvm_severity*)

`function void reset_severity_counts()`

Function

reset_severity_counts

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

`virtual function int get_id_count(string id)`

Function

get_id_count

Parameters

`id` (*`string`*)

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

Function

set_id_count

Parameters

`id` (*`string`*)

`count` (*`int`*)

function void incr_id_count(string id)

Function

incr_id_count

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

Parameters
: **id** (*string*)

virtual function void set_message_database(uvm_tr_database database)

Function

set_message_database

sets the *uvm_tr_database* used for recording messages

Parameters
: **database** (*uvm_tr_database*)

virtual function uvm_tr_database get_message_database()

Function

get_message_database

returns the *uvm_tr_database* used for recording messages

Return type
: *uvm_tr_database*

virtual function void get_severity_set(uvm_severity q)

Parameters
: **q** (*uvm_severity*)

virtual function void get_id_set(string q)

Parameters
: **q** (*string*)

function void f_display(UVM_FILE file, string str)

Function- f_display

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

Parameters
: **file** (*UVM_FILE*)
: **str** (*string*)

virtual function void process_report_message(uvm_report_message report_message)

Function- process_report_message

Parameters
: **report_message** (*uvm_report_message*)

virtual function void execute_report_message(uvm_report_message report_message, string composed_message)

Function

execute_report_message

Processes the provided message per the actions contained within.

Expert users can overload this method to customize action processing.

Parameters
: **report_message** (*uvm_report_message*)
: **composed_message** (*string*)

virtual function string compose_report_message(uvm_report_message report_message, string report_object_name = "")

Function

compose_report_message

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.

Parameters

report_message (*uvm_report_message*)

report_object_name (`string`)

virtual function void report_summarize(UVM_FILE file = 0)

Function

report_summarize

Outputs statistical information on the reports issued by this 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.

Parameters

file (*UVM_FILE*)

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)

Function- process_report

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.

Parameters

severity (*uvm_severity*)

name (`string`)

id (`string`)

message (`string`)

action (*uvm_action*)

file (*UVM_FILE*)

filename (`string`)

line (`int`)

composed_message (`string`)

verbosity_level (`int`)

client (*uvm_report_object*)

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

Function- compose_message

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.

Parameters

severity (*uvm_severity*)

name (`string`)

id (`string`)

message (`string`)

filename (`string`)

line (`int`)

15.1.1.66 Class uvm_pkg::uvm_derived_callbacks

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_callbacks_base*
↪*uvm_pkg* :: *uvm_typed_callbacks*
↪*uvm_pkg* :: *uvm_callbacks*
↪*uvm_pkg* :: *uvm_derived_callbacks*

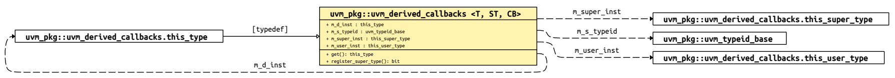

uvm_pkg::uvm_derived_callbacks.this_type
[typedef]
uvm_pkg::uvm_derived_callbacks <T, ST, CB>
+ m_d_inst : this_type
+ m_s_typeid : uvm_typeid_base
+ m_super_inst : this_super_type
+ m_user_inst : this_user_type
+ get(): this_type
+ register_super_type(): bit
m_d_inst
m_super_inst
m_s_typeid
m_user_inst
uvm_pkg::uvm_derived_callbacks.this_super_type
uvm_pkg::uvm_typeid_base
uvm_pkg::uvm_derived_callbacks.this_user_type

Fig. 16: Collaboration Diagram of uvm_derived_callbacks

Class- uvm_derived_callbacks (T, ST, CB)

This type is not really expected to be used directly by the user, instead they are expected to use the macro &#96;uvm_set_super_type. The sole purpose of this type is to allow for setting up of the derived_type/super_type mapping.

Table 72: Parameters

Name	Default value	Description
T	uvm_object	
ST	uvm_object	
CB	uvm_callback	

Table 73: Typedefs

Name	Actual Type	Description
this_type	*uvm_derived_call-backs#(T, ST, CB)*	
this_user_type	*uvm_callbacks#(T)*	
this_super_type	*uvm_callbacks#(ST)*	

Functions

static function this_type get()

Return type
this_type

static function bit register_super_type(string tname = "", string sname = "")

Parameters
tname (*string*)
sname (*string*)

15.1.1.67 Class uvm_pkg::uvm_domain

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_domain*

Class

uvm_domain

Phasing schedule node representing an independent branch of the schedule. Handle used to assign domains to components or hierarchies in the testbench

Constructors

`function new(string name)`

Function

new

Create a new instance of a phase domain.

Parameters
`name` (*`string`*)

Functions

`static function void get_domains(uvm_domain domains)`

Function

get_domains

Provides a list of all domains in the provided *domains* argument.

Parameters
`domains` (*uvm_domain*)

`static function uvm_phase get_uvm_schedule()`

Function

get_uvm_schedule

Get the "UVM" schedule, which consists of the run-time phases that all components execute when participating in the "UVM" domain.

Return type
uvm_phase

`static function uvm_domain get_common_domain()`

Function

get_common_domain

Get the "common" domain, which consists of the common phases that all components execute in sync with each other. Phases in the "common" domain are build, connect, end_of_elaboration, start_of_simulation, run, extract, check, report, and final.

Return type
uvm_domain

`static function void add_uvm_phases(uvm_phase schedule)`

Function

add_uvm_phases

Appends to the given *schedule* the built-in UVM phases.

Parameters
`schedule` (*uvm_phase*)

`static  function uvm_domain get_uvm_domain()`

Function

get_uvm_domain

Get a handle to the singleton *uvm* domain

Return type

uvm_domain

`function void jump(uvm_phase phase)`

Function

jump

jumps all active phases of this domain to to-phase if there is a path between active-phase and to-phase

Parameters

`phase` (*uvm_phase*)

`static  function void jump_all(uvm_phase phase)`

jump_all

Parameters

`phase` (*uvm_phase*)

15.1.1.68 Class uvm_pkg::uvm_driver

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_driver*

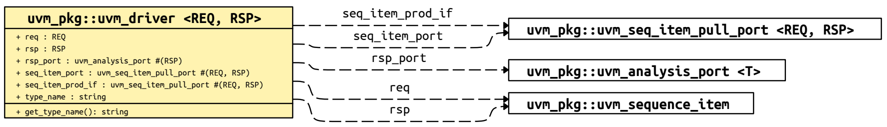

uvm_pkg::uvm_driver <REQ, RSP>
+ req : REQ
+ rsp : RSP
+ rsp_port : uvm_analysis_port #(RSP)
+ seq_item_port : uvm_seq_item_pull_port #(REQ, RSP)
+ seq_item_prod_if : uvm_seq_item_pull_port #(REQ, RSP)
+ type_name : string
+ get_type_name(): string
seq_item_prod_if
seq_item_port
rsp_port
req
rsp
uvm_pkg::uvm_seq_item_pull_port <REQ, RSP>
uvm_pkg::uvm_analysis_port <T>
uvm_pkg::uvm_sequence_item

Fig. 17: Collaboration Diagram of uvm_driver

CLASS

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.

Table 74: Parameters

Name	Default value	Description
REQ	uvm_sequence_item	
RSP	REQ	

Table 75: Variables

Name	Type	Description
seq_item_port	*uvm_seq_item_pull_port#(uvm_sequence_item, uvm_sequence_item)*	***Port*** 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.
seq_item_prod_if	*uvm_seq_item_pull_port#(uvm_sequence_item, uvm_sequence_item)*	alias

continues on next page

Table 75 – continued from previous page

Name	Type	Description
rsp_port	*uvm_analysis_-port#(uvm_sequence_-item)*	***Port*** 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.
req	*uvm_sequence_item*	
rsp	*uvm_sequence_item*	
type_name	string	

Constructors

`function new(string name, uvm_component parent)`

Function

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.

Parameters

`name` (*`string`*)

`parent` (*uvm_component*)

Functions

`virtual function string get_type_name()`

15.1.1.69 Class uvm_pkg::uvm_end_of_elaboration_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_bottomup_phase*
↪*uvm_pkg* :: *uvm_end_of_elaboration_phase*

Class

uvm_end_of_elaboration_phase

Fine-tune the testbench.

uvm_bottomup_phase that calls the *uvm_component::end_of_elaboration_phase* method.

Upon Entry

The verification environment has been completely assembled.
Current simulation time is still equal to 0 but some "delta cycles" may have occurred.

Typical Uses

Display environment topology.
Open files.
Define additional configuration settings for components.

Exit Criteria

- None.

Table 76: Variables

Name	Type	Description
type_name	string	

Functions

virtual function void exec_func(uvm_component comp, uvm_phase phase)

Parameters
comp (*uvm_component*)
phase (*uvm_phase*)

static function uvm_end_of_elaboration_phase get()

Function

get

Returns the singleton phase handle

Return type
uvm_end_of_elaboration_phase

virtual function string get_type_name()

15.1.1.70 Class uvm_pkg::uvm_enum_wrapper

Class

uvm_enum_wrapper(T)

The *uvm_enum_wrapper(T)* class is a utility mechanism provided as a convenience to the end user. It provides a *from_name* method which is the logical inverse of the System Verilog *name* method which is built into all enumerations.

Table 77: Parameters

Name	Default value	Description
T	uvm_active_passive_-enum	

Functions

`static function bit from_name(string name, uvm_active_passive_enum value)`

Function

from_name

Attempts to convert a string *name* to an enumerated value.

If the conversion is successful, the method will return 1, otherwise 0.

Note that the *name* passed in to the method must exactly match the value which would be produced by *enum::name* , and is case sensitive.

For example:

```
typedef uvm_enum_wrapper#(uvm_radix_enum) radix_wrapper;
uvm_radix_enum r_v;

// The following would return '0', as "foo" isn't a value
// in uvm_radix_enum:
radix_wrapper::from_name("foo", r_v);

// The following would return '0', as "uvm_bin" isn't a value
// in uvm_radix_enum (although the upper case "UVM_BIN" is):
radix_wrapper::from_name("uvm_bin", r_v);

// The following would return '1', and r_v would be set to
// the value of UVM_BIN
radix_wrapper::from_name("UVM_BIN", r_v);
```

Parameters

name (`string`)
value (*uvm_active_passive_enum*)

15.1.1.71 Class uvm_pkg::uvm_env

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_env*

CLASS

uvm_env

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

Table 78: Variables

Name	Type	Description
type_name	string	

Constructors

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

Function

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.

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)

Functions

`virtual function string get_type_name()`

15.1.1.72 Class uvm_pkg::uvm_event

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_event_base*
↪*uvm_pkg* :: *uvm_event*

CLASS

uvm_event(T)

The uvm_event class is an extension of the abstract uvm_event_base class.

The optional parameter *T* allows the user to define a data type which can be passed during an event trigger.

Table 79: Parameters

Name	Default value	Description
T	uvm_object	

Table 80: Variables

Name	Type	Description
type_name	string	

Constructors

`function new(string name = "")`

Function

new

Creates a new event object.

Parameters

`name` (*`string`*)

Functions

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

Function

trigger

Triggers the event, resuming all waiting processes.

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

Parameters

`data` (*uvm_object*)

`virtual function T get_trigger_data()`

Function

get_trigger_data

Gets the data, if any, provided by the last call to *trigger*.

Return type

T

`virtual function string get_type_name()`

virtual function void add_callback(uvm_event_callback#(uvm_object) cb, bit append = 1)

Function

add_callback

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.

Parameters

- **cb** (*uvm_event_callback#(uvm_object)*)
- **append** (`bit`)

virtual function void delete_callback(uvm_event_callback#(uvm_object) cb)

Function

delete_callback

Unregisters the given callback, *cb* , from this event.

Parameters

- **cb** (*uvm_event_callback#(uvm_object)*)

virtual function void do_print(uvm_printer printer)

Parameters

- **printer** (*uvm_printer*)

virtual function void do_copy(uvm_object rhs)

Parameters

- **rhs** (*uvm_object*)

virtual function uvm_object create(string name = "")

Parameters

- **name** (`string`)

Return type

uvm_object

Tasks

virtual function wait_trigger_data(uvm_object data)

Task

wait_trigger_data

This method calls *uvm_event_base::wait_trigger* followed by *get_trigger_data*.

Parameters

- **data** (*uvm_object*)

virtual function wait_ptrigger_data(uvm_object data)

Task

wait_ptrigger_data

This method calls *uvm_event_base::wait_ptrigger* followed by *get_trigger_data*.

Parameters

- **data** (*uvm_object*)

15.1.1.73 Class uvm_pkg::uvm_event_base

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_event_base*

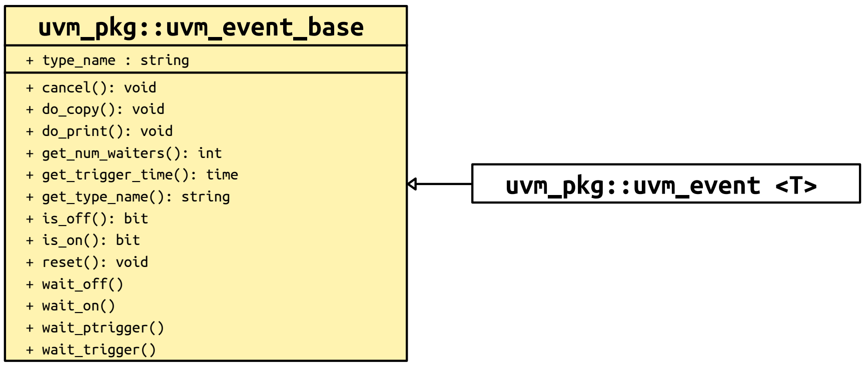


Fig. 18: Inheritance Diagram of uvm_event_base

CLASS

uvm_event_base

The uvm_event_base class is an abstract wrapper class around the SystemVerilog event construct. It provides some additional services such as setting callbacks and maintaining the number of waiters.

Table 81: Variables

Name	Type	Description
type_name	string	

Constructors

`function new(string name = "")`

Function

new

Creates a new event object.

Parameters

name (*string*)

Functions

`virtual function time get_trigger_time()`

Function

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.

`virtual function bit is_on()`

Function

is_on

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

A return of 1 indicates that the event has triggered.

`virtual function bit is_off()`

Function

is_off

Indicates whether the event has been triggered or been reset.

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

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

Function

reset

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.

Parameters

`wakeup` (*`bit`*)

`virtual function void cancel()`

Function

cancel

Decrements the number of waiters on the event.

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

`virtual function int get_num_waiters()`

Function

get_num_waiters

Returns the number of processes waiting on the event.

`virtual function string get_type_name()`

`virtual function void do_print(uvm_printer printer)`

Parameters

`printer` (*uvm_printer*)

`virtual function void do_copy(uvm_object rhs)`

Parameters

`rhs` (*uvm_object*)

Tasks

`virtual function wait_on(bit delta = 0)`

Task

wait_on

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 be remain "on" until the event is *reset*.

Parameters

`delta` (*`bit`*)

`virtual function wait_off(bit delta = 0)`

Task

wait_off

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.

Parameters

`delta` (*`bit`*)

`virtual function wait_trigger()`

Task

wait_trigger

Waits for the event to be triggered.

If one process calls wait_trigger in the same delta as another process calls <uvm_event(T)::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.

`virtual function wait_ptrigger()`

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.

15.1.1.74 Class uvm_pkg::uvm_event_callback

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_event_callback*

CLASS

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_event(T)>s. 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.

Table 82: Parameters

Name	Default value	Description
T	uvm_object	

Constructors

`function new(string name = "")`

Function

new

Creates a new callback object.
Parameters
`name` (*`string`*)

Functions

`virtual function bit pre_trigger(uvm_event#(uvm_object) e, uvm_object data)`

Function

pre_trigger

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(T)> that is being triggered, and *data* is the optional data associated with the event trigger.
Parameters
`e` (*uvm_event#(uvm_object)*)
`data` (*uvm_object*)

`virtual function void post_trigger(uvm_event#(uvm_object) e, uvm_object data)`

Function

post_trigger

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(T)> that is being triggered, and *data* is the optional data associated with the event trigger.

Parameters
- **`e`** (*uvm_event#(uvm_object)*)
- **`data`** (*uvm_object*)

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

Parameters
- **`name`** (`string`)

Return type
uvm_object

15.1.1.75 Class uvm_pkg::uvm_exhaustive_sequence

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*
↪*uvm_pkg* :: *uvm_exhaustive_sequence*

CLASS- 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_exaustive_sequence".

Constructors

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

new

Parameters

`name` (*`string`*)

Functions

`virtual function void do_copy(uvm_object rhs)`

Implement data functions

Parameters

`rhs` (*uvm_object*)

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

Parameters

`rhs` (*uvm_object*)

`comparer` (*uvm_comparer*)

`virtual function void do_print(uvm_printer printer)`

Parameters

`printer` (*uvm_printer*)

`virtual function void do_record(uvm_recorder recorder)`

Parameters

`recorder` (*uvm_recorder*)

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

Parameters

`name` (*`string`*)

Return type

uvm_object

`virtual function string get_type_name()`

Tasks

`virtual function body()`

body

15.1.1.76 Class uvm_pkg::uvm_extract_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_bottomup_phase*
↪*uvm_pkg* :: *uvm_extract_phase*

Class

uvm_extract_phase

Extract data from different points of the verification environment.

uvm_bottomup_phase that calls the *uvm_component::extract_phase* method.

Upon Entry

The DUT no longer needs to be simulated.
Simulation time will no longer advance.

Typical Uses

Extract any remaining data and final state information from scoreboard and testbench components
Probe the DUT (via zero-time hierarchical references and/or backdoor accesses) for final state information.
Compute statistics and summaries.
Display final state information
Close files.

Exit Criteria

- All data has been collected and summarized.

Table 83: Variables

Name	Type	Description
type_name	string	

Functions

`virtual function void exec_func(uvm_component comp, uvm_phase phase)`

Parameters
`comp` (*uvm_component*)
`phase` (*uvm_phase*)

`static function uvm_extract_phase get()`

Function

get

Returns the singleton phase handle
Return type
uvm_extract_phase

`virtual function string get_type_name()`

15.1.1.77 Class uvm_pkg::uvm_factory

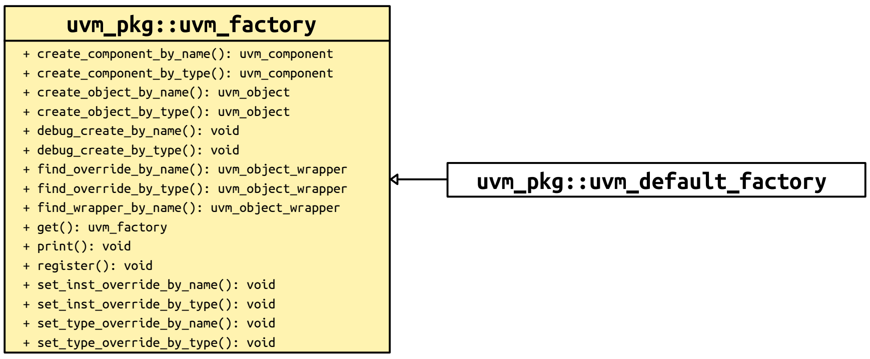

uvm_pkg::uvm_factory
+ create_component_by_name(): uvm_component
+ create_component_by_type(): uvm_component
+ create_object_by_name(): uvm_object
+ create_object_by_type(): uvm_object
+ debug_create_by_name(): void
+ debug_create_by_type(): void
+ find_override_by_name(): uvm_object_wrapper
+ find_override_by_type(): uvm_object_wrapper
+ find_wrapper_by_name(): uvm_object_wrapper
+ get(): uvm_factory
+ print(): void
+ register(): void
+ set_inst_override_by_name(): void
+ set_inst_override_by_type(): void
+ set_type_override_by_name(): void
+ set_type_override_by_type(): void
uvm_pkg::uvm_default_factory

Fig. 19: Inheritance Diagram of uvm_factory

CLASS

uvm_factory

As the name implies, uvm_factory is used to manufacture (create) UVM objects and components. 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.

type-based

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

name-based

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

The *uvm_factory* is an abstract class which declares many of its methods as *pure virtual* . The UVM uses the *uvm_default_factory* class as its default factory implementation.

See <uvm_default_factory::Usage> section for details on configuring and using the factory.

Functions

`static function uvm_factory get()`

Function

get

Static accessor for *uvm_factory*

The static accessor is provided as a convenience wrapper around retrieving the factory via the *uvm_coreservice_t::get_factory* method.

```
// Using the uvm_coreservice_t:
uvm_coreservice_t cs;
uvm_factory f;
cs = uvm_coreservice_t::get();
f = cs.get_factory();
```

```
// Not using the uvm_coreservice_t:
uvm_factory f;
f = uvm_factory::get();
```

Return type

uvm_factory

`virtual function void register(uvm_object_wrapper obj)`

Function

register

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.

Parameters

`obj` (*uvm_object_wrapper*)

`virtual function void set_inst_override_by_type(uvm_object_wrapper original_type, uvm_object_wrapper override_type, string full_inst_path)`

Function

set_inst_override_by_type

Parameters

`original_type` (*uvm_object_wrapper*)

`override_type` (*uvm_object_wrapper*)

`full_inst_path` (`string`)

`virtual function void set_inst_override_by_name(string original_type_name, string override_type_name, string full_inst_path)`

Function

set_inst_override_by_name

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

Parameters

`original_type_name` (`string`)

`override_type_name` (`string`)

`full_inst_path` (`string`)

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

Function

set_type_override_by_type

Parameters
- **original_type** (*uvm_object_wrapper*)
- **override_type** (*uvm_object_wrapper*)
- **replace** (`bit`)

`virtual function void set_type_override_by_name(string original_type_name, string override_type_name, bit replace = 1)`

Function

set_type_override_by_name

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

Parameters
- **original_type_name** (`string`)
- **override_type_name** (`string`)
- **replace** (`bit`)

`virtual function uvm_object create_object_by_type(uvm_object_wrapper requested_type, string parent_inst_path = "", string name = "")`

Function

create_object_by_type

Parameters
- **requested_type** (*uvm_object_wrapper*)
- **parent_inst_path** (`string`)
- **name** (`string`)

Return type
uvm_object

`virtual function uvm_component create_component_by_type(uvm_object_wrapper requested_type, string parent_inst_path = "", string name, uvm_component parent)`

Function

create_component_by_type

Parameters
- **requested_type** (*uvm_object_wrapper*)
- **parent_inst_path** (`string`)
- **name** (`string`)
- **parent** (*uvm_component*)

Return type
uvm_component

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

Function

create_object_by_name

Parameters
- **requested_type_name** (`string`)
- **parent_inst_path** (`string`)
- **name** (`string`)

Return type

uvm_object

`virtual function uvm_component create_component_by_name(string requested_type_name, string parent_inst_path = "", string name, uvm_component parent)`

Function

create_component_by_name

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.

Parameters

- **`requested_type_name`** (*`string`*)
- **`parent_inst_path`** (*`string`*)
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)

Return type

uvm_component

`virtual function void debug_create_by_type(uvm_object_wrapper requested_type, string parent_inst_path = "", string name = "")`

Function

debug_create_by_type

Parameters

- **`requested_type`** (*uvm_object_wrapper*)
- **`parent_inst_path`** (*`string`*)
- **`name`** (*`string`*)

`virtual function void debug_create_by_name(string requested_type_name, string parent_inst_path = "", string name = "")`

Function

debug_create_by_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.

Parameters

- **`requested_type_name`** (*`string`*)
- **`parent_inst_path`** (*`string`*)
- **`name`** (*`string`*)

`virtual function uvm_object_wrapper find_override_by_type(uvm_object_wrapper requested_type, string full_inst_path)`

Function

find_override_by_type

Parameters
- **`requested_type`** (*uvm_object_wrapper*)
- **`full_inst_path`** (*`string`*)

Return type
uvm_object_wrapper

`virtual function uvm_object_wrapper find_override_by_name(string requested_type_name, string full_inst_path)`

Function

find_override_by_name

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 }.

Parameters
- **`requested_type_name`** (*`string`*)
- **`full_inst_path`** (*`string`*)

Return type
uvm_object_wrapper

`virtual function uvm_object_wrapper find_wrapper_by_name(string type_name)`

Parameters
- **`type_name`** (*`string`*)

Return type
uvm_object_wrapper

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

Function

print

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.

Parameters
- **`all_types`** (*`int`*)

15.1.1.78 Class uvm_pkg::uvm_factory_override

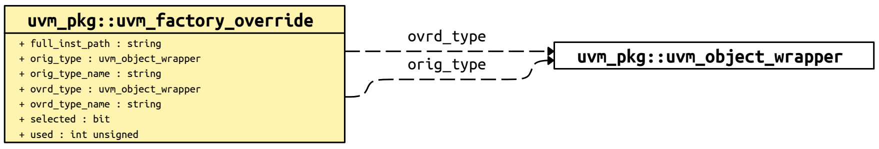

uvm_pkg::uvm_factory_override
+ full_inst_path : string
+ orig_type : uvm_object_wrapper
+ orig_type_name : string
+ ovrd_type : uvm_object_wrapper
+ ovrd_type_name : string
+ selected : bit
+ used : int unsigned
ovrd_type
orig_type
uvm_pkg::uvm_object_wrapper

Fig. 20: Collaboration Diagram of uvm_factory_override

CLASS- uvm_factory_override

Internal class.

Table 84: Variables

Name	Type	Description
full_inst_path	string	
orig_type_name	string	
ovrd_type_name	string	
selected	bit	
used	int unsigned	
orig_type	*uvm_object_wrapper*	
ovrd_type	*uvm_object_wrapper*	

Constructors

function new(string full_inst_path = "", string orig_type_name = "", uvm_object_-wrapper orig_type = null, uvm_object_wrapper ovrd_type)

Parameters

- **full_inst_path** (*string*)
- **orig_type_name** (*string*)
- **orig_type** (*uvm_object_wrapper*)
- **ovrd_type** (*uvm_object_wrapper*)

15.1.1.79 Class uvm_pkg::uvm_factory_queue_class

Instance overrides by requested type lookup

Table 85: Variables

Name	Type	Description
queue	*uvm_factory_override*	

15.1.1.80 Class uvm_pkg::uvm_final_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_topdown_phase*
↪*uvm_pkg* :: *uvm_final_phase*

Class

uvm_final_phase

Tie up loose ends.

uvm_topdown_phase that calls the *uvm_component::final_phase* method.

Upon Entry

- All test-related activity has completed.

Typical Uses

Close files.
Terminate co-simulation engines.

Exit Criteria

- Ready to exit simulator.

Table 86: Variables

Name	Type	Description
type_name	string	

Functions

virtual function void exec_func(uvm_component comp, uvm_phase phase)

Parameters
comp (*uvm_component*)
phase (*uvm_phase*)

static function uvm_final_phase get()

Function

get

Returns the singleton phase handle
Return type
uvm_final_phase

virtual function string get_type_name()

15.1.1.81 Class uvm_pkg::uvm_get_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_get_export*

Table 87: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters

- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.82 Class uvm_pkg::uvm_get_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_get_imp*

Table 88: Parameters

Name	Default value	Description
T	int	
IMP	int	

Constructors

`function  new(string name, int imp)`

Parameters
- **name** (*string*)
- **imp** (*int*)

15.1.1.83 Class uvm_pkg::uvm_get_peek_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_get_peek_export*

Table 89: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.1.1.84 Class uvm_pkg::uvm_get_peek_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_get_peek_imp*

Table 90: Parameters

Name	Default value	Description
T	int	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **`name`** (*`string`*)
- **`imp`** (*`int`*)

15.1.1.85 Class uvm_pkg::uvm_get_peek_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_get_peek_port*

Table 91: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.86 Class uvm_pkg::uvm_get_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_get_port*

Table 92: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters

name (*string*)
parent (*uvm_component*)
min_size (*int*)
max_size (*int*)

15.1.1.87 Class uvm_pkg::uvm_get_to_lock_dap

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_set_get_dap_base*
↪*uvm_pkg* :: *uvm_get_to_lock_dap*

Class

uvm_get_to_lock_dap

Provides a 'Get-To-Lock' Data Access Policy.

The 'Get-To-Lock' Data Access Policy allows for any number of 'sets', until the value is retrieved via a 'get'. Once 'get' has been called, it is illegal to 'set' a new value.

The UVM uses this policy to protect the *starting phase* and *automatic objection* values in *uvm_sequence_base*.

Table 93: Parameters

Name	Default value	Description
T	int	

Table 94: Typedefs

Name	Actual Type	Description
this_type	*uvm_get_to_lock_-dap#(T)*	Used for self-references

Constructors

```
function  new(string name = "unnamed-uvm_get_to_lock_dap#(T)")
```

Function

new

Constructor
Parameters
name (*string*)

Functions

```
virtual  function void set(int value)
```

Function

set

Updates the value stored within the DAP.

set will result in an error if the value has already been retrieved via a call to *get* .
Parameters
value (*int*)

```
virtual  function bit try_set(int value)
```

Function

try_set

Attempts to update the value stored within the DAP.

try_set will return a 1 if the value was successfully updated, or a 0 if the value can not be updated due to *get* having been called. No errors will be reported if *try_set* fails.

Parameters

`value` (*`int`*)

`virtual function T get()`

Function

get

Returns the current value stored within the DAP, and 'locks' the DAP.

After a 'get', the value contained within the DAP cannot be changed.

`virtual function bit try_get(int value)`

Function

try_get

Retrieves the current value stored within the DAP, and 'locks' the DAP.

try_get will always return 1.

Parameters

`value` (*`int`*)

`virtual function void do_copy(uvm_object rhs)`

Group

Introspection

The *uvm_get_to_lock_dap* cannot support the standard UVM instrumentation methods (*copy* , *clone* , *pack* and *unpack*), due to the fact that they would potentially violate the access policy.

A call to any of these methods will result in an error.

Parameters

`rhs` (*uvm_object*)

`virtual function void do_pack(uvm_packer packer)`

Parameters

`packer` (*uvm_packer*)

`virtual function void do_unpack(uvm_packer packer)`

Parameters

`packer` (*uvm_packer*)

`virtual function string convert2string()`

Function- convert2string

`virtual function void do_print(uvm_printer printer)`

Function- do_print

Parameters

`printer` (*uvm_printer*)

15.1.1.88 Class uvm_pkg::uvm_hdl_path_concat

Class

uvm_hdl_path_concat

Concatenation of HDL variables

A dArray of *uvm_hdl_path_slice* specifying a concatenation of HDL variables that implement a register in the HDL.

Slices must be specified in most-to-least significant order. Slices must not overlap. Gaps may exist in the concatenation if portions of the registers are not implemented.

For example, the following register l

```
         1  1  1  1  1  1  0  0  0  0  0  0  0  0  0  0
Bits:    5  4  3  2  1  0  9  8  7  6  5  4  3  2  1  0
        +-+---+-------------+---+-------+
        |A|xxx|      B      |xxx|   C   |
        +-+---+-------------+---+-------+
```

If the register is implemented using a single HDL variable, The array should specify a single slice with its *offset* and *size* specified as -1. For example:

```
concat.set('{ '{"r1", -1, -1} });
```

Table 95: Variables

Name	Type	Description
slices	*uvm_hdl_path_slice*	***Variable*** slices Array of individual slices, stored in most-to-least significant order

Functions

`function void set(uvm_hdl_path_slice t)`

Function

set

Initialize the concatenation using an array literal

Parameters

`t` (*uvm_hdl_path_slice*)

`function void add_slice(uvm_hdl_path_slice slice)`

Function

add_slice

Append the specified *slice* literal to the path concatenation

Parameters

`slice` (*uvm_hdl_path_slice*)

`function void add_path(string path, int unsigned offset = -1, int unsigned size = -1)`

Function

add_path

Append the specified *path* to the path concatenation, for the specified number of bits at the specified *offset* .

Parameters

`path` (*`string`*)
`offset` (*`int unsigned`*)
`size` (*`int unsigned`*)

15.1.1.89 Class uvm_pkg::uvm_heartbeat

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_heartbeat*

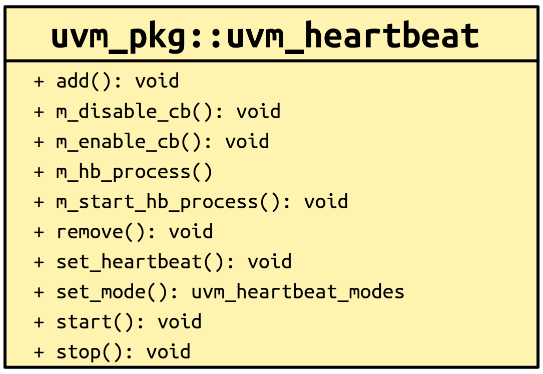


Fig. 21: Collaboration Diagram of uvm_heartbeat

Constructors

function new(string name, uvm_component cntxt, uvm_objection objection = null)

Function

new

Creates a new heartbeat instance associated with *cntxt* . The context is the hierarchical location that the heartbeat objections will flow through and be monitored at. The *objection* associated with the heartbeat is optional, if it is left *null* but it must be set before the heartbeat monitor will activate.

```
uvm_objection myobjection = new("myobjection"); //some shared objection
class myenv extends uvm_env;
   uvm_heartbeat hb = new("hb", this, myobjection);
   ...
endclass
```

Parameters
- **name** (*string*)
- **cntxt** (*uvm_component*)
- **objection** (*uvm_objection*)

Functions

function uvm_heartbeat_modes set_mode(uvm_heartbeat_modes mode = UVM_NO_HB_MODE)

Function

set_mode

Sets or retrieves the heartbeat mode. The current value for the heartbeat mode is returned. If an argument is specified to change the mode then the mode is changed to the new value.

Parameters
- **mode** (*uvm_heartbeat_modes*)

Return type
uvm_heartbeat_modes

function void set_heartbeat(uvm_event#(uvm_object) e, uvm_component comps)

Function

set_heartbeat

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*.

Parameters

`e` (*uvm_event#(uvm_object)*)

`comps` (*uvm_component*)

`function void add(uvm_component comp)`

Function

add

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.

Parameters

`comp` (*uvm_component*)

`function void remove(uvm_component comp)`

Function

remove

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).

Parameters

`comp` (*uvm_component*)

`function void start(uvm_event#(uvm_object) e = null)`

Function

start

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.

Parameters

`e` (*uvm_event#(uvm_object)*)

`function void stop()`

Function

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.

15.1.1.90 Class uvm_pkg::uvm_heartbeat_callback

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_callback*
↪*uvm_pkg* :: *uvm_objection_callback*
↪*uvm_pkg* :: *uvm_heartbeat_callback*

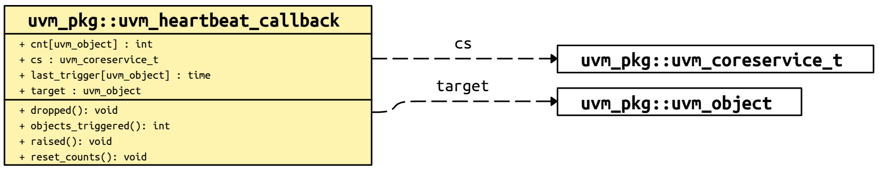


Fig. 22: Collaboration Diagram of uvm_heartbeat_callback

Table 96: Variables

Name	Type	Description
cnt	int	
last_trigger	time	
target	*uvm_object*	
cs	*uvm_coreservice_t*	

Constructors

function new(string name, uvm_object target)

Parameters
- **name** (*string*)
- **target** (*uvm_object*)

Functions

virtual function void raised(uvm_objection objection, uvm_object obj, uvm_-object source_obj, string description, int count)

Parameters
- **objection** (*uvm_objection*)
- **obj** (*uvm_object*)
- **source_obj** (*uvm_object*)
- **description** (*string*)
- **count** (*int*)

virtual function void dropped(uvm_objection objection, uvm_object obj, uvm_-object source_obj, string description, int count)

Parameters
- **objection** (*uvm_objection*)
- **obj** (*uvm_object*)
- **source_obj** (*uvm_object*)
- **description** (*string*)
- **count** (*int*)

function void reset_counts()

function int objects_triggered()

15.1.1.91 Class uvm_pkg::uvm_in_order_built_in_comparator

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_in_order_comparator*
↪*uvm_pkg* :: *uvm_in_order_built_in_comparator*

CLASS

uvm_in_order_built_in_comparator (T)

This class uses the uvm_built_in_* comparison, converter, and pair classes. Use this class for built-in types (int, bit, string, etc.)

Table 97: Parameters

Name	Default value	Description
T	int	

Table 98: Variables

Name	Type	Description
type_name	string	

Table 99: Typedefs

Name	Actual Type	Description
this_type	*uvm_in_order_built_in_-comparator#(T)*	

Constructors

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

Parameters
- **name** (`string`)
- **parent** (*uvm_component*)

Functions

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

15.1.1.92 Class uvm_pkg::uvm_in_order_class_comparator

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_in_order_comparator*
↪*uvm_pkg* :: *uvm_in_order_class_comparator*

CLASS

uvm_in_order_class_comparator (T)

This class uses the uvm_class_* comparison, converter, and pair classes. Use this class for comparing user-defined objects of type T, which must provide compare() and convert2string() method.

Table 100: Parameters

Name	Default value	Description
T	int	

Table 101: Variables

Name	Type	Description
type_name	string	

Table 102: Typedefs

Name	Actual Type	Description
this_type	*uvm_in_order_class_-comparator#(T)*	

Constructors

`function new(string name, uvm_component parent)`

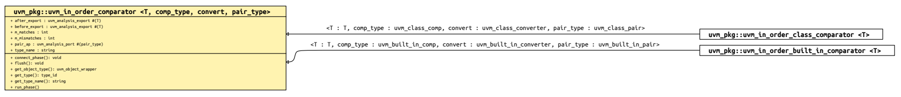
Parameters
- **`name`** (`string`)
- **`parent`** (*uvm_component*)

Functions

`virtual function string get_type_name()`

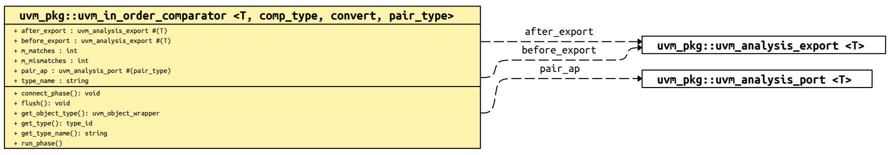
15.1.1.93 Class uvm_pkg::uvm_in_order_comparator

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_in_order_comparator*



uvm_pkg::uvm_in_order_comparator <T, comp_type, convert, pair_type>
+ after_export : uvm_analysis_export #(T)
+ before_export : uvm_analysis_export #(T)
+ m_matches : int
+ m_mismatches : int
+ pair_ap : uvm_analysis_port #(pair_type)
+ type_name : string
+ connect_phase(): void
+ flush(): void
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string
+ run_phase()
<T : T, comp_type : uvm_class_comp, convert : uvm_class_converter, pair_type : uvm_class_pair>
<T : T, comp_type : uvm_built_in_comp, convert : uvm_built_in_converter, pair_type : uvm_built_in_pair>
uvm_pkg::uvm_in_order_class_comparator <T>
uvm_pkg::uvm_in_order_built_in_comparator <T>

Fig. 23: Inheritance Diagram of uvm_in_order_comparator



uvm_pkg::uvm_in_order_comparator <T, comp_type, convert, pair_type>
+ after_export : uvm_analysis_export #(T)
+ before_export : uvm_analysis_export #(T)
+ m_matches : int
+ m_mismatches : int
+ pair_ap : uvm_analysis_port #(pair_type)
+ type_name : string
+ connect_phase(): void
+ flush(): void
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string
+ run_phase()
after_export
before_export
pair_ap
uvm_pkg::uvm_analysis_export <T>
uvm_pkg::uvm_analysis_port <T>

Fig. 24: Collaboration Diagram of uvm_in_order_comparator

CLASS

uvm_in_order_comparator (T, comp_type, convert, pair_type)

Compares two streams of data objects of the type parameter, T. These transactions may either be classes or built-in types. To be successfully compared, the two streams of data must be in the same order. Apart from that, there are no assumptions made about the relative timing of the two streams of data.

Type parameters

T

Specifies the type of transactions to be compared.

comp_type

A policy class to compare the two transaction streams. It must provide the static method "function bit comp(T a, T b)" which returns *TRUE* if *a* and *b* are the same.

convert

A policy class to convert the transactions being compared to a string. It must provide the static method "function string convert2string(T a)".

pair_type

A policy class to allow pairs of transactions to be handled as a single *uvm_object* type.

Built in types (such as ints, bits, logic, and structs) can be compared using the default values for comp_type, convert, and pair_type. For convenience, you can use the subtype, <uvm_in_order_built_in_comparator (T)> for built-in types.

When T is a *uvm_object*, you can use the convenience subtype <uvm_in_order_class_comparator (T)>.

Comparisons are commutative, meaning it does not matter which data stream is connected to which export, before_export or after_export.

Comparisons are done in order and as soon as a transaction is received from both streams. Internal fifos are used to buffer incoming transactions on one stream until a transaction to compare arrives on the other stream.

Table 103: Parameters

Name	Default value	Description
T	int	
comp_type	uvm_built_in_comp	
convert	uvm_built_in_converter	
pair_type	uvm_built_in_pair	

Table 104: Variables

Name	Type	Description
type_name	string	
before_export	*uvm_analysis_export#(int)*	***Port*** 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	*uvm_analysis_export#(int)*	***Port*** 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	*uvm_analysis_port#(uvm_built_in_pair#(int, int))*	***Port*** 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.

Table 105: Typedefs

Name	Actual Type	Description
this_type	*uvm_in_order_comparator#(T, comp_type, convert, pair_type)*	

Constructors

`function new(string name, uvm_component parent)`

Parameters

`name` (`string`)

`parent` (*uvm_component*)

Functions

`virtual function string get_type_name()`

`virtual function void connect_phase(uvm_phase phase)`

Parameters
: **`phase`** (*uvm_phase*)

`virtual function void flush()`

Function

flush

This method sets m_matches and m_mismatches back to zero. The <uvm_tlm_fifo(T)::flush> takes care of flushing the FIFOs.

Tasks

`virtual function run_phase(uvm_phase phase)`

Task- run_phase

Internal method.

Takes pairs of before and after transactions and compares them. Status information is updated according to the results of the comparison. Each pair is published to the pair_ap analysis port.

Parameters
: **`phase`** (*uvm_phase*)

15.1.1.94 Class uvm_pkg::uvm_int_rsrc

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_resource_base*
↪*uvm_pkg* :: *uvm_resource*
↪*uvm_pkg* :: *uvm_int_rsrc*

uvm_int_rsrc

specialization of uvm_resource (T) for T = int

Table 106: Typedefs

Name	Actual Type	Description
this_subtype	*uvm_int_rsrc*	

Constructors

`function new(string name, string s = "*")`

Parameters
- **`name`** (*`string`*)
- **`s`** (*`string`*)

Functions

`virtual function string convert2string()`

15.1.1.95 Class uvm_pkg::uvm_line_printer

uvm_pkg :: *uvm_printer*
↪*uvm_pkg* :: *uvm_tree_printer*
↪*uvm_pkg* :: *uvm_line_printer*

Class

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 }
```

Constructors

`function new()`

Variable

new

Creates a new instance of *uvm_line_printer* . It differs from the *uvm_tree_printer* only in that the output contains no line-feeds and indentation.

15.1.1.96 Class uvm_pkg::uvm_link_base

uvm_pkg :: *uvm_void*
  ↪*uvm_pkg* :: *uvm_object*
    ↪*uvm_pkg* :: *uvm_link_base*

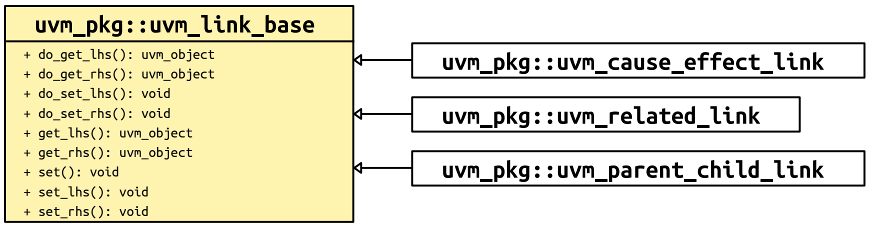


Fig. 25: Inheritance Diagram of uvm_link_base

CLASS

uvm_link_base

The *uvm_link_base* class presents a simple API for defining a link between any two objects.

Using extensions of this class, a *uvm_tr_database* can determine the type of links being passed, without relying on "magic" string names.

For example: |

```
virtual function void do_establish_link(uvm_link_base link);
  uvm_parent_child_link pc_link;
  uvm_cause_effect_link ce_link;

  if ($cast(pc_link, link)) begin
     // Record the parent-child relationship
  end
  else if ($cast(ce_link, link)) begin
     // Record the cause-effect relationship
  end
  else begin
     // Unsupported relationship!
  end
endfunction : do_establish_link
```

Constructors

`function new(string name = "unnamed-uvm_link_base")`

Function

new

Constructor

Parameters

name

Instance name

Parameters
  name (*string*)

Functions

`function void set_lhs(uvm_object lhs)`

Function

set_lhs

Sets the left-hand-side of the link

Triggers the *do_set_lhs* callback.

Parameters

`lhs` (*uvm_object*)

`function uvm_object get_lhs()`

Function

get_lhs

Gets the left-hand-side of the link

Triggers the *do_get_lhs* callback

Return type

uvm_object

`function void set_rhs(uvm_object rhs)`

Function

set_rhs

Sets the right-hand-side of the link

Triggers the *do_set_rhs* callback.

Parameters

`rhs` (*uvm_object*)

`function uvm_object get_rhs()`

Function

get_rhs

Gets the right-hand-side of the link

Triggers the *do_get_rhs* callback

Return type

uvm_object

`function void set(uvm_object lhs, uvm_object rhs)`

Function

set

Convenience method for setting both sides in one call.

Triggers both the *do_set_rhs* and *do_set_lhs* callbacks.

Parameters

`lhs` (*uvm_object*)

`rhs` (*uvm_object*)

`virtual  function void do_set_lhs(uvm_object lhs)`

Function

do_set_lhs

Callback for setting the left-hand-side

Parameters

`lhs` (*uvm_object*)

`virtual  function uvm_object do_get_lhs()`

Function

do_get_lhs

Callback for retrieving the left-hand-side

Return type

uvm_object

`virtual  function void do_set_rhs(uvm_object rhs)`

Function

do_set_rhs

Callback for setting the right-hand-side

Parameters

`rhs` (*uvm_object*)

`virtual  function uvm_object do_get_rhs()`

Function

do_get_rhs

Callback for retrieving the right-hand-side

Return type

uvm_object

15.1.1.97 Class uvm_pkg::uvm_main_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_task_phase*
↪*uvm_pkg* :: *uvm_main_phase*

Class

uvm_main_phase

Primary test stimulus.

uvm_task_phase that calls the *uvm_component::main_phase* method.

Upon Entry

- The stimulus associated with the test objectives is ready to be applied.

Typical Uses

Components execute transactions normally.
Data stimulus sequences are started.
Wait for a time-out or certain amount of time,

or completion of stimulus sequences.

Exit Criteria

- Enough stimulus has been applied to meet the primary stimulus objective of the test.

Table 107: Variables

Name	Type	Description
type_name	string	

Functions

`static function uvm_main_phase get()`

Function

get

Returns the singleton phase handle
Return type
uvm_main_phase

`virtual function string get_type_name()`

Tasks

`virtual function exec_task(uvm_component comp, uvm_phase phase)`

Parameters
`comp` (*uvm_component*)
`phase` (*uvm_phase*)

15.1.1.98 Class uvm_pkg::uvm_master_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_master_export*

Table 108: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.99 Class uvm_pkg::uvm_master_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_master_imp*

Table 109: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	
IMP	int	
REQ_IMP	IMP	
RSP_IMP	IMP	

Table 110: Typedefs

Name	Actual Type	Description
this_imp_type	IMP	
this_req_type	REQ_IMP	
this_rsp_type	RSP_IMP	

Constructors

`function new(string name, this_imp_type imp, this_req_type req_imp = null, this_rsp_type rsp_imp = null)`

Parameters
- **name** (`string`)
- **imp** (*this_imp_type*)
- **req_imp** (*this_req_type*)
- **rsp_imp** (*this_rsp_type*)

15.1.1.100 Class uvm_pkg::uvm_master_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_master_port*

Table 111: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.101 Class uvm_pkg::uvm_mem

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_mem*

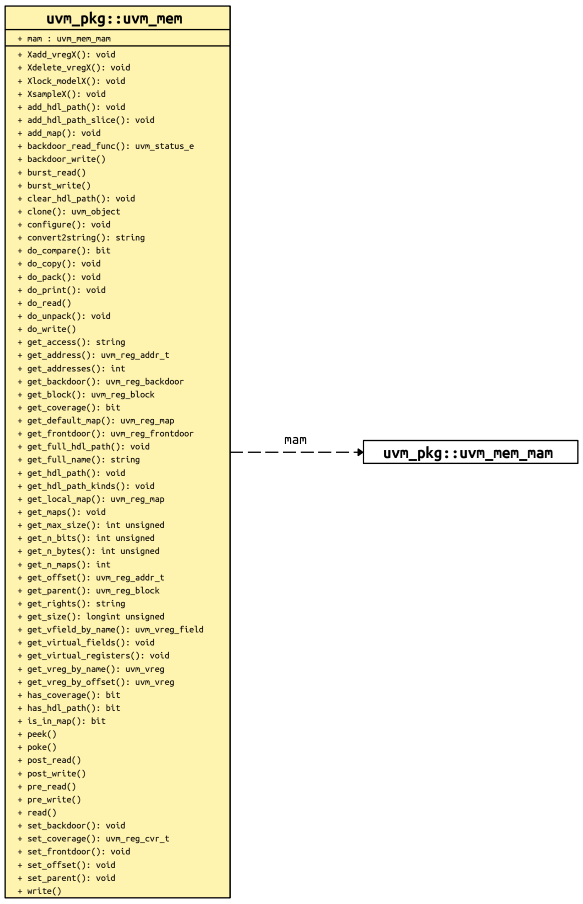


Fig. 26: Collaboration Diagram of uvm_mem

CLASS

uvm_mem

Memory abstraction base class

A memory is a collection of contiguous locations. A memory may be accessible via more than one address map.

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

Table 112: Variables

Name	Type	Description
mam	*uvm_mem_mam*	***variable*** mam Memory allocation manager Memory allocation manager for the memory corresponding to this abstraction class instance. Can be used to allocate regions of consecutive addresses of specific sizes, such as DMA buffers, or to locate virtual register array.

Constructors

`function new(string name, longint unsigned size, int unsigned n_bits, string access = "RW", int has_coverage = UVM_NO_COVERAGE)`

Function

new

Create a new instance and type-specific configuration

Creates an instance of a memory abstraction class with the specified name.

size specifies the total number of memory locations. *n_bits* specifies the total number of bits in each memory location. *access* specifies the access policy of this memory and may be one of "RW for RAMs and "RO" for ROMs.

has_coverage specifies which functional coverage models are present in the extension of the register abstraction class. Multiple functional coverage models may be specified by adding their symbolic names, as defined by the *uvm_coverage_model_e* type. New

Parameters
- **`name`** (*`string`*)
- **`size`** (*`longint unsigned`*)
- **`n_bits`** (*`int unsigned`*)
- **`access`** (*`string`*)
- **`has_coverage`** (*`int`*)

Enums

`init_e`

Enum Items
- UNKNOWNS
- ZEROES
- ONES
- ADDRESS
- VALUE
- INCR
- DECR

Functions

`function void configure(uvm_reg_block parent, string hdl_path = "")`

Function

configure

Instance-specific configuration

Specify the parent block of this memory.

If this memory is implemented in a single HDL variable, its name is specified as the *hdl_path* . Otherwise, if the memory is implemented as a concatenation of variables (usually one per bank), then the HDL path must be specified using the *add_hdl_path()* or *add_hdl_path_slice()* method. Configure

Parameters

- **`parent`** (*uvm_reg_block*)
- **`hdl_path`** (*`string`*)

`virtual function void set_offset(uvm_reg_map map, uvm_reg_addr_t offset, bit unmapped = 0)`

Function

set_offset

Modify the offset of the memory

The offset of a memory within an address map is set using the *uvm_reg_map::add_mem()* method. This method is used to modify that offset dynamically.

Note

Modifying the offset of a memory will make the abstract model

diverge from the specification that was used to create it. Set_offset

Parameters

- **`map`** (*uvm_reg_map*)
- **`offset`** (*uvm_reg_addr_t*)
- **`unmapped`** (*`bit`*)

`virtual function void set_parent(uvm_reg_block parent)`

Set_parent

Parameters

- **`parent`** (*uvm_reg_block*) -- Local

`function void add_map(uvm_reg_map map)`

Add_map

Parameters

- **`map`** (*uvm_reg_map*) -- Local

`function void Xlock_modelX()`

Xlock_modelXlocal

`function void Xadd_vregX(uvm_vreg vreg)`

Xadd_vregX

Parameters

- **`vreg`** (*uvm_vreg*) -- Local

`function void Xdelete_vregX(uvm_vreg vreg)`

Xdelete_vregX

Parameters

- **`vreg`** (*uvm_vreg*) -- Local

`virtual function string get_full_name()`

Function

get_full_name

Get the hierarchical name

Return the hierarchal name of this memory. The base of the hierarchical name is the root block. Get_full_name

`virtual function uvm_reg_block get_parent()`

Function

get_parent

Get the parent block. Get_parent

Return type

uvm_reg_block

`virtual function uvm_reg_block get_block()`

Get_block

Return type

uvm_reg_block

`virtual function int get_n_maps()`

Function

get_n_maps

Returns the number of address maps this memory is mapped in. Get_n_maps

`function bit is_in_map(uvm_reg_map map)`

Function

is_in_map

Return TRUE if this memory is in the specified address *map* . Is_in_map

Parameters

`map` (*uvm_reg_map*)

`virtual function void get_maps(uvm_reg_map maps)`

Function

get_maps

Returns all of the address *maps* where this memory is mapped. Get_maps

Parameters

`maps` (*uvm_reg_map*)

`function uvm_reg_map get_local_map(uvm_reg_map map, string caller = "")`

Get_local_map

Parameters

`map` (*uvm_reg_map*) -- Local

`caller` (`string`)

Return type

uvm_reg_map

`function uvm_reg_map get_default_map(string caller = "")`

Get_default_map

Parameters

`caller` (`string`) -- Local

Return type

uvm_reg_map

`virtual function string get_rights(uvm_reg_map map = null)`

Function

get_rights

Returns the access rights of this memory.

Returns "RW", "RO" or "WO". The access rights of a memory is always "RW", unless it is a shared memory with access restriction in a particular address map.

If no address map is specified and the memory is mapped in only one address map, that address map is used. If the memory is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the memory is not mapped in the specified address map, an error message is issued and "RW" is returned. Get_rights

Parameters

`map` (*uvm_reg_map*)

`virtual function string get_access(uvm_reg_map map = null)`

Function

get_access

Returns the access policy of the memory when written and read via an address map.

If the memory is mapped in more than one address map, an address *map* must be specified. If access restrictions are present when accessing a memory through the specified address map, the access mode returned takes the access restrictions into account. For example, a read-write memory accessed through a domain with read-only restrictions would return "RO". Get_access

Parameters

map (*uvm_reg_map*)

function longint unsigned get_size()

Function

get_size

Returns the number of unique memory locations in this memory. Get_size

function int unsigned get_n_bytes()

Function

get_n_bytes

Return the width, in number of bytes, of each memory location. Get_n_bytes

function int unsigned get_n_bits()

Function

get_n_bits

Returns the width, in number of bits, of each memory location. Get_n_bits

static function int unsigned get_max_size()

Function

get_max_size

Returns the maximum width, in number of bits, of all memories. Get_max_size

virtual function void get_virtual_registers(uvm_vreg regs)

Function

get_virtual_registers

Return the virtual registers in this memory

Fills the specified array with the abstraction class for all of the virtual registers implemented in this memory. The order in which the virtual registers are located in the array is not specified. Get_virtual_registers

Parameters

regs (*uvm_vreg*)

virtual function void get_virtual_fields(uvm_vreg_field fields)

Function

get_virtual_fields

Return the virtual fields in the memory

Fills the specified dynamic array with the abstraction class for all of the virtual fields implemented in this memory. The order in which the virtual fields are located in the array is not specified. Get_virtual_fields

Parameters

fields (*uvm_vreg_field*)

virtual function uvm_vreg get_vreg_by_name(string name)

Function

get_vreg_by_name

Find the named virtual register

Finds a virtual register with the specified name implemented in this memory and returns its abstraction class instance. If no virtual register with the specified name is found, returns *null* . Get_vreg_by_name

Parameters

name (`string`)

Return type

uvm_vreg

virtual function uvm_vreg_field get_vfield_by_name(string name)

Function

get_vfield_by_name

Find the named virtual field

Finds a virtual field with the specified name implemented in this memory and returns its abstraction class instance. If no virtual field with the specified name is found, returns *null* . Get_vfield_by_name

Parameters
name (*string*)

Return type
uvm_vreg_field

virtual function uvm_vreg get_vreg_by_offset(uvm_reg_addr_t offset, uvm_reg_-map map = null)

Function

get_vreg_by_offset

Find the virtual register implemented at the specified offset

Finds the virtual register implemented in this memory at the specified *offset* in the specified address *map* and returns its abstraction class instance. If no virtual register at the offset is found, returns *null* . Get_vreg_by_offset

Parameters
offset (*uvm_reg_addr_t*)
map (*uvm_reg_map*)

Return type
uvm_vreg

virtual function uvm_reg_addr_t get_offset(uvm_reg_addr_t offset = 0, uvm_reg_-map map = null)

Function

get_offset

Returns the base offset of a memory location

Returns the base offset of the specified location in this memory in an address *map* .

If no address map is specified and the memory is mapped in only one address map, that address map is used. If the memory is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the memory is not mapped in the specified address map, an error message is issued. Get_offset

Parameters
offset (*uvm_reg_addr_t*)
map (*uvm_reg_map*)

Return type
uvm_reg_addr_t

virtual function uvm_reg_addr_t get_address(uvm_reg_addr_t offset = 0, uvm_reg_-map map = null)

Function

get_address

Returns the base external physical address of a memory location

Returns the base external physical address of the specified location in this memory if accessed through the specified address *map* .

If no address map is specified and the memory is mapped in only one address map, that address map is used. If the memory is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the memory is not mapped in the specified address map, an error message is issued. Get_address

Parameters
- **offset** (*uvm_reg_addr_t*)
- **map** (*uvm_reg_map*)

Return type
uvm_reg_addr_t

`virtual function int get_addresses(uvm_reg_addr_t offset = 0, uvm_reg_map map = null, uvm_reg_addr_t addr)`

Function

get_addresses

Identifies the external physical address(es) of a memory location

Computes all of the external physical addresses that must be accessed to completely read or write the specified location in this memory. The addressed are specified in little endian order. Returns the number of bytes transferred on each access.

If no address map is specified and the memory is mapped in only one address map, that address map is used. If the memory is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the memory is not mapped in the specified address map, an error message is issued. Get_addresses

Parameters
- **offset** (*uvm_reg_addr_t*)
- **map** (*uvm_reg_map*)
- **addr** (*uvm_reg_addr_t*)

`function void set_frontdoor(uvm_reg_frontdoor ftdr, uvm_reg_map map = null, string fname = "", int lineno = 0)`

Function

set_frontdoor

Set a user-defined frontdoor for this memory

By default, memories are mapped linearly into the address space of the address maps that instantiate them. If memories are accessed using a different mechanism, a user-defined access mechanism must be defined and associated with the corresponding memory abstraction class

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

Parameters
- **ftdr** (*uvm_reg_frontdoor*)
- **map** (*uvm_reg_map*)
- **fname** (`string`)
- **lineno** (`int`)

`function uvm_reg_frontdoor get_frontdoor(uvm_reg_map map = null)`

Function

get_frontdoor

Returns the user-defined frontdoor for this memory

If *null* , no user-defined frontdoor has been defined. A user-defined frontdoor is defined by using the *uvm_mem::set_frontdoor()* method.

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

Parameters
- **map** (*uvm_reg_map*)

Return type
uvm_reg_frontdoor

`function void set_backdoor(uvm_reg_backdoor bkdr, string fname = "", int lineno = 0)`

Function

set_backdoor

Set a user-defined backdoor for this memory

By default, memories are accessed via the built-in string-based DPI routines if an HDL path has been specified using the *uvm_mem::configure()* or *uvm_mem::add_hdl_path()* method. If this default mechanism is not suitable (e.g. because the memory is not implemented in pure SystemVerilog) a user-defined access mechanism must be defined and associated with the corresponding memory abstraction class. Set_backdoor

Parameters

- **`bkdr`** (*uvm_reg_backdoor*)
- **`fname`** (*`string`*)
- **`lineno`** (*`int`*)

`function uvm_reg_backdoor get_backdoor(bit inherited = 1)`

Function

get_backdoor

Returns the user-defined backdoor for this memory

If *null* , no user-defined backdoor has been defined. A user-defined backdoor is defined by using the *uvm_reg::set_backdoor()* method.

If *inherit* is TRUE, returns the backdoor of the parent block if none have been specified for this memory. Get_backdoor

Parameters

- **`inherited`** (*`bit`*)

Return type

uvm_reg_backdoor

`function void clear_hdl_path(string kind = "RTL")`

Function

clear_hdl_path

Delete HDL paths

Remove any previously specified HDL path to the memory instance for the specified design abstraction. Clear_hdl_path

Parameters

- **`kind`** (*`string`*)

`function void add_hdl_path(uvm_hdl_path_slice slices, string kind = "RTL")`

Function

add_hdl_path

Add an HDL path

Add the specified HDL path to the memory instance for the specified design abstraction. This method may be called more than once for the same design abstraction if the memory is physically duplicated in the design abstraction. Add_hdl_path

Parameters

- **`slices`** (*uvm_hdl_path_slice*)
- **`kind`** (*`string`*)

`function void add_hdl_path_slice(string name, int offset, int size, bit first = 0, string kind = "RTL")`

Function

add_hdl_path_slice

Add the specified HDL slice to the HDL path for the specified design abstraction. If *first* is TRUE, starts the specification of a duplicate HDL implementation of the memory. Add_hdl_path_slice

Parameters

- **`name`** (*`string`*)
- **`offset`** (*`int`*)
- **`size`** (*`int`*)
- **`first`** (*`bit`*)
- **`kind`** (*`string`*)

`function bit has_hdl_path(string kind = "")`

Function

has_hdl_path

Check if a HDL path is specified

Returns TRUE if the memory instance has a HDL path defined for the specified design abstraction. If no design abstraction is specified, uses the default design abstraction specified for the parent block. Has_hdl_path

Parameters

`kind` (`string`)

`function void get_hdl_path(uvm_hdl_path_concat paths, string kind = "")`

Function

get_hdl_path

Get the incremental HDL path(s)

Returns the HDL path(s) defined for the specified design abstraction in the memory instance. Returns only the component of the HDL paths that corresponds to the memory, not a full hierarchical path

If no design abstraction is specified, the default design abstraction for the parent block is used. Get_hdl_path

Parameters

`paths` (*uvm_hdl_path_concat*)
`kind` (`string`)

`function void get_full_hdl_path(uvm_hdl_path_concat paths, string kind = "", string separator = ".")`

Function

get_full_hdl_path

Get the full hierarchical HDL path(s)

Returns the full hierarchical HDL path(s) defined for the specified design abstraction in the memory instance. There may be more than one path returned even if only one path was defined for the memory instance, if any of the parent components have more than one path defined for the same design abstraction

If no design abstraction is specified, the default design abstraction for each ancestor block is used to get each incremental path. Get_full_hdl_path

Parameters

`paths` (*uvm_hdl_path_concat*)
`kind` (`string`)
`separator` (`string`)

`function void get_hdl_path_kinds(string kinds)`

Function

get_hdl_path_kinds

Get design abstractions for which HDL paths have been defined. Get_hdl_path_kinds

Parameters

`kinds` (`string`)

`virtual function uvm_status_e backdoor_read_func(uvm_reg_item rw)`

Function

backdoor_read_func

User-defined backdoor read access

Override the default string-based DPI backdoor access read for this memory type. Backdoor_read_func

Parameters

`rw` (*uvm_reg_item*)

Return type

uvm_status_e

virtual function bit has_coverage(uvm_reg_cvr_t models)

Function

has_coverage

Check if memory has coverage model(s)

Returns TRUE if the memory abstraction class contains a coverage model for all of the models specified. Models are specified by adding the symbolic value of individual coverage model as defined in *uvm_coverage_model_e*. Has_coverage

Parameters

models (*uvm_reg_cvr_t*)

virtual function uvm_reg_cvr_t set_coverage(uvm_reg_cvr_t is_on)

Function

set_coverage

Turns on coverage measurement.

Turns the collection of functional coverage measurements on or off for this memory. The functional coverage measurement is turned on for every coverage model specified using *uvm_coverage_model_e* symbolic identifiers. Multiple functional coverage models can be specified by adding the functional coverage model identifiers. All other functional coverage models are turned off. Returns the sum of all functional coverage models whose measurements were previously on.

This method can only control the measurement of functional coverage models that are present in the memory abstraction classes, then enabled during construction. See the *uvm_mem::has_coverage()* method to identify the available functional coverage models. Set_coverage

Parameters

is_on (*uvm_reg_cvr_t*)

Return type

uvm_reg_cvr_t

virtual function bit get_coverage(uvm_reg_cvr_t is_on)

Function

get_coverage

Check if coverage measurement is on.

Returns TRUE if measurement for all of the specified functional coverage models are currently on. Multiple functional coverage models can be specified by adding the functional coverage model identifiers.

See *uvm_mem::set_coverage()* for more details. Get_coverage

Parameters

is_on (*uvm_reg_cvr_t*)

function void XsampleX(uvm_reg_addr_t addr, bit is_read, uvm_reg_map map)

Parameters

addr (*uvm_reg_addr_t*) -- Local

is_read (`bit`)

map (*uvm_reg_map*)

virtual function void do_print(uvm_printer printer)

Core ovm_object operations. Do_print

Parameters

printer (*uvm_printer*)

virtual function string convert2string()

Convert2string

virtual function uvm_object clone()

Clone

Return type

uvm_object

virtual function void do_copy(uvm_object rhs)

Do_copy

Parameters
- **rhs** (*uvm_object*)

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

Do_compare

Parameters
- **rhs** (*uvm_object*)
- **comparer** (*uvm_comparer*)

`virtual function void do_pack(uvm_packer packer)`

Do_pack

Parameters
- **packer** (*uvm_packer*)

`virtual function void do_unpack(uvm_packer packer)`

Do_unpack

Parameters
- **packer** (*uvm_packer*)

Tasks

`virtual function write(uvm_status_e status, uvm_reg_addr_t offset, uvm_reg_data_t value, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)`

Task

write

Write the specified value in a memory location

Write *value* in the memory location that corresponds to this abstraction class instance at the specified *offset* using the specified access *path* . If the memory is mapped in more than one address map, an address *map* must be specified if a physical access is used (front-door access). If a back-door access path is used, the effect of writing the register through a physical access is mimicked. For example, a read-only memory will not be written. Write

Parameters
- **status** (*uvm_status_e*)
- **offset** (*uvm_reg_addr_t*)
- **value** (*uvm_reg_data_t*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **parent** (*uvm_sequence_base*)
- **prior** (`int`)
- **extension** (*uvm_object*)
- **fname** (`string`)
- **lineno** (`int`)

`virtual function read(uvm_status_e status, uvm_reg_addr_t offset, uvm_reg_data_t value, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)`

Task

read

Read the current value from a memory location

Read and return *value* from the memory location that corresponds to this abstraction class instance at the specified *offset* using the specified access *path* . If the register is mapped in more than one address map, an address *map* must be specified if a physical access is used (front-door access). Read

Parameters
- **status** (*uvm_status_e*)
- **offset** (*uvm_reg_addr_t*)

value (*uvm_reg_data_t*)
path (*uvm_path_e*)
map (*uvm_reg_map*)
parent (*uvm_sequence_base*)
prior (*int*)
extension (*uvm_object*)
fname (*string*)
lineno (*int*)

```
virtual  function  burst_write(uvm_status_e status, uvm_reg_addr_t offset, uvm_reg_-
data_t value, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_-
sequence_base parent = null, int prior = -1, uvm_object extension = null,
string fname = "", int lineno = 0)
```

Task

burst_write

Write the specified values in memory locations

Burst-write the specified *values* in the memory locations beginning at the specified *offset* . If the memory is mapped in more than one address map, an address *map* must be specified if not using the backdoor. If a back-door access path is used, the effect of writing the register through a physical access is mimicked. For example, a read-only memory will not be written. Burst_write

Parameters

status (*uvm_status_e*)
offset (*uvm_reg_addr_t*)
value (*uvm_reg_data_t*)
path (*uvm_path_e*)
map (*uvm_reg_map*)
parent (*uvm_sequence_base*)
prior (*int*)
extension (*uvm_object*)
fname (*string*)
lineno (*int*)

```
virtual  function  burst_read(uvm_status_e status, uvm_reg_addr_t offset, uvm_reg_-
data_t value, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_-
sequence_base parent = null, int prior = -1, uvm_object extension = null,
string fname = "", int lineno = 0)
```

Task

burst_read

Read values from memory locations

Burst-read into *values* the data the memory locations beginning at the specified *offset* . If the memory is mapped in more than one address map, an address *map* must be specified if not using the backdoor. If a back-door access path is used, the effect of writing the register through a physical access is mimicked. For example, a read-only memory will not be written. Burst_read

Parameters

status (*uvm_status_e*)
offset (*uvm_reg_addr_t*)
value (*uvm_reg_data_t*)
path (*uvm_path_e*)
map (*uvm_reg_map*)
parent (*uvm_sequence_base*)
prior (*int*)
extension (*uvm_object*)
fname (*string*)
lineno (*int*)

```
virtual  function  poke(uvm_status_e status, uvm_reg_addr_t offset, uvm_reg_data_-
t value, string kind = "", uvm_sequence_base parent = null, uvm_-
object extension = null, string fname = "", int lineno = 0)
```

Task

poke

Deposit the specified value in a memory location

Deposit the value in the DUT memory location corresponding to this abstraction class instance at the specified *offset* , as-is, using a back-door access.

Uses the HDL path for the design abstraction specified by *kind* . Poke

Parameters

- **`status`** (*uvm_status_e*)
- **`offset`** (*uvm_reg_addr_t*)
- **`value`** (*uvm_reg_data_t*)
- **`kind`** (*`string`*)
- **`parent`** (*uvm_sequence_base*)
- **`extension`** (*uvm_object*)
- **`fname`** (*`string`*)
- **`lineno`** (*`int`*)

```
virtual  function  peek(uvm_status_e status, uvm_reg_addr_t offset, uvm_reg_data_-
t value, string kind = "", uvm_sequence_base parent = null, uvm_-
object extension = null, string fname = "", int lineno = 0)
```

Task

peek

Read the current value from a memory location

Sample the value in the DUT memory location corresponding to this abstraction class instance at the specified *offset* using a back-door access. The memory location value is sampled, not modified.

Uses the HDL path for the design abstraction specified by *kind* . Peek

Parameters

- **`status`** (*uvm_status_e*)
- **`offset`** (*uvm_reg_addr_t*)
- **`value`** (*uvm_reg_data_t*)
- **`kind`** (*`string`*)
- **`parent`** (*uvm_sequence_base*)
- **`extension`** (*uvm_object*)
- **`fname`** (*`string`*)
- **`lineno`** (*`int`*)

```
virtual  function  do_write(uvm_reg_item rw)
```

Do_write

Parameters

- **`rw`** (*uvm_reg_item*)

```
virtual  function  do_read(uvm_reg_item rw)
```

Do_read

Parameters

- **`rw`** (*uvm_reg_item*)

```
virtual  function  backdoor_write(uvm_reg_item rw)
```

Function

backdoor_write

User-defined backdoor read access

Override the default string-based DPI backdoor access write for this memory type. Backdoor_write

Parameters

- **`rw`** (*uvm_reg_item*)

```
virtual  function  pre_write(uvm_reg_item rw)
```

Task

pre_write

Called before memory write.

If the *offset* , *value* , access *path* , or address *map* are modified, the updated offset, data value, access path or address map will be used to perform the memory operation. If the *status* is modified to anything other than <UVM_IS_OK>, the operation is aborted.

The registered callback methods are invoked after the invocation of this method.

Parameters

`rw` (*uvm_reg_item*)

`virtual function post_write(uvm_reg_item rw)`

Task

post_write

Called after memory write.

If the *status* is modified, the updated status will be returned by the memory operation.

The registered callback methods are invoked before the invocation of this method.

Parameters

`rw` (*uvm_reg_item*)

`virtual function pre_read(uvm_reg_item rw)`

Task

pre_read

Called before memory read.

If the *offset* , access *path* or address *map* are modified, the updated offset, access path or address map will be used to perform the memory operation. If the *status* is modified to anything other than <UVM_IS_OK>, the operation is aborted.

The registered callback methods are invoked after the invocation of this method.

Parameters

`rw` (*uvm_reg_item*)

`virtual function post_read(uvm_reg_item rw)`

Task

post_read

Called after memory read.

If the readback data or *status* is modified, the updated readback //data or status will be returned by the memory operation.

The registered callback methods are invoked before the invocation of this method.

Parameters

`rw` (*uvm_reg_item*)

15.1.1.102 Class uvm_pkg::uvm_mem_access_seq

uvm_pkg :: *uvm_void*
  ↪*uvm_pkg* :: *uvm_object*
    ↪*uvm_pkg* :: *uvm_transaction*
      ↪*uvm_pkg* :: *uvm_sequence_item*
        ↪*uvm_pkg* :: *uvm_sequence_base*
          ↪*uvm_pkg* :: *uvm_sequence*
            ↪*uvm_pkg* :: *uvm_reg_sequence*
              ↪*uvm_pkg* :: *uvm_mem_access_seq*

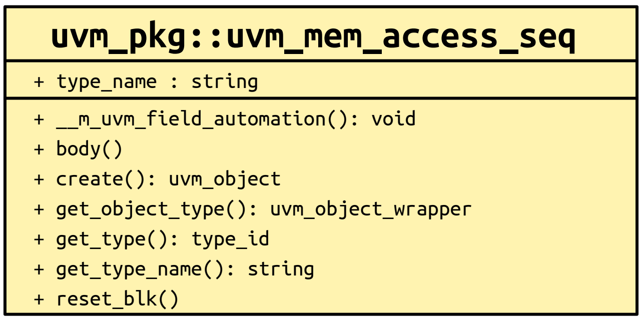

uvm_pkg::uvm_mem_access_seq
+ type_name : string
+ __m_uvm_field_automation(): void
+ body()
+ create(): uvm_object
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string
+ reset_blk()

Fig. 27: Collaboration Diagram of uvm_mem_access_seq

class

uvm_mem_access_seq

Verify the accessibility of all memories in a block by executing the *uvm_mem_single_access_seq* sequence on every memory within it.

If bit-type resource named "NO_REG_TESTS", "NO_MEM_TESTS", or "NO_MEM_ACCESS_TEST" in the "REG::" namespace matches the full name of the block, the block is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.get_full_name(),".*"},
                           "NO_MEM_TESTS", 1, this);
```

Constructors

function new(string name = "uvm_mem_access_seq")

Parameters
name (*string*)

Tasks

virtual function body()

Task

body

Execute the Memory Access sequence. Do not call directly. Use seq.start() instead.

virtual function reset_blk(uvm_reg_block blk)

Task

reset_blk

Reset the DUT that corresponds to the specified block abstraction class.

Currently empty. Will rollback the environment's phase to the *reset* phase once the new phasing is available.

In the meantime, the DUT should be reset before executing this test sequence or this method should be implemented in an extension to reset the DUT.

Parameters

`blk` (*uvm_reg_block*)

15.1.1.103 Class uvm_pkg::uvm_mem_mam

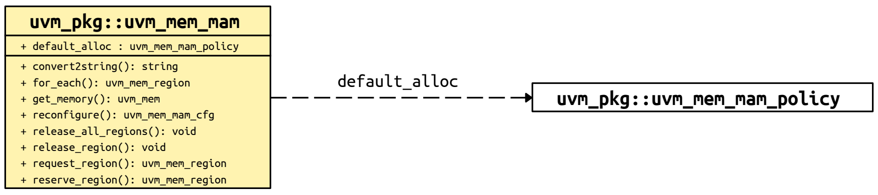

uvm_pkg::uvm_mem_mam
+ default_alloc : uvm_mem_mam_policy
+ convert2string(): string
+ for_each(): uvm_mem_region
+ get_memory(): uvm_mem
+ reconfigure(): uvm_mem_mam_cfg
+ release_all_regions(): void
+ release_region(): void
+ request_region(): uvm_mem_region
+ reserve_region(): uvm_mem_region
default_alloc
uvm_pkg::uvm_mem_mam_policy

Fig. 28: Collaboration Diagram of uvm_mem_mam

CLASS

uvm_mem_mam

Memory allocation manager

Memory allocation management utility class similar to C's malloc() and free(). A single instance of this class is used to manage a single, contiguous address space.

Table 113: Variables

Name	Type	Description
default_alloc	*uvm_mem_mam_policy*	***Variable*** default_alloc Region allocation policy This object is repeatedly randomized when allocating new regions.

Constructors

`function new(string name, uvm_mem_mam_cfg cfg, uvm_mem mem = null)`

Function

new

Create a new manager instance

Create an instance of a memory allocation manager with the specified name and configuration. This instance manages all memory region allocation within the address range specified in the configuration descriptor.

If a reference to a memory abstraction class is provided, the memory locations within the regions can be accessed through the region descriptor, using the *uvm_mem_region::read()* and *uvm_mem_region::write()* methods.

Parameters

- **`name`** (`string`)
- **`cfg`** (*uvm_mem_mam_cfg*)
- **`mem`** (*uvm_mem*)

Enums

`alloc_mode_e`

Type

alloc_mode_e

Memory allocation mode

Specifies how to allocate a memory region

GREEDY

Consume new, previously unallocated memory

THRIFTY

Reused previously released memory as much as possible (not yet implemented)

Enum Items
- GREEDY
- THRIFTY

`locality_e`

Type

locality_e

Location of memory regions

Specifies where to locate new memory regions

BROAD

Locate new regions randomly throughout the address space

NEARBY

Locate new regions adjacent to existing regions

Enum Items
- BROAD
- NEARBY

Functions

`function uvm_mem_mam_cfg reconfigure(uvm_mem_mam_cfg cfg = null)`

Function

reconfigure

Reconfigure the manager

Modify the maximum and minimum addresses of the address space managed by the allocation manager, allocation mode, or locality. The number of bytes per memory location cannot be modified once an allocation manager has been constructed. All currently allocated regions must fall within the new address space.

Returns the previous configuration.

if no new configuration is specified, simply returns the current configuration.

Parameters
- **`cfg`** (*uvm_mem_mam_cfg*)

Return type
- *uvm_mem_mam_cfg*

`function uvm_mem_region reserve_region(bit[63:0] start_offset, int unsigned n_bytes, string fname = "", int lineno = 0)`

Function

reserve_region

Reserve a specific memory region

Reserve a memory region of the specified number of bytes starting at the specified offset. A descriptor of the reserved region is returned. If the specified region cannot be reserved, *null* is returned.

It may not be possible to reserve a region because it overlaps with an already-allocated region or it lies outside the address range managed by the memory manager.

Regions can be reserved to create "holes" in the managed address space.

Parameters
- **start_offset** (*bit[63:0]*)
- **n_bytes** (*int unsigned*)
- **fname** (*string*)
- **lineno** (*int*)

Return type
uvm_mem_region

function uvm_mem_region request_region(int unsigned n_bytes, uvm_mem_mam_-policy alloc = null, string fname = "", int lineno = 0)

Function

request_region

Request and reserve a memory region

Request and reserve a memory region of the specified number of bytes starting at a random location. If an policy is specified, it is randomized to determine the start offset of the region. If no policy is specified, the policy found in the *uvm_mem_mam::default_alloc* class property is randomized.

A descriptor of the allocated region is returned. If no region can be allocated, *null* is returned.

It may not be possible to allocate a region because there is no area in the memory with enough consecutive locations to meet the size requirements or because there is another contradiction when randomizing the policy.

If the memory allocation is configured to *THRIFTY* or *NEARBY* , a suitable region is first sought procedurally.

Parameters
- **n_bytes** (*int unsigned*)
- **alloc** (*uvm_mem_mam_policy*)
- **fname** (*string*)
- **lineno** (*int*)

Return type
uvm_mem_region

function void release_region(uvm_mem_region region)

Function

release_region

Release the specified region

Release a previously allocated memory region. An error is issued if the specified region has not been previously allocated or is no longer allocated.

Parameters
- **region** (*uvm_mem_region*)

function void release_all_regions()

Function

release_all_regions

Forcibly release all allocated memory regions.

function string convert2string()

Function

convert2string

Image of the state of the manager

Create a human-readable description of the state of the memory manager and the currently allocated regions.

function uvm_mem_region for_each(bit reset = 0)

Function

for_each

Iterate over all currently allocated regions

If reset is *TRUE* , reset the iterator and return the first allocated region. Returns *null* when there are no additional allocated regions to iterate on.

Parameters
`reset` (*`bit`*)

Return type
uvm_mem_region

`function uvm_mem get_memory()`

Function

get_memory

Get the managed memory implementation

Return the reference to the memory abstraction class for the memory implementing the locations managed by this instance of the allocation manager. Returns *null* if no memory abstraction class was specified at construction time.

Return type
uvm_mem

15.1.1.104 Class uvm_pkg::uvm_mem_mam_cfg

CLASS

uvm_mem_mam_cfg

Specifies the memory managed by an instance of a *uvm_mem_mam* memory allocation manager class.

Table 114: Variables

Name	Type	Description
n_bytes	int unsigned	***variable*** n_bytes Number of bytes in each memory location
start_offset	bit[63:0]	FIXME start_offset and end_offset should be "longint unsigned" to match the memory addr types variable: start_offset Lowest address of managed space
end_offset	bit[63:0]	***variable*** end_offset Last address of managed space
mode	*uvm_mem_mam::alloc_-mode_e*	***variable*** mode Region allocation mode
locality	*uvm_mem_mam::local-ity_e*	***variable*** locality Region location mode

Table 115: Constraints

Name	Description
uvm_mem_mam_cfg_-valid	

15.1.1.105 Class uvm_pkg::uvm_mem_mam_policy

Class

uvm_mem_mam_policy

An instance of this class is randomized to determine the starting offset of a randomly allocated memory region. This class can be extended to provide additional constraints on the starting offset, such as word alignment or location of the region within a memory page. If a procedural region allocation policy is required, it can be implemented in the pre/post_randomize() method.

Table 116: Variables

Name	Type	Description
len	int unsigned	***variable*** len Number of addresses required
start_offset	bit[63:0]	***variable*** start_offset The starting offset of the region
min_offset	bit[63:0]	***variable*** min_offset Minimum address offset in the managed address space
max_offset	bit[63:0]	***variable*** max_offset Maximum address offset in the managed address space
in_use	*uvm_mem_region*	***variable*** in_use Regions already allocated in the managed address space

Table 117: Constraints

Name	Description
uvm_mem_mam_pol-icy_no_overlap	
uvm_mem_mam_pol-icy_valid	

15.1.1.106 Class uvm_pkg::uvm_mem_region

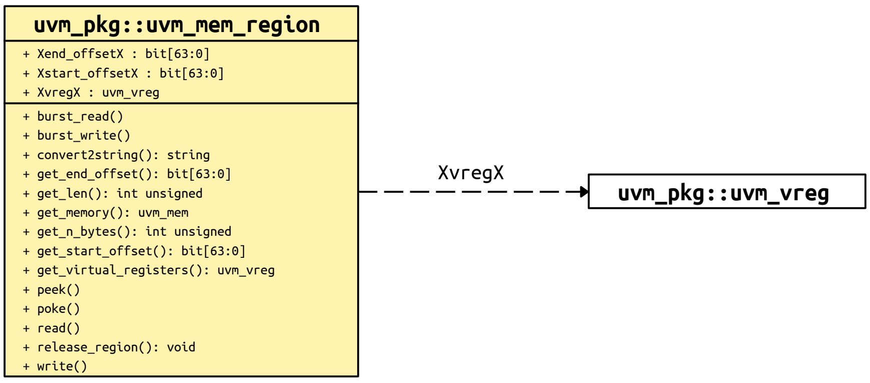

uvm_pkg::uvm_mem_region
+ Xend_offsetX : bit[63:0]
+ Xstart_offsetX : bit[63:0]
+ XvregX : uvm_vreg
+ burst_read()
+ burst_write()
+ convert2string(): string
+ get_end_offset(): bit[63:0]
+ get_len(): int unsigned
+ get_memory(): uvm_mem
+ get_n_bytes(): int unsigned
+ get_start_offset(): bit[63:0]
+ get_virtual_registers(): uvm_vreg
+ peek()
+ poke()
+ read()
+ release_region(): void
+ write()
XvregX
uvm_pkg::uvm_vreg

Fig. 29: Collaboration Diagram of uvm_mem_region

CLASS

uvm_mem_region

Allocated memory region descriptor

Each instance of this class describes an allocated memory region. Instances of this class are created only by the memory manager, and returned by the *uvm_mem_mam::reserve_region()* and *uvm_mem_mam::request_region()* methods.

Table 118: Variables

Name	Type	Description
Xstart_offsetX	bit[63:0]	Can't be local since function
Xend_offsetX	bit[63:0]	calls not supported in constraints
XvregX	*uvm_vreg*	Local

Constructors

`function new(bit[63:0] start_offset, bit[63:0] end_offset, int unsigned len, int unsigned n_bytes, uvm_mem_mam parent)`

Implementation

Parameters

- **`start_offset`** (*`bit[63:0]`*) -- Local
- **`end_offset`** (*`bit[63:0]`*)
- **`len`** (*`int unsigned`*)
- **`n_bytes`** (*`int unsigned`*)
- **`parent`** (*uvm_mem_mam*)

Functions

`function bit[63:0] get_start_offset()`

> ***Function***
>
> get_start_offset
>
> Get the start offset of the region
>
> Return the address offset, within the memory, where this memory region starts.

`function bit[63:0] get_end_offset()`

> ***Function***
>
> get_end_offset
>
> Get the end offset of the region
>
> Return the address offset, within the memory, where this memory region ends.

`function int unsigned get_len()`

> ***Function***
>
> get_len
>
> Size of the memory region
>
> Return the number of consecutive memory locations (not necessarily bytes) in the allocated region.

`function int unsigned get_n_bytes()`

> ***Function***
>
> get_n_bytes
>
> Number of bytes in the region
>
> Return the number of consecutive bytes in the allocated region. If the managed memory contains more than one byte per address, the number of bytes in an allocated region may be greater than the number of requested or reserved bytes.

`function void release_region()`

> ***Function***
>
> release_region
>
> Release this region

`function uvm_mem get_memory()`

> ***Function***
>
> get_memory
>
> Get the memory where the region resides
>
> Return a reference to the memory abstraction class for the memory implementing this allocated memory region. Returns *null* if no memory abstraction class was specified for the allocation manager that allocated this region.
>
> Return type
>
> *uvm_mem*

`function uvm_vreg get_virtual_registers()`

> ***Function***
>
> get_virtual_registers
>
> Get the virtual register array in this region
>
> Return a reference to the virtual register array abstraction class implemented in this region. Returns *null* if the memory region is not known to implement virtual registers.
>
> Return type
>
> *uvm_vreg*

`function string convert2string()`

Tasks

function write(uvm_status_e status, uvm_reg_addr_t offset, uvm_reg_data_t value, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_-base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)

Task

write

Write to a memory location in the region.

Write to the memory location that corresponds to the specified *offset* within this region. Requires that the memory abstraction class be associated with the memory allocation manager that allocated this region.

See *uvm_mem::write()* for more details.

Parameters

- **status** (*uvm_status_e*)
- **offset** (*uvm_reg_addr_t*)
- **value** (*uvm_reg_data_t*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **parent** (*uvm_sequence_base*)
- **prior** (`int`)
- **extension** (*uvm_object*)
- **fname** (`string`)
- **lineno** (`int`)

function read(uvm_status_e status, uvm_reg_addr_t offset, uvm_reg_data_t value, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_-base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)

Task

read

Read from a memory location in the region.

Read from the memory location that corresponds to the specified *offset* within this region. Requires that the memory abstraction class be associated with the memory allocation manager that allocated this region.

See *uvm_mem::read()* for more details.

Parameters

- **status** (*uvm_status_e*)
- **offset** (*uvm_reg_addr_t*)
- **value** (*uvm_reg_data_t*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **parent** (*uvm_sequence_base*)
- **prior** (`int`)
- **extension** (*uvm_object*)
- **fname** (`string`)
- **lineno** (`int`)

function burst_write(uvm_status_e status, uvm_reg_addr_t offset, uvm_reg_data_-t value, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_-base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)

Task

burst_write

Write to a set of memory location in the region.

Write to the memory locations that corresponds to the specified *burst* within this region. Requires that the memory abstraction class be associated with the memory allocation manager that allocated this region.

See *uvm_mem::burst_write()* for more details.

Parameters

- **`status`** (*uvm_status_e*)
- **`offset`** (*uvm_reg_addr_t*)
- **`value`** (*uvm_reg_data_t*)
- **`path`** (*uvm_path_e*)
- **`map`** (*uvm_reg_map*)
- **`parent`** (*uvm_sequence_base*)
- **`prior`** (`int`)
- **`extension`** (*uvm_object*)
- **`fname`** (`string`)
- **`lineno`** (`int`)

```
function burst_read(uvm_status_e status, uvm_reg_addr_t offset, uvm_reg_data_-
t value, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_-
base parent = null, int prior = -1, uvm_object extension = null, string fname = "",
int lineno = 0)
```

Task

burst_read

Read from a set of memory location in the region.

Read from the memory locations that corresponds to the specified *burst* within this region. Requires that the memory abstraction class be associated with the memory allocation manager that allocated this region.

See *uvm_mem::burst_read()* for more details.

Parameters

- **`status`** (*uvm_status_e*)
- **`offset`** (*uvm_reg_addr_t*)
- **`value`** (*uvm_reg_data_t*)
- **`path`** (*uvm_path_e*)
- **`map`** (*uvm_reg_map*)
- **`parent`** (*uvm_sequence_base*)
- **`prior`** (`int`)
- **`extension`** (*uvm_object*)
- **`fname`** (`string`)
- **`lineno`** (`int`)

```
function poke(uvm_status_e status, uvm_reg_addr_t offset, uvm_reg_data_t value,
uvm_sequence_base parent = null, uvm_object extension = null, string fname = "",
int lineno = 0)
```

Task

poke

Deposit in a memory location in the region.

Deposit the specified value in the memory location that corresponds to the specified *offset* within this region. Requires that the memory abstraction class be associated with the memory allocation manager that allocated this region.

See *uvm_mem::poke()* for more details.

Parameters

- **`status`** (*uvm_status_e*)
- **`offset`** (*uvm_reg_addr_t*)
- **`value`** (*uvm_reg_data_t*)
- **`parent`** (*uvm_sequence_base*)
- **`extension`** (*uvm_object*)
- **`fname`** (`string`)
- **`lineno`** (`int`)

```
function peek(uvm_status_e status, uvm_reg_addr_t offset, uvm_reg_data_t value,
uvm_sequence_base parent = null, uvm_object extension = null, string fname = "",
int lineno = 0)
```

Task

peek

Sample a memory location in the region.

Sample the memory location that corresponds to the specified *offset* within this region. Requires that the memory abstraction class be associated with the memory allocation manager that allocated this region.

See *uvm_mem::peek()* for more details.

Parameters

- **`status`** (*uvm_status_e*)
- **`offset`** (*uvm_reg_addr_t*)
- **`value`** (*uvm_reg_data_t*)
- **`parent`** (*uvm_sequence_base*)
- **`extension`** (*uvm_object*)
- **`fname`** (*`string`*)
- **`lineno`** (*`int`*)

15.1.1.107 Class uvm_pkg::uvm_mem_shared_access_seq

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*
↪*uvm_pkg* :: *uvm_reg_sequence*
↪*uvm_pkg* :: *uvm_mem_shared_access_seq*

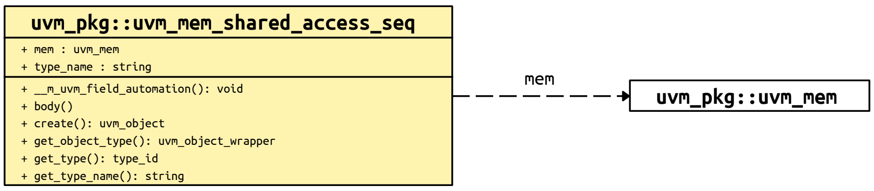

uvm_pkg::uvm_mem_shared_access_seq
+ mem : uvm_mem
+ type_name : string
+ __m_uvm_field_automation(): void
+ body()
+ create(): uvm_object
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string
mem
uvm_pkg::uvm_mem

Fig. 30: Collaboration Diagram of uvm_mem_shared_access_seq

Class

uvm_mem_shared_access_seq

Verify the accessibility of a shared memory by writing through each address map then reading it via every other address maps in which the memory is readable and the backdoor, making sure that the resulting value matches the written value.

If bit-type resource named "NO_REG_TESTS", "NO_MEM_TESTS", "NO_REG_SHARED_ACCESS_TEST" or "NO_MEM_SHARED_ACCESS_TEST" in the "REG::" namespace matches the full name of the memory, the memory is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.mem0.get_full_name()},
                           "NO_MEM_TESTS", 1, this);
```

The DUT should be idle and not modify the memory during this test.

Table 119: Variables

Name	Type	Description
mem	*uvm_mem*	***variable*** mem The memory to be tested

Constructors

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

Parameters
`name` (`string`)

Tasks

`virtual function body()`

15.1.1.108 Class uvm_pkg::uvm_mem_single_access_seq

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*
↪*uvm_pkg* :: *uvm_reg_sequence*
↪*uvm_pkg* :: *uvm_mem_single_access_seq*

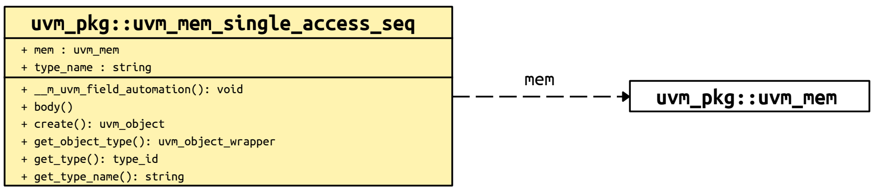

uvm_pkg::uvm_mem_single_access_seq
+ mem : uvm_mem
+ type_name : string
+ __m_uvm_field_automation(): void
+ body()
+ create(): uvm_object
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string
mem
uvm_pkg::uvm_mem

Fig. 31: Collaboration Diagram of uvm_mem_single_access_seq

class

uvm_mem_single_access_seq

Verify the accessibility of a memory by writing through its default address map then reading it via the backdoor, then reversing the process, making sure that the resulting value matches the written value.

If bit-type resource named "NO_REG_TESTS", "NO_MEM_TESTS", or "NO_MEM_ACCESS_TEST" in the "REG::" namespace matches the full name of the memory, the memory is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.mem0.get_full_name()},
                           "NO_MEM_TESTS", 1, this);
```

Memories without an available backdoor cannot be tested.

The DUT should be idle and not modify the memory during this test.

Table 120: Variables

Name	Type	Description
mem	*uvm_mem*	***Variable*** mem The memory to be tested

Constructors

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

Parameters
name (*string*)

Tasks

`virtual function body()`

15.1.1.109 Class uvm_pkg::uvm_mem_single_walk_seq

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*
↪*uvm_pkg* :: *uvm_reg_sequence*
↪*uvm_pkg* :: *uvm_mem_single_walk_seq*

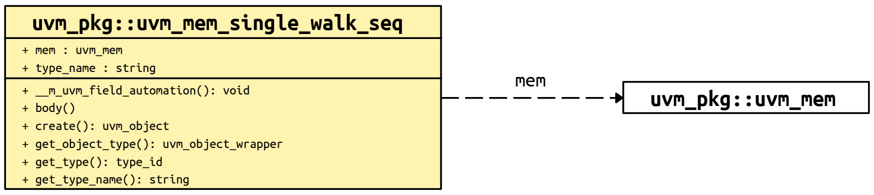

uvm_pkg::uvm_mem_single_walk_seq
+ mem : uvm_mem
+ type_name : string
+ __m_uvm_field_automation(): void
+ body()
+ create(): uvm_object
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string
mem
uvm_pkg::uvm_mem

Fig. 32: Collaboration Diagram of uvm_mem_single_walk_seq

Class

uvm_mem_single_walk_seq

Runs the walking-ones algorithm on the memory given by the *mem* property, which must be assigned prior to starting this sequence.

If bit-type resource named "NO_REG_TESTS", "NO_MEM_TESTS", or "NO_MEM_WALK_TEST" in the "REG::" namespace matches the full name of the memory, the memory is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.mem0.get_full_name()},
                           "NO_MEM_TESTS", 1, this);
```

The walking ones algorithm is performed for each map in which the memory is defined.

```
for (k = 0 thru memsize-1)
  write addr=k data=~k
  if (k > 0) {
    read addr=k-1, expect data=~(k-1)
    write addr=k-1 data=k-1
  if (k == last addr)
    read addr=k, expect data=~k
```

Table 121: Variables

Name	Type	Description
mem	*uvm_mem*	***Variable*** mem The memory to test; must be assigned prior to starting sequence.

Constructors

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

Function

new

Creates a new instance of the class with the given name.

Parameters

`name` (*`string`*)

Tasks

`virtual function body()`

Task

body

Performs the walking-ones algorithm on each map of the memory specified in *mem*.

15.1.1.110 Class uvm_pkg::uvm_mem_walk_seq

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*
↪*uvm_pkg* :: *uvm_reg_sequence*
↪*uvm_pkg* :: *uvm_mem_walk_seq*

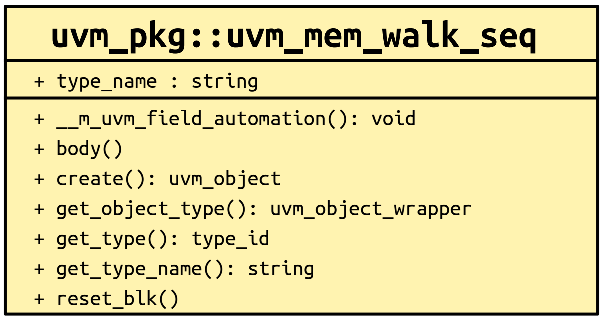

uvm_pkg::uvm_mem_walk_seq
+ type_name : string
+ __m_uvm_field_automation(): void
+ body()
+ create(): uvm_object
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string
+ reset_blk()

Fig. 33: Collaboration Diagram of uvm_mem_walk_seq

Class

uvm_mem_walk_seq

Verifies the all memories in a block by executing the *uvm_mem_single_walk_seq* sequence on every memory within it.

If bit-type resource named "NO_REG_TESTS", "NO_MEM_TESTS", or "NO_MEM_WALK_TEST" in the "REG::" namespace matches the full name of the block, the block is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.get_full_name(),".*"},
                           "NO_MEM_TESTS", 1, this);
```

Constructors

function new(string name = "uvm_mem_walk_seq")

Parameters
name (*string*)

Tasks

virtual function body()

Task

body

Executes the mem walk sequence, one block at a time. Do not call directly. Use seq.start() instead.

virtual function reset_blk(uvm_reg_block blk)

Task

reset_blk

Reset the DUT that corresponds to the specified block abstraction class.

Currently empty. Will rollback the environment's phase to the *reset* phase once the new phasing is available.

In the meantime, the DUT should be reset before executing this test sequence or this method should be implemented in an extension to reset the DUT.

Parameters

`blk` (*uvm_reg_block*)

15.1.1.111 Class uvm_pkg::uvm_monitor

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_monitor*

CLASS

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.

Table 122: Variables

Name	Type	Description
type_name	string	

Constructors

`function new(string name, uvm_component parent)`

Function

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.

Parameters

`name` (`string`)
`parent` (*uvm_component*)

Functions

`virtual function string get_type_name()`

15.1.1.112 Class uvm_pkg::uvm_nonblocking_get_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_get_export*

Table 123: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.1.1.113 Class uvm_pkg::uvm_nonblocking_get_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_get_imp*

Table 124: Parameters

Name	Default value	Description
T	int	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **`name`** (*`string`*)
- **`imp`** (*`int`*)

15.1.1.114 Class uvm_pkg::uvm_nonblocking_get_peek_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_get_peek_export*

Table 125: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.115 Class uvm_pkg::uvm_nonblocking_get_peek_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_get_peek_imp*

Table 126: Parameters

Name	Default value	Description
T	int	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **`name`** (*`string`*)
- **`imp`** (*`int`*)

15.1.1.116 Class uvm_pkg::uvm_nonblocking_get_peek_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_get_peek_port*

Table 127: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.1.1.117 Class uvm_pkg::uvm_nonblocking_get_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_get_port*

Table 128: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.118 Class uvm_pkg::uvm_nonblocking_master_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_master_export*

Table 129: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.119 Class uvm_pkg::uvm_nonblocking_master_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_master_imp*

Table 130: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	
IMP	int	
REQ_IMP	IMP	
RSP_IMP	IMP	

Table 131: Typedefs

Name	Actual Type	Description
this_imp_type	IMP	
this_req_type	REQ_IMP	
this_rsp_type	RSP_IMP	

Constructors

`function new(string name, this_imp_type imp, this_req_type req_imp = null, this_rsp_type rsp_imp = null)`

Parameters
- **`name`** (`string`)
- **`imp`** (*this_imp_type*)
- **`req_imp`** (*this_req_type*)
- **`rsp_imp`** (*this_rsp_type*)

15.1.1.120 Class uvm_pkg::uvm_nonblocking_master_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_master_port*

Table 132: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.1.1.121 Class uvm_pkg::uvm_nonblocking_peek_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_peek_export*

Table 133: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters

- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.1.1.122 Class uvm_pkg::uvm_nonblocking_peek_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_peek_imp*

Table 134: Parameters

Name	Default value	Description
T	int	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **`name`** (*`string`*)
- **`imp`** (*`int`*)

15.1.1.123 Class uvm_pkg::uvm_nonblocking_peek_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_peek_port*

Table 135: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.124 Class uvm_pkg::uvm_nonblocking_put_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_put_export*

Table 136: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters

- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.1.1.125 Class uvm_pkg::uvm_nonblocking_put_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_put_imp*

Table 137: Parameters

Name	Default value	Description
T	int	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **`name`** (*`string`*)
- **`imp`** (*`int`*)

15.1.1.126 Class uvm_pkg::uvm_nonblocking_put_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_put_port*

Table 138: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.1.1.127 Class uvm_pkg::uvm_nonblocking_slave_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_slave_export*

Table 139: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.128 Class uvm_pkg::uvm_nonblocking_slave_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_slave_imp*

Table 140: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	
IMP	int	
REQ_IMP	IMP	
RSP_IMP	IMP	

Table 141: Typedefs

Name	Actual Type	Description
this_imp_type	IMP	
this_req_type	REQ_IMP	
this_rsp_type	RSP_IMP	

Constructors

function new(string name, this_imp_type imp, this_req_type req_imp = null, this_-rsp_type rsp_imp = null)

Parameters

name (`string`)
imp (*this_imp_type*)
req_imp (*this_req_type*)
rsp_imp (*this_rsp_type*)

15.1.1.129 Class uvm_pkg::uvm_nonblocking_slave_port

uvm_pkg :: *uvm_tlm_if_base*
  ↪*uvm_pkg* :: *uvm_port_base*
    ↪*uvm_pkg* :: *uvm_nonblocking_slave_port*

Table 142: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.1.1.130 Class uvm_pkg::uvm_nonblocking_transport_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_transport_export*

Table 143: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.131 Class uvm_pkg::uvm_nonblocking_transport_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_transport_imp*

Table 144: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **name** (*string*)
- **imp** (*int*)

15.1.1.132 Class uvm_pkg::uvm_nonblocking_transport_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_nonblocking_transport_port*

Table 145: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.133 Class uvm_pkg::uvm_obj_rsrc

uvm_pkg :: *uvm_void*
  ↪*uvm_pkg* :: *uvm_object*
    ↪*uvm_pkg* :: *uvm_resource_base*
      ↪*uvm_pkg* :: *uvm_resource*
        ↪*uvm_pkg* :: *uvm_obj_rsrc*

uvm_obj_rsrc

specialization of uvm_resource (T) for T = uvm_object

Table 146: Typedefs

Name	Actual Type	Description
this_subtype	*uvm_obj_rsrc*	

Constructors

```
function  new(string name, string s = "*")
```

Parameters
- **name** (*string*)
- **s** (*string*)

15.1.1.134 Class uvm_pkg::uvm_object_registry

uvm_pkg :: *uvm_object_wrapper*
↪*uvm_pkg* :: *uvm_object_registry*

CLASS

uvm_object_registry (T, Tname)

The uvm_object_registry serves as a lightweight proxy for a *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.

Table 147: Parameters

Name	Default value	Description
T	uvm_object	
Tname	"<unknown>"	

Table 148: Variables

Name	Type	Description
type_name	string	

Table 149: Typedefs

Name	Actual Type	Description
this_type	*uvm_object_registry#(T, Tname)*	

Functions

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

Function

create_object

Creates an object of type *T* and returns it as a handle to a *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.

Parameters
: **name** (`string`)

Return type
: *uvm_object*

`virtual function string get_type_name()`

Function

get_type_name

Returns the value given by the string parameter, *Tname* . This method overrides the method in *uvm_object_wrapper*.

`static function this_type get()`

> ***Function***
>
> get
>
> Returns the singleton instance of this type. Type-based factory operation depends on there being a single proxy instance for each registered type.
>
> Return type
>
> *this_type*

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

> ***Function***
>
> 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. The *contxt* argument, if supplied, supersedes the *parent* 's context. The new instance will have the given leaf *name* , if provided.
>
> Parameters
>
> **`name`** (`string`)
> **`parent`** (*uvm_component*)
> **`contxt`** (`string`)
>
> Return type
>
> *T*

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

> ***Function***
>
> 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. The original type, *T* , is typically a super class of the override type.
>
> Parameters
>
> **`override_type`** (*uvm_object_wrapper*)
> **`replace`** (`bit`)

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

> ***Function***
>
> 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. 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.
>
> Parameters
>
> **`override_type`** (*uvm_object_wrapper*)
> **`inst_path`** (`string`)
> **`parent`** (*uvm_component*)

15.1.1.135 Class uvm_pkg::uvm_object_string_pool

uvm_pkg :: *uvm_void*
  ↪*uvm_pkg* :: *uvm_object*
    ↪*uvm_pkg* :: *uvm_pool*
      ↪*uvm_pkg* :: *uvm_object_string_pool*

CLASS

uvm_object_string_pool (T)

This provides a specialization of the generic <uvm_pool (KEY, T)> class for an associative array of *uvm_object*-based objects indexed by string. Specializations of this class include the *uvm_event_pool* (a uvm_object_string_pool storing *uvm_event(uvm_object)*) and *uvm_barrier_pool* (a uvm_obejct_string_pool storing *uvm_barrier*).

Table 150: Parameters

Name	Default value	Description
T	uvm_object	

Table 151: Variables

Name	Type	Description
type_name	string	

Table 152: Typedefs

Name	Actual Type	Description
this_type	*uvm_object_string_pool#(T)*	

Constructors

`function new(string name = "")`

Function

new

Creates a new pool with the given *name* .

Parameters

`name` (`string`)

Functions

`virtual function string get_type_name()`

Function

get_type_name

Returns the type name of this object.

`static function this_type get_global_pool()`

Function

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.

Return type
: *this_type*

`static function T get_global(string key)`

Function

get_global

Returns the specified item instance from the global item pool.

Parameters
: **`key`** (*`string`*)

Return type
: *T*

`virtual function T get(string key)`

Function

get

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.

Parameters
: **`key`** (*`string`*)

Return type
: *T*

`virtual function void delete(string key)`

Function

delete

Removes the item with the given string *key* from the pool.

Parameters
: **`key`** (*`string`*)

`virtual function void do_print(uvm_printer printer)`

Function- do_print

Parameters
: **`printer`** (*uvm_printer*)

15.1.1.136 Class uvm_pkg::uvm_object_wrapper

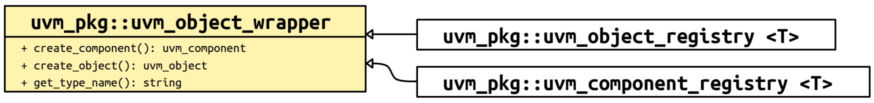

uvm_pkg::uvm_object_wrapper
+ create_component(): uvm_component
+ create_object(): uvm_object
+ get_type_name(): string
uvm_pkg::uvm_object_registry <T>
uvm_pkg::uvm_component_registry <T>

Fig. 34: Inheritance Diagram of uvm_object_wrapper

CLASS

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.

Functions

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

Function

create_object

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.

Parameters
: **name** (*string*)

Return type
: *uvm_object*

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

Function

create_component

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.

Parameters
: **name** (*string*)
: **parent** (*uvm_component*)

Return type
: *uvm_component*

`virtual function string get_type_name()`

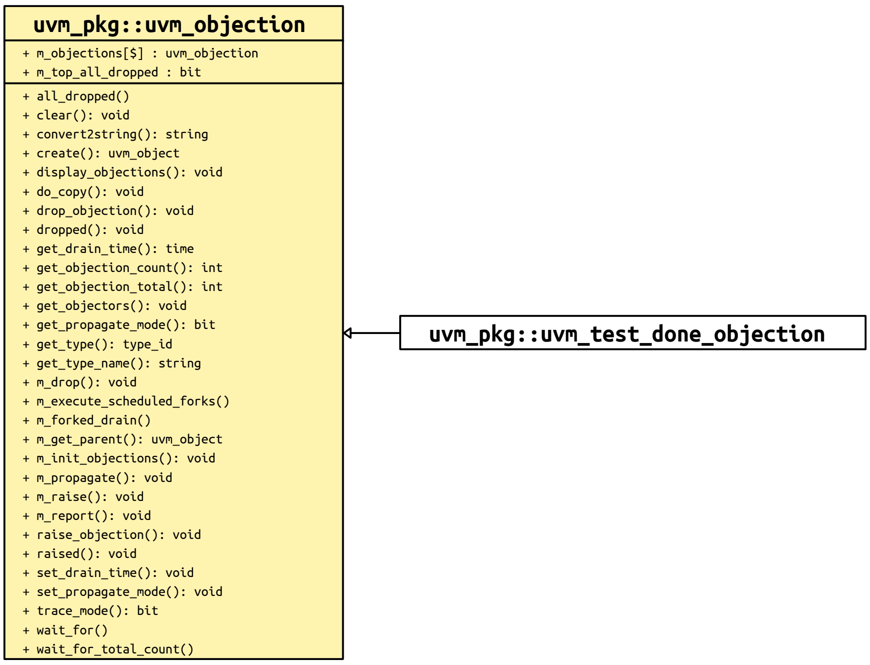
Function

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.

15.1.1.137 Class uvm_pkg::uvm_objection

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_objection*



uvm_pkg::uvm_objection
+ m_objections[$] : uvm_objection
+ m_top_all_dropped : bit
+ all_dropped()
+ clear(): void
+ convert2string(): string
+ create(): uvm_object
+ display_objections(): void
+ do_copy(): void
+ drop_objection(): void
+ dropped(): void
+ get_drain_time(): time
+ get_objection_count(): int
+ get_objection_total(): int
+ get_objectors(): void
+ get_propagate_mode(): bit
+ get_type(): type_id
+ get_type_name(): string
+ m_drop(): void
+ m_execute_scheduled_forks()
+ m_forked_drain()
+ m_get_parent(): uvm_object
+ m_init_objections(): void
+ m_propagate(): void
+ m_raise(): void
+ m_report(): void
+ raise_objection(): void
+ raised(): void
+ set_drain_time(): void
+ set_propagate_mode(): void
+ trace_mode(): bit
+ wait_for()
+ wait_for_total_count()
uvm_pkg::uvm_test_done_objection

Fig. 35: Inheritance Diagram of uvm_objection

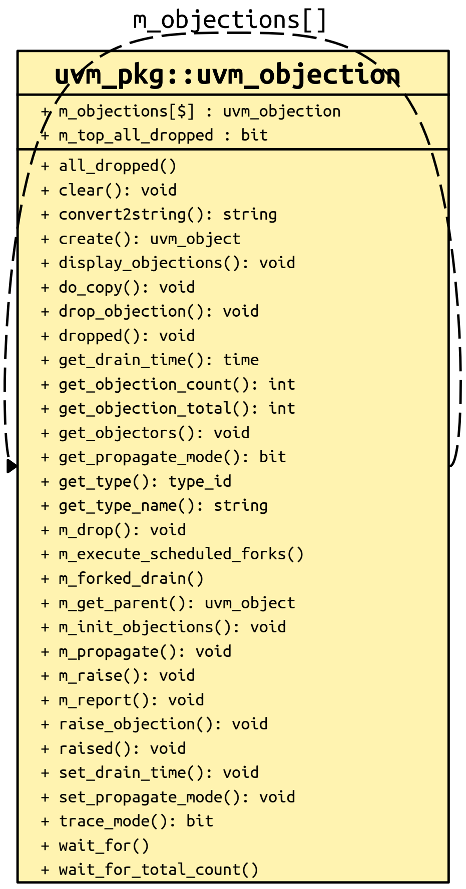


Fig. 36: Collaboration Diagram of uvm_objection

Class

uvm_objection

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

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.

Table 153: Typedefs

Name	Actual Type	Description
type_id	*uvm_object_registry#(uvm_objection, "uvm_objection")*	Below is all of the basic data stuff that is needed for a uvm_object for factory registration, printing, comparing, etc.

Constructors

`function  new(string name = "")`

Function

new

Creates a new objection instance. Accesses the command line argument +UVM_OBJECTION_TRACE to turn tracing on for all objection objects.

Parameters

name (*string*)

Functions

`function bit trace_mode(int mode = -1)`

Function

trace_mode

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.

Parameters

mode (*int*)

`function void set_propagate_mode(bit prop_mode)`

Function

set_propagate_mode

Sets the propagation mode for this objection.

By default, objections support hierarchical propagation for components. For example, if we have the following basic component tree:

```
uvm_top.parent.child
```

Any objections raised by 'child' would get propagated down to parent, and then to uvm_test_top. Resulting in the following counts and totals:

```
                        | count  | total  |
uvm_top.parent.child |      1 |     1   |
uvm_top.parent        |      0 |     1   |
uvm_top               |      0 |     1   |
```

While propagations such as these can be useful, if they are unused by the testbench then they are simply an unnecessary performance hit. If the testbench is not going to use this functionality, then the performance can be improved by setting the propagation mode to 0.

When propagation mode is set to 0, all intermediate callbacks between the *source* and *top* will be skipped. This would result in the following counts and totals for the above objection:

```
                        | count  | total  |
uvm_top.parent.child |      1 |     1   |
uvm_top.parent        |      0 |     0   |
uvm_top               |      0 |     1   |
```

Since the propagation mode changes the behavior of the objection, it can only be safely changed if there are no objections *raised* or *draining* . Any attempts to change the mode while objections are *raised* or *draining* will result in an error.

Parameters

prop_mode (*bit*)

`function bit get_propagate_mode()`

Function

get_propagate_mode

Returns the propagation mode for this objection.

`virtual  function void raise_objection(uvm_object obj = null, string description = "", int count = 1)`

Function

raise_objection

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_root*, 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.

Parameters

obj (*uvm_object*)
description (*string*)
count (*int*)

```
virtual  function void drop_objection(uvm_object obj = null,
string description = "", int count = 1)
```

Function

drop_objection

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_root*, 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.

Parameters

obj (*uvm_object*)
description (*string*)
count (*int*)

virtual function void clear(uvm_object obj = null)

Function

clear

Immediately clears the objection state. All counts are cleared and the any processes waiting on a call to wait_for(UVM_ALL_DROPPED, uvm_top) are released.

The caller, if a uvm_object-based object, should pass its 'this' handle to the *obj* argument to document who cleared the objection. Any drain_times set by the user are not affected.

Parameters
- **obj** (*uvm_object*)

function void set_drain_time(uvm_object obj = null, time drain)

AE

set_drain_time(drain, obj = null)?

Parameters
- **obj** (*uvm_object*)
- **drain** (`time`)

virtual function void raised(uvm_object obj, uvm_object source_obj, string description, int count)

Function

raised

Objection callback that is called when a *raise_objection* has reached *obj* . The default implementation calls *uvm_component::raised*.

Parameters
- **obj** (*uvm_object*)
- **source_obj** (*uvm_object*)
- **description** (`string`)
- **count** (`int`)

virtual function void dropped(uvm_object obj, uvm_object source_obj, string description, int count)

Function

dropped

Objection callback that is called when a *drop_objection* has reached *obj* . The default implementation calls *uvm_component::dropped*.

Parameters
- **obj** (*uvm_object*)
- **source_obj** (*uvm_object*)
- **description** (`string`)
- **count** (`int`)

function void get_objectors(uvm_object list)

Function

get_objectors

Returns the current list of objecting objects (objects that raised an objection but have not dropped it).

Parameters
- **list** (*uvm_object*)

function int get_objection_count(uvm_object obj = null)

Function

get_objection_count

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

Parameters
- **obj** (*uvm_object*)

`function int get_objection_total(uvm_object obj = null)`

Function

get_objection_total

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

Parameters

`obj` (*uvm_object*)

`function time get_drain_time(uvm_object obj = null)`

Function

get_drain_time

Returns the current drain time set for the given object (default

0 ns).

Parameters

`obj` (*uvm_object*)

`virtual function string convert2string()`

`function void display_objections(uvm_object obj = null, bit show_header = 1)`

Function

display_objections

Displays objection information about the given *object* . If *object* is not specified or *null* , the implicit top-level component, *uvm_root*, is chosen. The *show_header* argument allows control of whether a header is output.

Parameters

`obj` (*uvm_object*)

`show_header` (`bit`)

`static function type_id get_type()`

Return type

type_id

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

Parameters

`name` (`string`)

Return type

uvm_object

`virtual function string get_type_name()`

`virtual function void do_copy(uvm_object rhs)`

Parameters

`rhs` (*uvm_object*)

Tasks

`virtual function all_dropped(uvm_object obj, uvm_object source_obj, string description, int count)`

Function

all_dropped

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*.

Parameters

`obj` (*uvm_object*)

`source_obj` (*uvm_object*)

`description` (`string`)

`count` (`int`)

`function wait_for(uvm_objection_event objt_event, uvm_object obj = null)`

Task

wait_for

Waits for the raised, dropped, or all_dropped *event* to occur in the given *obj* . The task returns after all corresponding callbacks for that event have been executed.

Parameters

objt_event (*uvm_objection_event*)

obj (*uvm_object*)

```
function  wait_for_total_count(uvm_object obj = null, int count = 0)
```

Parameters

obj (*uvm_object*)

count (*int*)

15.1.1.138 Class uvm_pkg::uvm_objection_callback

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_callback*
↪*uvm_pkg* :: *uvm_objection_callback*

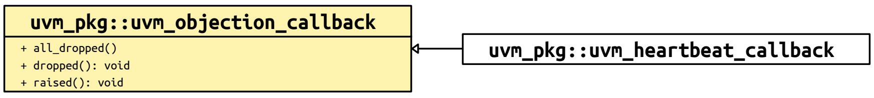

uvm_pkg::uvm_objection_callback
+ all_dropped()
+ dropped(): void
+ raised(): void
uvm_pkg::uvm_heartbeat_callback

Fig. 37: Inheritance Diagram of uvm_objection_callback

Constructors

function new(string name)

Parameters
name (*string*)

Functions

virtual function void raised(uvm_objection objection, uvm_object obj, uvm_-object source_obj, string description, int count)

Function

raised

Objection raised callback function. Called by *uvm_objection::raised*.

Parameters
objection (*uvm_objection*)
obj (*uvm_object*)
source_obj (*uvm_object*)
description (*string*)
count (*int*)

virtual function void dropped(uvm_objection objection, uvm_object obj, uvm_-object source_obj, string description, int count)

Function

dropped

Objection dropped callback function. Called by *uvm_objection::dropped*.

Parameters
objection (*uvm_objection*)
obj (*uvm_object*)
source_obj (*uvm_object*)
description (*string*)
count (*int*)

Tasks

virtual function all_dropped(uvm_objection objection, uvm_object obj, uvm_-object source_obj, string description, int count)

Function

all_dropped

Objection all_dropped callback function. Called by *uvm_objection::all_dropped*.

Parameters
objection (*uvm_objection*)
obj (*uvm_object*)
source_obj (*uvm_object*)

description (*string*)
count (*int*)

15.1.1.139 Class uvm_pkg::uvm_objection_context_object

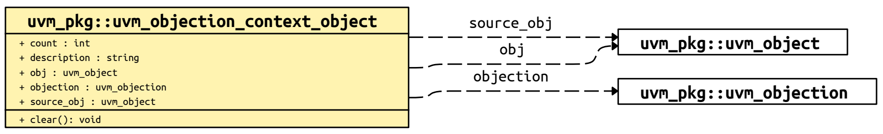


Fig. 38: Collaboration Diagram of uvm_objection_context_object

Have a pool of context objects to use

Table 154: Variables

Name	Type	Description
obj	*uvm_object*	
source_obj	*uvm_object*	
description	string	
count	int	
objection	*uvm_objection*	

Functions

`function void clear()`

Clears the values stored within the object, preventing memory leaks from reused objects

15.1.1.140 Class uvm_pkg::uvm_objection_events

Table 155: Variables

Name	Type	Description
waiters	int	

Events

`raised`

`dropped`

`all_dropped`

15.1.1.141 Class uvm_pkg::uvm_packer

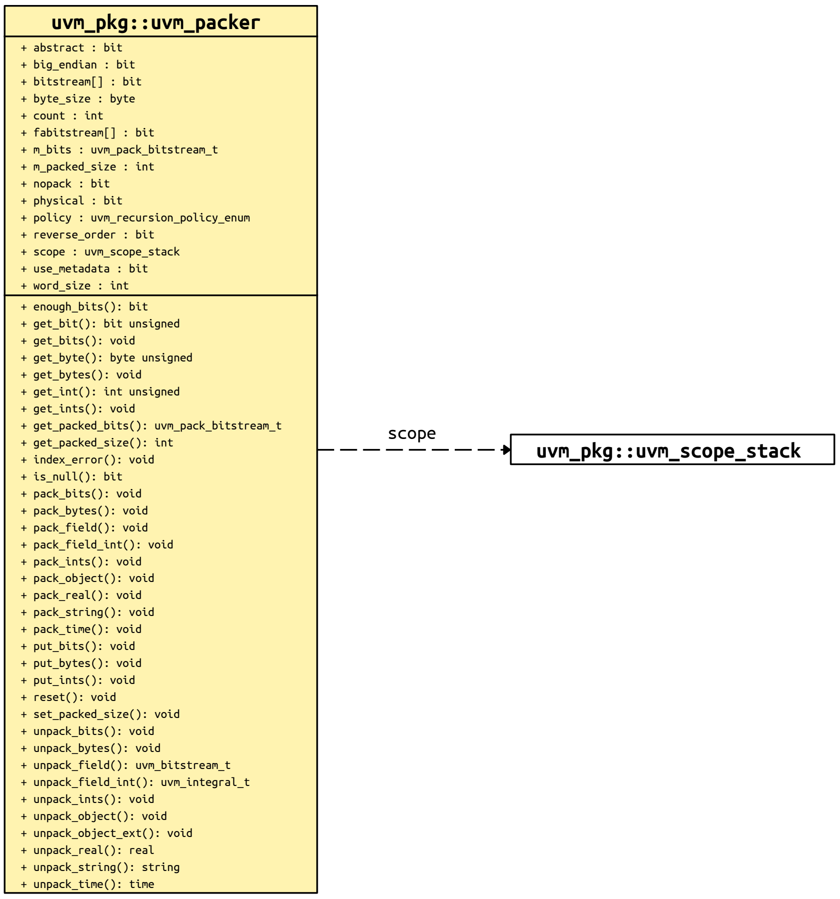


Fig. 39: Collaboration Diagram of uvm_packer

Table 156: Variables

Name	Type	Description
physical	bit	***Variable*** physical 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	***Variable*** abstract 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.

continues on next page

Table 156 – continued from previous page

Name	Type	Description
use_metadata	bit	***Variable*** use_metadata This flag indicates whether to encode metadata when packing dynamic data, or to decode metadata when unpacking. Implementations of *uvm_object::do_pack* and *uvm_object::do_unpack* should regard this bit when performing their respective operation. When set, metadata should be encoded as follows: For strings, pack an additional *null* byte after the string is packed. For objects, pack 4 bits prior to packing the object itself. Use 4'b0000 to indicate the object being packed is *null* , otherwise pack 4'b0001 (the remaining 3 bits are reserved). For queues, dynamic arrays, and associative arrays, pack 32 bits indicating the size of the array prior to packing individual elements.

continues on next page

Table 156 – continued from previous page

Name	Type	Description
big_endian	bit	***Variable*** 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*). 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
```

Name	Type	Description
bitstream	bit	variables and methods primarily for internal use local bits for (un)pack_bytes
fabitstream	bit	field automation bits for (un)pack_bytes
count	int	used to count the number of packed bits
scope	*uvm_scope_stack*	
reverse_order	bit	Flip the bit order around
byte_size	byte	Set up bytesize for endianess
word_size	int	Set up worksize for endianess
nopack	bit	Only count packable bits

continues on next page

Table 156 – continued from previous page

Name	Type	Description
policy	*uvm_recursion_policy_-enum*	

Functions

`virtual function void pack_field(uvm_bitstream_t value, int size)`

Function

pack_field

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

Parameters

- **`value`** (*uvm_bitstream_t*)
- **`size`** (`int`)

`virtual function void pack_field_int(uvm_integral_t value, int size)`

Function

pack_field_int

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_field_int

Parameters

- **`value`** (*uvm_integral_t*)
- **`size`** (`int`)

`virtual function void pack_bits(bit value, int size = -1)`

Function

pack_bits

Packs bits from upacked array of bits into the pack array.

See *pack_ints* for additional information. Pack_bits

Parameters

- **`value`** (`bit`)
- **`size`** (`int`)

`virtual function void pack_bytes(byte value, int size = -1)`

Function

pack_bytes

Packs bits from an upacked array of bytes into the pack array.

See *pack_ints* for additional information. Pack_bytes

Parameters

- **`value`** (`byte`)
- **`size`** (`int`)

`virtual function void pack_ints(int value, int size = -1)`

Function

pack_ints

Packs bits from an unpacked array of ints into the pack array.

The bits are appended to the internal pack array. This method allows for fields of arbitrary length to be passed in, using the SystemVerilog *stream* operator.

For example

```
bit[511:0] my_field;
begin
  int my_stream[];
  { << int {my_stream}} = my_field;
  packer.pack_ints(my_stream);
end
```

When appending the stream to the internal pack array, the packer will obey the value of *big_endian* (appending the array from MSB to LSB if set).

An optional *size* parameter is provided, which defaults to '-1'. If set to any value greater than '-1' (including 0), then the packer will use the size as the number of bits to pack, otherwise the packer will simply pack the entire stream.

An error will be asserted if the *size* has been specified, and exceeds the size of the source array. Pack_ints

Parameters

value (*int*)

size (*int*)

virtual function void pack_string(string value)

Function

pack_string

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_string

Parameters

value (*string*)

virtual function void pack_time(time value)

Function

pack_time

Packs a time *value* as 64 bits into the pack array. Pack_time

Parameters

value (*time*)

virtual function void pack_real(real value)

Function

pack_real

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_real

Parameters

value (*real*)

virtual function void pack_object(uvm_object value)

Function

pack_object

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. Pack_object

Parameters

value (*uvm_object*)

`virtual function bit is_null()`

Function

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. Is_null

`virtual function uvm_bitstream_t unpack_field(int size)`

Function

unpack_field

Unpacks bits from the pack array and returns the bit-stream that was unpacked. *size* is the number of bits to unpack; the maximum is 4096 bits. Unpack_field

Parameters

`size` (*`int`*)

Return type

uvm_bitstream_t

`virtual function uvm_integral_t unpack_field_int(int size)`

Function

unpack_field_int

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_int

Parameters

`size` (*`int`*)

Return type

uvm_integral_t

`virtual function void unpack_bits(bit value, int size = -1)`

Function

unpack_bits

Unpacks bits from the pack array into an unpacked array of bits. Unpack_bits

Parameters

`value` (*`bit`*)

`size` (*`int`*)

`virtual function void unpack_bytes(byte value, int size = -1)`

Function

unpack_bytes

Unpacks bits from the pack array into an unpacked array of bytes. Unpack_bytes

Parameters

`value` (*`byte`*)

`size` (*`int`*)

`virtual function void unpack_ints(int value, int size = -1)`

Function

unpack_ints

Unpacks bits from the pack array into an unpacked array of ints.

The unpacked array is unpacked from the internal pack array. This method allows for fields of arbitrary length to be passed in without expanding into a pre-defined integral type first.

For example

```
bit[511:0] my_field;
begin
  int my_stream[] = new[16]; // 512/32 = 16
  packer.unpack_ints(my_stream);
  my_field = {<<{my_stream}};
end
```

When unpacking the stream from the internal pack array, the packer will obey the value of *big_endian* (unpacking the array from MSB to LSB if set).

An optional *size* parameter is provided, which defaults to '-1'. If set to any value greater than '-1' (including 0), then the packer will use the size as the number of bits to unpack, otherwise the packer will simply unpack the entire stream.

An error will be asserted if the *size* has been specified, and exceeds the size of the target array. Unpack_ints

Parameters

value (*int*)

size (*int*)

virtual function string unpack_string(int num_chars = -1)

Function

unpack_string

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. If num_chars is not -1, then the user only wants to unpack a specific number of bytes into the string.

Parameters

num_chars (*int*)

virtual function time unpack_time()

Function

unpack_time

Unpacks the next 64 bits of the pack array and places them into a time variable. Unpack_time

virtual function real unpack_real()

Function

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_real

virtual function void unpack_object(uvm_object value)

Function

unpack_object

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.

Parameters

value (*uvm_object*)

virtual function int get_packed_size()

Function

get_packed_size

Returns the number of bits that were packed. Get_packed_size

virtual function void unpack_object_ext(uvm_object value)

Unpack_object

Parameters

value (*uvm_object*)

virtual function uvm_pack_bitstream_t get_packed_bits()

Get_packed_bits

Return type

uvm_pack_bitstream_t

virtual function bit unsigned get_bit(int unsigned index)

Get_bit

Parameters

index (*int unsigned*)

virtual function byte unsigned get_byte(int unsigned index)

Get_byte

Parameters

index (*int unsigned*)

virtual function int unsigned get_int(int unsigned index)

Get_int

Parameters

index (*int unsigned*)

virtual function void get_bits(bit unsigned bits)

Get_bits

Parameters

bits (*bit unsigned*)

virtual function void get_bytes(byte unsigned bytes)

Get_bytes

Parameters

bytes (*byte unsigned*)

virtual function void get_ints(int unsigned ints)

Get_ints

Parameters

ints (*int unsigned*)

virtual function void put_bits(bit unsigned bitstream)

Put_bits

Parameters

bitstream (*bit unsigned*)

virtual function void put_bytes(byte unsigned bytestream)

Put_bytes

Parameters

bytestream (*byte unsigned*)

virtual function void put_ints(int unsigned intstream)

Put_ints

Parameters

intstream (*int unsigned*)

virtual function void set_packed_size()

Set_packed_size

function void index_error(int index, string id, int sz)

Index_ok

Parameters

index (*int*)

id (*string*)

sz (*int*)

function bit enough_bits(int needed, string id)

Enough_bits

Parameters

`needed` (*`int`*)

`id` (*`string`*)

`function void reset()`

Reset

15.1.1.142 Class uvm_pkg::uvm_parent_child_link

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_link_base*
↪*uvm_pkg* :: *uvm_parent_child_link*

CLASS

uvm_parent_child_link

The *uvm_parent_child_link* is used to represent a Parent/Child relationship between two objects.

Constructors

function new(string name = "unnamed-uvm_parent_child_link")

Function

new

Constructor

Parameters

name

Instance name

Parameters
name (*string*)

Functions

static function uvm_parent_child_link get_link(uvm_object lhs, uvm_object rhs, string name = "pc_link")

Function

get_link

Constructs a pre-filled link

This allows for simple one-line link creations.

```
my_db.establish_link(uvm_parent_child_link::get_link(record1, record2));
```

Parameters:

lhs

Left hand side reference

rhs

Right hand side reference

name

Optional name for the link object

Parameters
lhs (*uvm_object*)
rhs (*uvm_object*)
name (*string*)

Return type
uvm_parent_child_link

`virtual function void do_set_lhs(uvm_object lhs)`

Function

do_set_lhs

Sets the left-hand-side (Parent)

Parameters

`lhs` (*uvm_object*)

`virtual function uvm_object do_get_lhs()`

Function

do_get_lhs

Retrieves the left-hand-side (Parent)

Return type

uvm_object

`virtual function void do_set_rhs(uvm_object rhs)`

Function

do_set_rhs

Sets the right-hand-side (Child)

Parameters

`rhs` (*uvm_object*)

`virtual function uvm_object do_get_rhs()`

Function

do_get_rhs

Retrieves the right-hand-side (Child)

Return type

uvm_object

15.1.1.143 Class uvm_pkg::uvm_peek_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_peek_export*

Table 157: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.1.1.144 Class uvm_pkg::uvm_peek_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_peek_imp*

Table 158: Parameters

Name	Default value	Description
T	int	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **`name`** (*`string`*)
- **`imp`** (*`int`*)

15.1.1.145 Class uvm_pkg::uvm_peek_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_peek_port*

Table 159: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters

- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.146 Class uvm_pkg::uvm_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*

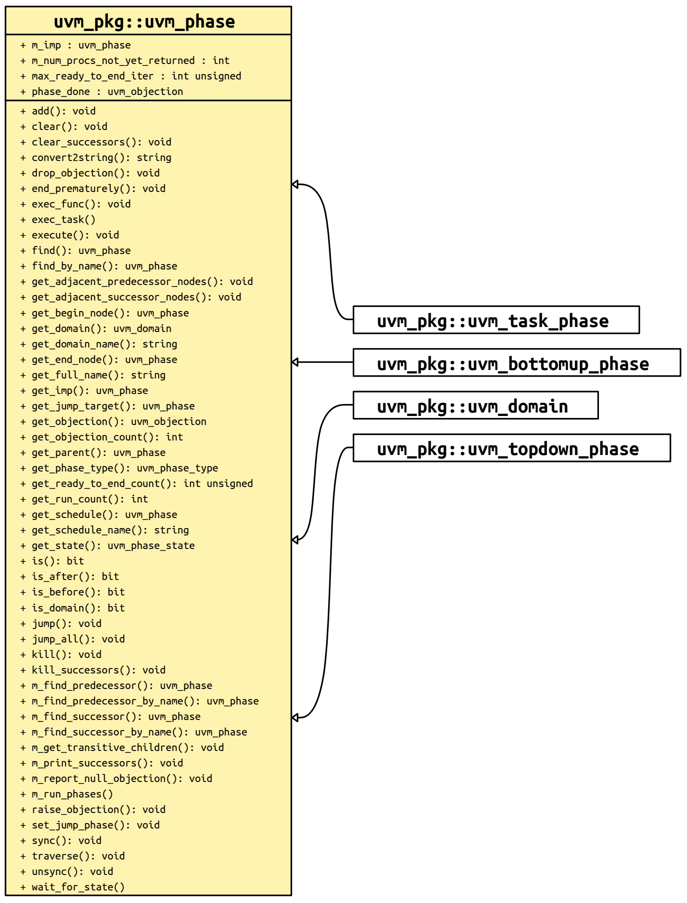

uvm_pkg::uvm_phase
+ m_imp : uvm_phase
+ m_num_procs_not_yet_returned : int
+ max_ready_to_end_iter : int unsigned
+ phase_done : uvm_objection
+ add(): void
+ clear(): void
+ clear_successors(): void
+ convert2string(): string
+ drop_objection(): void
+ end_prematurely(): void
+ exec_func(): void
+ exec_task()
+ execute(): void
+ find(): uvm_phase
+ find_by_name(): uvm_phase
+ get_adjacent_predecessor_nodes(): void
+ get_adjacent_successor_nodes(): void
+ get_begin_node(): uvm_phase
+ get_domain(): uvm_domain
+ get_domain_name(): string
+ get_end_node(): uvm_phase
+ get_full_name(): string
+ get_imp(): uvm_phase
+ get_jump_target(): uvm_phase
+ get_objection(): uvm_objection
+ get_objection_count(): int
+ get_parent(): uvm_phase
+ get_phase_type(): uvm_phase_type
+ get_ready_to_end_count(): int unsigned
+ get_run_count(): int
+ get_schedule(): uvm_phase
+ get_schedule_name(): string
+ get_state(): uvm_phase_state
+ is(): bit
+ is_after(): bit
+ is_before(): bit
+ is_domain(): bit
+ jump(): void
+ jump_all(): void
+ kill(): void
+ kill_successors(): void
+ m_find_predecessor(): uvm_phase
+ m_find_predecessor_by_name(): uvm_phase
+ m_find_successor(): uvm_phase
+ m_find_successor_by_name(): uvm_phase
+ m_get_transitive_children(): void
+ m_print_successors(): void
+ m_report_null_objection(): void
+ m_run_phases()
+ raise_objection(): void
+ set_jump_phase(): void
+ sync(): void
+ traverse(): void
+ unsync(): void
+ wait_for_state()
uvm_pkg::uvm_task_phase
uvm_pkg::uvm_bottomup_phase
uvm_pkg::uvm_domain
uvm_pkg::uvm_topdown_phase

Fig. 40: Inheritance Diagram of uvm_phase

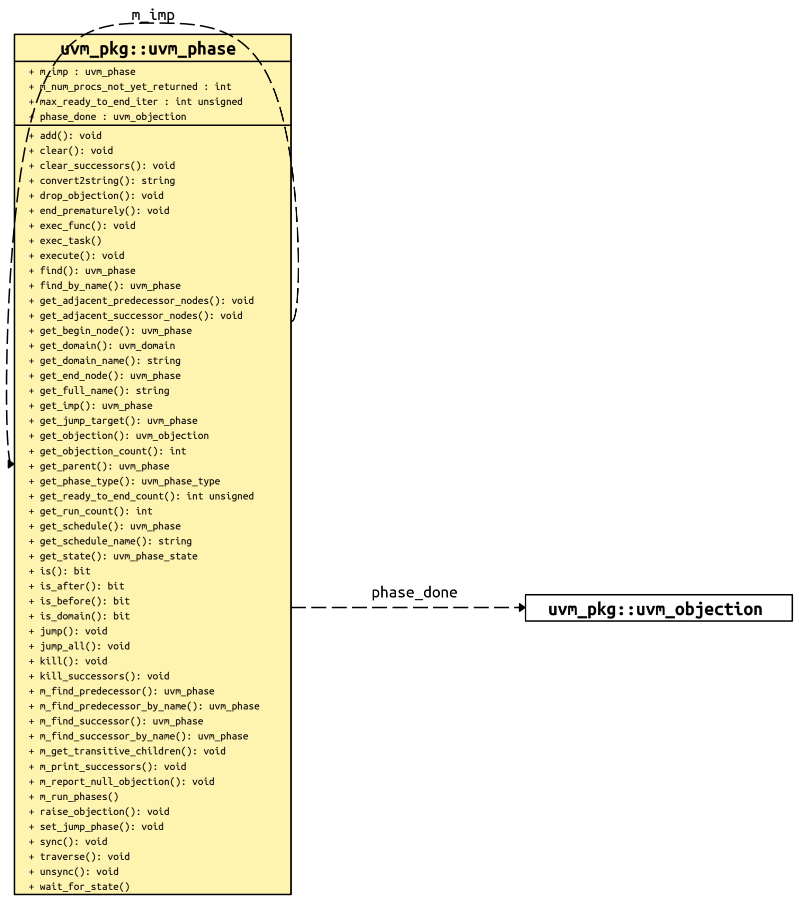

m_imp
uvm_pkg::uvm_phase
+ m_imp : uvm_phase
+ m_num_procs_not_yet_returned : int
+ max_ready_to_end_iter : int unsigned
+ phase_done : uvm_objection
+ add(): void
+ clear(): void
+ clear_successors(): void
+ convert2string(): string
+ drop_objection(): void
+ end_prematurely(): void
+ exec_func(): void
+ exec_task()
+ execute(): void
+ find(): uvm_phase
+ find_by_name(): uvm_phase
+ get_adjacent_predecessor_nodes(): void
+ get_adjacent_successor_nodes(): void
+ get_begin_node(): uvm_phase
+ get_domain(): uvm_domain
+ get_domain_name(): string
+ get_end_node(): uvm_phase
+ get_full_name(): string
+ get_imp(): uvm_phase
+ get_jump_target(): uvm_phase
+ get_objection(): uvm_objection
+ get_objection_count(): int
+ get_parent(): uvm_phase
+ get_phase_type(): uvm_phase_type
+ get_ready_to_end_count(): int unsigned
+ get_run_count(): int
+ get_schedule(): uvm_phase
+ get_schedule_name(): string
+ get_state(): uvm_phase_state
+ is(): bit
+ is_after(): bit
+ is_before(): bit
+ is_domain(): bit
+ jump(): void
+ jump_all(): void
+ kill(): void
+ kill_successors(): void
+ m_find_predecessor(): uvm_phase
+ m_find_predecessor_by_name(): uvm_phase
+ m_find_successor(): uvm_phase
+ m_find_successor_by_name(): uvm_phase
+ m_get_transitive_children(): void
+ m_print_successors(): void
+ m_report_null_objection(): void
+ m_run_phases()
+ raise_objection(): void
+ set_jump_phase(): void
+ sync(): void
+ traverse(): void
+ unsync(): void
+ wait_for_state()
phase_done
uvm_pkg::uvm_objection

Fig. 41: Collaboration Diagram of uvm_phase

Class

uvm_phase

This base class defines everything about a phase

behavior, state, and context.

To define behavior, it is extended by UVM or the user to create singleton objects which capture the definition of what the phase does and how it does it. These are then cloned to produce multiple nodes which are hooked up in a graph structure to provide context: which phases follow which, and to hold the state of the phase throughout its lifetime. UVM provides default extensions of this class for the standard runtime phases. VIP Providers can likewise extend this class to define the phase functor for a particular component context as required.

This base class defines everything about a phase

behavior, state, and context.

To define behavior, it is extended by UVM or the user to create singleton objects which capture the definition of what the phase does and how it does it. These are then cloned to produce multiple nodes which are hooked up in a graph structure to provide context: which phases follow which, and to hold the state of the phase throughout its lifetime. UVM provides default extensions of this class for the standard runtime phases. VIP Providers can likewise extend this class to define the phase functor for a particular component context as required.

Phase Definition

Singleton instances of those extensions are provided as package variables. These instances define the attributes of the phase (not what state it is in) They are then cloned into schedule nodes which point back to one of these implementations, and calls its virtual task or function methods on each participating component. It is the base class for phase functors, for both predefined and user-defined phases. Per-component overrides can use a customized imp.

To create custom phases, do not extend uvm_phase directly

see the

three predefined extended classes below which encapsulate behavior for different phase types: task, bottom-up function and top-down function.

Extend the appropriate one of these to create a uvm_YOURNAME_phase class (or YOURPREFIX_NAME_phase class) for each phase, containing the default implementation of the new phase, which must be a uvm_component-compatible delegate, and which may be a *null* implementation. Instantiate a singleton instance of that class for your code to use when a phase handle is required. If your custom phase depends on methods that are not in uvm_component, but are within an extended class, then extend the base YOURPREFIX_NAME_phase class with parameterized component class context as required, to create a specialized functor which calls your extended component class methods. This scheme ensures compile-safety for your extended component classes while providing homogeneous base types for APIs and underlying data structures.

Phase Context

A schedule is a coherent group of one or mode phase/state nodes linked together by a graph structure, allowing arbitrary linear/parallel relationships to be specified, and executed by stepping through them in the graph order. Each schedule node points to a phase and holds the execution state of that phase, and has optional links to other nodes for synchronization.

The main operations are

construct, add phases, and instantiate

hierarchically within another schedule.

Structure is a DAG (Directed Acyclic Graph). Each instance is a node connected to others to form the graph. Hierarchy is overlaid with m_parent. Each node in the graph has zero or more successors, and zero or more predecessors. No nodes are completely isolated from others. Exactly one node has zero predecessors. This is the root node. Also the graph is acyclic, meaning for all nodes in the graph, by following the forward arrows you will never end up back where you started but you will eventually reach a node that has no successors.

Phase State

A given phase may appear multiple times in the complete phase graph, due to the multiple independent domain feature, and the ability for different VIP to customize their own phase schedules perhaps reusing existing phases. Each node instance in the graph maintains its own state of execution.

Phase Handle

Handles of this type uvm_phase are used frequently in the API, both by the user, to access phasing-specific API, and also as a parameter to some APIs. In many cases, the singleton phase handles can be used (eg. *uvm_run_phase::get()*) in APIs. For those APIs that need to look up that phase in the graph, this is done automatically.

Table 160: Variables

Name	Type	Description
max_ready_to_end_iter	int unsigned	
phase_done	*uvm_objection*	phase done objection

Constructors

`function  new(string name = "uvm_phase", uvm_phase_type phase_type = UVM_PHASE_-SCHEDULE, uvm_phase parent = null)`

Function

new

Create a new phase node, with a name and a note of its type

name

name of this phase

type

a value in *uvm_phase_type*. New

Parameters

- **`name`** (*`string`*)
- **`phase_type`** (*uvm_phase_type*)
- **`parent`** (*uvm_phase*)

Functions

`function uvm_phase_type get_phase_type()`

Function

get_phase_type

Returns the phase type as defined by *uvm_phase_type*. Get_phase_type

Return type

uvm_phase_type

`function uvm_phase_state get_state()`

Function

get_state

Accessor to return current state of this phase. Get_state

Return type

uvm_phase_state

`function int get_run_count()`

Function

get_run_count

Accessor to return the integer number of times this phase has executed. Get_run_count

`function uvm_phase find_by_name(string name, bit stay_in_scope = 1)`

Function

find_by_name

Locate a phase node with the specified *name* and return its handle. With *stay_in_scope* set, searches only within this phase's schedule or domain. Find_by_name

Parameters

- **`name`** (*`string`*)
- **`stay_in_scope`** (*`bit`*)

Return type

uvm_phase

`function uvm_phase find(uvm_phase phase, bit stay_in_scope = 1)`

Function

find

Locate the phase node with the specified *phase* IMP and return its handle. With *stay_in_scope* set, searches only within this phase's schedule or domain. Find

Parameters

phase (*uvm_phase*)
stay_in_scope (*bit*)
Return type
uvm_phase

`function bit is(uvm_phase phase)`

Function

is

returns 1 if the containing uvm_phase refers to the same phase as the phase argument, 0 otherwise. Is
Parameters
phase (*uvm_phase*)

`function bit is_before(uvm_phase phase)`

Function

is_before

Returns 1 if the containing uvm_phase refers to a phase that is earlier than the phase argument, 0 otherwise. Is_before
Parameters
phase (*uvm_phase*)

`function bit is_after(uvm_phase phase)`

Function

is_after

returns 1 if the containing uvm_phase refers to a phase that is later than the phase argument, 0 otherwise. Is_after
Parameters
phase (*uvm_phase*)

`virtual  function void exec_func(uvm_component comp, uvm_phase phase)`

Function

exec_func

Implements the functor/delegate functionality for a function phase type

comp

the component to execute the functionality upon

phase

the phase schedule that originated this phase call
Parameters
comp (*uvm_component*)
phase (*uvm_phase*)

`function void add(uvm_phase phase, uvm_phase with_phase = null, uvm_phase after_-phase = null, uvm_phase before_phase = null)`

Function

add

Build up a schedule structure inserting phase by phase, specifying linkage

Phases can be added anywhere, in series or parallel with existing nodes

phase

handle of singleton derived imp containing actual functor. by default the new phase is appended to the schedule

with_phase

specify to add the new phase in parallel with this one

after_phase

specify to add the new phase as successor to this one

before_phase

specify to add the new phase as predecessor to this one. Add

TBD error checks if param nodes are actually in this schedule or not

Parameters

- **`phase`** (*uvm_phase*)
- **`with_phase`** (*uvm_phase*)
- **`after_phase`** (*uvm_phase*)
- **`before_phase`** (*uvm_phase*)

`function uvm_phase get_parent()`

Function

get_parent

Returns the parent schedule node, if any, for hierarchical graph traversal. Get_parent

Return type

uvm_phase

`virtual function string get_full_name()`

Function

get_full_name

Returns the full path from the enclosing domain down to this node. The singleton IMP phases have no hierarchy. Get_full_name

`function uvm_phase get_schedule(bit hier = 0)`

Function

get_schedule

Returns the topmost parent schedule node, if any, for hierarchical graph traversal. Get_schedule

Parameters

`hier` (`bit`)

Return type

uvm_phase

`function string get_schedule_name(bit hier = 0)`

Function

get_schedule_name

Returns the schedule name associated with this phase node. Get_schedule_name

Parameters

`hier` (`bit`)

`function uvm_domain get_domain()`

Function

get_domain

Returns the enclosing domain. Get_domain

Return type

uvm_domain

`function uvm_phase get_imp()`

Function

get_imp

Returns the phase implementation for this this node. Returns *null* if this phase type is not a UVM_PHASE_LEAF_NODE. Get_imp

Return type

uvm_phase

`function string get_domain_name()`

Function

get_domain_name

Returns the domain name associated with this phase node. Get_domain_name

function void get_adjacent_predecessor_nodes(uvm_phase pred)

Function

get_adjacent_predecessor_nodes

Provides an array of nodes which are predecessors to *this* phase node. A 'predecessor node' is defined as any phase node which lies prior to *this* node in the phase graph, with no nodes between *this* node and the predecessor node.

Parameters

pred (*uvm_phase*)

function void get_adjacent_successor_nodes(uvm_phase succ)

Function

get_adjacent_successor_nodes

Provides an array of nodes which are successors to *this* phase node. A 'successor's node' is defined as any phase node which comes after *this* node in the phase graph, with no nodes between *this* node and the successor node.

Parameters

succ (*uvm_phase*)

function uvm_objection get_objection()

Function

get_objection

Return the *uvm_objection* that gates the termination of the phase.

Return type

uvm_objection

virtual function void raise_objection(uvm_object obj, string description = "", int count = 1)

Function

raise_objection

Raise an objection to ending this phase Provides components with greater control over the phase flow for processes which are not implicit objectors to the phase.

```
while(1) begin
  some_phase.raise_objection(this);
  ...
  some_phase.drop_objection(this);
end
... Raise_objection
```

Parameters

obj (*uvm_object*)

description (*string*)

count (*int*)

virtual function void drop_objection(uvm_object obj, string description = "", int count = 1)

Function

drop_objection

Drop an objection to ending this phase

The drop is expected to be matched with an earlier raise. Drop_objection

Parameters

obj (*uvm_object*)

description (*string*)

count (*int*)

virtual function int get_objection_count(uvm_object obj = null)

Function

get_objection_count

Returns the current number of objections to ending this phase raised by the given *object* . Get_objection_count

Parameters
obj (*uvm_object*)

function void sync(uvm_domain target, uvm_phase phase = null, uvm_phase with_-phase = null)

Function

sync

Synchronize two domains, fully or partially

target

handle of target domain to synchronize this one to

phase

optional single phase in this domain to synchronize, otherwise sync all

with_phase

optional different target-domain phase to synchronize with, otherwise use *phase* in the target domain. Sync

Parameters
target (*uvm_domain*)
phase (*uvm_phase*)
with_phase (*uvm_phase*)

function void unsync(uvm_domain target, uvm_phase phase = null, uvm_phase with_-phase = null)

Function

unsync

Remove synchronization between two domains, fully or partially

target

handle of target domain to remove synchronization from

phase

optional single phase in this domain to un-synchronize, otherwise unsync all

with_phase

optional different target-domain phase to un-synchronize with, otherwise use *phase* in the target domain. Un-sync

Parameters
target (*uvm_domain*)
phase (*uvm_phase*)
with_phase (*uvm_phase*)

function void jump(uvm_phase phase)

Function

jump

Jump to a specified *phase* . If the destination *phase* is within the current phase schedule, a simple local jump takes place. If the jump-to *phase* is outside of the current schedule then the jump affects other schedules which share the phase. Jump

Note that this function does not directly alter flow of control. That is, the new phase is not initiated in this function. Rather, flags are set which execute_phase() uses to determine that a jump has been requested and performs the jump.

Parameters
phase (*uvm_phase*)

function void set_jump_phase(uvm_phase phase)

Function

set_jump_phase

Specify a phase to transition to when phase is complete. Note that this function is part of what jump() does; unlike jump() it does not set the flag to terminate the phase prematurely. Set_jump_phase

Specify a phase to transition to when phase is complete.

Parameters

phase (*uvm_phase*)

`function void end_prematurely()`

Function

end_prematurely

Set a flag to cause the phase to end prematurely. Note that this function is part of what jump() does; unlike jump() it does not set a jump_phase to go to after the phase ends. End_prematurely

Set a flag to cause the phase to end prematurely.

`static function void jump_all(uvm_phase phase)`

Function- jump_all

Make all schedules jump to a specified *phase* , even if the jump target is local. The jump happens to all phase schedules that contain the jump-to *phase* , i.e. a global jump. Jump_all

Parameters

phase (*uvm_phase*)

`function uvm_phase get_jump_target()`

Function

get_jump_target

Return handle to the target phase of the current jump, or *null* if no jump is in progress. Valid for use during the phase_ended() callback. Get_jump_target

Return type

uvm_phase

`virtual function void traverse(uvm_component comp, uvm_phase phase, uvm_phase_state state)`

Implementation

Callbacks

Provide the required component traversal behavior. Called by execute()

Parameters

comp (*uvm_component*)

phase (*uvm_phase*)

state (*uvm_phase_state*)

`virtual function void execute(uvm_component comp, uvm_phase phase)`

Provide the required per-component execution flow. Called by traverse()

Parameters

comp (*uvm_component*)

phase (*uvm_phase*)

`function uvm_phase get_begin_node()`

Return type

uvm_phase

`function uvm_phase get_end_node()`

Return type

uvm_phase

`function int unsigned get_ready_to_end_count()`

`function void clear(uvm_phase_state state = UVM_PHASE_DORMANT)`

Clear

for internal graph maintenance after a forward jump

Parameters

state (*uvm_phase_state*)

function void clear_successors(uvm_phase_state state = UVM_PHASE_DORMANT, uvm_-phase end_state = null)

Clear_successors

for internal graph maintenance after a forward jump called only by execute_phase() depth-first traversal of the DAG, calliing clear() on each node do not clear the end phase or beyond

Parameters

- **state** (*uvm_phase_state*)
- **end_state** (*uvm_phase*)

function void kill()

Kill

function void kill_successors()

Using a depth-first traversal, kill all the successor phases of the current phase.

virtual function string convert2string()

function bit is_domain()

Tasks

virtual function exec_task(uvm_component comp, uvm_phase phase)

Function

exec_task

Implements the functor/delegate functionality for a task phase type

comp

the component to execute the functionality upon

phase

the phase schedule that originated this phase call

Parameters

- **comp** (*uvm_component*)
- **phase** (*uvm_phase*)

function wait_for_state(uvm_phase_state state, uvm_wait_op op = UVM_EQ)

Function

wait_for_state

Wait until this phase compares with the given *state* and *op* operand. For <UVM_EQ> and <UVM_NE> operands, several *uvm_phase_states* can be supplied by ORing their enum constants, in which case the caller will wait until the phase state is any of (UVM_EQ) or none of (UVM_NE) the provided states.

To wait for the phase to be at the started state or after

```
wait_for_state(UVM_PHASE_STARTED, UVM_GTE);
```

To wait for the phase to be either started or executing

```
wait_for_state(UVM_PHASE_STARTED | UVM_PHASE_EXECUTING, UVM_EQ);. Wait_for_
↪state
```

Parameters

- **state** (*uvm_phase_state*)
- **op** (*uvm_wait_op*)

15.1.1.147 Class uvm_pkg::uvm_phase_cb

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_callback*
↪*uvm_pkg* :: *uvm_phase_cb*

Class

uvm_phase_cb

This class defines a callback method that is invoked by the phaser during the execution of a specific node in the phase graph or all phase nodes. User-defined callback extensions can be used to integrate data types that are not natively phase-aware with the UVM phasing.

Constructors

`function new(string name = "unnamed-uvm_phase_cb")`

Function

new

Constructor
Parameters
`name` (`string`)

Functions

`virtual function void phase_state_change(uvm_phase phase, uvm_phase_state_-change change)`

Function

phase_state_change

Called whenever a *phase* changes state. The *change* descriptor describes the transition that was just completed. The callback method is invoked immediately after the phase state has changed, but before the phase implementation is executed.

An extension may interact with the phase, such as raising the phase objection to prolong the phase, in a manner that is consistent with the current phase state.

By default, the callback method does nothing. Unless otherwise specified, modifying the phase transition descriptor has no effect on the phasing schedule or execution.
Parameters
`phase` (*uvm_phase*)
`change` (*uvm_phase_state_change*)

15.1.1.148 Class uvm_pkg::uvm_phase_state_change

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase_state_change*

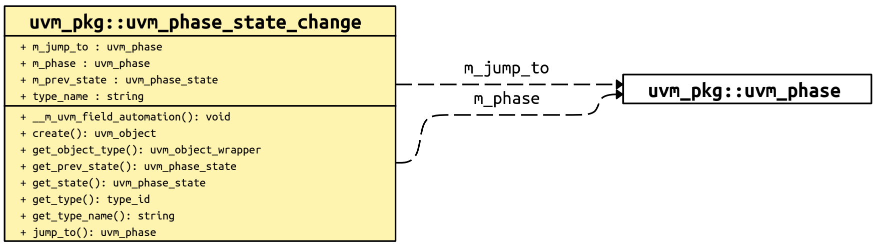

uvm_pkg::uvm_phase_state_change
+ m_jump_to : uvm_phase
+ m_phase : uvm_phase
+ m_prev_state : uvm_phase_state
+ type_name : string
+ __m_uvm_field_automation(): void
+ create(): uvm_object
+ get_object_type(): uvm_object_wrapper
+ get_prev_state(): uvm_phase_state
+ get_state(): uvm_phase_state
+ get_type(): type_id
+ get_type_name(): string
+ jump_to(): uvm_phase
m_jump_to
m_phase
uvm_pkg::uvm_phase

Fig. 42: Collaboration Diagram of uvm_phase_state_change

Class

uvm_phase_state_change

Phase state transition descriptor. Used to describe the phase transition that caused a <uvm_phase_cb::phase_state_changed()> callback to be invoked.

Constructors

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

Parameters
name (*`string`*)

Functions

`virtual function uvm_phase_state get_state()`

Function

get_state()

Returns the state the phase just transitioned to. Functionally equivalent to *uvm_phase::get_state()*.

Return type
uvm_phase_state

`virtual function uvm_phase_state get_prev_state()`

Function

get_prev_state()

Returns the state the phase just transitioned from.

Return type
uvm_phase_state

`function uvm_phase jump_to()`

Function

jump_to()

If the current state is *UVM_PHASE_ENDED* or *UVM_PHASE_JUMPING* because of a phase jump, returns the phase that is the target of jump. Returns *null* otherwise.

Return type
uvm_phase

15.1.1.149 Class uvm_pkg::uvm_pool

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_pool*

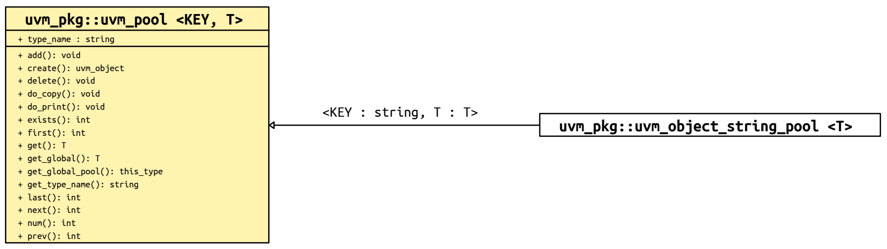


Fig. 43: Inheritance Diagram of uvm_pool

CLASS

uvm_pool (KEY, T)

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

Table 161: Parameters

Name	Default value	Description
KEY	int	
T	uvm_void	

Table 162: Variables

Name	Type	Description
type_name	string	

Table 163: Typedefs

Name	Actual Type	Description
this_type	*uvm_pool#(KEY, T)*	

Constructors

`function new(string name = "")`

Function

new

Creates a new pool with the given *name* .

Parameters

`name` (*`string`*)

Functions

`static function this_type get_global_pool()`

Function

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.

Return type
: *this_type*

`static function T get_global(int key)`

Function

get_global

Returns the specified item instance from the global item pool.

Parameters
: **key** (*int*)

Return type
: *T*

`virtual function T get(int key)`

Function

get

Returns the item with the given *key* .

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

Parameters
: **key** (*int*)

Return type
: *T*

`virtual function void add(int key, uvm_void item)`

Function

add

Adds the given (*key* , *item*) pair to the pool. If an item already exists at the given *key* it is overwritten with the new *item* .

Parameters
: **key** (*int*)
: **item** (*uvm_void*)

`virtual function int num()`

Function

num

Returns the number of uniquely keyed items stored in the pool.

`virtual function void delete(int key)`

Function

delete

Removes the item with the given *key* from the pool.

Parameters
: **key** (*int*)

`virtual function int exists(int key)`

Function

exists

Returns 1 if an item with the given *key* exists in the pool, 0 otherwise.

Parameters
: **key** (*int*)

`virtual function int first(int key)`

Function

first

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.

Parameters

`key` (*`int`*)

`virtual function int last(int key)`

Function

last

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.

Parameters

`key` (*`int`*)

`virtual function int next(int key)`

Function

next

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.

Parameters

`key` (*`int`*)

`virtual function int prev(int key)`

Function

prev

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.

Parameters

`key` (*`int`*)

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

Parameters

`name` (*`string`*)

Return type

uvm_object

`virtual function string get_type_name()`

`virtual function void do_copy(uvm_object rhs)`

Parameters

`rhs` (*uvm_object*)

`virtual function void do_print(uvm_printer printer)`

Parameters

`printer` (*uvm_printer*)

15.1.1.150 Class uvm_pkg::uvm_port_component

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_port_component_base*
↪*uvm_pkg* :: *uvm_port_component*

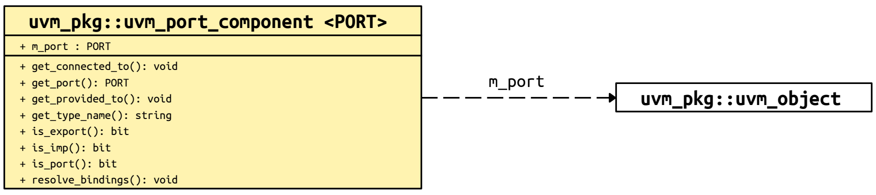

uvm_pkg::uvm_port_component <PORT>
+ m_port : PORT
+ get_connected_to(): void
+ get_port(): PORT
+ get_provided_to(): void
+ get_type_name(): string
+ is_export(): bit
+ is_imp(): bit
+ is_port(): bit
+ resolve_bindings(): void
m_port
uvm_pkg::uvm_object

Fig. 44: Collaboration Diagram of uvm_port_component

Table 164: Parameters

Name	Default value	Description
PORT	uvm_object	

Constructors

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

Parameters
- **`name`** (`string`)
- **`parent`** (*uvm_component*)
- **`port`** (*uvm_object*)

Functions

`virtual function string get_type_name()`

`virtual function void resolve_bindings()`

`function PORT get_port()`

Function

get_port

Retrieve the actual port object that this proxy refers to.

Return type
PORT

`virtual function void get_connected_to(uvm_port_list list)`

Parameters
- **`list`** (*uvm_port_list*)

`virtual function void get_provided_to(uvm_port_list list)`

Parameters
- **`list`** (*uvm_port_list*)

`virtual function bit is_port()`

`virtual function bit is_export()`

`virtual function bit is_imp()`

15.1.1.151 Class uvm_pkg::uvm_port_component_base

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_port_component_base*

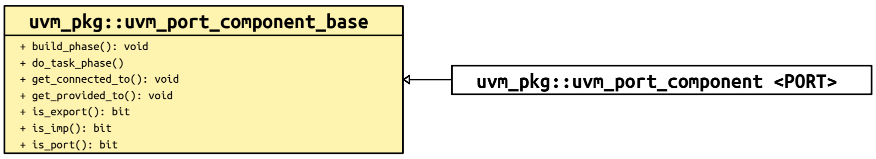

uvm_pkg::uvm_port_component_base
+ build_phase(): void
+ do_task_phase()
+ get_connected_to(): void
+ get_provided_to(): void
+ is_export(): bit
+ is_imp(): bit
+ is_port(): bit
uvm_pkg::uvm_port_component <PORT>

Fig. 45: Inheritance Diagram of uvm_port_component_base

CLASS

uvm_port_component_base

This class defines an interface for obtaining a port's connectivity lists after or during the end_of_elaboration phase. The sub-class, <uvm_port_component (PORT)>, implements this interface.

The connectivity lists are returned in the form of handles to objects of this type. This allowing traversal of any port's fan-out and fan-in network through recursive calls to *get_connected_to* and *get_provided_to*. Each port's full name and type name can be retrieved using *get_full_name* and *get_type_name* methods inherited from *uvm_component*.

Constructors

`function new(string name, uvm_component parent)`

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)

Functions

`virtual function void get_connected_to(uvm_port_list list)`

Function

get_connected_to

For a port or export type, this function fills *list* with all of the ports, exports and implementations that this port is connected to.

Parameters
- **`list`** (*uvm_port_list*)

`virtual function void get_provided_to(uvm_port_list list)`

Function

get_provided_to

For an implementation or export type, this function fills *list* with all of the ports, exports and implementations that this port is provides its implementation to.

Parameters
- **`list`** (*uvm_port_list*)

`virtual function bit is_port()`

Function

is_port

`virtual function bit is_export()`

Function

is_export

`virtual function bit is_imp()`

Function

is_imp

These function determine the type of port. The functions are mutually exclusive; one will return 1 and the other two will return 0.

`virtual function void build_phase(uvm_phase phase)`

Turn off auto config by not calling build_phase()

Parameters

`phase` (*uvm_phase*)

Tasks

`virtual function do_task_phase(uvm_phase phase)`

Parameters

`phase` (*uvm_phase*)

15.1.1.152 Class uvm_pkg::uvm_post_configure_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_task_phase*
↪*uvm_pkg* :: *uvm_post_configure_phase*

Class

uvm_post_configure_phase

After the SW has configured the DUT.

uvm_task_phase that calls the *uvm_component::post_configure_phase* method.

Upon Entry

- Indicates that the configuration information has been fully uploaded.

Typical Uses

Wait for configuration information to fully propagate and take effect.
Wait for components to complete training and rate negotiation.
Enable the DUT.
Sample DUT configuration coverage.

Exit Criteria

- The DUT has been fully configured and enabled and is ready to start operating normally.

Table 165: Variables

Name	Type	Description
type_name	string	

Functions

`static function uvm_post_configure_phase get()`

Function

get

Returns the singleton phase handle
Return type
uvm_post_configure_phase

`virtual function string get_type_name()`

Tasks

`virtual function exec_task(uvm_component comp, uvm_phase phase)`

Parameters
`comp` (*uvm_component*)
`phase` (*uvm_phase*)

15.1.1.153 Class uvm_pkg::uvm_post_main_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_task_phase*
↪*uvm_pkg* :: *uvm_post_main_phase*

Class

uvm_post_main_phase

After enough of the primary test stimulus.

uvm_task_phase that calls the *uvm_component::post_main_phase* method.

Upon Entry

- The primary stimulus objective of the test has been met.

Typical Uses

- Included for symmetry.

Exit Criteria

- None.

Table 166: Variables

Name	Type	Description
type_name	string	

Functions

`static function uvm_post_main_phase get()`

Function

get

Returns the singleton phase handle

Return type
uvm_post_main_phase

`virtual function string get_type_name()`

Tasks

`virtual function exec_task(uvm_component comp, uvm_phase phase)`

Parameters
`comp` (*uvm_component*)
`phase` (*uvm_phase*)

15.1.1.154 Class uvm_pkg::uvm_post_reset_phase

uvm_pkg :: *uvm_void*
  ↪*uvm_pkg* :: *uvm_object*
    ↪*uvm_pkg* :: *uvm_phase*
      ↪*uvm_pkg* :: *uvm_task_phase*
        ↪*uvm_pkg* :: *uvm_post_reset_phase*

Class

uvm_post_reset_phase

After reset is de-asserted.

uvm_task_phase that calls the *uvm_component::post_reset_phase* method.

Upon Entry

- Indicates that the DUT reset signal has been de-asserted.

Typical Uses

- Components should start behavior appropriate for reset being inactive. For example, components may start to transmit idle transactions

or interface training and rate negotiation. This behavior typically continues beyond the end of this phase.

Exit Criteria

- The testbench and the DUT are in a known, active state.

Table 167: Variables

Name	Type	Description
type_name	string	

Functions

`static function uvm_post_reset_phase get()`

Function

get

Returns the singleton phase handle

Return type

uvm_post_reset_phase

`virtual function string get_type_name()`

Tasks

`virtual function exec_task(uvm_component comp, uvm_phase phase)`

Parameters

`comp` (*uvm_component*)

`phase` (*uvm_phase*)

15.1.1.155 Class uvm_pkg::uvm_post_shutdown_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_task_phase*
↪*uvm_pkg* :: *uvm_post_shutdown_phase*

Class

uvm_post_shutdown_phase

After things have settled down.

uvm_task_phase that calls the *uvm_component::post_shutdown_phase* method. The end of this phase is synchronized to the end of the *uvm_run_phase* phase unless a user defined phase is added after this phase.

Upon Entry

- No more "data" stimulus is applied to the DUT.

Typical Uses

- Perform final checks that require run-time access to the DUT (e.g. read accounting registers or dump the content of memories).

Exit Criteria

All run-time checks have been satisfied.
The *uvm_run_phase* phase is ready to end.

Table 168: Variables

Name	Type	Description
type_name	string	

Functions

static function uvm_post_shutdown_phase get()

Function

get

Returns the singleton phase handle
Return type
uvm_post_shutdown_phase

virtual function string get_type_name()

Tasks

virtual function exec_task(uvm_component comp, uvm_phase phase)

Parameters
comp (*uvm_component*)
phase (*uvm_phase*)

15.1.1.156 Class uvm_pkg::uvm_pre_configure_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_task_phase*
↪*uvm_pkg* :: *uvm_pre_configure_phase*

Class

uvm_pre_configure_phase

Before the DUT is configured by the SW.

uvm_task_phase that calls the *uvm_component::pre_configure_phase* method.

Upon Entry

- Indicates that the DUT has been completed reset and is ready to be configured.

Typical Uses

Procedurally modify the DUT configuration information as described in the environment (and that will be eventually uploaded into the DUT).
Wait for components required for DUT configuration to complete training and rate negotiation.

Exit Criteria

- DUT configuration information is defined.

Table 169: Variables

Name	Type	Description
type_name	string	

Functions

`static function uvm_pre_configure_phase get()`

Function

get

Returns the singleton phase handle
Return type
uvm_pre_configure_phase

`virtual function string get_type_name()`

Tasks

`virtual function exec_task(uvm_component comp, uvm_phase phase)`

Parameters
`comp` (*uvm_component*)
`phase` (*uvm_phase*)

15.1.1.157 Class uvm_pkg::uvm_pre_main_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
  ↪*uvm_pkg* :: *uvm_phase*
    ↪*uvm_pkg* :: *uvm_task_phase*
      ↪*uvm_pkg* :: *uvm_pre_main_phase*

Class

uvm_pre_main_phase

Before the primary test stimulus starts.

uvm_task_phase that calls the *uvm_component::pre_main_phase* method.

Upon Entry

- Indicates that the DUT has been fully configured.

Typical Uses

- Wait for components to complete training and rate negotiation.

Exit Criteria

All components have completed training and rate negotiation.
All components are ready to generate and/or observe normal stimulus.

Table 170: Variables

Name	Type	Description
type_name	string	

Functions

`static function uvm_pre_main_phase get()`

Function

get

Returns the singleton phase handle
Return type
uvm_pre_main_phase

`virtual function string get_type_name()`

Tasks

`virtual function exec_task(uvm_component comp, uvm_phase phase)`

Parameters
`comp` (*uvm_component*)
`phase` (*uvm_phase*)

15.1.1.158 Class uvm_pkg::uvm_pre_reset_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_task_phase*
↪*uvm_pkg* :: *uvm_pre_reset_phase*

Class

uvm_pre_reset_phase

Before reset is asserted.

uvm_task_phase that calls the *uvm_component::pre_reset_phase* method. This phase starts at the same time as the *uvm_run_phase* unless a user defined phase is inserted in front of this phase.

Upon Entry

Indicates that power has been applied but not necessarily valid or stable.
There should not have been any active clock edges before entry into this phase.

Typical Uses

Wait for power good.
Components connected to virtual interfaces should initialize their output to X's or Z's.
Initialize the clock signals to a valid value
Assign reset signals to X (power-on reset).
Wait for reset signal to be asserted if not driven by the verification environment.

Exit Criteria

Reset signal, if driven by the verification environment, is ready to be asserted.
Reset signal, if not driven by the verification environment, is asserted.

Table 171: Variables

Name	Type	Description
type_name	string	

Functions

`static function uvm_pre_reset_phase get()`

Function

get

Returns the singleton phase handle

Return type

uvm_pre_reset_phase

`virtual function string get_type_name()`

Tasks

`virtual function exec_task(uvm_component comp, uvm_phase phase)`

Parameters

`comp` (*uvm_component*)
`phase` (*uvm_phase*)

15.1.1.159 Class uvm_pkg::uvm_pre_shutdown_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_task_phase*
↪*uvm_pkg* :: *uvm_pre_shutdown_phase*

Class

uvm_pre_shutdown_phase

Before things settle down.

uvm_task_phase that calls the *uvm_component::pre_shutdown_phase* method.

Upon Entry

- None.

Typical Uses

- Included for symmetry.

Exit Criteria

- None.

Table 172: Variables

Name	Type	Description
type_name	string	

Functions

`static function uvm_pre_shutdown_phase get()`

Function

get

Returns the singleton phase handle

Return type
uvm_pre_shutdown_phase

`virtual function string get_type_name()`

Tasks

`virtual function exec_task(uvm_component comp, uvm_phase phase)`

Parameters
comp (*uvm_component*)
phase (*uvm_phase*)

15.1.1.160 Class uvm_pkg::uvm_predict_s

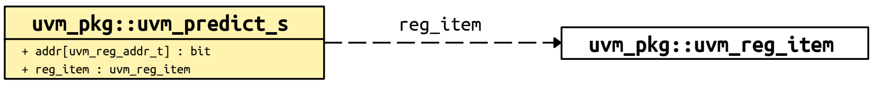


Fig. 46: Collaboration Diagram of uvm_predict_s

TITLE

Explicit Register Predictor

The *uvm_reg_predictor* class defines a predictor component, which is used to update the register model's mirror values based on transactions explicitly observed on a physical bus.

Table 173: Variables

Name	Type	Description
addr	bit	
reg_item	*uvm_reg_item*	

15.1.1.161 Class uvm_pkg::uvm_printer

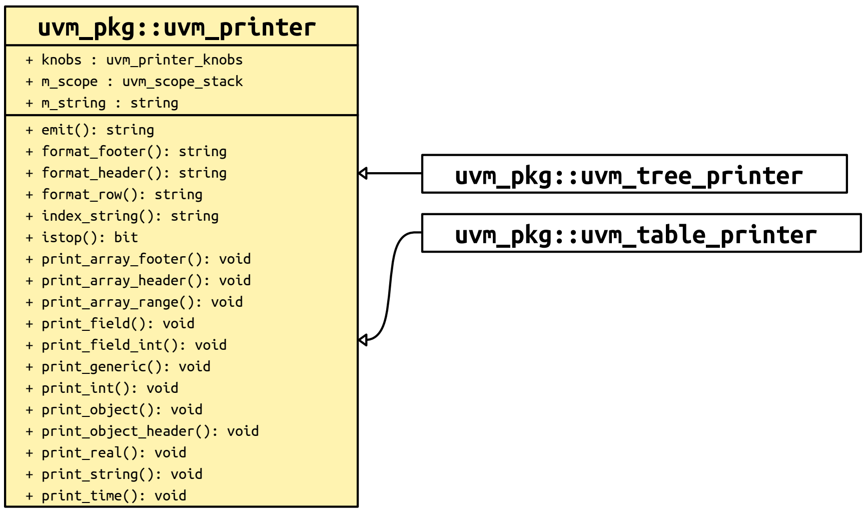

uvm_pkg::uvm_printer
+ knobs : uvm_printer_knobs
+ m_scope : uvm_scope_stack
+ m_string : string
+ emit(): string
+ format_footer(): string
+ format_header(): string
+ format_row(): string
+ index_string(): string
+ istop(): bit
+ print_array_footer(): void
+ print_array_header(): void
+ print_array_range(): void
+ print_field(): void
+ print_field_int(): void
+ print_generic(): void
+ print_int(): void
+ print_object(): void
+ print_object_header(): void
+ print_real(): void
+ print_string(): void
+ print_time(): void
uvm_pkg::uvm_tree_printer
uvm_pkg::uvm_table_printer

Fig. 47: Inheritance Diagram of uvm_printer

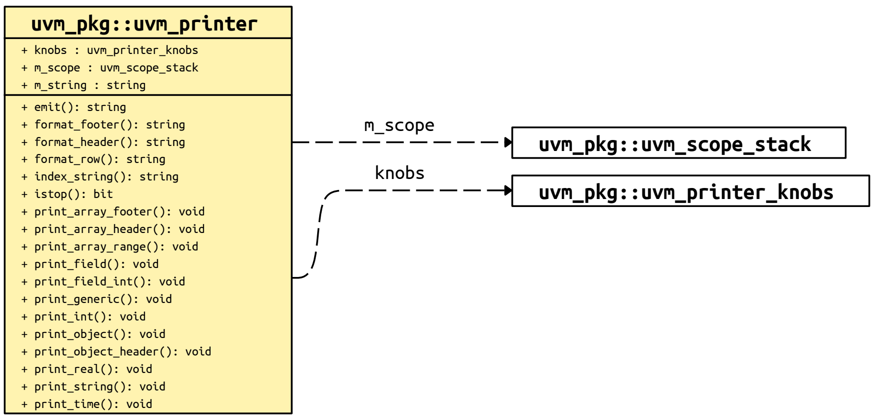


Fig. 48: Collaboration Diagram of uvm_printer

Class

uvm_printer

The uvm_printer class provides an interface for printing *uvm_objects* in various formats. Subtypes of uvm_printer implement different print formats, or policies.

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

uvm_printer

provides base printer functionality; must be overridden.

uvm_table_printer

prints the object in a tabular form.

uvm_tree_printer

prints the object in a tree form.

uvm_line_printer

prints the information on a single line, but uses the same object separators as the tree printer.

Printers have knobs that you use to control what and how information is printed. These knobs are contained in a separate knob class:

uvm_printer_knobs

common printer settings

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

uvm_default_tree_printer
uvm_default_line_printer
uvm_default_table_printer
uvm_default_printer (set to default_table_printer by default)

When *uvm_object::print* and *uvm_object::sprint* are called without specifying a printer, the *uvm_default_printer* is used.

Table 174: Variables

Name	Type	Description
knobs	*uvm_printer_knobs*	***Variable*** knobs The knob object provides access to the variety of knobs associated with a specific printer instance.

Functions

`virtual function void print_field(string name, uvm_bitstream_t value, int size, uvm_radix_enum radix = UVM_NORADIX, byte scope_separator = ".", string type_name = "")`

Function

print_field

Prints an integral field (up to 4096 bits).

name

The name of the field.

value

The value of the field.

size

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

radix

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

scope_separator

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

Parameters

- **`name`** (`string`)
- **`value`** (*uvm_bitstream_t*)
- **`size`** (`int`)
- **`radix`** (*uvm_radix_enum*)
- **`scope_separator`** (`byte`)
- **`type_name`** (`string`)

`virtual function void print_int(string name, uvm_bitstream_t value, int size, uvm_-radix_enum radix = UVM_NORADIX, byte scope_separator = ".", string type_name = "")`

backward compatibility

Parameters

- **`name`** (*`string`*)
- **`value`** (*uvm_bitstream_t*)
- **`size`** (*`int`*)
- **`radix`** (*uvm_radix_enum*)
- **`scope_separator`** (*`byte`*)
- **`type_name`** (*`string`*)

`virtual function void print_field_int(string name, uvm_integral_t value, int size, uvm_radix_enum radix = UVM_NORADIX, byte scope_separator = ".", string type_name = "")`

Function

print_field_int

Prints an integral field (up to 64 bits).

name

The name of the field.

value

The value of the field.

size

The number of bits of the field (maximum is 64).

radix

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

scope_separator

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

Parameters

- **`name`** (*`string`*)
- **`value`** (*uvm_integral_t*)
- **`size`** (*`int`*)
- **`radix`** (*uvm_radix_enum*)
- **`scope_separator`** (*`byte`*)
- **`type_name`** (*`string`*)

`virtual function void print_object(string name, uvm_object value, byte scope_-separator = ".")`

Function

print_object

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_object

Parameters

- **`name`** (*`string`*)
- **`value`** (*uvm_object*)
- **`scope_separator`** (*`byte`*)

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

Print_object_header

Parameters

- **`name`** (*`string`*)

value (*uvm_object*)
scope_separator (*byte*)

virtual function void print_string(string name, string value, byte scope_-separator = ".")

Function

print_string

Prints a string field. Print_string

Parameters

name (*string*)
value (*string*)
scope_separator (*byte*)

virtual function void print_time(string name, time value, byte scope_separator = ".")

Function

print_time

Prints a time value. name is the name of the field, and value is the value to print.

The print is subject to the *$timeformat* system task for formatting time values. Print_time

Parameters

name (*string*)
value (*time*)
scope_separator (*byte*)

virtual function void print_real(string name, real value, byte scope_separator = ".")

Function

print_real

Prints a real field. Print_real

Parameters

name (*string*)
value (*real*)
scope_separator (*byte*)

virtual function void print_generic(string name, string type_name, int size, string value, byte scope_separator = ".")

Function

print_generic

Prints a field having the given *name* , *type_name* , *size* , and *value* . Print_generic

Parameters

name (*string*)
type_name (*string*)
size (*int*)
value (*string*)
scope_separator (*byte*)

virtual function string emit()

Function

emit

Emits a string representing the contents of an object in a format defined by an extension of this object. Emit

virtual function string format_row(uvm_printer_row_info row)

Function

format_row

Hook for producing custom output of a single field (row). Format_row

Parameters

row (*uvm_printer_row_info*)

`virtual function string format_header()`

Function

format_header

Hook to override base header with a custom header.

`virtual function string format_footer()`

Function

format_footer

Hook to override base footer with a custom footer.

`virtual function void print_array_header(string name, int size, string arraytype = "array", byte scope_separator = ".")`

Function

print_array_header

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_header

Parameters

- **`name`** (*`string`*)
- **`size`** (*`int`*)
- **`arraytype`** (*`string`*)
- **`scope_separator`** (*`byte`*)

`virtual function void print_array_range(int min, int max)`

Function

print_array_range

Prints a range using ellipses for values. This method is used when honoring the array knobs for partial printing of large arrays, *uvm_printer_knobs::begin_elements* and *uvm_printer_knobs::end_elements*.

This function should be called after begin_elements have been printed and before end_elements have been printed. Print_array_range

Parameters

- **`min`** (*`int`*)
- **`max`** (*`int`*)

`virtual function void print_array_footer(int size = 0)`

Function

print_array_footer

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 let's the printer know that the array printing is complete. Print_array_footer

Parameters

- **`size`** (*`int`*)

`function bit istop()`

Utility methods. Istop

`function string index_string(int index, string name = "")`

Index_string

Parameters

- **`index`** (*`int`*)
- **`name`** (*`string`*)

15.1.1.162 Class uvm_pkg::uvm_printer_knobs

Class

uvm_printer_knobs

The *uvm_printer_knobs* class defines the printer settings available to all printer subtypes.

Table 175: Variables

Name	Type	Description
header	bit	***Variable*** header Indicates whether the *uvm_printer::format_header* function should be called when printing an object.
footer	bit	***Variable*** footer Indicates whether the *uvm_printer::format_footer* function should be called when printing an object.
full_name	bit	***Variable*** full_name Indicates whether uvm_printer::adjust_name should print the full name of an identifier or just the leaf name.
identifier	bit	***Variable*** identifier Indicates whether uvm_printer::adjust_name should print the identifier. This is useful in cases where you just want the values of an object, but no identifiers.
type_name	bit	***Variable*** type_name Controls whether to print a field's type name.
size	bit	***Variable*** size Controls whether to print a field's size.
depth	int	***Variable*** depth Indicates how deep to recurse when printing objects. A depth of -1 means to print everything.
reference	bit	***Variable*** reference Controls whether to print a unique reference ID for object handles. The behavior of this knob is simulator-dependent.

continues on next page

Table 175 – continued from previous page

Name	Type	Description
begin_elements	int	***Variable*** begin_elements Defines the number of elements at the head of a list to print. Use -1 for no max.
end_elements	int	***Variable*** end_elements This defines the number of elements at the end of a list that should be printed.
prefix	string	***Variable*** prefix Specifies the string prepended to each output line
indent	int	***Variable*** indent This knob specifies the number of spaces to use for level indentation. The default level indentation is two spaces.
show_root	bit	***Variable*** show_root This setting indicates whether or not the initial object that is printed (when current depth is 0) prints the full path name. By default, the first object is treated like all other objects and only the leaf name is printed.
mcd	int	***Variable*** mcd This is a file descriptor, or multi-channel descriptor, that specifies where the print output should be directed. By default, the output goes to the standard output of the simulator.
separator	string	***Variable*** separator For tree printers only, determines the opening and closing separators used for nested objects.
show_radix	bit	***Variable*** show_radix Indicates whether the radix string ('h, and so on) should be prepended to an integral value when one is printed.
default_radix	*uvm_radix_enum*	***Variable*** default_radix This knob sets the default radix to use for integral values when no radix enum is explicitly supplied to the *uvm_printer::print_field* or *uvm_printer::print_field_int* methods.

continues on next page

Table 175 – continued from previous page

Name	Type	Description
dec_radix	string	***Variable*** dec_radix 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	***Variable*** bin_radix 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	***Variable*** oct_radix 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	***Variable*** unsigned_radix 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	***Variable*** hex_radix 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.
max_width	int	Deprecated knobs, hereafter ignored
truncation	string	
name_width	int	
type_width	int	
size_width	int	
value_width	int	
sprint	bit	

Functions

`function string get_radix_str(uvm_radix_enum radix)`

Function

get_radix_str

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

Parameters

`radix` (*uvm_radix_enum*)

15.1.1.163 Class uvm_pkg::uvm_push_driver

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_push_driver*

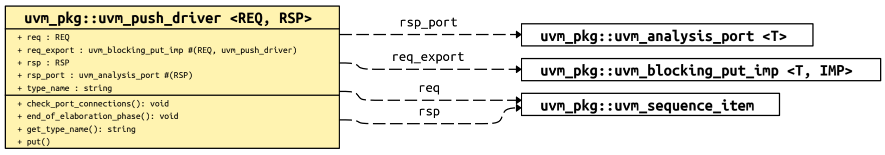

uvm_pkg::uvm_push_driver <REQ, RSP>
+ req : REQ
+ req_export : uvm_blocking_put_imp #(REQ, uvm_push_driver)
+ rsp : RSP
+ rsp_port : uvm_analysis_port #(RSP)
+ type_name : string
+ check_port_connections(): void
+ end_of_elaboration_phase(): void
+ get_type_name(): string
+ put()
rsp_port
req_export
req
rsp
uvm_pkg::uvm_analysis_port <T>
uvm_pkg::uvm_blocking_put_imp <T, IMP>
uvm_pkg::uvm_sequence_item

Fig. 49: Collaboration Diagram of uvm_push_driver

CLASS

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.

Table 176: Parameters

Name	Default value	Description
REQ	uvm_sequence_item	
RSP	REQ	

Table 177: Variables

Name	Type	Description
req_export	*uvm_blocking_put_imp#(uvm_sequence_item, uvm_push_driver#(uvm_sequence_item, uvm_sequence_item))*	***Port*** 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	*uvm_analysis_port#(uvm_sequence_item)*	***Port*** rsp_port This analysis port is used to send response transactions back to the originating sequencer.
req	*uvm_sequence_item*	

continues on next page

Table 177 – continued from previous page

Name	Type	Description
rsp	*uvm_sequence_item*	
type_name	string	

Constructors

`function new(string name, uvm_component parent)`

Function

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.

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)

Functions

`function void check_port_connections()`

`virtual function void end_of_elaboration_phase(uvm_phase phase)`

Parameters
- **`phase`** (*uvm_phase*)

`virtual function string get_type_name()`

Tasks

`virtual function put(uvm_sequence_item item)`

Parameters
- **`item`** (*uvm_sequence_item*)

15.1.1.164 Class uvm_pkg::uvm_push_sequencer

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_sequencer_base*
↪*uvm_pkg* :: *uvm_sequencer_param_base*
↪*uvm_pkg* :: *uvm_push_sequencer*

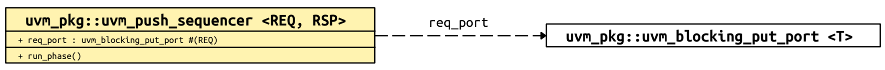

uvm_pkg::uvm_push_sequencer <REQ, RSP>
+ req_port : uvm_blocking_put_port #(REQ)
+ run_phase()
req_port
uvm_pkg::uvm_blocking_put_port <T>

Fig. 50: Collaboration Diagram of uvm_push_sequencer

CLASS

uvm_push_sequencer (REQ, RSP)

Table 178: Parameters

Name	Default value	Description
REQ	uvm_sequence_item	
RSP	REQ	

Table 179: Variables

Name	Type	Description
req_port	*uvm_blocking_put_-port#(uvm_sequence_-item)*	***Port*** req_port The push sequencer requires access to a blocking put interface. A continuous stream of sequence items are sent out this port, based on the list of available sequences loaded into this sequencer.

Table 180: Typedefs

Name	Actual Type	Description
this_type	*uvm_push_se-quencer#(REQ, RSP)*	

Constructors

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

Function

new

Standard component constructor that creates an instance of this class using the given *name* and *parent* , if any.

Parameters

`name` (*`string`*)

parent (*uvm_component*)

Tasks

`virtual function run_phase(uvm_phase phase)`

Task

run_phase

The push sequencer continuously selects from its list of available sequences and sends the next item from the selected sequence out its *req_port* using req_port.put(item). Typically, the req_port would be connected to the req_export on an instance of a <uvm_push_driver (REQ, RSP)>, which would be responsible for executing the item.

Parameters

phase (*uvm_phase*)

15.1.1.165 Class uvm_pkg::uvm_put_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_put_export*

Table 181: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.166 Class uvm_pkg::uvm_put_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_put_imp*

Table 182: Parameters

Name	Default value	Description
T	int	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **`name`** (*`string`*)
- **`imp`** (*`int`*)

15.1.1.167 Class uvm_pkg::uvm_put_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_put_port*

Table 183: Parameters

Name	Default value	Description
T	int	

Constructors

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

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.1.1.168 Class uvm_pkg::uvm_queue

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_queue*

CLASS

uvm_queue (T)

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

Table 184: Parameters

Name	Default value	Description
T	int	

Table 185: Variables

Name	Type	Description
type_name	string	

Table 186: Typedefs

Name	Actual Type	Description
this_type	*uvm_queue#(T)*	

Constructors

`function new(string name = "")`

Function

new

Creates a new queue with the given *name* .

Parameters

name (`string`)

Functions

`static function this_type get_global_queue()`

Function

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.

Return type

this_type

`static function T get_global(int index)`

Function

get_global

Returns the specified item instance from the global item queue.

Parameters

index (*int*)

virtual function T get(int index)

Function

get

Returns the item at the given *index* .

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

Parameters

index (*int*)

virtual function int size()

Function

size

Returns the number of items stored in the queue.

virtual function void insert(int index, int item)

Function

insert

Inserts the item at the given *index* in the queue.

Parameters

index (*int*)
item (*int*)

virtual function void delete(int index = -1)

Function

delete

Removes the item at the given *index* from the queue; if *index* is not provided, the entire contents of the queue are deleted.

Parameters

index (*int*)

virtual function T pop_front()

Function

pop_front

Returns the first element in the queue (index = 0), or *null* if the queue is empty.

virtual function T pop_back()

Function

pop_back

Returns the last element in the queue (index = size()-1), or *null* if the queue is empty.

virtual function void push_front(int item)

Function

push_front

Inserts the given *item* at the front of the queue.

Parameters

item (*int*)

virtual function void push_back(int item)

Function

push_back

Inserts the given *item* at the back of the queue.

Parameters

item (*int*)

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

Parameters
: **`name`** (*`string`*)

Return type
: *uvm_object*

`virtual function string get_type_name()`

`virtual function void do_copy(uvm_object rhs)`

Parameters
: **`rhs`** (*uvm_object*)

`virtual function string convert2string()`

15.1.1.169 Class uvm_pkg::uvm_random_sequence

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*
↪*uvm_pkg* :: *uvm_random_sequence*

CLASS- 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.

Constructors

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

new

Parameters

`name` (`string`)

Functions

`function int unsigned get_count()`

Function- get_count

Returns the count of the number of sub-sequences which are randomly generated. By default, count is equal to the value from the sequencer's count variable. However, if the sequencer's count variable is -1, then a random value between 0 and sequencer.max_random_count (exclusive) is chosen. The sequencer's count variable is subsequently reset to the random value that was used. If get_count() is call before the sequence has started, the return value will be sequencer.count, which may be -1.

`virtual function void do_copy(uvm_object rhs)`

Implement data functions

Parameters

`rhs` (*uvm_object*)

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

Parameters

`rhs` (*uvm_object*)

`comparer` (*uvm_comparer*)

`virtual function void do_print(uvm_printer printer)`

Parameters

`printer` (*uvm_printer*)

`virtual function void do_record(uvm_recorder recorder)`

Parameters

`recorder` (*uvm_recorder*)

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

Parameters

`name` (`string`)

Return type

uvm_object

`virtual  function string get_type_name()`

Tasks

`virtual  function  body()`

body

15.1.1.170 Class uvm_pkg::uvm_random_stimulus

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_random_stimulus*

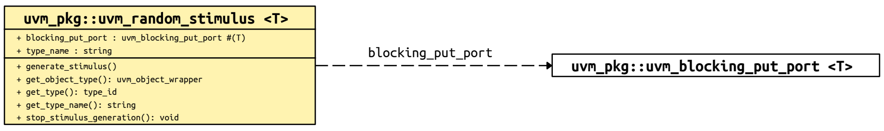

uvm_pkg::uvm_random_stimulus <T>
+ blocking_put_port : uvm_blocking_put_port #(T)
+ type_name : string
+ generate_stimulus()
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string
+ stop_stimulus_generation(): void
blocking_put_port
uvm_pkg::uvm_blocking_put_port <T>

Fig. 51: Collaboration Diagram of uvm_random_stimulus

CLASS

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.

Table 187: Parameters

Name	Default value	Description
T	uvm_transaction	

Table 188: Variables

Name	Type	Description
type_name	string	
blocking_put_port	*uvm_blocking_put_-port#(uvm_transaction)*	***Port*** blocking_put_port The blocking_put_port is used to send the generated stimulus to the rest of the testbench.

Table 189: Typedefs

Name	Actual Type	Description
this_type	*uvm_random_stimulus#(T)*	

Constructors

`function new(string name, uvm_component parent)`

Function

new

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.

Parameters

`name` (*`string`*)

`parent` (*uvm_component*)

Functions

`virtual function void stop_stimulus_generation()`

Function

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

`virtual function string get_type_name()`

Tasks

`virtual function generate_stimulus(uvm_transaction t = null, int max_count = 0)`

Function

generate_stimulus

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

Parameters

`t` (*uvm_transaction*)

`max_count` (*`int`*)

15.1.1.171 Class uvm_pkg::uvm_recorder

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_recorder*

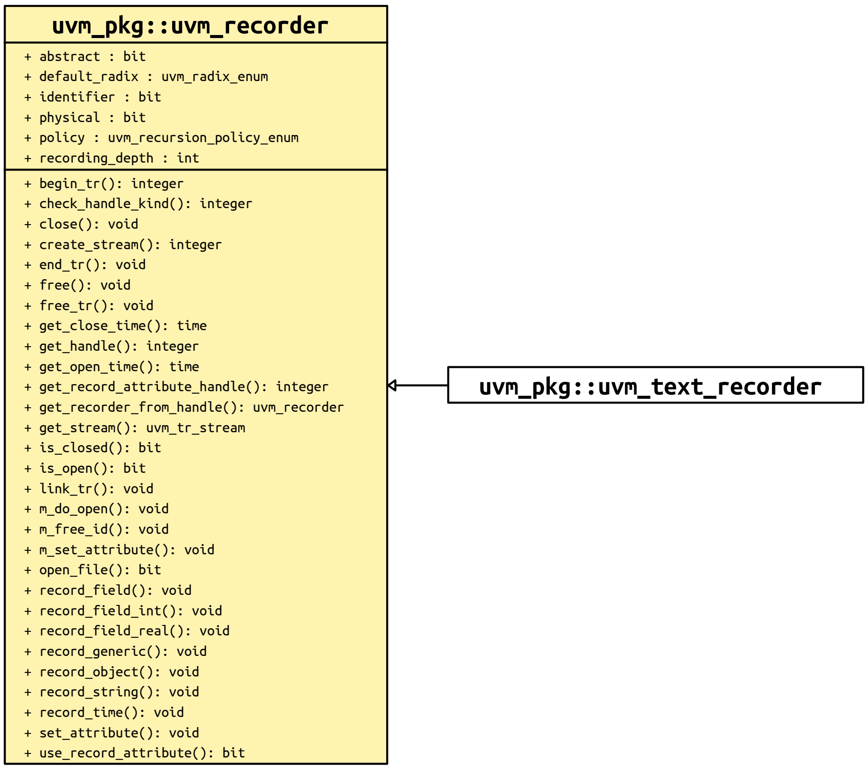


Fig. 52: Inheritance Diagram of uvm_recorder

CLASS

uvm_recorder

Abstract class which defines the *recorder* API.

Table 190: Variables

Name	Type	Description
recording_depth	int	Variable- recording_depth
default_radix	*uvm_radix_enum*	***Variable*** default_radix This is the default radix setting if *record_field* is called without a radix.

continues on next page

Table 190 – continued from previous page

Name	Type	Description
physical	bit	***Variable*** physical 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	***Variable*** abstract 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	***Variable*** identifier This bit is used to specify whether or not an object's reference should be recorded when the object is recorded.
policy	*uvm_recursion_policy_-enum*	***Variable*** recursion_policy Sets the recursion policy for recording objects. The default policy is deep (which means to recurse an object).

Constructors

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

Parameters
: **name** (*string*)

Functions

`function uvm_tr_stream get_stream()`

Function

get_stream

Returns a reference to the stream which created this record.

A warning will be asserted if get_stream is called prior to the record being initialized via do_open.

Return type
: *uvm_tr_stream*

`function void close(time close_time = 0)`

Function

close

Closes this recorder.

Closing a recorder marks the end of the transaction in the stream.

Parameters

close_time

Optional time to record as the closing time of this transaction.

This method will trigger a do_close call.
Parameters
`close_time` (*time*)

```
function void free(time close_time = 0)
```

Function

free

Frees this recorder

Freeing a recorder indicates that the stream and database can release any references to the recorder.

Parameters

close_time

Optional time to record as the closing time of this transaction.

If a recorder has not yet been closed (via a call to *close*), then *close* will automatically be called, and passed the *close_time* . If the recorder has already been closed, then the *close_time* will be ignored.

This method will trigger a do_free call.
Parameters
`close_time` (*time*)

```
function bit is_open()
```

Function

is_open

Returns true if this *uvm_recorder* was opened on its stream, but has not yet been closed.

```
function time get_open_time()
```

Function

get_open_time

Returns the *open_time*

```
function bit is_closed()
```

Function

is_closed

Returns true if this *uvm_recorder* was closed on its stream, but has not yet been freed.

```
function time get_close_time()
```

Function

get_close_time

Returns the *close_time*

```
function integer get_handle()
```

Function

get_handle

Returns a unique ID for this recorder.

A value of *0* indicates that the recorder has been *freed* , and no longer has a valid ID.

```
static  function uvm_recorder get_recorder_from_handle(integer id)
```

Function

get_recorder_from_handle

Static accessor, returns a recorder reference for a given unique id.

If no recorder exists with the given *id* , or if the recorder with that *id* has been freed, then *null* is returned.

This method can be used to access the recorder associated with a call to *uvm_transaction::begin_tr* or *uvm_component::begin_tr*.

```
integer handle = tr.begin_tr();
uvm_recorder recorder = uvm_recorder::get_recorder_from_handle(handle);
if (recorder != null) begin
  recorder.record_string("begin_msg", "Started recording transaction!");
end
```

Parameters
  id (*integer*)
Return type
  uvm_recorder

function void record_field(string name, uvm_bitstream_t value, int size, uvm_radix_enum radix = UVM_NORADIX)

Function

record_field

Records an integral field (less than or equal to 4096 bits).

Parameters

name

Name of the field

value

Value of the field to record.

size

Number of bits of the field which apply (Usually obtained via $bits).

radix

The *uvm_radix_enum* to use.

This method will trigger a do_record_field call.

Parameters
  name (*string*)
  value (*uvm_bitstream_t*)
  size (*int*)
  radix (*uvm_radix_enum*)

function void record_field_int(string name, uvm_integral_t value, int size, uvm_radix_enum radix = UVM_NORADIX)

Function

record_field_int

Records an integral field (less than or equal to 64 bits).

This optimized version of *record_field* is useful for sizes up to 64 bits.

Parameters

name

Name of the field

value

Value of the field to record

size

Number of bits of the wfield which apply (Usually obtained via $bits).

radix

The *uvm_radix_enum* to use.

This method will trigger a do_record_field_int call.

Parameters

- `name` (*string*)
- `value` (*uvm_integral_t*)
- `size` (*int*)
- `radix` (*uvm_radix_enum*)

`function void record_field_real(string name, real value)`

Function

record_field_real

Records a real field.

Parameters

name

Name of the field

value

Value of the field to record

This method will trigger a do_record_field_real call.

Parameters

- `name` (*string*)
- `value` (*real*)

`function void record_object(string name, uvm_object value)`

Function

record_object

Records an object field.

Parameters

name

Name of the field

value

Object to record

The implementation must use the <recursion_policy> and *identifier* to determine exactly what should be recorded.

Parameters

- `name` (*string*)
- `value` (*uvm_object*)

`function void record_string(string name, string value)`

Function

record_string

Records a string field.

Parameters

name

Name of the field

value

Value of the field

Parameters

- `name` (*string*)
- `value` (*string*)

`function void record_time(string name, time value)`

Function

record_time

Records a time field.

Parameters

name

Name of the field

value

Value of the field

Parameters
- **`name`** (*`string`*)
- **`value`** (*`time`*)

`function void record_generic(string name, string value, string type_name = "")`

Function

record_generic

Records a name/value pair, where *value* has been converted to a string.

For example:

```
recorder.record_generic("myvar","var_type", $sformatf("%0d",myvar), 32);
```

Parameters:

name

Name of the field

value

Value of the field

type_name

optional Type name of the field

Parameters
- **`name`** (*`string`*)
- **`value`** (*`string`*)
- **`type_name`** (*`string`*)

`virtual function bit use_record_attribute()`

Function

use_record_attribute

Indicates that this recorder does (or does not) support usage of the *`uvm_record_attribute* macro.

The default return value is *0* (not supported), developers can optionally extend *uvm_recorder* and set the value to *1* if they support the *`uvm_record_attribute* macro.

`virtual function integer get_record_attribute_handle()`

Function

get_record_attribute_handle

Provides a tool-specific handle which is compatible with *`uvm_record_attribute*.

By default, this method will return the same value as *get_handle*, however tool vendors can override this method to provide tool-specific handles which will be passed to the *`uvm_record_attribute* macro.

`virtual function bit open_file()`

Function- open_file

Opens the file in the <filename> property and assigns to the file descriptor <file>.

`virtual function integer create_stream(string name, string t, string scope)`

Function- create_stream

Parameters

- **`name`** (*`string`*)
- **`t`** (*`string`*)
- **`scope`** (*`string`*)

`virtual function void set_attribute(integer txh, string nm, logic[1023:0] value, uvm_radix_enum radix, integer numbits = 1024)`

Function- set_attribute

Parameters

- **`txh`** (*`integer`*)
- **`nm`** (*`string`*)
- **`value`** (*`logic[1023:0]`*)
- **`radix`** (*uvm_radix_enum*)
- **`numbits`** (*`integer`*)

`virtual function integer check_handle_kind(string htype, integer handle)`

Function- check_handle_kind

Parameters

- **`htype`** (*`string`*)
- **`handle`** (*`integer`*)

`virtual function integer begin_tr(string txtype, integer stream, string nm, string label = "", string desc = "", time begin_time = 0)`

Function- begin_tr

Parameters

- **`txtype`** (*`string`*)
- **`stream`** (*`integer`*)
- **`nm`** (*`string`*)
- **`label`** (*`string`*)
- **`desc`** (*`string`*)
- **`begin_time`** (*`time`*)

`virtual function void end_tr(integer handle, time end_time = 0)`

Function- end_tr

Parameters

- **`handle`** (*`integer`*)
- **`end_time`** (*`time`*)

`virtual function void link_tr(integer h1, integer h2, string relation = "")`

Function- link_tr

Parameters

- **`h1`** (*`integer`*)
- **`h2`** (*`integer`*)
- **`relation`** (*`string`*)

`virtual function void free_tr(integer handle)`

Function- free_tr

Parameters

- **`handle`** (*`integer`*)

15.1.1.172 Class uvm_pkg::uvm_reg

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_reg*

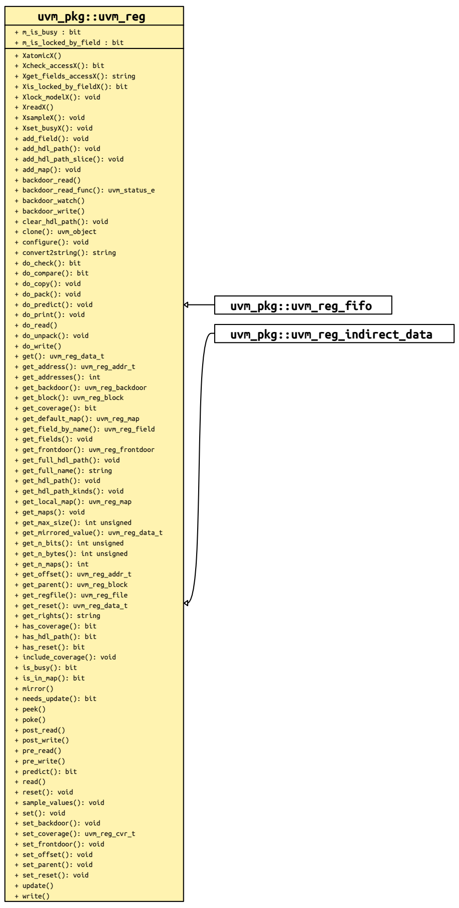


Fig. 53: Inheritance Diagram of uvm_reg

CLASS

uvm_reg

Register abstraction base class

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

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

Constructors

`function new(string name = "", int unsigned n_bits, int has_coverage)`

Function

new

Create a new instance and type-specific configuration

Creates an instance of a register abstraction class with the specified name.

n_bits specifies the total number of bits in the register. Not all bits need to be implemented. This value is usually a multiple of 8.

has_coverage specifies which functional coverage models are present in the extension of the register abstraction class. Multiple functional coverage models may be specified by adding their symbolic names, as defined by the *uvm_coverage_model_e* type. New

Parameters

- **`name`** (*`string`*)
- **`n_bits`** (*`int unsigned`*)
- **`has_coverage`** (*`int`*)

Functions

`function void configure(uvm_reg_block blk_parent, uvm_reg_file regfile_-parent = null, string hdl_path = "")`

Function

configure

Instance-specific configuration

Specify the parent block of this register. May also set a parent register file for this register,

If the register is implemented in a single HDL variable, its name is specified as the *hdl_path* . Otherwise, if the register is implemented as a concatenation of variables (usually one per field), then the HDL path must be specified using the *add_hdl_path()* or *add_hdl_path_slice* method. Configure

Parameters

- **`blk_parent`** (*uvm_reg_block*)
- **`regfile_parent`** (*uvm_reg_file*)
- **`hdl_path`** (*`string`*)

`virtual function void set_offset(uvm_reg_map map, uvm_reg_addr_t offset, bit unmapped = 0)`

Function

set_offset

Modify the offset of the register

The offset of a register within an address map is set using the *uvm_reg_map::add_reg()* method. This method is used to modify that offset dynamically.

Modifying the offset of a register will make the register model diverge from the specification that was used to create it. Set_offset

Parameters

- **`map`** (*uvm_reg_map*)
- **`offset`** (*uvm_reg_addr_t*)
- **`unmapped`** (*`bit`*)

`virtual function void set_parent(uvm_reg_block blk_parent, uvm_reg_file regfile_-parent)`

Set_parent

Parameters

- **`blk_parent`** (*uvm_reg_block*) -- Local
- **`regfile_parent`** (*uvm_reg_file*)

`virtual function void add_field(uvm_reg_field field)`

Add_field

Parameters

- **`field`** (*uvm_reg_field*) -- Local

`virtual function void add_map(uvm_reg_map map)`

Add_map

Parameters

- **`map`** (*uvm_reg_map*) -- Local

`function void Xlock_modelX()`

Xlock_modelXlocal

`virtual function string get_full_name()`

Function

get_full_name

Get the hierarchical name

Return the hierarchal name of this register. The base of the hierarchical name is the root block. Get_full_name

`virtual function uvm_reg_block get_parent()`

Function

get_parent

Get the parent block. Get_parent

Return type
uvm_reg_block

`virtual function uvm_reg_block get_block()`

Get_block

Return type
uvm_reg_block

`virtual function uvm_reg_file get_regfile()`

Function

get_regfile

Get the parent register file

Returns *null* if this register is instantiated in a block. Get_regfile

Return type
uvm_reg_file

`virtual function int get_n_maps()`

Function

get_n_maps

Returns the number of address maps this register is mapped in. Get_n_maps

`function bit is_in_map(uvm_reg_map map)`

Function

is_in_map

Returns 1 if this register is in the specified address *map* . Is_in_map

Parameters
`map` (*uvm_reg_map*)

`virtual function void get_maps(uvm_reg_map maps)`

Function

get_maps

Returns all of the address *maps* where this register is mapped. Get_maps

Parameters
`maps` (*uvm_reg_map*)

`virtual function uvm_reg_map get_local_map(uvm_reg_map map, string caller = "")`

Get_local_map

Parameters
- **`map`** (*uvm_reg_map*) -- Local
- **`caller`** (*`string`*)

Return type
uvm_reg_map

`virtual function uvm_reg_map get_default_map(string caller = "")`

Get_default_map

Parameters

`caller` (*`string`*) -- Local

Return type

uvm_reg_map

`virtual function string get_rights(uvm_reg_map map = null)`

Function

get_rights

Returns the accessibility ("RW, "RO", or "WO") of this register in the given *map* .

If no address map is specified and the register is mapped in only one address map, that address map is used. If the register is mapped in more than one address map, the default address map of the parent block is used.

Whether a register field can be read or written depends on both the field's configured access policy (refer to *uvm_reg_field::configure*) and the register's accessibility rights in the map being used to access the field.

If an address map is specified and the register is not mapped in the specified address map, an error message is issued and "RW" is returned. Get_rights

Parameters

`map` (*uvm_reg_map*)

`virtual function int unsigned get_n_bits()`

Function

get_n_bits

Returns the width, in bits, of this register. Get_n_bits

`virtual function int unsigned get_n_bytes()`

Function

get_n_bytes

Returns the width, in bytes, of this register. Rounds up to next whole byte if register is not a multiple of 8. Get_n_bytes

`static function int unsigned get_max_size()`

Function

get_max_size

Returns the maximum width, in bits, of all registers. Get_max_size

`virtual function void get_fields(uvm_reg_field fields)`

Function

get_fields

Return the fields in this register

Fills the specified array with the abstraction class for all of the fields contained in this register. Fields are ordered from least-significant position to most-significant position within the register. Get_fields

Parameters

`fields` (*uvm_reg_field*)

`virtual function uvm_reg_field get_field_by_name(string name)`

Function

get_field_by_name

Return the named field in this register

Finds a field with the specified name in this register and returns its abstraction class. If no fields are found, returns *null* . Get_field_by_name

Parameters

`name` (*`string`*)

Return type

uvm_reg_field

`function string Xget_fields_accessX(uvm_reg_map map)`

Xget_field_accessX

Returns "WO" if all of the fields in the registers are write-only Returns "RO" if all of the fields in the registers are read-only Returns "RW" otherwise.

Parameters

`map` (*uvm_reg_map*) -- Local

`virtual function uvm_reg_addr_t get_offset(uvm_reg_map map = null)`

Function

get_offset

Returns the offset of this register

Returns the offset of this register in an address *map* .

If no address map is specified and the register is mapped in only one address map, that address map is used. If the register is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the register is not mapped in the specified address map, an error message is issued. Get_offset

Parameters

`map` (*uvm_reg_map*)

Return type

uvm_reg_addr_t

`virtual function uvm_reg_addr_t get_address(uvm_reg_map map = null)`

Function

get_address

Returns the base external physical address of this register

Returns the base external physical address of this register if accessed through the specified address *map* .

If no address map is specified and the register is mapped in only one address map, that address map is used. If the register is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the register is not mapped in the specified address map, an error message is issued. Get_address

Parameters

`map` (*uvm_reg_map*)

Return type

uvm_reg_addr_t

`virtual function int get_addresses(uvm_reg_map map = null, uvm_reg_addr_t addr)`

Function

get_addresses

Identifies the external physical address(es) of this register

Computes all of the external physical addresses that must be accessed to completely read or write this register. The addressed are specified in little endian order. Returns the number of bytes transferred on each access.

If no address map is specified and the register is mapped in only one address map, that address map is used. If the register is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the register is not mapped in the specified address map, an error message is issued. Get_addresses

Parameters

`map` (*uvm_reg_map*)

`addr` (*uvm_reg_addr_t*)

`virtual function void set(uvm_reg_data_t value, string fname = "", int lineno = 0)`

Function

set

Set the desired value for this register

Sets the desired value of the fields in the register to the specified value. Does not actually set the value of the register in the design, only the desired value in its corresponding abstraction class in the RegModel model. Use the *uvm_reg::update()* method to update the actual register with the mirrored value or the *uvm_reg::write()* method to set the actual register and its mirrored value.

Unless this method is used, the desired value is equal to the mirrored value.

Refer *uvm_reg_field::set()* for more details on the effect of setting mirror values on fields with different access policies.

To modify the mirrored field values to a specific value, and thus use the mirrored as a scoreboard for the register values in the DUT, use the *uvm_reg::predict()* method. Set

Parameters

- **`value`** (*uvm_reg_data_t*)
- **`fname`** (`string`)
- **`lineno`** (`int`)

`virtual function uvm_reg_data_t get(string fname = "", int lineno = 0)`

Function

get

Return the desired value of the fields in the register.

Does not actually read the value of the register in the design, only the desired value in the abstraction class. Unless set to a different value using the *uvm_reg::set()*, the desired value and the mirrored value are identical.

Use the *uvm_reg::read()* or *uvm_reg::peek()* method to get the actual register value.

If the register contains write-only fields, the desired/mirrored value for those fields are the value last written and assumed to reside in the bits implementing these fields. Although a physical read operation would something different for these fields, the returned value is the actual content. Get

Parameters

- **`fname`** (`string`)
- **`lineno`** (`int`)

Return type

uvm_reg_data_t

`virtual function uvm_reg_data_t get_mirrored_value(string fname = "", int lineno = 0)`

Function

get_mirrored_value

Return the mirrored value of the fields in the register.

Does not actually read the value of the register in the design

If the register contains write-only fields, the desired/mirrored value for those fields are the value last written and assumed to reside in the bits implementing these fields. Although a physical read operation would something different for these fields, the returned value is the actual content. Get_mirrored_value

Parameters

- **`fname`** (`string`)
- **`lineno`** (`int`)

Return type

uvm_reg_data_t

`virtual function bit needs_update()`

Function

needs_update

Returns 1 if any of the fields need updating

See *uvm_reg_field::needs_update()* for details. Use the *uvm_reg::update()* to actually update the DUT register. Needs_update

`virtual function void reset(string kind = "HARD")`

Function

reset

Reset the desired/mirrored value for this register.

Sets the desired and mirror value of the fields in this register to the reset value for the specified reset *kind* . See *uvm_reg_field.reset()* for more details.

Also resets the semaphore that prevents concurrent access to the register. This semaphore must be explicitly reset if a thread accessing this register array was killed in before the access was completed. Reset

Parameters

kind (*string*)

virtual function uvm_reg_data_t get_reset(string kind = "HARD")

Function

get_reset

Get the specified reset value for this register

Return the reset value for this register for the specified reset *kind* . Get_reset

Parameters

kind (*string*)

Return type

uvm_reg_data_t

virtual function bit has_reset(string kind = "HARD", bit delete = 0)

Function

has_reset

Check if any field in the register has a reset value specified for the specified reset *kind* . If *delete* is TRUE, removes the reset value, if any. Has_reset

Parameters

kind (*string*)
delete (*bit*)

virtual function void set_reset(uvm_reg_data_t value, string kind = "HARD")

Function

set_reset

Specify or modify the reset value for this register

Specify or modify the reset value for all the fields in the register corresponding to the cause specified by *kind* . Set_reset

Parameters

value (*uvm_reg_data_t*)
kind (*string*)

virtual function bit predict(uvm_reg_data_t value, uvm_reg_byte_en_t be = -1, uvm_predict_e kind = UVM_PREDICT_DIRECT, uvm_path_e path = UVM_FRONTDOOR, uvm_reg_map map = null, string fname = "", int lineno = 0)

Function

predict

Update the mirrored and desired value for this register.

Predict the mirror (and desired) value of the fields in the register based on the specified observed *value* on a specified address *map* , or based on a calculated value. See *uvm_reg_field::predict()* for more details.

Returns TRUE if the prediction was successful for each field in the register. Predict

Parameters

value (*uvm_reg_data_t*)
be (*uvm_reg_byte_en_t*)
kind (*uvm_predict_e*)
path (*uvm_path_e*)
map (*uvm_reg_map*)
fname (*string*)
lineno (*int*)

`function bit is_busy()`

Function

is_busy

Returns 1 if register is currently being read or written. Is_busy

`function void Xset_busyX(bit busy)`

Xset_busyX

Parameters

`busy` (`bit`) -- Local

`virtual function bit Xcheck_accessX(uvm_reg_item rw, uvm_reg_map_info map_info, string caller)`

Xcheck_accessXlocal

Parameters

`rw` (*uvm_reg_item*)
`map_info` (*uvm_reg_map_info*)
`caller` (`string`)

`function bit Xis_locked_by_fieldX()`

Xis_loacked_by_fieldXlocal

`virtual function bit do_check(uvm_reg_data_t expected, uvm_reg_data_t actual, uvm_reg_map map)`

Do_check

Parameters

`expected` (*uvm_reg_data_t*)
`actual` (*uvm_reg_data_t*)
`map` (*uvm_reg_map*)

`virtual function void do_predict(uvm_reg_item rw, uvm_predict_e kind = UVM_PREDICT_DIRECT, uvm_reg_byte_en_t be = -1)`

Do_predict

Parameters

`rw` (*uvm_reg_item*)
`kind` (*uvm_predict_e*)
`be` (*uvm_reg_byte_en_t*)

`function void set_frontdoor(uvm_reg_frontdoor ftdr, uvm_reg_map map = null, string fname = "", int lineno = 0)`

Function

set_frontdoor

Set a user-defined frontdoor for this register

By default, registers are mapped linearly into the address space of the address maps that instantiate them. If registers are accessed using a different mechanism, a user-defined access mechanism must be defined and associated with the corresponding register abstraction class

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

Parameters

`ftdr` (*uvm_reg_frontdoor*)
`map` (*uvm_reg_map*)
`fname` (`string`)
`lineno` (`int`)

`function uvm_reg_frontdoor get_frontdoor(uvm_reg_map map = null)`

Function

get_frontdoor

Returns the user-defined frontdoor for this register

If *null* , no user-defined frontdoor has been defined. A user-defined frontdoor is defined by using the *uvm_reg::set_frontdoor()* method.

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

Parameters
`map` (*uvm_reg_map*)

Return type
uvm_reg_frontdoor

`function void set_backdoor(uvm_reg_backdoor bkdr, string fname = "", int lineno = 0)`

Function

set_backdoor

Set a user-defined backdoor for this register

By default, registers are accessed via the built-in string-based DPI routines if an HDL path has been specified using the *uvm_reg::configure()* or *uvm_reg::add_hdl_path()* method.

If this default mechanism is not suitable (e.g. because the register is not implemented in pure SystemVerilog) a user-defined access mechanism must be defined and associated with the corresponding register abstraction class

A user-defined backdoor is required if active update of the mirror of this register abstraction class, based on observed changes of the corresponding DUT register, is used. Set_backdoor

Parameters
`bkdr` (*uvm_reg_backdoor*)
`fname` (*`string`*)
`lineno` (*`int`*)

`function uvm_reg_backdoor get_backdoor(bit inherited = 1)`

Function

get_backdoor

Returns the user-defined backdoor for this register

If *null* , no user-defined backdoor has been defined. A user-defined backdoor is defined by using the *uvm_reg::set_backdoor()* method.

If *inherited* is TRUE, returns the backdoor of the parent block if none have been specified for this register. Get_backdoor

Parameters
`inherited` (*`bit`*)

Return type
uvm_reg_backdoor

`function void clear_hdl_path(string kind = "RTL")`

Function

clear_hdl_path

Delete HDL paths

Remove any previously specified HDL path to the register instance for the specified design abstraction. Clear_hdl_path

Parameters
`kind` (*`string`*)

`function void add_hdl_path(uvm_hdl_path_slice slices, string kind = "RTL")`

Function

add_hdl_path

Add an HDL path

Add the specified HDL path to the register instance for the specified design abstraction. This method may be called more than once for the same design abstraction if the register is physically duplicated in the design abstraction

For example, the following register

```
        1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
Bits:   5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
      +-+---+-------------+---+-------+
      |A|xxx|      B      |xxx|   C   |
      +-+---+-------------+---+-------+
```

would be specified using the following literal value:

```
add_hdl_path('{ '{"A_reg", 15, 1},
                '{"B_reg",  6, 7},
                '{'C_reg",  0, 4} } );
```

If the register is implemented using a single HDL variable, The array should specify a single slice with its *offset* and *size* specified as -1. For example:

```
r1.add_hdl_path('{ '{"r1", -1, -1} });. Add_hdl_path
```

Parameters

slices (*uvm_hdl_path_slice*)

kind (*string*)

function void add_hdl_path_slice(string name, int offset, int size, bit first = 0, string kind = "RTL")

Function

add_hdl_path_slice

Append the specified HDL slice to the HDL path of the register instance for the specified design abstraction. If *first* is TRUE, starts the specification of a duplicate HDL implementation of the register. Add_hdl_path_slice

Parameters

name (*string*)

offset (*int*)

size (*int*)

first (*bit*)

kind (*string*)

function bit has_hdl_path(string kind = "")

Function

has_hdl_path

Check if a HDL path is specified

Returns TRUE if the register instance has a HDL path defined for the specified design abstraction. If no design abstraction is specified, uses the default design abstraction specified for the parent block. Has_hdl_path

Parameters

kind (*string*)

function void get_hdl_path(uvm_hdl_path_concat paths, string kind = "")

Function

get_hdl_path

Get the incremental HDL path(s)

Returns the HDL path(s) defined for the specified design abstraction in the register instance. Returns only the component of the HDL paths that corresponds to the register, not a full hierarchical path

If no design abstraction is specified, the default design abstraction for the parent block is used. Get_hdl_path

Parameters

paths (*uvm_hdl_path_concat*)

kind (*string*)

function void get_hdl_path_kinds(string kinds)

Function

get_hdl_path_kinds

Get design abstractions for which HDL paths have been defined. Get_hdl_path_kinds

Parameters
    kinds (*string*)

function void get_full_hdl_path(uvm_hdl_path_concat paths, string kind = "", string separator = ".")

Function

get_full_hdl_path

Get the full hierarchical HDL path(s)

Returns the full hierarchical HDL path(s) defined for the specified design abstraction in the register instance. There may be more than one path returned even if only one path was defined for the register instance, if any of the parent components have more than one path defined for the same design abstraction

If no design abstraction is specified, the default design abstraction for each ancestor block is used to get each incremental path. Get_full_hdl_path

Parameters
    paths (*uvm_hdl_path_concat*)
    kind (*string*)
    separator (*string*)

virtual function uvm_status_e backdoor_read_func(uvm_reg_item rw)

Function

backdoor_read_func

User-defined backdoor read access

Override the default string-based DPI backdoor access read for this register type. Backdoor_read_func

Parameters
    rw (*uvm_reg_item*)

Return type
    uvm_status_e

static function void include_coverage(string scope, uvm_reg_cvr_t models, uvm_-object accessor = null)

Function

include_coverage

Specify which coverage model that must be included in various block, register or memory abstraction class instances.

The coverage models are specified by OR'ing or adding the *uvm_coverage_model_e* coverage model identifiers corresponding to the coverage model to be included.

The scope specifies a hierarchical name or pattern identifying a block, memory or register abstraction class instances. Any block, memory or register whose full hierarchical name matches the specified scope will have the specified functional coverage models included in them.

The scope can be specified as a POSIX regular expression or simple pattern. See <uvm_resource_base::Scope Interface> for more details.

```
uvm_reg::include_coverage("*", UVM_CVR_ALL);
```

The specification of which coverage model to include in which abstraction class is stored in a *uvm_reg_cvr_t* resource in the *uvm_resource_db* resource database, in the "uvm_reg::" scope namespace. Include_coverage

Parameters
    scope (*string*)
    models (*uvm_reg_cvr_t*)
    accessor (*uvm_object*)

virtual function bit has_coverage(uvm_reg_cvr_t models)

Function

has_coverage

Check if register has coverage model(s)

Returns TRUE if the register abstraction class contains a coverage model for all of the models specified. Models are specified by adding the symbolic value of individual coverage model as defined in *uvm_coverage_model_e*. Has_coverage

Parameters

models (*uvm_reg_cvr_t*)

virtual function uvm_reg_cvr_t set_coverage(uvm_reg_cvr_t is_on)

Function

set_coverage

Turns on coverage measurement.

Turns the collection of functional coverage measurements on or off for this register. The functional coverage measurement is turned on for every coverage model specified using *uvm_coverage_model_e* symbolic identifiers. Multiple functional coverage models can be specified by adding the functional coverage model identifiers. All other functional coverage models are turned off. Returns the sum of all functional coverage models whose measurements were previously on.

This method can only control the measurement of functional coverage models that are present in the register abstraction classes, then enabled during construction. See the *uvm_reg::has_coverage()* method to identify the available functional coverage models. Set_coverage

Parameters

is_on (*uvm_reg_cvr_t*)

Return type

uvm_reg_cvr_t

virtual function bit get_coverage(uvm_reg_cvr_t is_on)

Function

get_coverage

Check if coverage measurement is on.

Returns TRUE if measurement for all of the specified functional coverage models are currently on. Multiple functional coverage models can be specified by adding the functional coverage model identifiers.

See *uvm_reg::set_coverage()* for more details. Get_coverage

Parameters

is_on (*uvm_reg_cvr_t*)

virtual function void sample_values()

Function

sample_values

Functional coverage measurement method for field values

This method is invoked by the user or by the *uvm_reg_block::sample_values()* method of the parent block to trigger the sampling of the current field values in the register-level functional coverage model.

This method may be extended by the abstraction class generator to perform the required sampling in any provided field-value functional coverage model.

function void XsampleX(uvm_reg_data_t data, uvm_reg_data_t byte_en, bit is_read, uvm_reg_map map)

Parameters

- **data** (*uvm_reg_data_t*) -- Local
- **byte_en** (*uvm_reg_data_t*)
- **is_read** (`bit`)
- **map** (*uvm_reg_map*)

virtual function void do_print(uvm_printer printer)

Do_print

Parameters

printer (*uvm_printer*)

virtual function string convert2string()

Convert2string

virtual function uvm_object clone()

Clone

Return type

uvm_object

virtual function void do_copy(uvm_object rhs)

Do_copy

Parameters

rhs (*uvm_object*)

virtual function bit do_compare(uvm_object rhs, uvm_comparer comparer)

Do_compare

Parameters

rhs (*uvm_object*)
comparer (*uvm_comparer*)

virtual function void do_pack(uvm_packer packer)

Do_pack

Parameters

packer (*uvm_packer*)

virtual function void do_unpack(uvm_packer packer)

Do_unpack

Parameters

packer (*uvm_packer*)

Tasks

virtual function write(uvm_status_e status, uvm_reg_data_t value, uvm_path_-e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)

Task

write

Write the specified value in this register

Write *value* in the DUT register that corresponds to this abstraction class instance using the specified access *path* . If the register is mapped in more than one address map, an address *map* must be specified if a physical access is used (front-door access). If a back-door access path is used, the effect of writing the register through a physical access is mimicked. For example, read-only bits in the registers will not be written.

The mirrored value will be updated using the *uvm_reg::predict()* method. Write

Parameters

status (*uvm_status_e*)
value (*uvm_reg_data_t*)
path (*uvm_path_e*)
map (*uvm_reg_map*)
parent (*uvm_sequence_base*)
prior (`int`)
extension (*uvm_object*)
fname (`string`)
lineno (`int`)

virtual function read(uvm_status_e status, uvm_reg_data_t value, uvm_path_-e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)

Task

read

Read the current value from this register

Read and return *value* from the DUT register that corresponds to this abstraction class instance using the specified access *path* . If the register is mapped in more than one address map, an address *map* must be specified if a physical access is used (front-door access). If a back-door access path is used, the effect of reading the register through a physical access is mimicked. For example, clear-on-read bits in the registers will be set to zero.

The mirrored value will be updated using the *uvm_reg::predict()* method. Read

Parameters

- **`status`** (*uvm_status_e*)
- **`value`** (*uvm_reg_data_t*)
- **`path`** (*uvm_path_e*)
- **`map`** (*uvm_reg_map*)
- **`parent`** (*uvm_sequence_base*)
- **`prior`** (`int`)
- **`extension`** (*uvm_object*)
- **`fname`** (`string`)
- **`lineno`** (`int`)

```
virtual function poke(uvm_status_e status, uvm_reg_data_t value, string kind = "",
uvm_sequence_base parent = null, uvm_object extension = null, string fname = "",
int lineno = 0)
```

Task

poke

Deposit the specified value in this register

Deposit the value in the DUT register corresponding to this abstraction class instance, as-is, using a back-door access.

Uses the HDL path for the design abstraction specified by *kind* .

The mirrored value will be updated using the *uvm_reg::predict()* method. Poke

Parameters

- **`status`** (*uvm_status_e*)
- **`value`** (*uvm_reg_data_t*)
- **`kind`** (`string`)
- **`parent`** (*uvm_sequence_base*)
- **`extension`** (*uvm_object*)
- **`fname`** (`string`)
- **`lineno`** (`int`)

```
virtual function peek(uvm_status_e status, uvm_reg_data_t value, string kind = "",
uvm_sequence_base parent = null, uvm_object extension = null, string fname = "",
int lineno = 0)
```

Task

peek

Read the current value from this register

Sample the value in the DUT register corresponding to this abstraction class instance using a back-door access. The register value is sampled, not modified.

Uses the HDL path for the design abstraction specified by *kind* .

The mirrored value will be updated using the *uvm_reg::predict()* method. Peek

Parameters

- **`status`** (*uvm_status_e*)
- **`value`** (*uvm_reg_data_t*)
- **`kind`** (`string`)
- **`parent`** (*uvm_sequence_base*)
- **`extension`** (*uvm_object*)
- **`fname`** (`string`)
- **`lineno`** (`int`)

```
virtual function update(uvm_status_e status, uvm_path_e path = UVM_DEFAULT_PATH,
uvm_reg_map map = null, uvm_sequence_base parent = null, int prior = -1, uvm_-
object extension = null, string fname = "", int lineno = 0)
```

Task

update

Updates the content of the register in the design to match the desired value

This method performs the reverse operation of *uvm_reg::mirror()*. Write this register if the DUT register is out-of-date with the desired/mirrored value in the abstraction class, as determined by the *uvm_reg::needs_update()* method.

The update can be performed using the using the physical interfaces (frontdoor) or *uvm_reg::poke()* (backdoor) access. If the register is mapped in multiple address maps and physical access is used (front-door), an address *map* must be specified. Update

Parameters

- **status** (*uvm_status_e*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **parent** (*uvm_sequence_base*)
- **prior** (*int*)
- **extension** (*uvm_object*)
- **fname** (*string*)
- **lineno** (*int*)

```
virtual function mirror(uvm_status_e status, uvm_check_e check = UVM_NO_CHECK,
uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_-
base parent = null, int prior = -1, uvm_object extension = null, string fname = "",
int lineno = 0)
```

Task

mirror

Read the register and update/check its mirror value

Read the register and optionally compared the readback value with the current mirrored value if *check* is <UVM_CHECK>. The mirrored value will be updated using the *uvm_reg::predict()* method based on the readback value.

The mirroring can be performed using the physical interfaces (frontdoor) or *uvm_reg::peek()* (backdoor).

If *check* is specified as UVM_CHECK, an error message is issued if the current mirrored value does not match the readback value. Any field whose check has been disabled with *uvm_reg_field::set_compare()* will not be considered in the comparison.

If the register is mapped in multiple address maps and physical access is used (front-door access), an address *map* must be specified. If the register contains write-only fields, their content is mirrored and optionally checked only if a UVM_BACKDOOR access path is used to read the register. Mirror

Parameters

- **status** (*uvm_status_e*)
- **check** (*uvm_check_e*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **parent** (*uvm_sequence_base*)
- **prior** (*int*)
- **extension** (*uvm_object*)
- **fname** (*string*)
- **lineno** (*int*)

```
function XreadX(uvm_status_e status, uvm_reg_data_t value, uvm_path_e path, uvm_-
reg_map map, uvm_sequence_base parent = null, int prior = -1, uvm_-
object extension = null, string fname = "", int lineno = 0)
```

XreadX

Parameters

- **status** (*uvm_status_e*) -- Local
- **value** (*uvm_reg_data_t*)

path (*uvm_path_e*)
map (*uvm_reg_map*)
parent (*uvm_sequence_base*)
prior (*int*)
extension (*uvm_object*)
fname (*string*)
lineno (*int*)

function XatomicX(bit on)

XatomicX

Parameters

on (*bit*) -- Local

virtual function do_write(uvm_reg_item rw)

Do_write

Parameters

rw (*uvm_reg_item*)

virtual function do_read(uvm_reg_item rw)

Do_read

Parameters

rw (*uvm_reg_item*)

virtual function backdoor_read(uvm_reg_item rw)

Function

backdoor_read

User-define backdoor read access

Override the default string-based DPI backdoor access read for this register type. By default calls *uvm_reg::backdoor_read_func()*. Backdoor_read

Parameters

rw (*uvm_reg_item*)

virtual function backdoor_write(uvm_reg_item rw)

Function

backdoor_write

User-defined backdoor read access

Override the default string-based DPI backdoor access write for this register type. Backdoor_write

Parameters

rw (*uvm_reg_item*)

virtual function backdoor_watch()

Function

backdoor_watch

User-defined DUT register change monitor

Watch the DUT register corresponding to this abstraction class instance for any change in value and return when a value-change occurs. This may be implemented a string-based DPI access if the simulation tool provide a value-change callback facility. Such a facility does not exists in the standard SystemVerilog DPI and thus no default implementation for this method can be provided.

virtual function pre_write(uvm_reg_item rw)

Task

pre_write

Called before register write.

If the specified data value, access *path* or address *map* are modified, the updated data value, access path or address map will be used to perform the register operation. If the *status* is modified to anything other than <UVM_IS_OK>, the operation is aborted.

The registered callback methods are invoked after the invocation of this method. All register callbacks are executed before the corresponding field callbacks

Parameters

`rw` (*uvm_reg_item*)

`virtual function post_write(uvm_reg_item rw)`

Task

post_write

Called after register write.

If the specified *status* is modified, the updated status will be returned by the register operation.

The registered callback methods are invoked before the invocation of this method. All register callbacks are executed before the corresponding field callbacks

Parameters

`rw` (*uvm_reg_item*)

`virtual function pre_read(uvm_reg_item rw)`

Task

pre_read

Called before register read.

If the specified access *path* or address *map* are modified, the updated access path or address map will be used to perform the register operation. If the *status* is modified to anything other than <UVM_IS_OK>, the operation is aborted.

The registered callback methods are invoked after the invocation of this method. All register callbacks are executed before the corresponding field callbacks

Parameters

`rw` (*uvm_reg_item*)

`virtual function post_read(uvm_reg_item rw)`

Task

post_read

Called after register read.

If the specified readback data or *status* is modified, the updated readback data or status will be returned by the register operation.

The registered callback methods are invoked before the invocation of this method. All register callbacks are executed before the corresponding field callbacks

Parameters

`rw` (*uvm_reg_item*)

15.1.1.173 Class uvm_pkg::uvm_reg_access_seq

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*
↪*uvm_pkg* :: *uvm_reg_sequence*
↪*uvm_pkg* :: *uvm_reg_access_seq*

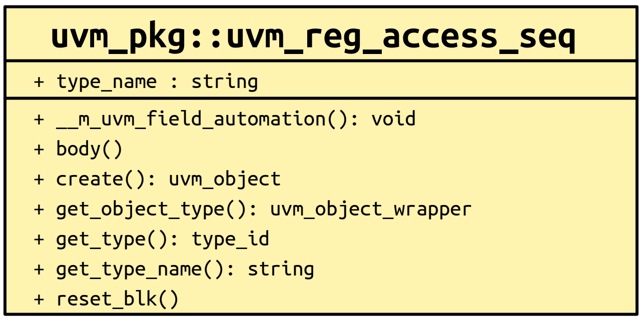

uvm_pkg::uvm_reg_access_seq
+ type_name : string
+ __m_uvm_field_automation(): void
+ body()
+ create(): uvm_object
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string
+ reset_blk()

Fig. 54: Collaboration Diagram of uvm_reg_access_seq

Class

uvm_reg_access_seq

Verify the accessibility of all registers in a block by executing the *uvm_reg_single_access_seq* sequence on every register within it.

If bit-type resource named "NO_REG_TESTS" or "NO_REG_ACCESS_TEST" in the "REG::" namespace matches the full name of the block, the block is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.get_full_name(),".*"},
                           "NO_REG_TESTS", 1, this);
```

Constructors

function new(string name = "uvm_reg_access_seq")

Parameters
name (*string*)

Tasks

virtual function body()

Task

body

Executes the Register Access sequence. Do not call directly. Use seq.start() instead.

virtual function reset_blk(uvm_reg_block blk)

Task

reset_blk

Reset the DUT that corresponds to the specified block abstraction class.

Currently empty. Will rollback the environment's phase to the *reset* phase once the new phasing is available.

In the meantime, the DUT should be reset before executing this test sequence or this method should be implemented in an extension to reset the DUT.

Parameters

`blk` (*uvm_reg_block*)

15.1.1.174 Class uvm_pkg::uvm_reg_adapter

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_reg_adapter*

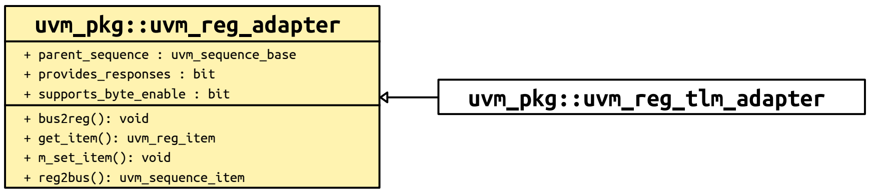

uvm_pkg::uvm_reg_adapter
+ parent_sequence : uvm_sequence_base
+ provides_responses : bit
+ supports_byte_enable : bit
+ bus2reg(): void
+ get_item(): uvm_reg_item
+ m_set_item(): void
+ reg2bus(): uvm_sequence_item
uvm_pkg::uvm_reg_tlm_adapter

Fig. 55: Inheritance Diagram of uvm_reg_adapter

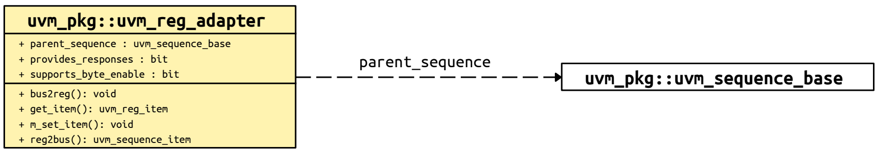

uvm_pkg::uvm_reg_adapter
+ parent_sequence : uvm_sequence_base
+ provides_responses : bit
+ supports_byte_enable : bit
+ bus2reg(): void
+ get_item(): uvm_reg_item
+ m_set_item(): void
+ reg2bus(): uvm_sequence_item
parent_sequence
uvm_pkg::uvm_sequence_base

Fig. 56: Collaboration Diagram of uvm_reg_adapter

Class

uvm_reg_adapter

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

Table 191: Variables

Name	Type	Description
supports_byte_enable	bit	***Variable*** supports_byte_enable Set this bit in extensions of this class if the bus protocol supports byte enables.
provides_responses	bit	***Variable*** provides_responses Set this bit in extensions of this class if the bus driver provides separate response items.
parent_sequence	*uvm_sequence_base*	***Variable*** parent_sequence Set this member in extensions of this class if the bus driver requires bus items be executed via a particular sequence base type. The sequence assigned to this member must implement do_clone().

Constructors

`function new(string name = "")`

Function

new

Create a new instance of this type, giving it the optional *name* .

Parameters
: **name** (*string*)

Functions

`virtual function uvm_sequence_item reg2bus(uvm_reg_bus_op rw)`

Function

reg2bus

Extensions of this class *must* implement this method to convert the specified *uvm_reg_bus_op* to a corresponding *uvm_sequence_item* subtype that defines the bus transaction.

The method must allocate a new bus-specific *uvm_sequence_item*, assign its members from the corresponding members from the given generic *rw* bus operation, then return it.

Parameters
: **rw** (*uvm_reg_bus_op*)

Return type
: *uvm_sequence_item*

`virtual function void bus2reg(uvm_sequence_item bus_item, uvm_reg_bus_op rw)`

Function

bus2reg

Extensions of this class *must* implement this method to copy members of the given bus-specific *bus_item* to corresponding members of the provided *bus_rw* instance. Unlike *reg2bus*, the resulting transaction is not allocated from scratch. This is to accommodate applications where the bus response must be returned in the original request.

Parameters
: **bus_item** (*uvm_sequence_item*)
: **rw** (*uvm_reg_bus_op*)

`virtual function uvm_reg_item get_item()`

function

get_item

Returns the bus-independent read/write information that corresponds to the generic bus transaction currently translated to a bus-specific transaction. This function returns a value reference only when called in the *uvm_reg_adapter::reg2bus()* method. It returns *null* at all other times. The content of the return *uvm_reg_item* instance must not be modified and used strictly to obtain additional information about the operation.

Return type
: *uvm_reg_item*

15.1.1.175 Class uvm_pkg::uvm_reg_backdoor

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_reg_backdoor*

Class

uvm_reg_backdoor

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

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

Table 192: Variables

Name	Type	Description
fname	string	
lineno	int	

Constructors

`function new(string name = "")`

Function

new

Create an instance of this class

Create an instance of the user-defined backdoor class for the specified register or memory
Parameters
`name` (`string`)

Functions

`virtual function void read_func(uvm_reg_item rw)`

Function

read_func

User-defined backdoor read operation.

Peek the current value in the HDL implementation. Returns the current value and an indication of the success of the operation. Read_func
Parameters
`rw` (*uvm_reg_item*)

`virtual function bit is_auto_updated(uvm_reg_field field)`

Function

is_auto_updated

Indicates if wait_for_change() method is implemented

Implement to return TRUE if and only if wait_for_change() is implemented to watch for changes in the HDL implementation of the specified field. Is_auto_updated
Parameters
`field` (*uvm_reg_field*)

`function void start_update_thread(uvm_object element)`

Start_update_thread

Parameters

`element` (*uvm_object*) -- Local

`function void kill_update_thread(uvm_object element)`

Kill_update_thread

Parameters

`element` (*uvm_object*) -- Local

`function bit has_update_threads()`

Has_update_threadslocal

Tasks

`virtual function write(uvm_reg_item rw)`

Task

write

User-defined backdoor write operation.

Call do_pre_write(). Deposit the specified value in the specified register HDL implementation. Call do_post_write(). Returns an indication of the success of the operation. Write

Parameters

`rw` (*uvm_reg_item*)

`virtual function read(uvm_reg_item rw)`

Task

read

User-defined backdoor read operation.

Overload this method only if the backdoor requires the use of task.

Call do_pre_read(). Peek the current value of the specified HDL implementation. Call do_post_read(). Returns the current value and an indication of the success of the operation.

By default, calls *read_func()*. Read

Parameters

`rw` (*uvm_reg_item*)

`virtual function pre_read(uvm_reg_item rw)`

Task

pre_read

Called before user-defined backdoor register read.

The registered callback methods are invoked after the invocation of this method.

Parameters

`rw` (*uvm_reg_item*)

`virtual function post_read(uvm_reg_item rw)`

Task

post_read

Called after user-defined backdoor register read.

The registered callback methods are invoked before the invocation of this method.

Parameters

`rw` (*uvm_reg_item*)

`virtual function pre_write(uvm_reg_item rw)`

Task

pre_write

Called before user-defined backdoor register write.

The registered callback methods are invoked after the invocation of this method.

The written value, if modified, modifies the actual value that will be written.

Parameters

`rw` (*uvm_reg_item*)

```
virtual function post_write(uvm_reg_item rw)
```

Task

post_write

Called after user-defined backdoor register write.

The registered callback methods are invoked before the invocation of this method.

Parameters

`rw` (*uvm_reg_item*)

15.1.1.176 Class uvm_pkg::uvm_reg_bit_bash_seq

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*
↪*uvm_pkg* :: *uvm_reg_sequence*
↪*uvm_pkg* :: *uvm_reg_bit_bash_seq*

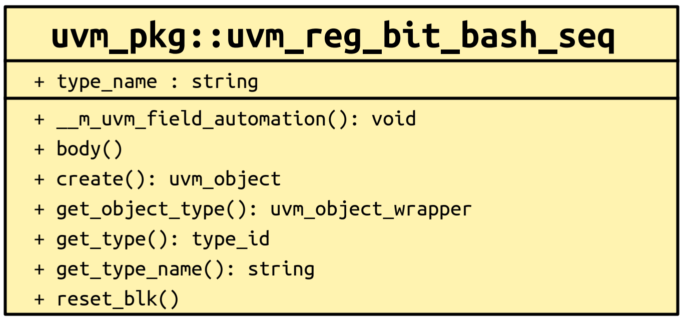

uvm_pkg::uvm_reg_bit_bash_seq
+ type_name : string
+ __m_uvm_field_automation(): void
+ body()
+ create(): uvm_object
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string
+ reset_blk()

Fig. 57: Collaboration Diagram of uvm_reg_bit_bash_seq

Class

uvm_reg_bit_bash_seq

Verify the implementation of all registers in a block by executing the *uvm_reg_single_bit_bash_seq* sequence on it.

If bit-type resource named "NO_REG_TESTS" or "NO_REG_BIT_BASH_TEST" in the "REG::" namespace matches the full name of the block, the block is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.get_full_name(),".*"},
                          "NO_REG_TESTS", 1, this);
```

Constructors

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

Parameters
name (*string*)

Tasks

`virtual function body()`

Task

body

Executes the Register Bit Bash sequence. Do not call directly. Use seq.start() instead.

`virtual function reset_blk(uvm_reg_block blk)`

Task

reset_blk

Reset the DUT that corresponds to the specified block abstraction class.

Currently empty. Will rollback the environment's phase to the *reset* phase once the new phasing is available.

In the meantime, the DUT should be reset before executing this test sequence or this method should be implemented in an extension to reset the DUT.

Parameters

`blk` (*uvm_reg_block*)

15.1.1.177 Class uvm_pkg::uvm_reg_block

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_reg_block*

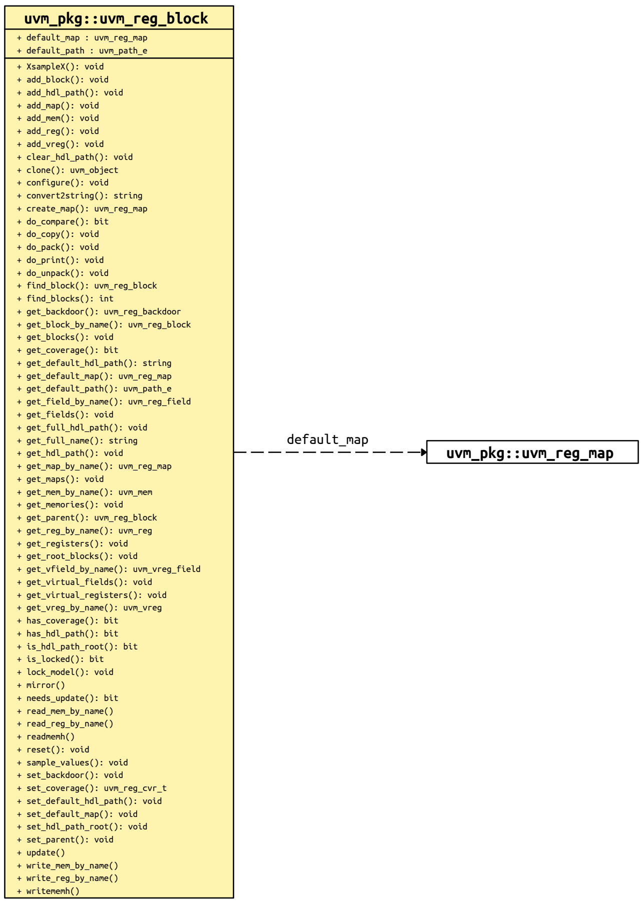


Fig. 58: Collaboration Diagram of uvm_reg_block

Class

uvm_reg_block

Block abstraction base class

A block represents a design hierarchy. It can contain registers, register files, memories and sub-blocks.

A block has one or more address maps, each corresponding to a physical interface on the block.

Table 193: Variables

Name	Type	Description
default_path	*uvm_path_e*	***Variable*** default_path Default access path for the registers and memories in this block.
default_map	*uvm_reg_map*	***Variable*** default_map Default address map Default address map for this block, to be used when no address map is specified for a register operation and that register is accessible from more than one address map. It is also the implicit address map for a block with a single, unnamed address map because it has only one physical interface.

Constructors

`function new(string name = "", int has_coverage = UVM_NO_COVERAGE)`

Function

new

Create a new instance and type-specific configuration

Creates an instance of a block abstraction class with the specified name.

has_coverage specifies which functional coverage models are present in the extension of the block abstraction class. Multiple functional coverage models may be specified by adding their symbolic names, as defined by the *uvm_coverage_model_e* type. New

Parameters

`name` (*`string`*)
`has_coverage` (*`int`*)

Functions

`function void configure(uvm_reg_block parent = null, string hdl_path = "")`

Function

configure

Instance-specific configuration

Specify the parent block of this block. A block without parent is a root block.

If the block file corresponds to a hierarchical RTL structure, its contribution to the HDL path is specified as the *hdl_path* . Otherwise, the block does not correspond to a hierarchical RTL structure (e.g. it is physically flattened) and does not contribute to the hierarchical HDL path of any contained registers or memories. Configure

Parameters

`parent` (*uvm_reg_block*)
`hdl_path` (*`string`*)

`virtual function uvm_reg_map create_map(string name, uvm_reg_addr_t base_addr, int unsigned n_bytes, uvm_endianness_e endian, bit byte_addressing = 1)`

Function

create_map

Create an address map in this block

Create an address map with the specified *name* , then configures it with the following properties.

base_addr

the base address for the map. All registers, memories, and sub-blocks within the map will be at offsets to this address

n_bytes

the byte-width of the bus on which this map is used

endian

the endian format. See *uvm_endianness_e* for possible values

byte_addressing

specifies whether consecutive addresses refer are 1 byte apart (TRUE) or *n_bytes* apart (FALSE). Default is TRUE.

```
APB = create_map("APB", 0, 1, UVM_LITTLE_ENDIAN, 1);. Create_map
```

Parameters
- **name** (*string*)
- **base_addr** (*uvm_reg_addr_t*)
- **n_bytes** (*int unsigned*)
- **endian** (*uvm_endianness_e*)
- **byte_addressing** (*bit*)

Return type
uvm_reg_map

function void set_default_map(uvm_reg_map map)

Function

set_default_map

Defines the default address map

Set the specified address map as the *default_map* for this block. The address map must be a map of this address block. Set_default_map

Parameters
map (*uvm_reg_map*)

function uvm_reg_map get_default_map()

Get_default_map

Return type
uvm_reg_map

virtual function void set_parent(uvm_reg_block parent)

Set_parent

Parameters
parent (*uvm_reg_block*)

function void add_block(uvm_reg_block blk)

Add_block

Parameters
blk (*uvm_reg_block*) -- Local

function void add_map(uvm_reg_map map)

Add_map

Parameters
map (*uvm_reg_map*) -- Local

function void add_reg(uvm_reg rg)

Add_reg

Parameters
rg (*uvm_reg*) -- Local

function void add_vreg(uvm_vreg vreg)

Add_vreg

Parameters
vreg (*uvm_vreg*) -- Local

function void add_mem(uvm_mem mem)

Add_mem
Parameters
mem (*uvm_mem*) -- Local

virtual function void lock_model()

Function

lock_model

Lock a model and build the address map.

Recursively lock an entire register model and build the address maps to enable the *uvm_reg_map::get_reg_by_offset()* and *uvm_reg_map::get_mem_by_offset()* methods.

Once locked, no further structural changes, such as adding registers or memories, can be made.

It is not possible to unlock a model. Lock_model

function bit is_locked()

Function

is_locked

Return TRUE if the model is locked. Is_locked

virtual function string get_full_name()

Function

get_full_name

Get the hierarchical name

Return the hierarchal name of this block. The base of the hierarchical name is the root block. Get Hierarchical Elements

virtual function uvm_reg_block get_parent()

Function

get_parent

Get the parent block

If this a top-level block, returns *null* . Get_parent
Return type
uvm_reg_block

static function void get_root_blocks(uvm_reg_block blks)

Function

get_root_blocks

Get the all root blocks

Returns an array of all root blocks in the simulation. Get_root_blocks
Parameters
blks (*uvm_reg_block*)

static function int find_blocks(string name, uvm_reg_block blks, uvm_reg_-block root = null, uvm_object accessor = null)

Function

find_blocks

Find the blocks whose hierarchical names match the specified *name* glob. If a *root* block is specified, the name of the blocks are relative to that block, otherwise they are absolute.

Returns the number of blocks found. Find_blocks
Parameters
name (`string`)
blks (*uvm_reg_block*)
root (*uvm_reg_block*)
accessor (*uvm_object*)

static function uvm_reg_block find_block(string name, uvm_reg_block root = null, uvm_object accessor = null)

Function

find_block

Find the first block whose hierarchical names match the specified *name* glob. If a *root* block is specified, the name of the blocks are relative to that block, otherwise they are absolute.

Returns the first block found or *null* otherwise. A warning is issued if more than one block is found. Find_blocks

Parameters
- **name** (*string*)
- **root** (*uvm_reg_block*)
- **accessor** (*uvm_object*)

Return type
uvm_reg_block

virtual function void get_blocks(uvm_reg_block blks, uvm_hier_e hier = UVM_HIER)

Function

get_blocks

Get the sub-blocks

Get the blocks instantiated in this blocks. If *hier* is TRUE, recursively includes any sub-blocks. Get_blocks

Parameters
- **blks** (*uvm_reg_block*)
- **hier** (*uvm_hier_e*)

virtual function void get_maps(uvm_reg_map maps)

Function

get_maps

Get the address maps

Get the address maps instantiated in this block. Get_maps

Parameters
maps (*uvm_reg_map*)

virtual function void get_registers(uvm_reg regs, uvm_hier_e hier = UVM_HIER)

Function

get_registers

Get the registers

Get the registers instantiated in this block. If *hier* is TRUE, recursively includes the registers in the sub-blocks.

Note that registers may be located in different and/or multiple address maps. To get the registers in a specific address map, use the *uvm_reg_map::get_registers()* method. Get_registers

Parameters
- **regs** (*uvm_reg*)
- **hier** (*uvm_hier_e*)

virtual function void get_fields(uvm_reg_field fields, uvm_hier_e hier = UVM_HIER)

Function

get_fields

Get the fields

Get the fields in the registers instantiated in this block. If *hier* is TRUE, recursively includes the fields of the registers in the sub-blocks. Get_fields

Parameters
- **fields** (*uvm_reg_field*)
- **hier** (*uvm_hier_e*)

`virtual function void get_memories(uvm_mem mems, uvm_hier_e hier = UVM_HIER)`

Function

get_memories

Get the memories

Get the memories instantiated in this block. If *hier* is TRUE, recursively includes the memories in the sub-blocks.

Note that memories may be located in different and/or multiple address maps. To get the memories in a specific address map, use the *uvm_reg_map::get_memories()* method. Get_memories

Parameters

- **`mems`** (*uvm_mem*)
- **`hier`** (*uvm_hier_e*)

`virtual function void get_virtual_registers(uvm_vreg regs, uvm_hier_e hier = UVM_-HIER)`

Function

get_virtual_registers

Get the virtual registers

Get the virtual registers instantiated in this block. If *hier* is TRUE, recursively includes the virtual registers in the sub-blocks. Get_virtual_registers

Parameters

- **`regs`** (*uvm_vreg*)
- **`hier`** (*uvm_hier_e*)

`virtual function void get_virtual_fields(uvm_vreg_field fields, uvm_hier_-e hier = UVM_HIER)`

Function

get_virtual_fields

Get the virtual fields

Get the virtual fields from the virtual registers instantiated in this block. If *hier* is TRUE, recursively includes the virtual fields in the virtual registers in the sub-blocks. Get_virtual_fields

Parameters

- **`fields`** (*uvm_vreg_field*)
- **`hier`** (*uvm_hier_e*)

`virtual function uvm_reg_block get_block_by_name(string name)`

Function

get_block_by_name

Finds a sub-block with the specified simple name.

The name is the simple name of the block, not a hierarchical name. relative to this block. If no block with that name is found in this block, the sub-blocks are searched for a block of that name and the first one to be found is returned.

If no blocks are found, returns *null* . Get_block_by_name

Parameters

- **`name`** (*`string`*)

Return type

uvm_reg_block

`virtual function uvm_reg_map get_map_by_name(string name)`

Function

get_map_by_name

Finds an address map with the specified simple name.

The name is the simple name of the address map, not a hierarchical name. relative to this block. If no map with that name is found in this block, the sub-blocks are searched for a map of that name and the first one to be found is returned.

If no address maps are found, returns *null* . Get_map_by_name

Parameters
: **name** (*string*)

Return type
: *uvm_reg_map*

`virtual function uvm_reg get_reg_by_name(string name)`

Function

get_reg_by_name

Finds a register with the specified simple name.

The name is the simple name of the register, not a hierarchical name. relative to this block. If no register with that name is found in this block, the sub-blocks are searched for a register of that name and the first one to be found is returned.

If no registers are found, returns *null* . Get_reg_by_name

Parameters
: **name** (*string*)

Return type
: *uvm_reg*

`virtual function uvm_reg_field get_field_by_name(string name)`

Function

get_field_by_name

Finds a field with the specified simple name.

The name is the simple name of the field, not a hierarchical name. relative to this block. If no field with that name is found in this block, the sub-blocks are searched for a field of that name and the first one to be found is returned.

If no fields are found, returns *null* . Get_field_by_name

Parameters
: **name** (*string*)

Return type
: *uvm_reg_field*

`virtual function uvm_mem get_mem_by_name(string name)`

Function

get_mem_by_name

Finds a memory with the specified simple name.

The name is the simple name of the memory, not a hierarchical name. relative to this block. If no memory with that name is found in this block, the sub-blocks are searched for a memory of that name and the first one to be found is returned.

If no memories are found, returns *null* . Get_mem_by_name

Parameters
: **name** (*string*)

Return type
: *uvm_mem*

`virtual function uvm_vreg get_vreg_by_name(string name)`

Function

get_vreg_by_name

Finds a virtual register with the specified simple name.

The name is the simple name of the virtual register, not a hierarchical name. relative to this block. If no virtual register with that name is found in this block, the sub-blocks are searched for a virtual register of that name and the first one to be found is returned.

If no virtual registers are found, returns *null* . Get_vreg_by_name

Parameters
name (*string*)
Return type
uvm_vreg

virtual function uvm_vreg_field get_vfield_by_name(string name)

Function

get_vfield_by_name

Finds a virtual field with the specified simple name.

The name is the simple name of the virtual field, not a hierarchical name. relative to this block. If no virtual field with that name is found in this block, the sub-blocks are searched for a virtual field of that name and the first one to be found is returned.

If no virtual fields are found, returns *null* . Get_vfield_by_name
Parameters
name (*string*)
Return type
uvm_vreg_field

virtual function bit has_coverage(uvm_reg_cvr_t models)

Function

has_coverage

Check if block has coverage model(s)

Returns TRUE if the block abstraction class contains a coverage model for all of the models specified. Models are specified by adding the symbolic value of individual coverage model as defined in *uvm_coverage_model_e*. Has_coverage
Parameters
models (*uvm_reg_cvr_t*)

virtual function uvm_reg_cvr_t set_coverage(uvm_reg_cvr_t is_on)

Function

set_coverage

Turns on coverage measurement.

Turns the collection of functional coverage measurements on or off for this block and all blocks, registers, fields and memories within it. The functional coverage measurement is turned on for every coverage model specified using *uvm_coverage_model_e* symbolic identifiers. Multiple functional coverage models can be specified by adding the functional coverage model identifiers. All other functional coverage models are turned off. Returns the sum of all functional coverage models whose measurements were previously on.

This method can only control the measurement of functional coverage models that are present in the various abstraction classes, then enabled during construction. See the *uvm_reg_block::has_coverage()* method to identify the available functional coverage models. Set_coverage
Parameters
is_on (*uvm_reg_cvr_t*)
Return type
uvm_reg_cvr_t

virtual function bit get_coverage(uvm_reg_cvr_t is_on = UVM_CVR_ALL)

Function

get_coverage

Check if coverage measurement is on.

Returns TRUE if measurement for all of the specified functional coverage models are currently on. Multiple functional coverage models can be specified by adding the functional coverage model identifiers.

See *uvm_reg_block::set_coverage()* for more details. Get_coverage
Parameters
is_on (*uvm_reg_cvr_t*)

`virtual function void sample_values()`

Function

sample_values

Functional coverage measurement method for field values

This method is invoked by the user or by the *uvm_reg_block::sample_values()* method of the parent block to trigger the sampling of the current field values in the block-level functional coverage model. It recursively invokes the *uvm_reg_block::sample_values()* and *uvm_reg::sample_values()* methods in the blocks and registers in this block.

This method may be extended by the abstraction class generator to perform the required sampling in any provided field-value functional coverage model. If this method is extended, it MUST call super.sample_values(). Sample_values

`function void XsampleX(uvm_reg_addr_t addr, bit is_read, uvm_reg_map map)`

XsampleX

Parameters

- **`addr`** (*uvm_reg_addr_t*) -- Local
- **`is_read`** (`bit`)
- **`map`** (*uvm_reg_map*)

`virtual function uvm_path_e get_default_path()`

Function

get_default_path

Default access path

Returns the default access path for this block. Get_default_path

Return type

uvm_path_e

`virtual function void reset(string kind = "HARD")`

Function

reset

Reset the mirror for this block.

Sets the mirror value of all registers in the block and sub-blocks to the reset value corresponding to the specified reset event. See *uvm_reg_field::reset()* for more details. Does not actually set the value of the registers in the design, only the values mirrored in their corresponding mirror. Reset

Parameters

`kind` (`string`)

`virtual function bit needs_update()`

Function

needs_update

Check if DUT registers need to be written

If a mirror value has been modified in the abstraction model without actually updating the actual register (either through randomization or via the *uvm_reg::set()* method, the mirror and state of the registers are outdated. The corresponding registers in the DUT need to be updated.

This method returns TRUE if the state of at least one register in the block or sub-blocks needs to be updated to match the mirrored values. The mirror values, or actual content of registers, are not modified. For additional information, see *uvm_reg_block::update()* method. Needs_update

`function uvm_reg_backdoor get_backdoor(bit inherited = 1)`

Function

get_backdoor

Get the user-defined backdoor for all registers in this block

Return the user-defined backdoor for all register in this block and all sub-blocks -- unless overridden by a backdoor set in a lower-level block or in the register itself.

If *inherited* is TRUE, returns the backdoor of the parent block if none have been specified for this block. Get_backdoor

Parameters

`inherited` (*`bit`*)

Return type

uvm_reg_backdoor

`function void set_backdoor(uvm_reg_backdoor bkdr, string fname = "", int lineno = 0)`

Function

set_backdoor

Set the user-defined backdoor for all registers in this block

Defines the backdoor mechanism for all registers instantiated in this block and sub-blocks, unless overridden by a definition in a lower-level block or register. Set_backdoor

Parameters

- **`bkdr`** (*uvm_reg_backdoor*)
- **`fname`** (*`string`*)
- **`lineno`** (*`int`*)

`function void clear_hdl_path(string kind = "RTL")`

Function

clear_hdl_path

Delete HDL paths

Remove any previously specified HDL path to the block instance for the specified design abstraction. Clear_hdl_path

Parameters

`kind` (*`string`*)

`function void add_hdl_path(string path, string kind = "RTL")`

Function

add_hdl_path

Add an HDL path

Add the specified HDL path to the block instance for the specified design abstraction. This method may be called more than once for the same design abstraction if the block is physically duplicated in the design abstraction. Add_hdl_path

Parameters

- **`path`** (*`string`*)
- **`kind`** (*`string`*)

`function bit has_hdl_path(string kind = "")`

Function

has_hdl_path

Check if a HDL path is specified

Returns TRUE if the block instance has a HDL path defined for the specified design abstraction. If no design abstraction is specified, uses the default design abstraction specified for this block or the nearest block ancestor with a specified default design abstraction. Has_hdl_path

Parameters

`kind` (*`string`*)

`function void get_hdl_path(string paths, string kind = "")`

Function

get_hdl_path

Get the incremental HDL path(s)

Returns the HDL path(s) defined for the specified design abstraction in the block instance. Returns only the component of the HDL paths that corresponds to the block, not a full hierarchical path

If no design abstraction is specified, the default design abstraction for this block is used. Get_hdl_path

Parameters
  paths (*string*)
  kind (*string*)

function void get_full_hdl_path(string paths, string kind = "", string separator = ".")

Function

get_full_hdl_path

Get the full hierarchical HDL path(s)

Returns the full hierarchical HDL path(s) defined for the specified design abstraction in the block instance. There may be more than one path returned even if only one path was defined for the block instance, if any of the parent components have more than one path defined for the same design abstraction

If no design abstraction is specified, the default design abstraction for each ancestor block is used to get each incremental path. Get_full_hdl_path

Parameters
  paths (*string*)
  kind (*string*)
  separator (*string*)

function void set_default_hdl_path(string kind)

Function

set_default_hdl_path

Set the default design abstraction

Set the default design abstraction for this block instance. Set_default_hdl_path

Parameters
  kind (*string*)

function string get_default_hdl_path()

Function

get_default_hdl_path

Get the default design abstraction

Returns the default design abstraction for this block instance. If a default design abstraction has not been explicitly set for this block instance, returns the default design abstraction for the nearest block ancestor. Returns "" if no default design abstraction has been specified. Get_default_hdl_path

function void set_hdl_path_root(string path, string kind = "RTL")

Function

set_hdl_path_root

Specify a root HDL path

Set the specified path as the absolute HDL path to the block instance for the specified design abstraction. This absolute root path is prepended to all hierarchical paths under this block. The HDL path of any ancestor block is ignored. This method overrides any incremental path for the same design abstraction specified using *add_hdl_path*. Set_hdl_path_root

Parameters
  path (*string*)
  kind (*string*)

function bit is_hdl_path_root(string kind = "")

Function

is_hdl_path_root

Check if this block has an absolute path

Returns TRUE if an absolute HDL path to the block instance for the specified design abstraction has been defined. If no design abstraction is specified, the default design abstraction for this block is used. Is_hdl_path_root

Parameters
  kind (*string*)

`virtual function void do_print(uvm_printer printer)`

Do_print

Parameters

`printer` (*uvm_printer*)

`virtual function void do_copy(uvm_object rhs)`

Do_copy

Parameters

`rhs` (*uvm_object*)

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

Do_compare

Parameters

`rhs` (*uvm_object*)

`comparer` (*uvm_comparer*)

`virtual function void do_pack(uvm_packer packer)`

Do_pack

Parameters

`packer` (*uvm_packer*)

`virtual function void do_unpack(uvm_packer packer)`

Do_unpack

Parameters

`packer` (*uvm_packer*)

`virtual function string convert2string()`

Convert2string

`virtual function uvm_object clone()`

Clone

Return type

uvm_object

Tasks

`virtual function update(uvm_status_e status, uvm_path_e path = UVM_DEFAULT_PATH, uvm_sequence_base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)`

Task

update

Batch update of register.

Using the minimum number of write operations, updates the registers in the design to match the mirrored values in this block and sub-blocks. The update can be performed using the physical interfaces (front-door access) or back-door accesses. This method performs the reverse operation of *uvm_reg_block::mirror()*. Update

Parameters

`status` (*uvm_status_e*)

`path` (*uvm_path_e*)

`parent` (*uvm_sequence_base*)

`prior` (`int`)

`extension` (*uvm_object*)

`fname` (`string`)

`lineno` (`int`)

`virtual function mirror(uvm_status_e status, uvm_check_e check = UVM_NO_CHECK, uvm_path_e path = UVM_DEFAULT_PATH, uvm_sequence_base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)`

Task

mirror

Update the mirrored values

Read all of the registers in this block and sub-blocks and update their mirror values to match their corresponding values in the design. The mirroring can be performed using the physical interfaces (front-door access) or back-door accesses. If the *check* argument is specified as <UVM_CHECK>, an error message is issued if the current mirrored value does not match the actual value in the design. This method performs the reverse operation of *uvm_reg_block::update()*. Mirror

Parameters

- **status** (*uvm_status_e*)
- **check** (*uvm_check_e*)
- **path** (*uvm_path_e*)
- **parent** (*uvm_sequence_base*)
- **prior** (`int`)
- **extension** (*uvm_object*)
- **fname** (`string`)
- **lineno** (`int`)

`virtual function write_reg_by_name(uvm_status_e status, string name, uvm_reg_data_t data, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)`

Task

write_reg_by_name

Write the named register

Equivalent to *get_reg_by_name()* followed by *uvm_reg::write()*. Write_reg_by_name

Parameters

- **status** (*uvm_status_e*)
- **name** (`string`)
- **data** (*uvm_reg_data_t*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **parent** (*uvm_sequence_base*)
- **prior** (`int`)
- **extension** (*uvm_object*)
- **fname** (`string`)
- **lineno** (`int`)

`virtual function read_reg_by_name(uvm_status_e status, string name, uvm_reg_data_t data, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)`

Task

read_reg_by_name

Read the named register

Equivalent to *get_reg_by_name()* followed by *uvm_reg::read()*. Read_reg_by_name

Parameters

- **status** (*uvm_status_e*)
- **name** (`string`)
- **data** (*uvm_reg_data_t*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **parent** (*uvm_sequence_base*)
- **prior** (`int`)
- **extension** (*uvm_object*)
- **fname** (`string`)
- **lineno** (`int`)

`virtual function write_mem_by_name(uvm_status_e status, string name, uvm_reg_addr_t offset, uvm_reg_data_t data, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)`

Task

write_mem_by_name

Write the named memory

Equivalent to *get_mem_by_name()* followed by *uvm_mem::write()*. Write_mem_by_name
Parameters
- **status** (*uvm_status_e*)
- **name** (*string*)
- **offset** (*uvm_reg_addr_t*)
- **data** (*uvm_reg_data_t*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **parent** (*uvm_sequence_base*)
- **prior** (*int*)
- **extension** (*uvm_object*)
- **fname** (*string*)
- **lineno** (*int*)

virtual function read_mem_by_name(uvm_status_e status, string name, uvm_reg_addr_t offset, uvm_reg_data_t data, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)

Task

read_mem_by_name

Read the named memory

Equivalent to *get_mem_by_name()* followed by *uvm_mem::read()*. Read_mem_by_name
Parameters
- **status** (*uvm_status_e*)
- **name** (*string*)
- **offset** (*uvm_reg_addr_t*)
- **data** (*uvm_reg_data_t*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **parent** (*uvm_sequence_base*)
- **prior** (*int*)
- **extension** (*uvm_object*)
- **fname** (*string*)
- **lineno** (*int*)

virtual function readmemh(string filename)

Readmemh
Parameters
- **filename** (*string*)

virtual function writememh(string filename)

Writememh
Parameters
- **filename** (*string*)

15.1.1.178 Class uvm_pkg::uvm_reg_cbs

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_callback*
↪*uvm_pkg* :: *uvm_reg_cbs*

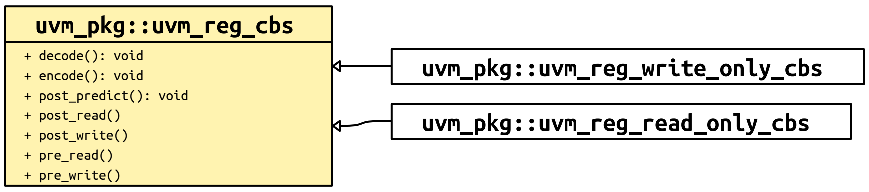


Fig. 59: Inheritance Diagram of uvm_reg_cbs

Class

uvm_reg_cbs

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

Constructors

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

Parameters
`name` (*`string`*)

Functions

`virtual function void post_predict(uvm_reg_field fld, uvm_reg_data_t previous, uvm_reg_data_t value, uvm_predict_e kind, uvm_path_e path, uvm_reg_map map)`

Task

post_predict

Called by the *uvm_reg_field::predict()* method after a successful UVM_PREDICT_READ or UVM_PREDICT_WRITE prediction.

previous is the previous value in the mirror and *value* is the latest predicted value. Any change to *value* will modify the predicted mirror value.

Parameters
`fld` (*uvm_reg_field*)
`previous` (*uvm_reg_data_t*)
`value` (*uvm_reg_data_t*)
`kind` (*uvm_predict_e*)
`path` (*uvm_path_e*)
`map` (*uvm_reg_map*)

`virtual function void encode(uvm_reg_data_t data)`

Function

encode

Data encoder

The registered callback methods are invoked in order of registration after all the *pre_write* methods have been called. The encoded data is passed through each invocation in sequence. This allows the *pre_write* methods to deal with clear-text data.

By default, the data is not modified.

Parameters

`data` (*uvm_reg_data_t*)

`virtual function void decode(uvm_reg_data_t data)`

Function

decode

Data decode

The registered callback methods are invoked in *reverse order* of registration before all the *post_read* methods are called. The decoded data is passed through each invocation in sequence. This allows the *post_read* methods to deal with clear-text data.

The reversal of the invocation order is to allow the decoding of the data to be performed in the opposite order of the encoding with both operations specified in the same callback extension.

By default, the data is not modified.

Parameters

`data` (*uvm_reg_data_t*)

Tasks

`virtual function pre_write(uvm_reg_item rw)`

Task

pre_write

Called before a write operation.

All registered *pre_write* callback methods are invoked after the invocation of the *pre_write* method of associated object (*uvm_reg*, *uvm_reg_field*, *uvm_mem*, or *uvm_reg_backdoor*). If the element being written is a *uvm_reg*, all *pre_write* callback methods are invoked before the contained *uvm_reg_fields*.

Backdoor

uvm_reg_backdoor::pre_write, *uvm_reg_cbs::pre_write* cbs for backdoor.

Register

uvm_reg::pre_write, *uvm_reg_cbs::pre_write* cbs for reg, then foreach field: *uvm_reg_field::pre_write*, *uvm_reg_cbs::pre_write* cbs for field

RegField

uvm_reg_field::pre_write, *uvm_reg_cbs::pre_write* cbs for field

Memory

uvm_mem::pre_write, *uvm_reg_cbs::pre_write* cbs for mem

The *rw* argument holds information about the operation.

Modifying the *value* modifies the actual value written.
For memories, modifying the *offset* modifies the offset used in the operation.
For non-backdoor operations, modifying the access *path* or address *map* modifies the actual path or map used in the

operation.

If the *rw.status* is modified to anything other than <UVM_IS_OK>, the operation is aborted.

See *uvm_reg_item* for details on *rw* information.

Parameters

`rw` (*uvm_reg_item*)

`virtual function post_write(uvm_reg_item rw)`

Task

post_write

Called after a write operation.

All registered *post_write* callback methods are invoked before the invocation of the *post_write* method of the associated object (*uvm_reg*, *uvm_reg_field*, *uvm_mem*, or *uvm_reg_backdoor*). If the element being written is a *uvm_reg*, all *post_write* callback methods are invoked before the contained *uvm_reg_fields*.

Summary of callback order:

Backdoor

uvm_reg_cbs::post_write cbs for backdoor, *uvm_reg_backdoor::post_write*

Register

uvm_reg_cbs::post_write cbs for reg, *uvm_reg::post_write*, then foreach field: *uvm_reg_cbs::post_write* cbs for field, *uvm_reg_field::post_read*

RegField

uvm_reg_cbs::post_write cbs for field, *uvm_reg_field::post_write*

Memory

uvm_reg_cbs::post_write cbs for mem, *uvm_mem::post_write*

The *rw* argument holds information about the operation.

Modifying the *status* member modifies the returned status.
Modifying the *value* or *offset* members has no effect, as the operation has already completed.

See *uvm_reg_item* for details on *rw* information.

Parameters

rw (*uvm_reg_item*)

```
virtual function pre_read(uvm_reg_item rw)
```

Task

pre_read

Callback called before a read operation.

All registered *pre_read* callback methods are invoked after the invocation of the *pre_read* method of associated object (*uvm_reg*, *uvm_reg_field*, *uvm_mem*, or *uvm_reg_backdoor*). If the element being read is a *uvm_reg*, all *pre_read* callback methods are invoked before the contained *uvm_reg_fields*.

Backdoor

uvm_reg_backdoor::pre_read, *uvm_reg_cbs::pre_read* cbs for backdoor

Register

uvm_reg::pre_read, *uvm_reg_cbs::pre_read* cbs for reg, then foreach field: *uvm_reg_field::pre_read*, *uvm_reg_cbs::pre_read* cbs for field

RegField

uvm_reg_field::pre_read, *uvm_reg_cbs::pre_read* cbs for field

Memory

uvm_mem::pre_read, *uvm_reg_cbs::pre_read* cbs for mem

The *rw* argument holds information about the operation.

The *value* member of *rw* is not used has no effect if modified.
For memories, modifying the *offset* modifies the offset used in the operation.
For non-backdoor operations, modifying the access *path* or address *map* modifies the actual path or map used in the

operation.

If the *rw.status* is modified to anything other than <UVM_IS_OK>, the operation is aborted.

See *uvm_reg_item* for details on *rw* information.

Parameters

rw (*uvm_reg_item*)

`virtual function post_read(uvm_reg_item rw)`

Task

post_read

Callback called after a read operation.

All registered *post_read* callback methods are invoked before the invocation of the *post_read* method of the associated object (*uvm_reg*, *uvm_reg_field*, *uvm_mem*, or *uvm_reg_backdoor*). If the element being read is a *uvm_reg*, all *post_read* callback methods are invoked before the contained *uvm_reg_fields*.

Backdoor

uvm_reg_cbs::post_read cbs for backdoor, *uvm_reg_backdoor::post_read*

Register

uvm_reg_cbs::post_read cbs for reg, *uvm_reg::post_read*, then foreach field: *uvm_reg_cbs::post_read* cbs for field, *uvm_reg_field::post_read*

RegField

uvm_reg_cbs::post_read cbs for field, *uvm_reg_field::post_read*

Memory

uvm_reg_cbs::post_read cbs for mem, *uvm_mem::post_read*

The *rw* argument holds information about the operation.

Modifying the readback *value* or *status* modifies the actual returned value and status.
Modifying the *value* or *offset* members has no effect, as the operation has already completed.

See *uvm_reg_item* for details on *rw* information.

Parameters

`rw` (*uvm_reg_item*)

15.1.1.179 Class uvm_pkg::uvm_reg_field

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_reg_field*

CLASS

uvm_reg_field

Field abstraction class

A field represents a set of bits that behave consistently as a single entity.

A field is contained within a single register, but may have different access policies depending on the address map use the access the register (thus the field).

Table 194: Variables

Name	Type	Description
value	*uvm_reg_data_t*	***Variable*** value Mirrored field value. This value can be sampled in a functional coverage model or constrained when randomized. Mirrored after randomize()

Table 195: Constraints

Name	Description
uvm_reg_field_valid	

Constructors

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

Function

new

Create a new field instance

This method should not be used directly. The *uvm_reg_field::type_id::create()* factory method should be used instead. New

Parameters

name (`string`)

Functions

```
function void configure(uvm_reg parent, int unsigned size, int unsigned lsb_pos,
string access, bit volatile, uvm_reg_data_t reset, bit has_reset, bit is_rand,
bit individually_accessible)
```

Function

configure

Instance-specific configuration

Specify the *parent* register of this field, its *size* in bits, the position of its least-significant bit within the register relative to the least-significant bit of the register, its *access* policy, volatility, "HARD" *reset* value, whether the field value is actually reset (the *reset* value is ignored if *FALSE*), whether the field value may be randomized and whether the field is the only one to occupy a byte lane in the register.

See *set_access* for a specification of the pre-defined field access policies.

If the field access policy is a pre-defined policy and NOT one of "RW", "WRC", "WRS", "WO", "W1", or "WO1", the value of *is_rand* is ignored and the rand_mode() for the field instance is turned off since it cannot be written. Configure

Parameters

- **parent** (*uvm_reg*)
- **size** (*int unsigned*)
- **lsb_pos** (*int unsigned*)
- **access** (*string*)
- **volatile** (*bit*)
- **reset** (*uvm_reg_data_t*)
- **has_reset** (*bit*)
- **is_rand** (*bit*)
- **individually_accessible** (*bit*)

virtual function string get_full_name()

Function

get_full_name

Get the hierarchical name

Return the hierarchal name of this field The base of the hierarchical name is the root block. Get_full_name

virtual function uvm_reg get_parent()

Function

get_parent

Get the parent register. Get_parent

Return type

uvm_reg

virtual function uvm_reg get_register()

Get_register

Return type

uvm_reg

virtual function int unsigned get_lsb_pos()

Function

get_lsb_pos

Return the position of the field

Returns the index of the least significant bit of the field in the register that instantiates it. An offset of 0 indicates a field that is aligned with the least-significant bit of the register. Get_lsb_pos

virtual function int unsigned get_n_bits()

Function

get_n_bits

Returns the width, in number of bits, of the field. Get_n_bits

static function int unsigned get_max_size()

FUNCTION

get_max_size

Returns the width, in number of bits, of the largest field. Get_max_size

virtual function string set_access(string mode)

Function

set_access

Modify the access policy of the field

Modify the access policy of the field to the specified one and return the previous access policy.

The pre-defined access policies are as follows. The effect of a read operation are applied after the current value of the field is sampled. The read operation will return the current value, not the value affected by the read operation (if any).

"RO"

W: no effect, R: no effect

"RW"

W: as-is, R: no effect

"RC"

W: no effect, R: clears all bits

"RS"

W: no effect, R: sets all bits

"WRC"

W: as-is, R: clears all bits

"WRS"

W: as-is, R: sets all bits

"WC"

W: clears all bits, R: no effect

"WS"

W: sets all bits, R: no effect

"WSRC"

W: sets all bits, R: clears all bits

"WCRS"

W: clears all bits, R: sets all bits

"W1C"

W: 1/0 clears/no effect on matching bit, R: no effect

"W1S"

W: 1/0 sets/no effect on matching bit, R: no effect

"W1T"

W: 1/0 toggles/no effect on matching bit, R: no effect

"W0C"

W: 1/0 no effect on/clears matching bit, R: no effect

"W0S"

W: 1/0 no effect on/sets matching bit, R: no effect

"W0T"

W: 1/0 no effect on/toggles matching bit, R: no effect

"W1SRC"

W: 1/0 sets/no effect on matching bit, R: clears all bits

"W1CRS"

W: 1/0 clears/no effect on matching bit, R: sets all bits

"W0SRC"

W: 1/0 no effect on/sets matching bit, R: clears all bits

"W0CRS"

W: 1/0 no effect on/clears matching bit, R: sets all bits

"WO"

W: as-is, R: error

"WOC"

W: clears all bits, R: error

"WOS"

W: sets all bits, R: error

"W1"

W: first one after *HARD* reset is as-is, other W have no effects, R: no effect

"WO1"

W: first one after *HARD* reset is as-is, other W have no effects, R: error

"NOACCESS"

W: no effect, R: no effect

It is important to remember that modifying the access of a field will make the register model diverge from the specification that was used to create it. Set_access

Parameters

`mode` (*`string`*)

`static  function bit define_access(string name)`

Function

define_access

Define a new access policy value

Because field access policies are specified using string values, there is no way for SystemVerilog to verify if a specific access value is valid or not. To help catch typing errors, user-defined access values must be defined using this method to avoid begin reported as an invalid access policy.

The name of field access policies are always converted to all uppercase.

Returns TRUE if the new access policy was not previously defined. Returns FALSE otherwise but does not issue an error message. Define_access

Parameters

`name` (*`string`*)

`virtual  function string get_access(uvm_reg_map map = null)`

Function

get_access

Get the access policy of the field

Returns the current access policy of the field when written and read through the specified address *map* . If the register containing the field is mapped in multiple address map, an address map must be specified. The access policy of a field from a specific address map may be restricted by the register's access policy in that address map. For example, a RW field may only be writable through one of the address maps and read-only through all of the other maps. If the field access contradicts the map's access value (field access of WO, and map access value of RO, etc), the method's return value is NOACCESS. Get_access

Parameters

`map` (*uvm_reg_map*)

`virtual function bit is_known_access(uvm_reg_map map = null)`

Function

is_known_access

Check if access policy is a built-in one.

Returns TRUE if the current access policy of the field, when written and read through the specified address *map* , is a built-in access policy. Is_known_access

Parameters

`map` (*uvm_reg_map*)

`virtual function void set_volatility(bit volatile)`

Function

set_volatility

Modify the volatility of the field to the specified one.

It is important to remember that modifying the volatility of a field will make the register model diverge from the specification that was used to create it. Set_volatility

Parameters

`volatile` (*`bit`*)

`virtual function bit is_volatile()`

Function

is_volatile

Indicates if the field value is volatile

UVM uses the IEEE 1685-2009 IP-XACT definition of "volatility". If TRUE, the value of the register is not predictable because it may change between consecutive accesses. This typically indicates a field whose value is updated by the DUT. The nature or cause of the change is not specified. If FALSE, the value of the register is not modified between consecutive accesses. Is_volatile

`virtual function void set(uvm_reg_data_t value, string fname = "", int lineno = 0)`

Function

set

Set the desired value for this field

It sets the desired value of the field to the specified *value* modified by the field access policy. It does not actually set the value of the field in the design, only the desired value in the abstraction class. Use the *uvm_reg::update()* method to update the actual register with the desired value or the *uvm_reg_field::write()* method to actually write the field and update its mirrored value.

The final desired value in the mirror is a function of the field access policy and the set value, just like a normal physical write operation to the corresponding bits in the hardware. As such, this method (when eventually followed by a call to *uvm_reg::update()*) is a zero-time functional replacement for the *uvm_reg_field::write()* method. For example, the desired value of a read-only field is not modified by this method and the desired value of a write-once field can only be set if the field has not yet been written to using a physical (for example, front-door) write operation.

Use the *uvm_reg_field::predict()* to modify the mirrored value of the field. Set

Parameters

`value` (*uvm_reg_data_t*)
`fname` (*`string`*)
`lineno` (*`int`*)

`virtual function uvm_reg_data_t get(string fname = "", int lineno = 0)`

Function

get

Return the desired value of the field

It does not actually read the value of the field in the design, only the desired value in the abstraction class. Unless set to a different value using the *uvm_reg_field::set()*, the desired value and the mirrored value are identical.

Use the *uvm_reg_field::read()* or *uvm_reg_field::peek()* method to get the actual field value.

If the field is write-only, the desired/mirrored value is the value last written and assumed to reside in the bits implementing it. Although a physical read operation would something different, the returned value is the actual content. Get

Parameters

fname (*string*)

lineno (*int*)

Return type

uvm_reg_data_t

virtual function uvm_reg_data_t get_mirrored_value(string fname = "", int lineno = 0)

Function

get_mirrored_value

Return the mirrored value of the field

It does not actually read the value of the field in the design, only the mirrored value in the abstraction class.

If the field is write-only, the desired/mirrored value is the value last written and assumed to reside in the bits implementing it. Although a physical read operation would something different, the returned value is the actual content. Get_mirrored_value

Parameters

fname (*string*)

lineno (*int*)

Return type

uvm_reg_data_t

virtual function void reset(string kind = "HARD")

Function

reset

Reset the desired/mirrored value for this field.

It sets the desired and mirror value of the field to the reset event specified by *kind* . If the field does not have a reset value specified for the specified reset *kind* the field is unchanged.

It does not actually reset the value of the field in the design, only the value mirrored in the field abstraction class.

Write-once fields can be modified after a "HARD" reset operation. Reset

Parameters

kind (*string*)

virtual function uvm_reg_data_t get_reset(string kind = "HARD")

Function

get_reset

Get the specified reset value for this field

Return the reset value for this field for the specified reset *kind* . Returns the current field value is no reset value has been specified for the specified reset event. Get_reset

Parameters

kind (*string*)

Return type

uvm_reg_data_t

virtual function bit has_reset(string kind = "HARD", bit delete = 0)

Function

has_reset

Check if the field has a reset value specified

Return TRUE if this field has a reset value specified for the specified reset *kind* . If *delete* is TRUE, removes the reset value, if any. Has_reset

Parameters

kind (*string*)
delete (*bit*)

`virtual function void set_reset(uvm_reg_data_t value, string kind = "HARD")`

Function

set_reset

Specify or modify the reset value for this field

Specify or modify the reset value for this field corresponding to the cause specified by *kind* . Set_reset
Parameters
value (*uvm_reg_data_t*)
kind (*string*)

`virtual function bit needs_update()`

Function

needs_update

Check if the abstract model contains different desired and mirrored values.

If a desired field value has been modified in the abstraction class without actually updating the field in the DUT, the state of the DUT (more specifically what the abstraction class *thinks* the state of the DUT is) is outdated. This method returns TRUE if the state of the field in the DUT needs to be updated to match the desired value. The mirror values or actual content of DUT field are not modified. Use the *uvm_reg::update()* to actually update the DUT field. Needs_update

`function void set_compare(uvm_check_e check = UVM_CHECK)`

Function

set_compare

Sets the compare policy during a mirror update. The field value is checked against its mirror only when both the *check* argument in *uvm_reg_block::mirror*, *uvm_reg::mirror*, or *uvm_reg_field::mirror* and the compare policy for the field is <UVM_CHECK>. Set_compare
Parameters
check (*uvm_check_e*)

`function uvm_check_e get_compare()`

Function

get_compare

Returns the compare policy for this field. Get_compare
Return type
uvm_check_e

`function bit is_indv_accessible(uvm_path_e path, uvm_reg_map local_map)`

Function

is_indv_accessible

Check if this field can be written individually, i.e. without affecting other fields in the containing register. Is_indv_accessible
Parameters
path (*uvm_path_e*)
local_map (*uvm_reg_map*)

`function bit predict(uvm_reg_data_t value, uvm_reg_byte_en_t be = -1, uvm_predict_-e kind = UVM_PREDICT_DIRECT, uvm_path_e path = UVM_FRONTDOOR, uvm_reg_-map map = null, string fname = "", int lineno = 0)`

Function

predict

Update the mirrored and desired value for this field.

Predict the mirror and desired value of the field based on the specified observed *value* on a bus using the specified address *map* .

If *kind* is specified as <UVM_PREDICT_READ>, the value was observed in a read transaction on the specified address *map* or backdoor (if *path* is <UVM_BACKDOOR>). If *kind* is specified as <UVM_PREDICT_WRITE>, the value was observed in a write transaction on the specified address *map* or backdoor (if *path* is <UVM_BACKDOOR>). If *kind* is specified as <UVM_PREDICT_DIRECT>, the value was computed and is updated as-is, without regard to any access policy. For example, the mirrored value of a read-only field is modified by this method if *kind* is specified as <UVM_PREDICT_DIRECT>.

This method does not allow an update of the mirror (or desired) when the register containing this field is busy executing a transaction because the results are unpredictable and indicative of a race condition in the testbench.

Returns TRUE if the prediction was successful. Predict

Parameters

- **value** (*uvm_reg_data_t*)
- **be** (*uvm_reg_byte_en_t*)
- **kind** (*uvm_predict_e*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **fname** (`string`)
- **lineno** (`int`)

`virtual function uvm_reg_data_t XpredictX(uvm_reg_data_t cur_val, uvm_reg_data_t wr_val, uvm_reg_map map)`

Local. XpredictX

Parameters

- **cur_val** (*uvm_reg_data_t*)
- **wr_val** (*uvm_reg_data_t*)
- **map** (*uvm_reg_map*)

Return type

uvm_reg_data_t

`virtual function uvm_reg_data_t XupdateX()`

Local. XupdateX

Return type

uvm_reg_data_t

`function bit Xcheck_accessX(uvm_reg_item rw, uvm_reg_map_info map_info, string caller)`

Local. Xcheck_accessX

Parameters

- **rw** (*uvm_reg_item*)
- **map_info** (*uvm_reg_map_info*)
- **caller** (`string`)

`virtual function void do_predict(uvm_reg_item rw, uvm_predict_e kind = UVM_PREDICT_DIRECT, uvm_reg_byte_en_t be = -1)`

Do_predict

Parameters

- **rw** (*uvm_reg_item*)
- **kind** (*uvm_predict_e*)
- **be** (*uvm_reg_byte_en_t*)

`virtual function void do_print(uvm_printer printer)`

Do_print

Parameters

printer (*uvm_printer*)

`virtual function string convert2string()`

Convert2string

`virtual function uvm_object clone()`

Clone

Return type

uvm_object

`virtual function void do_copy(uvm_object rhs)`

Do_copy

Parameters

`rhs` (*uvm_object*)

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

Do_compare

Parameters

`rhs` (*uvm_object*)
`comparer` (*uvm_comparer*)

`virtual function void do_pack(uvm_packer packer)`

Do_pack

Parameters

`packer` (*uvm_packer*)

`virtual function void do_unpack(uvm_packer packer)`

Do_unpack

Parameters

`packer` (*uvm_packer*)

Tasks

`virtual function write(uvm_status_e status, uvm_reg_data_t value, uvm_path_- e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)`

Task

write

Write the specified value in this field

Write *value* in the DUT field that corresponds to this abstraction class instance using the specified access *path* . If the register containing this field is mapped in more than one address map, an address *map* must be specified if a physical access is used (front-door access). If a back-door access path is used, the effect of writing the field through a physical access is mimicked. For example, read-only bits in the field will not be written.

The mirrored value will be updated using the *uvm_reg_field::predict()* method.

If a front-door access is used, and if the field is the only field in a byte lane and if the physical interface corresponding to the address map used to access the field support byte-enabling, then only the field is written. Otherwise, the entire register containing the field is written, and the mirrored values of the other fields in the same register are used in a best-effort not to modify their value.

If a backdoor access is used, a peek-modify-poke process is used. in a best-effort not to modify the value of the other fields in the register. Write

Parameters

`status` (*uvm_status_e*)
`value` (*uvm_reg_data_t*)
`path` (*uvm_path_e*)
`map` (*uvm_reg_map*)
`parent` (*uvm_sequence_base*)
`prior` (`int`)
`extension` (*uvm_object*)
`fname` (`string`)
`lineno` (`int`)

`virtual function read(uvm_status_e status, uvm_reg_data_t value, uvm_path_- e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)`

Task

read

Read the current value from this field

Read and return *value* from the DUT field that corresponds to this abstraction class instance using the specified access *path* . If the register containing this field is mapped in more than one address map, an address *map* must be specified if a physical access is used (front-door access). If a back-door access path is used, the effect of reading the field through a physical access is mimicked. For example, clear-on-read bits in the field will be set to zero.

The mirrored value will be updated using the *uvm_reg_field::predict()* method.

If a front-door access is used, and if the field is the only field in a byte lane and if the physical interface corresponding to the address map used to access the field support byte-enabling, then only the field is read. Otherwise, the entire register containing the field is read, and the mirrored values of the other fields in the same register are updated.

If a backdoor access is used, the entire containing register is peeked and the mirrored value of the other fields in the register is updated. Read

Parameters

- **status** (*uvm_status_e*)
- **value** (*uvm_reg_data_t*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **parent** (*uvm_sequence_base*)
- **prior** (`int`)
- **extension** (*uvm_object*)
- **fname** (`string`)
- **lineno** (`int`)

```
virtual function poke(uvm_status_e status, uvm_reg_data_t value, string kind = "",
uvm_sequence_base parent = null, uvm_object extension = null, string fname = "",
int lineno = 0)
```

Task

poke

Deposit the specified value in this field

Deposit the value in the DUT field corresponding to this abstraction class instance, as-is, using a back-door access. A peek-modify-poke process is used in a best-effort not to modify the value of the other fields in the register.

The mirrored value will be updated using the *uvm_reg_field::predict()* method. Poke

Parameters

- **status** (*uvm_status_e*)
- **value** (*uvm_reg_data_t*)
- **kind** (`string`)
- **parent** (*uvm_sequence_base*)
- **extension** (*uvm_object*)
- **fname** (`string`)
- **lineno** (`int`)

```
virtual function peek(uvm_status_e status, uvm_reg_data_t value, string kind = "",
uvm_sequence_base parent = null, uvm_object extension = null, string fname = "",
int lineno = 0)
```

Task

peek

Read the current value from this field

Sample the value in the DUT field corresponding to this abstraction class instance using a back-door access. The field value is sampled, not modified.

Uses the HDL path for the design abstraction specified by *kind* .

The entire containing register is peeked and the mirrored value of the other fields in the register are updated using the *uvm_reg_field::predict()* method. Peek

Parameters

- **status** (*uvm_status_e*)
- **value** (*uvm_reg_data_t*)

kind (*string*)
parent (*uvm_sequence_base*)
extension (*uvm_object*)
fname (*string*)
lineno (*int*)

virtual function mirror(uvm_status_e status, uvm_check_e check = UVM_NO_CHECK, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_-base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)

Task

mirror

Read the field and update/check its mirror value

Read the field and optionally compared the readback value with the current mirrored value if *check* is <UVM_CHECK>. The mirrored value will be updated using the *predict()* method based on the readback value.

The *path* argument specifies whether to mirror using the <UVM_FRONTDOOR> (*read*) or <UVM_BACK-DOOR> (*peek()*).

If *check* is specified as <UVM_CHECK>, an error message is issued if the current mirrored value does not match the readback value, unless *set_compare* was used disable the check.

If the containing register is mapped in multiple address maps and physical access is used (front-door access), an address *map* must be specified. For write-only fields, their content is mirrored and optionally checked only if a UVM_BACKDOOR access path is used to read the field. Mirror

Parameters

status (*uvm_status_e*)
check (*uvm_check_e*)
path (*uvm_path_e*)
map (*uvm_reg_map*)
parent (*uvm_sequence_base*)
prior (*int*)
extension (*uvm_object*)
fname (*string*)
lineno (*int*)

virtual function do_write(uvm_reg_item rw)

Do_write

Parameters

rw (*uvm_reg_item*)

virtual function do_read(uvm_reg_item rw)

Do_read

Parameters

rw (*uvm_reg_item*)

virtual function pre_write(uvm_reg_item rw)

Task

pre_write

Called before field write.

If the specified data value, access *path* or address *map* are modified, the updated data value, access path or address map will be used to perform the register operation. If the *status* is modified to anything other than <UVM_IS_OK>, the operation is aborted.

The field callback methods are invoked after the callback methods on the containing register. The registered callback methods are invoked after the invocation of this method.

Parameters

rw (*uvm_reg_item*)

`virtual function post_write(uvm_reg_item rw)`

Task

post_write

Called after field write.

If the specified *status* is modified, the updated status will be returned by the register operation.

The field callback methods are invoked after the callback methods on the containing register. The registered callback methods are invoked before the invocation of this method.

Parameters

`rw` (*uvm_reg_item*)

`virtual function pre_read(uvm_reg_item rw)`

Task

pre_read

Called before field read.

If the access *path* or address *map* in the *rw* argument are modified, the updated access path or address map will be used to perform the register operation. If the *status* is modified to anything other than <UVM_IS_OK>, the operation is aborted.

The field callback methods are invoked after the callback methods on the containing register. The registered callback methods are invoked after the invocation of this method.

Parameters

`rw` (*uvm_reg_item*)

`virtual function post_read(uvm_reg_item rw)`

Task

post_read

Called after field read.

If the specified readback data or~status~ in the *rw* argument is modified, the updated readback data or status will be returned by the register operation.

The field callback methods are invoked after the callback methods on the containing register. The registered callback methods are invoked before the invocation of this method.

Parameters

`rw` (*uvm_reg_item*)

15.1.1.180 Class uvm_pkg::uvm_reg_fifo

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_reg*
↪*uvm_pkg* :: *uvm_reg_fifo*

Class

uvm_reg_fifo

This special register models a DUT FIFO accessed via write/read, where writes push to the FIFO and reads pop from it.

Backdoor access is not enabled, as it is not yet possible to force complete FIFO state, i.e. the write and read indexes used to access the FIFO data.

Table 196: Variables

Name	Type	Description
fifo	*uvm_reg_data_t*	***Variable*** fifo The abstract representation of the FIFO. Constrained to be no larger than the size parameter. It is public to enable subtypes to add constraints on it and randomize.

Table 197: Constraints

Name	Description
valid_fifo_size	

Constructors

`function new(string name = "reg_fifo", int unsigned size, int unsigned n_bits, int has_cover)`

Function

new

Creates an instance of a FIFO register having *size* elements of *n_bits* each.

Parameters
- **name** (*string*)
- **size** (*int unsigned*)
- **n_bits** (*int unsigned*)
- **has_cover** (*int*)

Functions

`virtual function void build()`

Funtion

build

Builds the abstract FIFO register object. Called by the instantiating block, a *uvm_reg_block* subtype.

`function void set_compare(uvm_check_e check = UVM_CHECK)`

Function

set_compare

Sets the compare policy during a mirror (read) of the DUT FIFO. The DUT read value is checked against its mirror only when both the *check* argument in the <mirror()> call and the compare policy for the field is <UVM_CHECK>.

Parameters

check (*uvm_check_e*)

function int unsigned size()

Function

size

The number of entries currently in the FIFO.

function int unsigned capacity()

Function

capacity

The maximum number of entries, or depth, of the FIFO.

virtual function void set(uvm_reg_data_t value, string fname = "", int lineno = 0)

Function

set

Pushes the given value to the abstract FIFO. You may call this method several times before an *update()* as a means of preloading the DUT FIFO. Calls to *set()* to a full FIFO are ignored. You must call *update()* to update the DUT FIFO with your set values.

Parameters

value (*uvm_reg_data_t*)
fname (*string*)
lineno (*int*)

virtual function uvm_reg_data_t get(string fname = "", int lineno = 0)

Function

get

Returns the next value from the abstract FIFO, but does not pop it. Used to get the expected value in a <mirror()> operation.

Parameters

fname (*string*)
lineno (*int*)

Return type

uvm_reg_data_t

virtual function void do_predict(uvm_reg_item rw, uvm_predict_e kind = UVM_-PREDICT_DIRECT, uvm_reg_byte_en_t be = -1)

Function

do_predict

Updates the abstract (mirror) FIFO based on <write()> and <read()> operations. When auto-prediction is on, this method is called before each read, write, peek, or poke operation returns. When auto-prediction is off, this method is called by a *uvm_reg_predictor* upon receipt and conversion of an observed bus operation to this register.

If a write prediction, the observed write value is pushed to the abstract FIFO as long as it is not full and the operation did not originate from an *update()*. If a read prediction, the observed read value is compared with the frontmost value in the abstract FIFO if *set_compare()* enabled comparison and the FIFO is not empty.

Parameters

rw (*uvm_reg_item*)
kind (*uvm_predict_e*)
be (*uvm_reg_byte_en_t*)

Tasks

`virtual function update(uvm_status_e status, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)`

Function

update

Pushes (writes) all values preloaded using *set()* to the DUT. You must *update* after *set* before any blocking statements, else other reads/writes to the DUT FIFO may cause the mirror to become out of sync with the DUT.

Parameters

- **`status`** (*uvm_status_e*)
- **`path`** (*uvm_path_e*)
- **`map`** (*uvm_reg_map*)
- **`parent`** (*uvm_sequence_base*)
- **`prior`** (*`int`*)
- **`extension`** (*uvm_object*)
- **`fname`** (*`string`*)
- **`lineno`** (*`int`*)

`virtual function pre_write(uvm_reg_item rw)`

Task

pre_write

Special pre-processing for a <write()> or *update()*. Called as a result of a <write()> or *update()*. It is an error to attempt a write to a full FIFO or a write while an update is still pending. An update is pending after one or more calls to *set()*. If in your application the DUT allows writes to a full FIFO, you must override *pre_write* as appropriate.

Parameters

- **`rw`** (*uvm_reg_item*)

`virtual function pre_read(uvm_reg_item rw)`

Task

pre_read

Special post-processing for a <write()> or *update()*. Aborts the operation if the internal FIFO is empty. If in your application the DUT does not behave this way, you must override *pre_write* as appropriate.

Parameters

- **`rw`** (*uvm_reg_item*)

15.1.1.181 Class uvm_pkg::uvm_reg_file

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_reg_file*

CLASS

uvm_reg_file

Register file abstraction base class

A register file is a collection of register files and registers used to create regular repeated structures.

Register files are usually instantiated as arrays.

Constructors

`function new(string name = "")`

Function

new

Create a new instance

Creates an instance of a register file abstraction class with the specified name. New
Parameters
`name` (`string`)

Functions

`function void configure(uvm_reg_block blk_parent, uvm_reg_file regfile_parent, string hdl_path = "")`

Function

configure

Configure a register file instance

Specify the parent block and register file of the register file instance. If the register file is instantiated in a block, *regfile_parent* is specified as *null* . If the register file is instantiated in a register file, *blk_parent* must be the block parent of that register file and *regfile_parent* is specified as that register file.

If the register file corresponds to a hierarchical RTL structure, its contribution to the HDL path is specified as the *hdl_path* . Otherwise, the register file does not correspond to a hierarchical RTL structure (e.g. it is physically flattened) and does not contribute to the hierarchical HDL path of any contained registers. Configure
Parameters
`blk_parent` (*uvm_reg_block*)
`regfile_parent` (*uvm_reg_file*)
`hdl_path` (`string`)

`virtual function string get_full_name()`

Function

get_full_name

Get the hierarchical name

Return the hierarchal name of this register file. The base of the hierarchical name is the root block. Get_full_name

`virtual function uvm_reg_block get_parent()`

Function

get_parent

Get the parent block. Get_parent

Return type

uvm_reg_block

`virtual function uvm_reg_block get_block()`

Get_block

Return type

uvm_reg_block

`virtual function uvm_reg_file get_regfile()`

Function

get_regfile

Get the parent register file

Returns *null* if this register file is instantiated in a block. Get_regfile

Return type

uvm_reg_file

`function void clear_hdl_path(string kind = "RTL")`

Function

clear_hdl_path

Delete HDL paths

Remove any previously specified HDL path to the register file instance for the specified design abstraction. Clear_hdl_path

Parameters

`kind` (*`string`*)

`function void add_hdl_path(string path, string kind = "RTL")`

Function

add_hdl_path

Add an HDL path

Add the specified HDL path to the register file instance for the specified design abstraction. This method may be called more than once for the same design abstraction if the register file is physically duplicated in the design abstraction. Add_hdl_path

Parameters

- **`path`** (*`string`*)
- **`kind`** (*`string`*)

`function bit has_hdl_path(string kind = "")`

Function

has_hdl_path

Check if a HDL path is specified

Returns TRUE if the register file instance has a HDL path defined for the specified design abstraction. If no design abstraction is specified, uses the default design abstraction specified for the nearest enclosing register file or block

If no design abstraction is specified, the default design abstraction for this register file is used. Has_hdl_path

Parameters

`kind` (*`string`*)

`function void get_hdl_path(string paths, string kind = "")`

Function

get_hdl_path

Get the incremental HDL path(s)

Returns the HDL path(s) defined for the specified design abstraction in the register file instance. If no design abstraction is specified, uses the default design abstraction specified for the nearest enclosing register file or block. Returns only the component of the HDL paths that corresponds to the register file, not a full hierarchical path

If no design abstraction is specified, the default design abstraction for this register file is used. Get_hdl_path

Parameters

paths (*string*)

kind (*string*)

function void get_full_hdl_path(string paths, string kind = "", string separator = ".")

Function

get_full_hdl_path

Get the full hierarchical HDL path(s)

Returns the full hierarchical HDL path(s) defined for the specified design abstraction in the register file instance. If no design abstraction is specified, uses the default design abstraction specified for the nearest enclosing register file or block. There may be more than one path returned even if only one path was defined for the register file instance, if any of the parent components have more than one path defined for the same design abstraction

If no design abstraction is specified, the default design abstraction for each ancestor register file or block is used to get each incremental path. Get_full_hdl_path

Parameters

paths (*string*)

kind (*string*)

separator (*string*)

function void set_default_hdl_path(string kind)

Function

set_default_hdl_path

Set the default design abstraction

Set the default design abstraction for this register file instance. Set_default_hdl_path

Parameters

kind (*string*)

function string get_default_hdl_path()

Function

get_default_hdl_path

Get the default design abstraction

Returns the default design abstraction for this register file instance. If a default design abstraction has not been explicitly set for this register file instance, returns the default design abstraction for the nearest register file or block ancestor. Returns "" if no default design abstraction has been specified. Get_default_hdl_path

virtual function void do_print(uvm_printer printer)

Do_print

Parameters

printer (*uvm_printer*)

virtual function string convert2string()

Convert2string

virtual function uvm_object clone()

Clone

Return type

uvm_object

virtual function void do_copy(uvm_object rhs)

Do_copy

Parameters

rhs (*uvm_object*)

virtual function bit do_compare(uvm_object rhs, uvm_comparer comparer)

Do_compare

Parameters

rhs (*uvm_object*)

comparer (*uvm_comparer*)

`virtual function void do_pack(uvm_packer packer)`

Do_pack

Parameters

`packer` (*uvm_packer*)

`virtual function void do_unpack(uvm_packer packer)`

Do_unpack

Parameters

`packer` (*uvm_packer*)

15.1.1.182 Class uvm_pkg::uvm_reg_frontdoor

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*
↪*uvm_pkg* :: *uvm_reg_sequence*
↪*uvm_pkg* :: *uvm_reg_frontdoor*

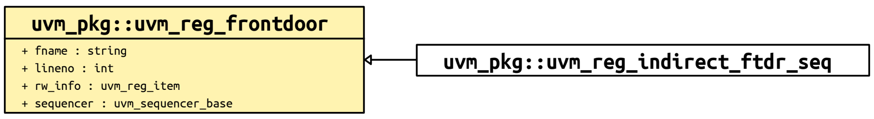

uvm_pkg::uvm_reg_frontdoor
+ fname : string
+ lineno : int
+ rw_info : uvm_reg_item
+ sequencer : uvm_sequencer_base
uvm_pkg::uvm_reg_indirect_ftdr_seq

Fig. 60: Inheritance Diagram of uvm_reg_frontdoor

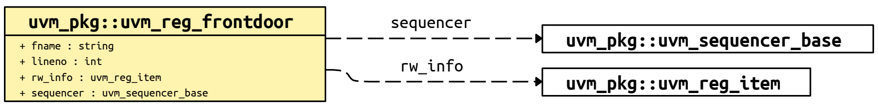


Fig. 61: Collaboration Diagram of uvm_reg_frontdoor

Class

uvm_reg_frontdoor

Facade class for register and memory frontdoor access.

User-defined frontdoor access sequence

Base class for user-defined access to register and memory reads and writes through a physical interface.

By default, different registers and memories are mapped to different addresses in the address space and are accessed via those exclusively through physical addresses.

The frontdoor allows access using a non-linear and/or non-mapped mechanism. Users can extend this class to provide the physical access to these registers.

Table 198: Variables

Name	Type	Description
rw_info	*uvm_reg_item*	***Variable*** rw_info Holds information about the register being read or written
sequencer	*uvm_sequencer_base*	***Variable*** sequencer Sequencer executing the operation
fname	string	

continues on next page

Table 198 – continued from previous page

Name	Type	Description
lineno	int	

Constructors

`function new(string name = "")`

Function

new

Constructor, new object given optional *name* .

Parameters

`name` (*`string`*)

15.1.1.183 Class uvm_pkg::uvm_reg_hw_reset_seq

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*
↪*uvm_pkg* :: *uvm_reg_sequence*
↪*uvm_pkg* :: *uvm_reg_hw_reset_seq*

class

uvm_reg_hw_reset_seq

Test the hard reset values of registers

The test sequence performs the following steps

1. resets the DUT and the block abstraction class associated with this sequence.
2. reads all of the registers in the block, via all of the available address maps, comparing the value read with the expected reset value.

If bit-type resource named "NO_REG_TESTS" or "NO_REG_HW_RESET_TEST" in the "REG::" namespace matches the full name of the block or register, the block or register is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.get_full_name(),".*"},
                                "NO_REG_TESTS", 1, this);
```

This is usually the first test executed on any DUT.

Constructors

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

Parameters
`name` (`string`)

Tasks

`virtual function body()`

Variable

body

Executes the Hardware Reset sequence. Do not call directly. Use seq.start() instead.

`virtual function reset_blk(uvm_reg_block blk)`

task

reset_blk

Reset the DUT that corresponds to the specified block abstraction class.

Currently empty. Will rollback the environment's phase to the *reset* phase once the new phasing is available.

In the meantime, the DUT should be reset before executing this test sequence or this method should be implemented in an extension to reset the DUT.

Parameters
`blk` (*uvm_reg_block*)

15.1.1.184 Class uvm_pkg::uvm_reg_indirect_data

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_reg*
↪*uvm_pkg* :: *uvm_reg_indirect_data*

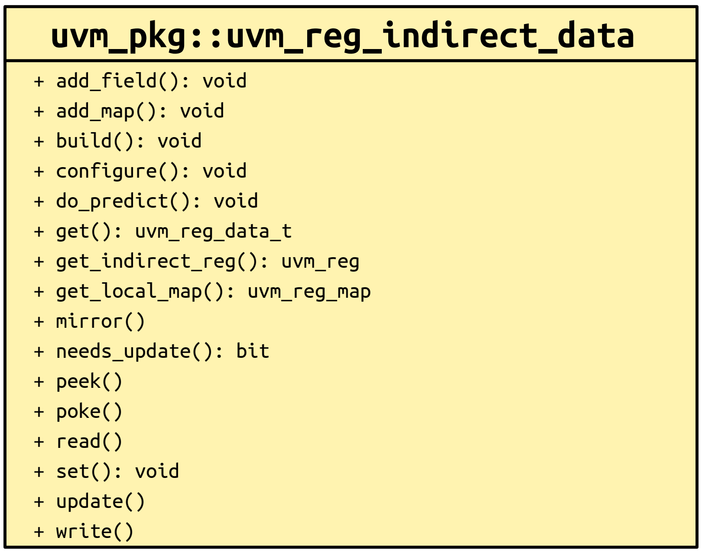


Fig. 62: Collaboration Diagram of uvm_reg_indirect_data

CLASS

uvm_reg_indirect_data

Indirect data access abstraction class

Models the behavior of a register used to indirectly access a register array, indexed by a second *address* register.

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

Constructors

function new(string name = "uvm_reg_indirect", int unsigned n_bits, int has_cover)

Function

new

Create an instance of this class

Should not be called directly, other than via super.new(). The value of *n_bits* must match the number of bits in the indirect register array.

Parameters

- **name** (*string*)
- **n_bits** (*int unsigned*)
- **has_cover** (*int*)

Functions

`virtual function void build()`

`function void configure(uvm_reg idx, uvm_reg reg_a, uvm_reg_block blk_parent, uvm_reg_file regfile_parent = null)`

Function

configure

Configure the indirect data register.

The *idx* register specifies the index, in the *reg_a* register array, of the register to access. The *idx* must be written to first. A read or write operation to this register will subsequently read or write the indexed register in the register array.

The number of bits in each register in the register array must be equal to *n_bits* of this register.

See *uvm_reg::configure()* for the remaining arguments.

Parameters
- **idx** (*uvm_reg*)
- **reg_a** (*uvm_reg*)
- **blk_parent** (*uvm_reg_block*)
- **regfile_parent** (*uvm_reg_file*)

`virtual function void add_map(uvm_reg_map map)`

Parameters
- **map** (*uvm_reg_map*) -- Local

`virtual function void do_predict(uvm_reg_item rw, uvm_predict_e kind = UVM_PREDICT_DIRECT, uvm_reg_byte_en_t be = -1)`

Parameters
- **rw** (*uvm_reg_item*)
- **kind** (*uvm_predict_e*)
- **be** (*uvm_reg_byte_en_t*)

`virtual function uvm_reg_map get_local_map(uvm_reg_map map, string caller = "")`

Parameters
- **map** (*uvm_reg_map*)
- **caller** (`string`)

Return type
uvm_reg_map

`virtual function void add_field(uvm_reg_field field)`

Just for good measure, to catch and short-circuit non-sensical uses

Parameters
- **field** (*uvm_reg_field*)

`virtual function void set(uvm_reg_data_t value, string fname = "", int lineno = 0)`

Parameters
- **value** (*uvm_reg_data_t*)
- **fname** (`string`)
- **lineno** (`int`)

`virtual function uvm_reg_data_t get(string fname = "", int lineno = 0)`

Parameters
- **fname** (`string`)
- **lineno** (`int`)

Return type
uvm_reg_data_t

`virtual function uvm_reg get_indirect_reg(string fname = "", int lineno = 0)`

Parameters
- **fname** (`string`)
- **lineno** (`int`)

Return type
uvm_reg

`virtual function bit needs_update()`

Tasks

virtual function write(uvm_status_e status, uvm_reg_data_t value, uvm_path_-e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)

Parameters

- **status** (*uvm_status_e*)
- **value** (*uvm_reg_data_t*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **parent** (*uvm_sequence_base*)
- **prior** (`int`)
- **extension** (*uvm_object*)
- **fname** (`string`)
- **lineno** (`int`)

virtual function read(uvm_status_e status, uvm_reg_data_t value, uvm_path_-e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_base parent = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)

Parameters

- **status** (*uvm_status_e*)
- **value** (*uvm_reg_data_t*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **parent** (*uvm_sequence_base*)
- **prior** (`int`)
- **extension** (*uvm_object*)
- **fname** (`string`)
- **lineno** (`int`)

virtual function poke(uvm_status_e status, uvm_reg_data_t value, string kind = "", uvm_sequence_base parent = null, uvm_object extension = null, string fname = "", int lineno = 0)

Parameters

- **status** (*uvm_status_e*)
- **value** (*uvm_reg_data_t*)
- **kind** (`string`)
- **parent** (*uvm_sequence_base*)
- **extension** (*uvm_object*)
- **fname** (`string`)
- **lineno** (`int`)

virtual function peek(uvm_status_e status, uvm_reg_data_t value, string kind = "", uvm_sequence_base parent = null, uvm_object extension = null, string fname = "", int lineno = 0)

Parameters

- **status** (*uvm_status_e*)
- **value** (*uvm_reg_data_t*)
- **kind** (`string`)
- **parent** (*uvm_sequence_base*)
- **extension** (*uvm_object*)
- **fname** (`string`)
- **lineno** (`int`)

virtual function update(uvm_status_e status, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_base parent = null, int prior = -1, uvm_-object extension = null, string fname = "", int lineno = 0)

Parameters

- **status** (*uvm_status_e*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)

parent (*uvm_sequence_base*)
prior (*int*)
extension (*uvm_object*)
fname (*string*)
lineno (*int*)

```
virtual function mirror(uvm_status_e status, uvm_check_e check = UVM_NO_CHECK,
uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_-
base parent = null, int prior = -1, uvm_object extension = null, string fname = "",
int lineno = 0)
```

Parameters

status (*uvm_status_e*)
check (*uvm_check_e*)
path (*uvm_path_e*)
map (*uvm_reg_map*)
parent (*uvm_sequence_base*)
prior (*int*)
extension (*uvm_object*)
fname (*string*)
lineno (*int*)

15.1.1.185 Class uvm_pkg::uvm_reg_indirect_ftdr_seq

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*
↪*uvm_pkg* :: *uvm_reg_sequence*
↪*uvm_pkg* :: *uvm_reg_frontdoor*
↪*uvm_pkg* :: *uvm_reg_indirect_ftdr_seq*

Constructors

`function new(uvm_reg addr_reg, int idx, uvm_reg data_reg)`

Parameters
- **`addr_reg`** (*uvm_reg*)
- **`idx`** (`int`)
- **`data_reg`** (*uvm_reg*)

Tasks

`virtual function body()`

15.1.1.186 Class uvm_pkg::uvm_reg_item

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_reg_item*

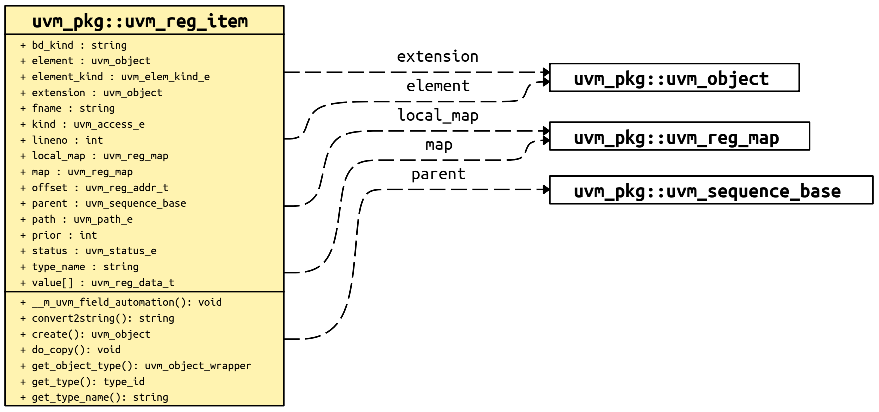


Fig. 63: Collaboration Diagram of uvm_reg_item

CLASS

uvm_reg_item

Defines an abstract register transaction item. No bus-specific information is present, although a handle to a *uvm_reg_map* is provided in case a user wishes to implement a custom address translation algorithm.

Table 199: Variables

Name	Type	Description
element_kind	*uvm_elem_kind_e*	***Variable*** element_kind ***Kind of element being accessed*** REG, MEM, or FIELD. See *uvm_elem_kind_e*.
element	*uvm_object*	***Variable*** element A handle to the RegModel model element associated with this transaction. Use *element_kind* to determine the type to cast to: *uvm_reg*, *uvm_mem*, or *uvm_reg_field*.
kind	*uvm_access_e*	***Variable*** kind ***Kind of access*** READ or WRITE.

continues on next page

Table 199 – continued from previous page

Name	Type	Description
value	*uvm_reg_data_t*	***Variable*** value The value to write to, or after completion, the value read from the DUT. Burst operations use the *values* property.
offset	*uvm_reg_addr_t*	***Variable*** offset For memory accesses, the offset address. For bursts, the *starting* offset address.
status	*uvm_status_e*	***Variable*** status ***The result of the transaction*** IS_OK, HAS_X, or ERROR. See *uvm_status_e*.
local_map	*uvm_reg_map*	***Variable*** local_map The local map used to obtain addresses. Users may customize address-translation using this map. Access to the sequencer and bus adapter can be obtained by getting this map's root map, then calling *uvm_reg_map::get_sequencer* and *uvm_reg_map::get_adapter*.
map	*uvm_reg_map*	***Variable*** map The original map specified for the operation. The actual *map* used may differ when a test or sequence written at the block level is reused at the system level.
path	*uvm_path_e*	***Variable*** path ***The path being used*** <UVM_FRONTDOOR> or <UVM_BACKDOOR>.
parent	*uvm_sequence_base*	***Variable*** parent The sequence from which the operation originated.
prior	int	***Variable*** prior The priority requested of this transfer, as defined by *uvm_sequence_base::start_item*.
extension	*uvm_object*	***Variable*** extension Handle to optional user data, as conveyed in the call to write(), read(), mirror(), or update() used to trigger the operation.

continues on next page

Table 199 – continued from previous page

Name	Type	Description
bd_kind	string	***Variable*** bd_kind If path is UVM_BACKDOOR, this member specifies the abstraction kind for the backdoor access, e.g. "RTL" or "GATES".
fname	string	***Variable*** fname The file name from where this transaction originated, if provided at the call site.
lineno	int	***Variable*** lineno The file name from where this transaction originated, if provided at the call site.

Table 200: Constraints

Name	Description
max_values	TODO: parameterize

Constructors

`function new(string name = "")`

Function

new

Create a new instance of this type, giving it the optional *name* .

Parameters

name (*string*)

Functions

`virtual function string convert2string()`

Function

convert2string

Returns a string showing the contents of this transaction.

`virtual function void do_copy(uvm_object rhs)`

Function

do_copy

Copy the *rhs* object into this object. The *rhs* object must derive from *uvm_reg_item*.

Parameters

rhs (*uvm_object*)

15.1.1.187 Class uvm_pkg::uvm_reg_map

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_reg_map*

Class

uvm_reg_map

```
Address map abstraction class
```

This class represents an address map. An address map is a collection of registers and memories accessible via a specific physical interface. Address maps can be composed into higher-level address maps.

Address maps are created using the *uvm_reg_block::create_map()* method.

Table 201: Assertions

Name	Kind	Description
uvm_pkg::uvm_reg_-map.[anonymous]	immediate assert	`$cast(seq, o)`
uvm_pkg::uvm_reg_-map.[anonymous]	immediate assert	`$cast(seq, o)`

Constructors

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

Function

new

Create a new instance. New

Parameters

name (*`string`*)

Functions

`function void Xinit_address_mapX()`

Xinit_address_mapXlocal

`static function uvm_reg_map backdoor()`

Function

backdoor

Return the backdoor pseudo-map singleton

This pseudo-map is used to specify or configure the backdoor instead of a real address map.

Return type

uvm_reg_map

`function void configure(uvm_reg_block parent, uvm_reg_addr_t base_addr, int unsigned n_bytes, uvm_endianness_e endian, bit byte_addressing = 1)`

Function

configure

Instance-specific configuration

Configures this map with the following properties.

parent

the block in which this map is created and applied

base_addr

the base address for this map. All registers, memories, and sub-blocks will be at offsets to this address

n_bytes

the byte-width of the bus on which this map is used

endian

the endian format. See *uvm_endianness_e* for possible values

byte_addressing

specifies whether the address increment is on a per-byte basis. For example, consecutive memory locations with *n_bytes* =4 (32-bit bus) are 4 apart: 0, 4, 8, and so on. Default is TRUE. Configure

Parameters

- `parent` (*uvm_reg_block*)
- `base_addr` (*uvm_reg_addr_t*)
- `n_bytes` (`int unsigned`)
- `endian` (*uvm_endianness_e*)
- `byte_addressing` (`bit`)

```
virtual  function void add_reg(uvm_reg rg, uvm_reg_addr_t offset,
string rights = "RW", bit unmapped = 0, uvm_reg_frontdoor frontdoor = null)
```

Function

add_reg

Add a register

Add the specified register instance *rg* to this address map.

The register is located at the specified address *offset* from this maps configured base address.

The *rights* specify the register's accessibility via this map. Valid values are "RW", "RO", and "WO". Whether a register field can be read or written depends on both the field's configured access policy (see *uvm_reg_field::configure* and the register's rights in the map being used to access the field.

The number of consecutive physical addresses occupied by the register depends on the width of the register and the number of bytes in the physical interface corresponding to this address map.

If *unmapped* is TRUE, the register does not occupy any physical addresses and the base address is ignored. Unmapped registers require a user-defined *frontdoor* to be specified.

A register may be added to multiple address maps if it is accessible from multiple physical interfaces. A register may only be added to an address map whose parent block is the same as the register's parent block. Add_reg

Parameters

- `rg` (*uvm_reg*)
- `offset` (*uvm_reg_addr_t*)
- `rights` (`string`)
- `unmapped` (`bit`)
- `frontdoor` (*uvm_reg_frontdoor*)

```
virtual  function void add_mem(uvm_mem mem, uvm_reg_addr_t offset,
string rights = "RW", bit unmapped = 0, uvm_reg_frontdoor frontdoor = null)
```

Function

add_mem

Add a memory

Add the specified memory instance to this address map. The memory is located at the specified base address and has the specified access rights ("RW", "RO" or "WO"). The number of consecutive physical addresses occupied by the memory depends on the width and size of the memory and the number of bytes in the physical interface corresponding to this address map.

If *unmapped* is TRUE, the memory does not occupy any physical addresses and the base address is ignored. Unmapped memories require a user-defined *frontdoor* to be specified.

A memory may be added to multiple address maps if it is accessible from multiple physical interfaces. A memory may only be added to an address map whose parent block is the same as the memory's parent block. Add_mem

Parameters

- **`mem`** (*uvm_mem*)
- **`offset`** (*uvm_reg_addr_t*)
- **`rights`** (*`string`*)
- **`unmapped`** (*`bit`*)
- **`frontdoor`** (*uvm_reg_frontdoor*)

`virtual function void add_submap(uvm_reg_map child_map, uvm_reg_addr_t offset)`

Function

add_submap

Add an address map

Add the specified address map instance to this address map. The address map is located at the specified base address. The number of consecutive physical addresses occupied by the submap depends on the number of bytes in the physical interface that corresponds to the submap, the number of addresses used in the submap and the number of bytes in the physical interface corresponding to this address map.

An address map may be added to multiple address maps if it is accessible from multiple physical interfaces. An address map may only be added to an address map in the grand-parent block of the address submap. Add_submap

Parameters

- **`child_map`** (*uvm_reg_map*)
- **`offset`** (*uvm_reg_addr_t*)

`virtual function void set_sequencer(uvm_sequencer_base sequencer, uvm_reg_adapter adapter = null)`

Function

set_sequencer

Set the sequencer and adapter associated with this map. This method *must* be called before starting any sequences based on uvm_reg_sequence. Set_sequencer

Parameters

- **`sequencer`** (*uvm_sequencer_base*)
- **`adapter`** (*uvm_reg_adapter*)

`virtual function void set_submap_offset(uvm_reg_map submap, uvm_reg_addr_t offset)`

Function

set_submap_offset

Set the offset of the given *submap* to *offset* . Set_submap_offset

Parameters

- **`submap`** (*uvm_reg_map*)
- **`offset`** (*uvm_reg_addr_t*)

`virtual function uvm_reg_addr_t get_submap_offset(uvm_reg_map submap)`

Function

get_submap_offset

Return the offset of the given *submap* . Get_submap_offset

Parameters

`submap` (*uvm_reg_map*)

Return type

uvm_reg_addr_t

`virtual function void set_base_addr(uvm_reg_addr_t offset)`

Function

set_base_addr

Set the base address of this map. Set_base_addr

Parameters

`offset` (*uvm_reg_addr_t*)

`virtual function void reset(string kind = "SOFT")`

Function

reset

Reset the mirror for all registers in this address map.

Sets the mirror value of all registers in this address map and all of its submaps to the reset value corresponding to the specified reset event. See *uvm_reg_field::reset()* for more details. Does not actually set the value of the registers in the design, only the values mirrored in their corresponding mirror.

Note that, unlike the other reset() method, the default reset event for this method is "SOFT". Reset

Parameters

`kind` (`string`)

`virtual function void add_parent_map(uvm_reg_map parent_map, uvm_reg_addr_t offset)`

Add_parent_map

Parameters

`parent_map` (*uvm_reg_map*) -- Local

`offset` (*uvm_reg_addr_t*)

`virtual function void Xverify_map_configX()`

Local

`virtual function string get_full_name()`

Function

get_full_name

Get the hierarchical name

Return the hierarchal name of this address map. The base of the hierarchical name is the root block. Get_full_name

`virtual function uvm_reg_map get_root_map()`

Function

get_root_map

Get the externally-visible address map

Get the top-most address map where this address map is instantiated. It corresponds to the externally-visible address map that can be accessed by the verification environment. Get_root_map

Return type

uvm_reg_map

`virtual function uvm_reg_block get_parent()`

Function

get_parent

Get the parent block

Return the block that is the parent of this address map. Get_parent

Return type

uvm_reg_block

`virtual function uvm_reg_map get_parent_map()`

Function

get_parent_map

Get the higher-level address map

Return the address map in which this address map is mapped. returns *null* if this is a top-level address map. Get_parent_map

Return type

uvm_reg_map

virtual function uvm_reg_addr_t get_base_addr(uvm_hier_e hier = UVM_HIER)

Function

get_base_addr

Get the base offset address for this map. If this map is the root map, the base address is that set with the *base_addr* argument to *uvm_reg_block::create_map()*. If this map is a submap of a higher-level map, the base address is offset given this submap by the parent map. See *set_submap_offset*. Get_base_addr

Parameters
: **hier** (*uvm_hier_e*)

Return type
: *uvm_reg_addr_t*

virtual function int unsigned get_n_bytes(uvm_hier_e hier = UVM_HIER)

Function

get_n_bytes

Get the width in bytes of the bus associated with this map. If *hier* is *UVM_HIER* , then gets the effective bus width relative to the system level. The effective bus width is the narrowest bus width from this map to the top-level root map. Each bus access will be limited to this bus width. Get_n_bytes

Parameters
: **hier** (*uvm_hier_e*)

virtual function int unsigned get_addr_unit_bytes()

Function

get_addr_unit_bytes

Get the number of bytes in the smallest addressable unit in the map. Returns 1 if the address map was configured using byte-level addressing. Returns *get_n_bytes()* otherwise. Get_addr_unit_bytes

virtual function uvm_endianness_e get_endian(uvm_hier_e hier = UVM_HIER)

Function

get_base_addr

Gets the endianness of the bus associated with this map. If *hier* is set to *UVM_HIER* , gets the system-level endianness. Get_endian

Parameters
: **hier** (*uvm_hier_e*)

Return type
: *uvm_endianness_e*

virtual function uvm_sequencer_base get_sequencer(uvm_hier_e hier = UVM_HIER)

Function

get_sequencer

Gets the sequencer for the bus associated with this map. If *hier* is set to *UVM_HIER* , gets the sequencer for the bus at the system-level. See *set_sequencer*. Get_sequencer

Parameters
: **hier** (*uvm_hier_e*)

Return type
: *uvm_sequencer_base*

virtual function uvm_reg_adapter get_adapter(uvm_hier_e hier = UVM_HIER)

Function

get_adapter

Gets the bus adapter for the bus associated with this map. If *hier* is set to *UVM_HIER* , gets the adapter for the bus used at the system-level. See *set_sequencer*. Get_adapter

Parameters
: **hier** (*uvm_hier_e*)

Return type
: *uvm_reg_adapter*

`virtual function void get_submaps(uvm_reg_map maps, uvm_hier_e hier = UVM_HIER)`

Function

get_submaps

Get the address sub-maps

Get the address maps instantiated in this address map. If *hier* is *UVM_HIER* , recursively includes the address maps, in the sub-maps. Get_submaps

Parameters

- **`maps`** (*uvm_reg_map*)
- **`hier`** (*uvm_hier_e*)

`virtual function void get_registers(uvm_reg regs, uvm_hier_e hier = UVM_HIER)`

Function

get_registers

Get the registers

Get the registers instantiated in this address map. If *hier* is *UVM_HIER* , recursively includes the registers in the sub-maps. Get_registers

Parameters

- **`regs`** (*uvm_reg*)
- **`hier`** (*uvm_hier_e*)

`virtual function void get_fields(uvm_reg_field fields, uvm_hier_e hier = UVM_HIER)`

Function

get_fields

Get the fields

Get the fields in the registers instantiated in this address map. If *hier* is *UVM_HIER* , recursively includes the fields of the registers in the sub-maps. Get_fields

Parameters

- **`fields`** (*uvm_reg_field*)
- **`hier`** (*uvm_hier_e*)

`virtual function void get_memories(uvm_mem mems, uvm_hier_e hier = UVM_HIER)`

Function

get_memories

Get the memories

Get the memories instantiated in this address map. If *hier* is *UVM_HIER* , recursively includes the memories in the sub-maps. Get_memories

Parameters

- **`mems`** (*uvm_mem*)
- **`hier`** (*uvm_hier_e*)

`virtual function void get_virtual_registers(uvm_vreg regs, uvm_hier_e hier = UVM_-HIER)`

Function

get_virtual_registers

Get the virtual registers

Get the virtual registers instantiated in this address map. If *hier* is *UVM_HIER* , recursively includes the virtual registers in the sub-maps. Get_virtual_registers

Parameters

- **`regs`** (*uvm_vreg*)
- **`hier`** (*uvm_hier_e*)

`virtual function void get_virtual_fields(uvm_vreg_field fields, uvm_hier_-e hier = UVM_HIER)`

Function

get_virtual_fields

Get the virtual fields

Get the virtual fields from the virtual registers instantiated in this address map. If *hier* is *UVM_HIER* , recursively includes the virtual fields in the virtual registers in the sub-maps. Get_virtual_fields

Parameters

- **`fields`** (*uvm_vreg_field*)
- **`hier`** (*uvm_hier_e*)

`virtual function uvm_reg_map_info get_reg_map_info(uvm_reg rg, bit error = 1)`

Get_reg_map_info

Parameters

- **`rg`** (*uvm_reg*)
- **`error`** (`bit`)

Return type

uvm_reg_map_info

`virtual function uvm_reg_map_info get_mem_map_info(uvm_mem mem, bit error = 1)`

Get_mem_map_info

Parameters

- **`mem`** (*uvm_mem*)
- **`error`** (`bit`)

Return type

uvm_reg_map_info

`virtual function int unsigned get_size()`

Get_size

`virtual function int get_physical_addresses(uvm_reg_addr_t base_addr, uvm_reg_-addr_t mem_offset, int unsigned n_bytes, uvm_reg_addr_t addr)`

Function

get_physical_addresses

Translate a local address into external addresses

Identify the sequence of addresses that must be accessed physically to access the specified number of bytes at the specified address within this address map. Returns the number of bytes of valid data in each access.

Returns in *addr* a list of address in little endian order, with the granularity of the top-level address map.

A register is specified using a base address with *mem_offset* as 0. A location within a memory is specified using the base address of the memory and the index of the location within that memory. Get_physical_addresses

Parameters

- **`base_addr`** (*uvm_reg_addr_t*)
- **`mem_offset`** (*uvm_reg_addr_t*)
- **`n_bytes`** (`int unsigned`)
- **`addr`** (*uvm_reg_addr_t*)

`virtual function uvm_reg get_reg_by_offset(uvm_reg_addr_t offset, bit read = 1)`

Function

get_reg_by_offset

Get register mapped at offset

Identify the register located at the specified offset within this address map for the specified type of access. Returns *null* if no such register is found.

The model must be locked using *uvm_reg_block::lock_model()* to enable this functionality. Get_reg_by_offset

Parameters

- **`offset`** (*uvm_reg_addr_t*)
- **`read`** (`bit`)

Return type

uvm_reg

`virtual function uvm_mem get_mem_by_offset(uvm_reg_addr_t offset)`

Function

get_mem_by_offset

Get memory mapped at offset

Identify the memory located at the specified offset within this address map. The offset may refer to any memory location in that memory. Returns *null* if no such memory is found.

The model must be locked using *uvm_reg_block::lock_model()* to enable this functionality. Get_mem_by_offset

Parameters

`offset` (*uvm_reg_addr_t*)

Return type

uvm_mem

`function void set_auto_predict(bit on = 1)`

Function

set_auto_predict

Sets the auto-predict mode for his map.

When *on* is *TRUE* , the register model will automatically update its mirror (what it thinks should be in the DUT) immediately after any bus read or write operation via this map. Before a *uvm_reg::write* or *uvm_reg::read* operation returns, the register's *uvm_reg::predict* method is called to update the mirrored value in the register.

When *on* is *FALSE* , bus reads and writes via this map do not automatically update the mirror. For real-time updates to the mirror in this mode, you connect a *uvm_reg_predictor* instance to the bus monitor. The predictor takes observed bus transactions from the bus monitor, looks up the associated *uvm_reg* register given the address, then calls that register's *uvm_reg::predict* method. While more complex, this mode will capture all register read/write activity, including that not directly descendant from calls to *uvm_reg::write* and *uvm_reg::read*.

By default, auto-prediction is turned off.

Parameters

`on` (`bit`)

`function bit get_auto_predict()`

Function

get_auto_predict

Gets the auto-predict mode setting for this map.

`function void set_check_on_read(bit on = 1)`

Function

set_check_on_read

Sets the check-on-read mode for his map and all of its submaps.

When *on* is *TRUE* , the register model will automatically check any value read back from a register or field against the current value in its mirror and report any discrepancy. This effectively combines the functionality of the *uvm_reg::read()* and *uvm_reg::mirror(UVM_CHECK)* method. This mode is useful when the register model is used passively.

When *on* is *FALSE* , no check is made against the mirrored value.

At the end of the read operation, the mirror value is updated based on the value that was read regardless of this mode setting.

By default, auto-prediction is turned off.

Parameters

`on` (`bit`)

`function bit get_check_on_read()`

Function

get_check_on_read

Gets the check-on-read mode setting for this map.

`function void Xget_bus_infoX(uvm_reg_item rw, uvm_reg_map_info map_info, int size, int lsb, int addr_skip)`

Bus Access

Parameters

- **`rw`** (*uvm_reg_item*)
- **`map_info`** (*uvm_reg_map_info*)
- **`size`** (*`int`*)
- **`lsb`** (*`int`*)
- **`addr_skip`** (*`int`*)

`virtual function string convert2string()`

Convert2string

`virtual function uvm_object clone()`

Clone

Return type

uvm_object

`virtual function void do_print(uvm_printer printer)`

Do_print

Parameters

`printer` (*uvm_printer*)

`virtual function void do_copy(uvm_object rhs)`

Do_copy

Parameters

`rhs` (*uvm_object*)

`function void set_transaction_order_policy(uvm_reg_transaction_order_policy pol)`

Function

set_transaction_order_policy

set the transaction order policy

Parameters

`pol` (*uvm_reg_transaction_order_policy*)

`function uvm_reg_transaction_order_policy get_transaction_order_policy()`

Function

get_transaction_order_policy

set the transaction order policy

Return type

uvm_reg_transaction_order_policy

Tasks

`virtual function do_bus_write(uvm_reg_item rw, uvm_sequencer_base sequencer, uvm_-reg_adapter adapter)`

Task

do_bus_write

Perform a bus write operation. Do_bus_write

Parameters

- **`rw`** (*uvm_reg_item*)
- **`sequencer`** (*uvm_sequencer_base*)
- **`adapter`** (*uvm_reg_adapter*)

`virtual function do_bus_read(uvm_reg_item rw, uvm_sequencer_base sequencer, uvm_-reg_adapter adapter)`

Task

do_bus_read

Perform a bus read operation. Do_bus_read

Parameters

`rw` (*uvm_reg_item*)

`sequencer` (*uvm_sequencer_base*)

`adapter` (*uvm_reg_adapter*)

`virtual function do_write(uvm_reg_item rw)`

Task

do_write

Perform a write operation. Do_write(uvm_reg_item rw)

Parameters

`rw` (*uvm_reg_item*)

`virtual function do_read(uvm_reg_item rw)`

Task

do_read

Perform a read operation. Do_read(uvm_reg_item rw)

Parameters

`rw` (*uvm_reg_item*)

15.1.1.188 Class uvm_pkg::uvm_reg_map_info

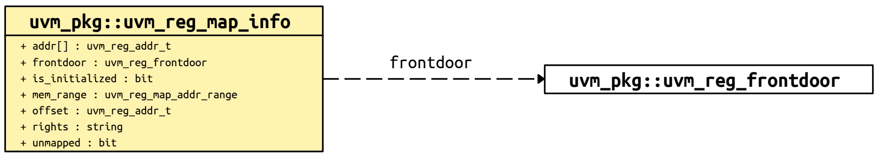

uvm_pkg::uvm_reg_map_info
+ addr[] : uvm_reg_addr_t
+ frontdoor : uvm_reg_frontdoor
+ is_initialized : bit
+ mem_range : uvm_reg_map_addr_range
+ offset : uvm_reg_addr_t
+ rights : string
+ unmapped : bit
frontdoor
uvm_pkg::uvm_reg_frontdoor

Fig. 64: Collaboration Diagram of uvm_reg_map_info

Table 202: Variables

Name	Type	Description
offset	*uvm_reg_addr_t*	
rights	string	
unmapped	bit	
addr	*uvm_reg_addr_t*	
frontdoor	*uvm_reg_frontdoor*	
mem_range	*uvm_reg_map_addr_-range*	
is_initialized	bit	if set marks the uvm_reg_map_info as initialized, prevents using an uninitialized map (for instance if the model has not been locked accidently and the maps have not been computed before)

15.1.1.189 Class uvm_pkg::uvm_reg_mem_access_seq

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*
↪*uvm_pkg* :: *uvm_reg_sequence*
↪*uvm_pkg* :: *uvm_reg_mem_access_seq*

Class

uvm_reg_mem_access_seq

Verify the accessibility of all registers and memories in a block by executing the *uvm_reg_access_seq* and *uvm_mem_access_seq* sequence respectively on every register and memory within it.

Blocks and registers with the NO_REG_TESTS or the NO_REG_ACCESS_TEST attribute are not verified.

Constructors

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

Parameters
`name` (*`string`*)

Tasks

`virtual function body()`

`virtual function reset_blk(uvm_reg_block blk)`

Any additional steps required to reset the block and make it accessibl
Parameters
`blk` (*uvm_reg_block*)

15.1.1.190 Class uvm_pkg::uvm_reg_mem_built_in_seq

uvm_pkg :: *uvm_void*
  ↪*uvm_pkg* :: *uvm_object*
    ↪*uvm_pkg* :: *uvm_transaction*
      ↪*uvm_pkg* :: *uvm_sequence_item*
        ↪*uvm_pkg* :: *uvm_sequence_base*
          ↪*uvm_pkg* :: *uvm_sequence*
            ↪*uvm_pkg* :: *uvm_reg_sequence*
              ↪*uvm_pkg* :: *uvm_reg_mem_built_in_seq*

Class

uvm_reg_mem_built_in_seq

Sequence that executes a user-defined selection of pre-defined register and memory test sequences.

Table 203: Variables

Name	Type	Description
tests	bit[63:0]	***Variable*** tests The pre-defined test sequences to be executed.

Constructors

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

Parameters
  `name` (`string`)

Tasks

`virtual function body()`

Task

body

Executes any or all the built-in register and memory sequences. Do not call directly. Use seq.start() instead.

15.1.1.191 Class uvm_pkg::uvm_reg_mem_hdl_paths_seq

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*
↪*uvm_pkg* :: *uvm_reg_sequence*
↪*uvm_pkg* :: *uvm_reg_mem_hdl_paths_seq*

class

uvm_reg_mem_hdl_paths_seq

Verify the correctness of HDL paths specified for registers and memories.

This sequence is be used to check that the specified backdoor paths are indeed accessible by the simulator. By default, the check is performed for the default design abstraction. If the simulation contains multiple models of the DUT, HDL paths for multiple design abstractions can be checked.

If a path is not accessible by the simulator, it cannot be used for read/write backdoor accesses. In that case a warning is produced. A simulator may have finer-grained access permissions such as separate read or write permissions. These extra access permissions are NOT checked.

The test is performed in zero time and does not require any reads/writes to/from the DUT.

Table 204: Variables

Name	Type	Description
abstractions	string	***Variable*** abstractions If set, check the HDL paths for the specified design abstractions. If empty, check the HDL path for the default design abstraction, as specified with *uvm_reg_block::set_default_hdl_path()*

Constructors

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

Parameters
`name` (`string`)

Tasks

`virtual function body()`

`virtual function reset_blk(uvm_reg_block blk)`

Any additional steps required to reset the block and make it accessible
Parameters
`blk` (*uvm_reg_block*)

15.1.1.192 Class uvm_pkg::uvm_reg_mem_shared_access_seq

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*
↪*uvm_pkg* :: *uvm_reg_sequence*
↪*uvm_pkg* :: *uvm_reg_mem_shared_access_seq*

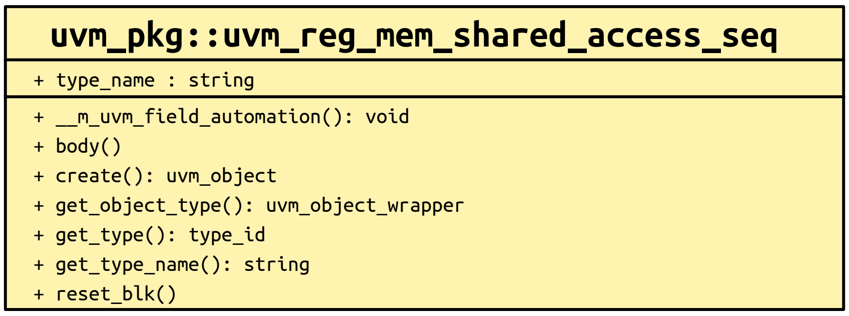

uvm_pkg::uvm_reg_mem_shared_access_seq
+ type_name : string
+ __m_uvm_field_automation(): void
+ body()
+ create(): uvm_object
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string
+ reset_blk()

Fig. 65: Collaboration Diagram of uvm_reg_mem_shared_access_seq

Class

uvm_reg_mem_shared_access_seq

Verify the accessibility of all shared registers and memories in a block by executing the *uvm_reg_shared_access_seq* and *uvm_mem_shared_access_seq* sequence respectively on every register and memory within it.

If bit-type resource named "NO_REG_TESTS", "NO_MEM_TESTS", "NO_REG_SHARED_ACCESS_TEST" or "NO_MEM_SHARED_ACCESS_TEST" in the "REG::" namespace matches the full name of the block, the block is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.get_full_name(),".*"},
                            "NO_REG_TESTS", 1, this);
```

Constructors

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

Parameters
name (*string*)

Tasks

`virtual function body()`

Task

body

Executes the Shared Register and Memory sequence

`virtual function reset_blk(uvm_reg_block blk)`

task

reset_blk

Reset the DUT that corresponds to the specified block abstraction class.

Currently empty. Will rollback the environment's phase to the *reset* phase once the new phasing is available.

In the meantime, the DUT should be reset before executing this test sequence or this method should be implemented in an extension to reset the DUT.

Parameters

`blk` (*uvm_reg_block*)

15.1.1.193 Class uvm_pkg::uvm_reg_predictor

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_reg_predictor*

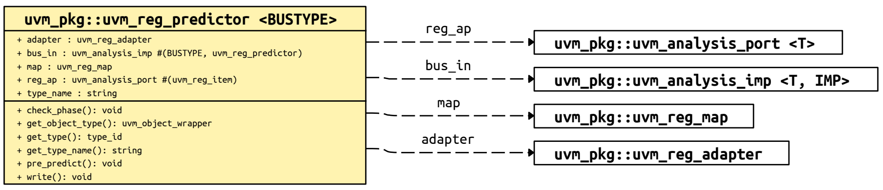

uvm_pkg::uvm_reg_predictor <BUSTYPE>
+ adapter : uvm_reg_adapter
+ bus_in : uvm_analysis_imp #(BUSTYPE, uvm_reg_predictor)
+ map : uvm_reg_map
+ reg_ap : uvm_analysis_port #(uvm_reg_item)
+ type_name : string
+ check_phase(): void
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string
+ pre_predict(): void
+ write(): void
reg_ap
bus_in
map
adapter
uvm_pkg::uvm_analysis_port <T>
uvm_pkg::uvm_analysis_imp <T, IMP>
uvm_pkg::uvm_reg_map
uvm_pkg::uvm_reg_adapter

Fig. 66: Collaboration Diagram of uvm_reg_predictor

CLASS

uvm_reg_predictor

Updates the register model mirror based on observed bus transactions

This class converts observed bus transactions of type *BUSTYPE* to generic registers transactions, determines the register being accessed based on the bus address, then updates the register's mirror value with the observed bus data, subject to the register's access mode. See *uvm_reg::predict* for details.

Memories can be large, so their accesses are not predicted.

Table 205: Parameters

Name	Default value	Description
BUSTYPE	int	

Table 206: Variables

Name	Type	Description
bus_in	*uvm_analysis_imp#(int, uvm_reg_predictor#(int))*	***Variable*** bus_in Observed bus transactions of type *BUSTYPE* are received from this port and processed. For each incoming transaction, the predictor will attempt to get the register or memory handle corresponding to the observed bus address. If there is a match, the predictor calls the register or memory's predict method, passing in the observed bus data. The register or memory mirror will be updated with this data, subject to its configured access behavior--RW, RO, WO, etc. The predictor will also convert the bus transaction to a generic *uvm_reg_item* and send it out the *reg_ap* analysis port. If the register is wider than the bus, the predictor will collect the multiple bus transactions needed to determine the value being read or written.
reg_ap	*uvm_analysis_-port#(uvm_reg_item)*	***Variable*** reg_ap Analysis output port that publishes *uvm_reg_item* transactions converted from bus transactions received on *bus_in* .
map	*uvm_reg_map*	***Variable*** map The map used to convert a bus address to the corresponding register or memory handle. Must be configured before the run phase.
adapter	*uvm_reg_adapter*	***Variable*** adapter The adapter used to convey the parameters of a bus operation in terms of a canonical *uvm_reg_bus_op* datum. The *uvm_reg_adapter* must be configured before the run phase.
type_name	string	This method is documented in uvm_object

Constructors

`function new(string name, uvm_component parent)`

Function

new

Create a new instance of this type, giving it the optional *name* and *parent* .

Parameters

`name` (`string`)
`parent` (*uvm_component*)

Functions

`virtual function string get_type_name()`

`virtual function void pre_predict(uvm_reg_item rw)`

Function

pre_predict

Override this method to change the value or re-direct the target register

Parameters

`rw` (*uvm_reg_item*)

`virtual function void write(int tr)`

Function- write

not a user-level method. Do not call directly. See documentation for the *bus_in* member.

Parameters

`tr` (*`int`*)

`virtual function void check_phase(uvm_phase phase)`

Function

check_phase

Checks that no pending register transactions are still queued.

Parameters

`phase` (*uvm_phase*)

15.1.1.194 Class uvm_pkg::uvm_reg_read_only_cbs

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_callback*
↪*uvm_pkg* :: *uvm_reg_cbs*
↪*uvm_pkg* :: *uvm_reg_read_only_cbs*

Class

uvm_reg_read_only_cbs

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

Constructors

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

Parameters
`name` (*`string`*)

Functions

`static function void add(uvm_reg rg)`

Function

add

Add this callback to the specified register and its contained fields.
Parameters
`rg` (*uvm_reg*)

`static function void remove(uvm_reg rg)`

Function

remove

Remove this callback from the specified register and its contained fields.
Parameters
`rg` (*uvm_reg*)

Tasks

`virtual function pre_write(uvm_reg_item rw)`

Function

pre_write

Produces an error message and sets status to <UVM_NOT_OK>.
Parameters
`rw` (*uvm_reg_item*)

15.1.1.195 Class uvm_pkg::uvm_reg_sequence

uvm_pkg :: *uvm_void*
 ↪*uvm_pkg* :: *uvm_object*
  ↪*uvm_pkg* :: *uvm_transaction*
   ↪*uvm_pkg* :: *uvm_sequence_item*
    ↪*uvm_pkg* :: *uvm_sequence_base*
     ↪*uvm_pkg* :: *uvm_sequence*
      ↪*uvm_pkg* :: *uvm_reg_sequence*

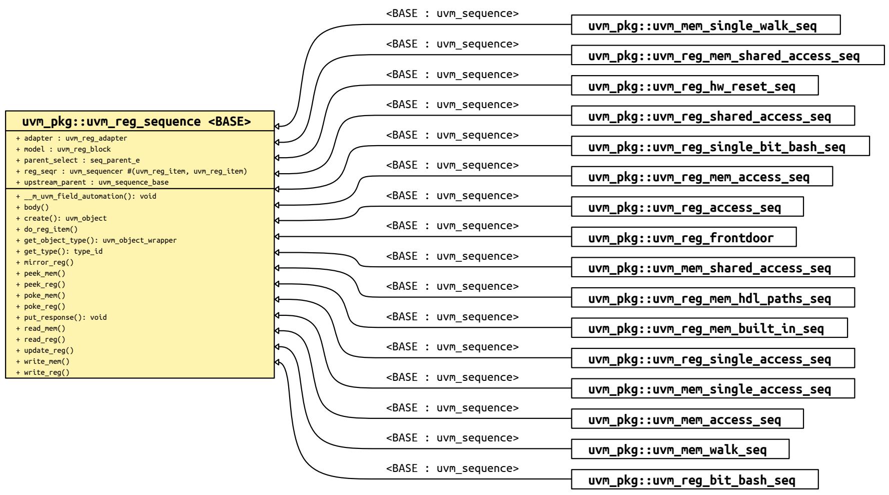

uvm_pkg::uvm_reg_sequence <BASE>
<BASE : uvm_sequence>
uvm_pkg::uvm_mem_single_walk_seq
uvm_pkg::uvm_reg_mem_shared_access_seq
uvm_pkg::uvm_reg_hw_reset_seq
uvm_pkg::uvm_reg_shared_access_seq
uvm_pkg::uvm_reg_single_bit_bash_seq
uvm_pkg::uvm_reg_mem_access_seq
uvm_pkg::uvm_reg_access_seq
uvm_pkg::uvm_reg_frontdoor
uvm_pkg::uvm_mem_shared_access_seq
uvm_pkg::uvm_reg_mem_hdl_paths_seq
uvm_pkg::uvm_reg_mem_built_in_seq
uvm_pkg::uvm_reg_single_access_seq
uvm_pkg::uvm_mem_single_access_seq
uvm_pkg::uvm_mem_access_seq
uvm_pkg::uvm_mem_walk_seq
uvm_pkg::uvm_reg_bit_bash_seq

Fig. 67: Inheritance Diagram of uvm_reg_sequence

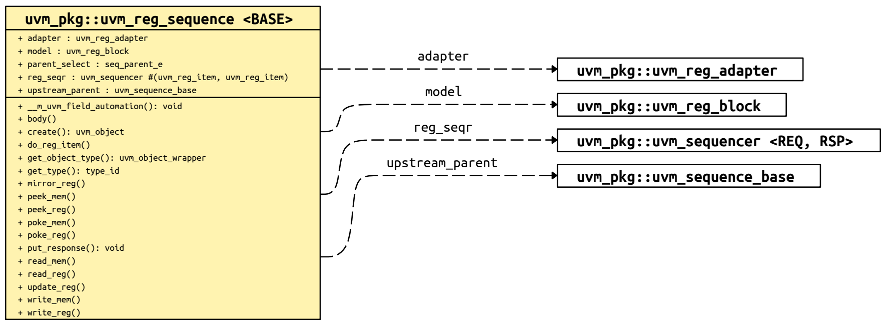


Fig. 68: Collaboration Diagram of uvm_reg_sequence

CLASS

uvm_reg_sequence

This class provides base functionality for both user-defined RegModel test sequences and "register translation sequences".

When used as a base for user-defined RegModel test sequences, this class provides convenience methods for reading and writing registers and memories. Users implement the body() method to interact directly with the RegModel model (held in the *model* property) or indirectly via the delegation methods in this class.
When used as a translation sequence, objects of this class are executed directly on a bus sequencer which are used in support of a layered sequencer use model, a pre-defined convert-and-execute algorithm is provided.

Register operations do not require extending this class if none of the above services are needed. Register test sequences can be extend from the base <uvm_sequence (REQ, RSP)> base class or even from outside a sequence.

Note- The convenience API not yet implemented.

Table 207: Parameters

Name	Default value	Description
BASE	uvm_sequence	

Table 208: Variables

Name	Type	Description
model	*uvm_reg_block*	***Variable*** model Block abstraction this sequence executes on, defined only when this sequence is a user-defined test sequence.
adapter	*uvm_reg_adapter*	***Variable*** adapter Adapter to use for translating between abstract register transactions and physical bus transactions, defined only when this sequence is a translation sequence.
reg_seqr	*uvm_sequencer#(uvm_-reg_item, uvm_reg_item)*	***Variable*** reg_seqr Layered upstream "register" sequencer. Specifies the upstream sequencer between abstract register transactions and physical bus transactions. Defined only when this sequence is a translation sequence, and we want to "pull" from an upstream sequencer.
parent_select	*seq_parent_e*	
upstream_parent	*uvm_sequence_base*	

Constructors

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

Function

new

Create a new instance, giving it the optional *name* .

Parameters

`name` (*`string`*)

Enums

`seq_parent_e`

Enum Items

LOCAL
UPSTREAM

Functions

`virtual function void put_response(uvm_sequence_item response_item)`

Function- put_response

not user visible. Needed to populate this sequence's response queue with any bus item type.

Parameters

`response_item` (*uvm_sequence_item*)

Tasks

`virtual function body()`

Task

body

Continually gets a register transaction from the configured upstream sequencer, *reg_seqr*, and executes the corresponding bus transaction via *do_reg_item*.

User-defined RegModel test sequences must override body() and not call super.body(), else a warning will be issued and the calling process not return.

`virtual function do_reg_item(uvm_reg_item rw)`

Function

do_reg_item

Executes the given register transaction, *rw* , via the sequencer on which this sequence was started (i.e. m_sequencer). Uses the configured *adapter* to convert the register transaction into the type expected by this sequencer.

Parameters

`rw` (*uvm_reg_item*)

`virtual function write_reg(uvm_reg rg, uvm_status_e status, uvm_reg_data_t value, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)`

Task

write_reg

Writes the given register *rg* using *uvm_reg::write*, supplying 'this' as the *parent* argument. Thus,

```
write_reg(model.regA, status, value);
```

is equivalent to

```
model.regA.write(status, value, .parent(this));
```

Parameters

`rg` (*uvm_reg*)
`status` (*uvm_status_e*)
`value` (*uvm_reg_data_t*)
`path` (*uvm_path_e*)
`map` (*uvm_reg_map*)
`prior` (`int`)
`extension` (*uvm_object*)
`fname` (`string`)
`lineno` (`int`)

`virtual function read_reg(uvm_reg rg, uvm_status_e status, uvm_reg_data_t value, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)`

Task

read_reg

Reads the given register *rg* using *uvm_reg::read*, supplying 'this' as the *parent* argument. Thus,

```
read_reg(model.regA, status, value);
```

is equivalent to

```
model.regA.read(status, value, .parent(this));
```

Parameters

- **rg** (*uvm_reg*)
- **status** (*uvm_status_e*)
- **value** (*uvm_reg_data_t*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **prior** (*int*)
- **extension** (*uvm_object*)
- **fname** (*string*)
- **lineno** (*int*)

virtual function poke_reg(uvm_reg rg, uvm_status_e status, uvm_reg_data_t value, string kind = "", uvm_object extension = null, string fname = "", int lineno = 0)

Task

poke_reg

Pokes the given register *rg* using *uvm_reg::poke*, supplying 'this' as the *parent* argument. Thus,

```
poke_reg(model.regA, status, value);
```

is equivalent to

```
model.regA.poke(status, value, .parent(this));
```

Parameters

- **rg** (*uvm_reg*)
- **status** (*uvm_status_e*)
- **value** (*uvm_reg_data_t*)
- **kind** (*string*)
- **extension** (*uvm_object*)
- **fname** (*string*)
- **lineno** (*int*)

virtual function peek_reg(uvm_reg rg, uvm_status_e status, uvm_reg_data_t value, string kind = "", uvm_object extension = null, string fname = "", int lineno = 0)

Task

peek_reg

Peeks the given register *rg* using *uvm_reg::peek*, supplying 'this' as the *parent* argument. Thus,

```
peek_reg(model.regA, status, value);
```

is equivalent to

```
model.regA.peek(status, value, .parent(this));
```

Parameters

- **rg** (*uvm_reg*)
- **status** (*uvm_status_e*)
- **value** (*uvm_reg_data_t*)
- **kind** (*string*)
- **extension** (*uvm_object*)
- **fname** (*string*)
- **lineno** (*int*)

virtual function update_reg(uvm_reg rg, uvm_status_e status, uvm_path_-e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, int prior = -1, uvm_-object extension = null, string fname = "", int lineno = 0)

Task

update_reg

Updates the given register *rg* using *uvm_reg::update*, supplying 'this' as the *parent* argument. Thus,

```
update_reg(model.regA, status, value);
```

is equivalent to

```
model.regA.update(status, value, .parent(this));
```

Parameters
- **rg** (*uvm_reg*)
- **status** (*uvm_status_e*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **prior** (*int*)
- **extension** (*uvm_object*)
- **fname** (*string*)
- **lineno** (*int*)

virtual function mirror_reg(uvm_reg rg, uvm_status_e status, uvm_check_e check = UVM_NO_CHECK, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)

Task

mirror_reg

Mirrors the given register *rg* using *uvm_reg::mirror*, supplying 'this' as the *parent* argument. Thus,

```
mirror_reg(model.regA, status, UVM_CHECK);
```

is equivalent to

```
model.regA.mirror(status, UVM_CHECK, .parent(this));
```

Parameters
- **rg** (*uvm_reg*)
- **status** (*uvm_status_e*)
- **check** (*uvm_check_e*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **prior** (*int*)
- **extension** (*uvm_object*)
- **fname** (*string*)
- **lineno** (*int*)

virtual function write_mem(uvm_mem mem, uvm_status_e status, uvm_reg_addr_t offset, uvm_reg_data_t value, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)

Task

write_mem

Writes the given memory *mem* using *uvm_mem::write*, supplying 'this' as the *parent* argument. Thus,

```
write_mem(model.regA, status, offset, value);
```

is equivalent to

```
model.regA.write(status, offset, value, .parent(this));
```

Parameters
- **mem** (*uvm_mem*)
- **status** (*uvm_status_e*)

offset (*uvm_reg_addr_t*)
value (*uvm_reg_data_t*)
path (*uvm_path_e*)
map (*uvm_reg_map*)
prior (`int`)
extension (*uvm_object*)
fname (`string`)
lineno (`int`)

`virtual function read_mem(uvm_mem mem, uvm_status_e status, uvm_reg_addr_t offset, uvm_reg_data_t value, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, int prior = -1, uvm_object extension = null, string fname = "", int lineno = 0)`

Task

read_mem

Reads the given memory *mem* using *uvm_mem::read*, supplying 'this' as the *parent* argument. Thus,

```
read_mem(model.regA, status, offset, value);
```

is equivalent to

```
model.regA.read(status, offset, value, .parent(this));
```

Parameters

mem (*uvm_mem*)
status (*uvm_status_e*)
offset (*uvm_reg_addr_t*)
value (*uvm_reg_data_t*)
path (*uvm_path_e*)
map (*uvm_reg_map*)
prior (`int`)
extension (*uvm_object*)
fname (`string`)
lineno (`int`)

`virtual function poke_mem(uvm_mem mem, uvm_status_e status, uvm_reg_addr_t offset, uvm_reg_data_t value, string kind = "", uvm_object extension = null, string fname = "", int lineno = 0)`

Task

poke_mem

Pokes the given memory *mem* using *uvm_mem::poke*, supplying 'this' as the *parent* argument. Thus,

```
poke_mem(model.regA, status, offset, value);
```

is equivalent to

```
model.regA.poke(status, offset, value, .parent(this));
```

Parameters

mem (*uvm_mem*)
status (*uvm_status_e*)
offset (*uvm_reg_addr_t*)
value (*uvm_reg_data_t*)
kind (`string`)
extension (*uvm_object*)
fname (`string`)
lineno (`int`)

`virtual function peek_mem(uvm_mem mem, uvm_status_e status, uvm_reg_addr_t offset, uvm_reg_data_t value, string kind = "", uvm_object extension = null, string fname = "", int lineno = 0)`

Task

peek_mem

Peeks the given memory *mem* using *uvm_mem::peek*, supplying 'this' as the *parent* argument. Thus,

```
peek_mem(model.regA, status, offset, value);
```

is equivalent to

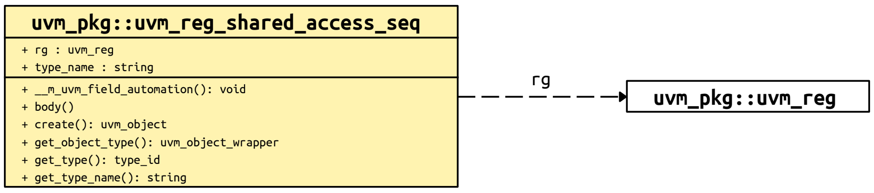
```
model.regA.peek(status, offset, value, .parent(this));
```

Parameters

- **`mem`** (*uvm_mem*)
- **`status`** (*uvm_status_e*)
- **`offset`** (*uvm_reg_addr_t*)
- **`value`** (*uvm_reg_data_t*)
- **`kind`** (*`string`*)
- **`extension`** (*uvm_object*)
- **`fname`** (*`string`*)
- **`lineno`** (*`int`*)

15.1.1.196 Class uvm_pkg::uvm_reg_shared_access_seq

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*
↪*uvm_pkg* :: *uvm_reg_sequence*
↪*uvm_pkg* :: *uvm_reg_shared_access_seq*



uvm_pkg::uvm_reg_shared_access_seq
+ rg : uvm_reg
+ type_name : string
+ __m_uvm_field_automation(): void
+ body()
+ create(): uvm_object
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string
rg
uvm_pkg::uvm_reg

Fig. 69: Collaboration Diagram of uvm_reg_shared_access_seq

Class

uvm_reg_shared_access_seq

Verify the accessibility of a shared register by writing through each address map then reading it via every other address maps in which the register is readable and the backdoor, making sure that the resulting value matches the mirrored value.

If bit-type resource named "NO_REG_TESTS" or "NO_REG_SHARED_ACCESS_TEST" in the "REG::" namespace matches the full name of the register, the register is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.r0.get_full_name()},
                           "NO_REG_TESTS", 1, this);
```

Registers that contain fields with unknown access policies cannot be tested.

The DUT should be idle and not modify any register during this test.

Table 209: Variables

Name	Type	Description
rg	*uvm_reg*	***Variable*** rg The register to be tested

Constructors

function new(string name = "uvm_reg_shared_access_seq")

Parameters
name (*string*)

Tasks

`virtual function body()`

15.1.1.197 Class uvm_pkg::uvm_reg_single_access_seq

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*
↪*uvm_pkg* :: *uvm_reg_sequence*
↪*uvm_pkg* :: *uvm_reg_single_access_seq*

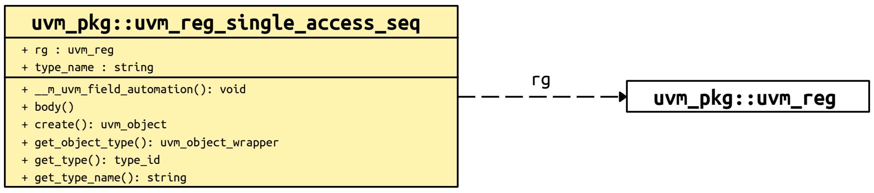

uvm_pkg::uvm_reg_single_access_seq
+ rg : uvm_reg
+ type_name : string
+ __m_uvm_field_automation(): void
+ body()
+ create(): uvm_object
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string
rg
uvm_pkg::uvm_reg

Fig. 70: Collaboration Diagram of uvm_reg_single_access_seq

Class

uvm_reg_single_access_seq

Verify the accessibility of a register by writing through its default address map then reading it via the backdoor, then reversing the process, making sure that the resulting value matches the mirrored value.

If bit-type resource named "NO_REG_TESTS" or "NO_REG_ACCESS_TEST" in the "REG::" namespace matches the full name of the register, the register is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.r0.get_full_name()},
                            "NO_REG_TESTS", 1, this);
```

Registers without an available backdoor or that contain read-only fields only, or fields with unknown access policies cannot be tested.

The DUT should be idle and not modify any register during this test.

Table 210: Variables

Name	Type	Description
rg	*uvm_reg*	***Variable*** rg The register to be tested

Constructors

function new(string name = "uvm_reg_single_access_seq")

Parameters
name (*string*)

Tasks

`virtual function body()`

15.1.1.198 Class uvm_pkg::uvm_reg_single_bit_bash_seq

uvm_pkg :: *uvm_void*
  ↪*uvm_pkg* :: *uvm_object*
    ↪*uvm_pkg* :: *uvm_transaction*
      ↪*uvm_pkg* :: *uvm_sequence_item*
        ↪*uvm_pkg* :: *uvm_sequence_base*
          ↪*uvm_pkg* :: *uvm_sequence*
            ↪*uvm_pkg* :: *uvm_reg_sequence*
              ↪*uvm_pkg* :: *uvm_reg_single_bit_bash_seq*

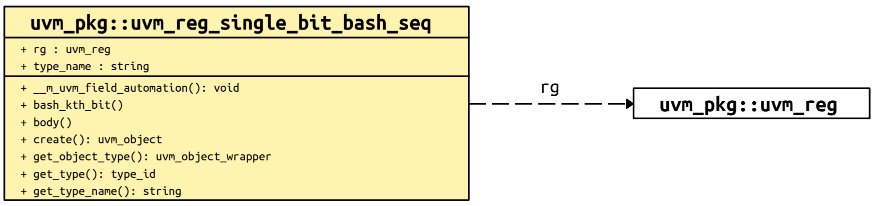

uvm_pkg::uvm_reg_single_bit_bash_seq
+ rg : uvm_reg
+ type_name : string
+ __m_uvm_field_automation(): void
+ bash_kth_bit()
+ body()
+ create(): uvm_object
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string
rg
uvm_pkg::uvm_reg

Fig. 71: Collaboration Diagram of uvm_reg_single_bit_bash_seq

Class

uvm_reg_single_bit_bash_seq

Verify the implementation of a single register by attempting to write 1's and 0's to every bit in it, via every address map in which the register is mapped, making sure that the resulting value matches the mirrored value.

If bit-type resource named "NO_REG_TESTS" or "NO_REG_BIT_BASH_TEST" in the "REG::" namespace matches the full name of the register, the register is not tested.

```
uvm_resource_db#(bit)::set({"REG::",regmodel.blk.r0.get_full_name()},
                           "NO_REG_TESTS", 1, this);
```

Registers that contain fields with unknown access policies cannot be tested.

The DUT should be idle and not modify any register during this test.

Table 211: Variables

Name	Type	Description
rg	*uvm_reg*	***Variable*** rg The register to be tested

Constructors

function new(string name = "uvm_reg_single_bit_bash_seq")

Parameters
  name (*string*)

Tasks

virtual function body()

function bash_kth_bit(uvm_reg rg, int k, string mode, uvm_reg_map map, uvm_reg_-data_t dc_mask)

Parameters

`rg` (*uvm_reg*)
`k` (*`int`*)
`mode` (*`string`*)
`map` (*uvm_reg_map*)
`dc_mask` (*uvm_reg_data_t*)

15.1.1.199 Class uvm_pkg::uvm_reg_tlm_adapter

uvm_pkg :: *uvm_void*
  ↪*uvm_pkg* :: *uvm_object*
    ↪*uvm_pkg* :: *uvm_reg_adapter*
      ↪*uvm_pkg* :: *uvm_reg_tlm_adapter*

Class

uvm_reg_tlm_adapter

For converting between *uvm_reg_bus_op* and *uvm_tlm_gp* items.

Constructors

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

Parameters
: **`name`** (*`string`*)

Functions

`virtual function uvm_sequence_item reg2bus(uvm_reg_bus_op rw)`

Function

reg2bus

Converts a *uvm_reg_bus_op* struct to a *uvm_tlm_gp* item.

Parameters
: **`rw`** (*uvm_reg_bus_op*)

Return type
: *uvm_sequence_item*

`virtual function void bus2reg(uvm_sequence_item bus_item, uvm_reg_bus_op rw)`

Function

bus2reg

Converts a *uvm_tlm_gp* item to a *uvm_reg_bus_op*. into the provided *rw* transaction.

Parameters
: **`bus_item`** (*uvm_sequence_item*)
: **`rw`** (*uvm_reg_bus_op*)

15.1.1.200 Class uvm_pkg::uvm_reg_transaction_order_policy

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_reg_transaction_order_policy*

Class

uvm_reg_transaction_order_policy

Constructors

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

Parameters
`name` (*`string`*)

Functions

`virtual function void order(uvm_reg_bus_op q)`

Function

order

the order() function may reorder the sequence of bus transactions produced by a single uvm_reg transaction (read/write). This can be used in scenarios when the register width differs from the bus width and one register access results in a series of bus transactions. the first item (0) of the queue will be the first bus transaction (the last($) will be the final transaction

Parameters
`q` (*uvm_reg_bus_op*)

15.1.1.201 Class uvm_pkg::uvm_reg_write_only_cbs

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_callback*
↪*uvm_pkg* :: *uvm_reg_cbs*
↪*uvm_pkg* :: *uvm_reg_write_only_cbs*

Class

uvm_reg_write_only_cbs

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

Constructors

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

Parameters
`name` (*`string`*)

Functions

`static function void add(uvm_reg rg)`

Function

add

Add this callback to the specified register and its contained fields.
Parameters
`rg` (*uvm_reg*)

`static function void remove(uvm_reg rg)`

Function

remove

Remove this callback from the specified register and its contained fields.
Parameters
`rg` (*uvm_reg*)

Tasks

`virtual function pre_read(uvm_reg_item rw)`

Function

pre_read

Produces an error message and sets status to <UVM_NOT_OK>.
Parameters
`rw` (*uvm_reg_item*)

15.1.1.202 Class uvm_pkg::uvm_related_link

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_link_base*
↪*uvm_pkg* :: *uvm_related_link*

CLASS

uvm_related_link

The *uvm_related_link* is used to represent a generic "is related" link between two objects.

Constructors

`function new(string name = "unnamed-uvm_related_link")`

Function

new

Constructor

Parameters

name

Instance name

Parameters

`name` (*`string`*)

Functions

`static function uvm_related_link get_link(uvm_object lhs, uvm_object rhs, string name = "ce_link")`

Function

get_link

Constructs a pre-filled link

This allows for simple one-line link creations.

```
my_db.establish_link(uvm_related_link::get_link(record1, record2));
```

Parameters:

lhs

Left hand side reference

rhs

Right hand side reference

name

Optional name for the link object

Parameters

`lhs` (*uvm_object*)

`rhs` (*uvm_object*)

`name` (*`string`*)

Return type

uvm_related_link

`virtual function void do_set_lhs(uvm_object lhs)`

Function

do_set_lhs

Sets the left-hand-side

Parameters

`lhs` (*uvm_object*)

`virtual function uvm_object do_get_lhs()`

Function

do_get_lhs

Retrieves the left-hand-side

Return type

uvm_object

`virtual function void do_set_rhs(uvm_object rhs)`

Function

do_set_rhs

Sets the right-hand-side

Parameters

`rhs` (*uvm_object*)

`virtual function uvm_object do_get_rhs()`

Function

do_get_rhs

Retrieves the right-hand-side

Return type

uvm_object

15.1.1.203 Class uvm_pkg::uvm_report_message_element_base

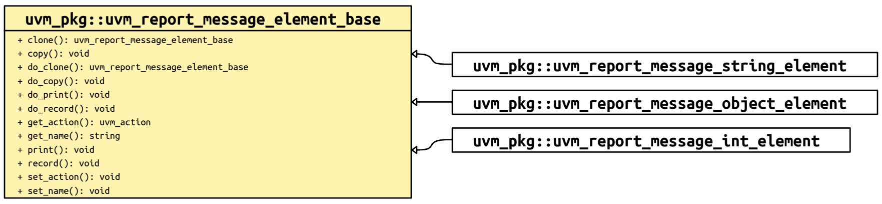

uvm_pkg::uvm_report_message_element_base
+ clone(): uvm_report_message_element_base
+ copy(): void
+ do_clone(): uvm_report_message_element_base
+ do_copy(): void
+ do_print(): void
+ do_record(): void
+ get_action(): uvm_action
+ get_name(): string
+ print(): void
+ record(): void
+ set_action(): void
+ set_name(): void
uvm_pkg::uvm_report_message_string_element
uvm_pkg::uvm_report_message_object_element
uvm_pkg::uvm_report_message_int_element

Fig. 72: Inheritance Diagram of uvm_report_message_element_base

CLASS

uvm_report_message_element_base

Base class for report message element. Defines common interface.

Functions

`virtual function string get_name()`

Function

get_name

`virtual function void set_name(string name)`

Function

set_name

Get or set the name of the element

Parameters
name (*string*)

`virtual function uvm_action get_action()`

Function

get_action

Return type
uvm_action

`virtual function void set_action(uvm_action action)`

Function

set_action

Get or set the authorized action for the element

Parameters
action (*uvm_action*)

`function void print(uvm_printer printer)`

Parameters
printer (*uvm_printer*)

`function void record(uvm_recorder recorder)`

Parameters
recorder (*uvm_recorder*)

`function void copy(uvm_report_message_element_base rhs)`

Parameters
rhs (*uvm_report_message_element_base*)

`function uvm_report_message_element_base clone()`

Return type
uvm_report_message_element_base

`virtual function void do_print(uvm_printer printer)`

Parameters
: **`printer`** (*uvm_printer*)

`virtual function void do_record(uvm_recorder recorder)`

Parameters
: **`recorder`** (*uvm_recorder*)

`virtual function void do_copy(uvm_report_message_element_base rhs)`

Parameters
: **`rhs`** (*uvm_report_message_element_base*)

`virtual function uvm_report_message_element_base do_clone()`

Return type
: *uvm_report_message_element_base*

15.1.1.204 Class uvm_pkg::uvm_report_message_element_container

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_message_element_container*

CLASS

uvm_report_message_element_container

A container used by report message to contain the dynamically added elements, with APIs to add and delete the elements.

Table 212: Typedefs

Name	Actual Type	Description
queue_of_element	*uvm_report_message_element_base*	Function: get_elements Get all the elements from the container and put them in a queue

Constructors

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

Function

new

Create a new uvm_report_message_element_container object
Parameters
`name` (*`string`*)

Functions

`virtual function int size()`

Function

size

Returns the size of the container, i.e. the number of elements

`virtual function void delete(int index)`

Function

delete

Delete the *index* -th element in the container
Parameters
`index` (*`int`*)

`virtual function void delete_elements()`

Function

delete_elements

Delete all the elements in the container

`virtual function queue_of_element get_elements()`

Return type
queue_of_element

`virtual function void add_int(string name, uvm_bitstream_t value, int size, uvm_radix_enum radix, uvm_action action = (UVM_LOG|UVM_RM_RECORD))`

Function

add_int

This method adds an integral type of the name *name* and value *value* to the container. The required *size* field indicates the size of *value* . The required *radix* field determines how to display and record the field. The optional print/record bit is to specify whether the element will be printed/recorded.

Parameters

- **name** (*string*)
- **value** (*uvm_bitstream_t*)
- **size** (*int*)
- **radix** (*uvm_radix_enum*)
- **action** (*uvm_action*)

virtual function void add_string(string name, string value, uvm_-action action = (UVM_LOG|UVM_RM_RECORD))

Function

add_string

This method adds a string of the name *name* and value *value* to the message. The optional print/record bit is to specify whether the element will be printed/recorded.

Parameters

- **name** (*string*)
- **value** (*string*)
- **action** (*uvm_action*)

virtual function void add_object(string name, uvm_object obj, uvm_-action action = (UVM_LOG|UVM_RM_RECORD))

Function

add_object

This method adds a uvm_object of the name *name* and reference *obj* to the message. The optional print/record bit is to specify whether the element will be printed/recorded.

Parameters

- **name** (*string*)
- **obj** (*uvm_object*)
- **action** (*uvm_action*)

virtual function void do_print(uvm_printer printer)

Parameters

- **printer** (*uvm_printer*)

virtual function void do_record(uvm_recorder recorder)

Parameters

- **recorder** (*uvm_recorder*)

virtual function void do_copy(uvm_object rhs)

Parameters

- **rhs** (*uvm_object*)

15.1.1.205 Class uvm_pkg::uvm_report_message_int_element

uvm_pkg :: *uvm_report_message_element_base*
↪*uvm_pkg* :: *uvm_report_message_int_element*

CLASS

uvm_report_message_int_element

Message element class for integral type

Table 213: Typedefs

Name	Actual Type	Description
this_type	*uvm_report_message_-int_element*	

Functions

`virtual function uvm_bitstream_t get_value(int size, uvm_radix_enum radix)`

Function

get_value

Parameters
- **`size`** (`int`)
- **`radix`** (*uvm_radix_enum*)

Return type
- *uvm_bitstream_t*

`virtual function void set_value(uvm_bitstream_t value, int size, uvm_radix_-enum radix)`

Function

set_value

Get or set the value (integral type) of the element, with size and radix

Parameters
- **`value`** (*uvm_bitstream_t*)
- **`size`** (`int`)
- **`radix`** (*uvm_radix_enum*)

`virtual function void do_print(uvm_printer printer)`

Parameters
- **`printer`** (*uvm_printer*)

`virtual function void do_record(uvm_recorder recorder)`

Parameters
- **`recorder`** (*uvm_recorder*)

`virtual function void do_copy(uvm_report_message_element_base rhs)`

Parameters
- **`rhs`** (*uvm_report_message_element_base*)

`virtual function uvm_report_message_element_base do_clone()`

Return type
- *uvm_report_message_element_base*

15.1.1.206 Class uvm_pkg::uvm_report_message_object_element

uvm_pkg :: *uvm_report_message_element_base*
↪*uvm_pkg* :: *uvm_report_message_object_element*

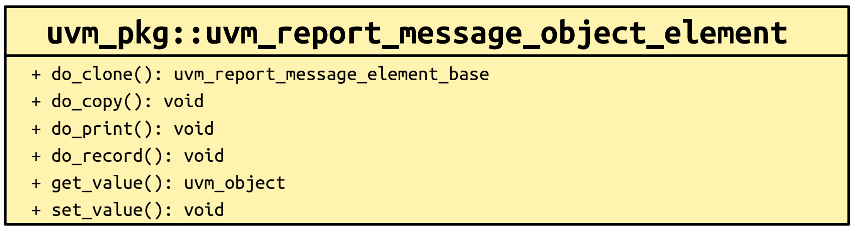

uvm_pkg::uvm_report_message_object_element
+ do_clone(): uvm_report_message_element_base
+ do_copy(): void
+ do_print(): void
+ do_record(): void
+ get_value(): uvm_object
+ set_value(): void

Fig. 73: Collaboration Diagram of uvm_report_message_object_element

CLASS

uvm_report_message_object_element

Message element class for object type

Table 214: Typedefs

Name	Actual Type	Description
this_type	*uvm_report_message_object_element*	

Functions

`virtual function uvm_object get_value()`

Function

get_value

Get the value (object reference) of the element

Return type
uvm_object

`virtual function void set_value(uvm_object value)`

Function

set_value

Get or set the value (object reference) of the element

Parameters
`value` (*uvm_object*)

`virtual function void do_print(uvm_printer printer)`

Parameters
`printer` (*uvm_printer*)

`virtual function void do_record(uvm_recorder recorder)`

Parameters
`recorder` (*uvm_recorder*)

`virtual function void do_copy(uvm_report_message_element_base rhs)`

Parameters
`rhs` (*uvm_report_message_element_base*)

`virtual function uvm_report_message_element_base do_clone()`

Return type
: *uvm_report_message_element_base*

15.1.1.207 Class uvm_pkg::uvm_report_message_string_element

uvm_pkg :: *uvm_report_message_element_base*
↪*uvm_pkg* :: *uvm_report_message_string_element*

CLASS

uvm_report_message_string_element

Message element class for string type

Table 215: Typedefs

Name	Actual Type	Description
this_type	*uvm_report_message_-string_element*	

Functions

`virtual function string get_value()`

Function

get_value

`virtual function void set_value(string value)`

Function

set_value

Get or set the value (string type) of the element

Parameters
`value` (*`string`*)

`virtual function void do_print(uvm_printer printer)`

Parameters
`printer` (*uvm_printer*)

`virtual function void do_record(uvm_recorder recorder)`

Parameters
`recorder` (*uvm_recorder*)

`virtual function void do_copy(uvm_report_message_element_base rhs)`

Parameters
`rhs` (*uvm_report_message_element_base*)

`virtual function uvm_report_message_element_base do_clone()`

Return type
uvm_report_message_element_base

15.1.1.208 Class uvm_pkg::uvm_report_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_bottomup_phase*
↪*uvm_pkg* :: *uvm_report_phase*

Class

uvm_report_phase

Report results of the test.

uvm_bottomup_phase that calls the *uvm_component::report_phase* method.

Upon Entry

- Test is known to have passed or failed.

Typical Uses

Report test results.
Write results to file.

Exit Criteria

- End of test.

Table 216: Variables

Name	Type	Description
type_name	string	

Functions

`virtual function void exec_func(uvm_component comp, uvm_phase phase)`

Parameters
`comp` (*uvm_component*)
`phase` (*uvm_phase*)

`static function uvm_report_phase get()`

Function

get

Returns the singleton phase handle
Return type
uvm_report_phase

`virtual function string get_type_name()`

15.1.1.209 Class uvm_pkg::uvm_reset_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_task_phase*
↪*uvm_pkg* :: *uvm_reset_phase*

Class

uvm_reset_phase

Reset is asserted.

uvm_task_phase that calls the *uvm_component::reset_phase* method.

Upon Entry

- Indicates that the hardware reset signal is ready to be asserted.

Typical Uses

Assert reset signals.
Components connected to virtual interfaces should drive their output to their specified reset or idle value.
Components and environments should initialize their state variables.
Clock generators start generating active edges.
De-assert the reset signal(s) just before exit.
Wait for the reset signal(s) to be de-asserted.

Exit Criteria

Reset signal has just been de-asserted.
Main or base clock is working and stable.
At least one active clock edge has occurred.
Output signals and state variables have been initialized.

Table 217: Variables

Name	Type	Description
type_name	string	

Functions

`static function uvm_reset_phase get()`

Function

get

Returns the singleton phase handle

Return type
uvm_reset_phase

`virtual function string get_type_name()`

Tasks

`virtual function exec_task(uvm_component comp, uvm_phase phase)`

Parameters
`comp` (*uvm_component*)
`phase` (*uvm_phase*)

15.1.1.210 Class uvm_pkg::uvm_resource

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_resource_base*
↪*uvm_pkg* :: *uvm_resource*

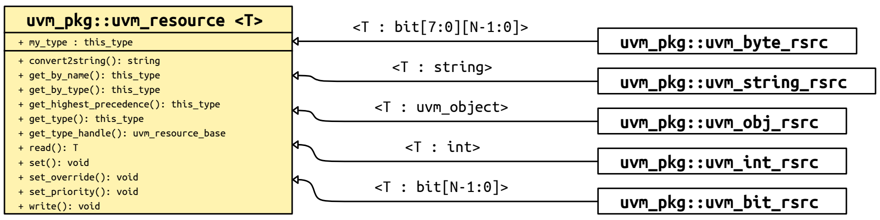


Fig. 74: Inheritance Diagram of uvm_resource

Class

uvm_resource (T)

Parameterized resource. Provides essential access methods to read from and write to the resource database.

Table 218: Parameters

Name	Default value	Description
T	int	

Table 219: Variables

Name	Type	Description
my_type	*this_type*	singleton handle that represents the type of this resource

Table 220: Typedefs

Name	Actual Type	Description
this_type	*uvm_resource#(T)*	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`scope`** (*`string`*)

Functions

`virtual function string convert2string()`

`static function this_type get_type()`

Function

get_type

Static function that returns the static type handle. The return type is this_type, which is the type of the parameterized class.

Return type

this_type

`virtual function uvm_resource_base get_type_handle()`

Function

get_type_handle

Returns the static type handle of this resource in a polymorphic fashion. The return type of get_type_handle() is uvm_resource_base. This function is not static and therefore can only be used by instances of a parameterized resource.

Return type

uvm_resource_base

`function void set()`

Function

set

Simply put this resource into the global resource pool

`function void set_override(uvm_resource_types::override_t override = 2'b11)`

Function

set_override

Put a resource into the global resource pool as an override. This means it gets put at the head of the list and is searched before other existing resources that occupy the same position in the name map or the type map. The default is to override both the name and type maps. However, using the *override* argument you can specify that either the name map or type map is overridden.

Parameters

`override` (*uvm_resource_types::override_t*)

`static function this_type get_by_name(string scope, string name, bit rpterr = 1)`

Function

get_by_name

looks up a resource by *name* in the name map. The first resource with the specified name, whose type is the current type, and is visible in the specified *scope* is returned, if one exists. The *rpterr* flag indicates whether or not an error should be reported if the search fails. If *rpterr* is set to one then a failure message is issued, including suggested spelling alternatives, based on resource names that exist in the database, gathered by the spell checker.

Parameters

`scope` (*`string`*)
`name` (*`string`*)
`rpterr` (*`bit`*)

Return type

this_type

`static function this_type get_by_type(string scope = "", uvm_resource_base type_-handle)`

Function

get_by_type

looks up a resource by *type_handle* in the type map. The first resource with the specified *type_handle* that is visible in the specified *scope* is returned, if one exists. If there is no resource matching the specifications, *null* is returned.

Parameters

`scope` (*`string`*)
`type_handle` (*uvm_resource_base*)

Return type

this_type

`function T read(uvm_object accessor = null)`

Function

read

Return the object stored in the resource container. If an *accessor* object is supplied then also update the accessor record for this resource.

Parameters

`accessor` (*uvm_object*)

`function void write(int t, uvm_object accessor = null)`

Function

write

Modify the object stored in this resource container. If the resource is read-only then issue an error message and return without modifying the object in the container. If the resource is not read-only and an *accessor* object has been supplied then also update the accessor record. Lastly, replace the object value in the container with the value supplied as the argument, *t* , and release any processes blocked on *uvm_resource_base::wait_modified*. If the value to be written is the same as the value already present in the resource then the write is not done. That also means that the accessor record is not updated and the modified bit is not set.

Parameters

`t` (*`int`*)

`accessor` (*uvm_object*)

`virtual function void set_priority(uvm_resource_types::priority_e pri)`

Function

set priority

Change the search priority of the resource based on the value of the priority enum argument, *pri* .

Parameters

`pri` (*uvm_resource_types::priority_e*)

`static function this_type get_highest_precedence(uvm_resource_types::rsrc_q_t q)`

Function

get_highest_precedence

In a queue of resources, locate the first one with the highest precedence whose type is T. This function is static so that it can be called from anywhere.

Parameters

`q` (*uvm_resource_types::rsrc_q_t*)

Return type

this_type

15.1.1.211 Class uvm_pkg::uvm_resource_base

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_resource_base*

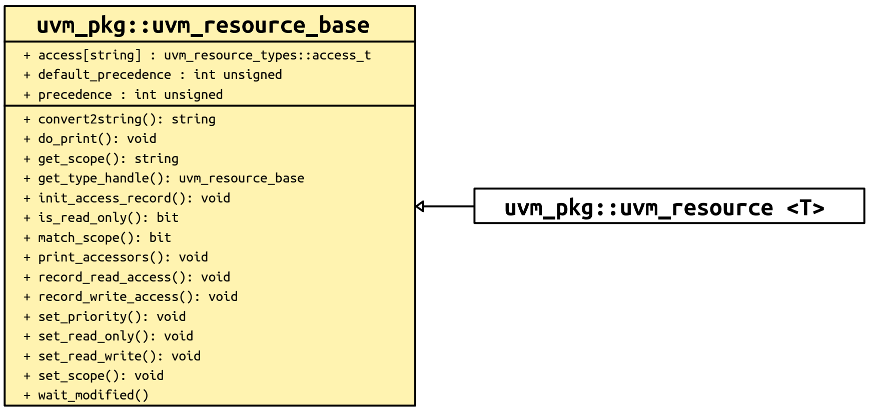

uvm_pkg::uvm_resource_base
+ access[string] : uvm_resource_types::access_t
+ default_precedence : int unsigned
+ precedence : int unsigned
+ convert2string(): string
+ do_print(): void
+ get_scope(): string
+ get_type_handle(): uvm_resource_base
+ init_access_record(): void
+ is_read_only(): bit
+ match_scope(): bit
+ print_accessors(): void
+ record_read_access(): void
+ record_write_access(): void
+ set_priority(): void
+ set_read_only(): void
+ set_read_write(): void
+ set_scope(): void
+ wait_modified()
uvm_pkg::uvm_resource <T>

Fig. 75: Inheritance Diagram of uvm_resource_base

Class

uvm_resource_base

Non-parameterized base class for resources. Supports interfaces for scope matching, and virtual functions for printing the resource and for printing the accessor list

Table 221: Variables

Name	Type	Description
access	*uvm_resource_types::access_t*	
precedence	int unsigned	***variable*** precedence This variable is used to associate a precedence that a resource has with respect to other resources which match the same scope and name. Resources are set to the *default_precedence* initially, and may be set to a higher or lower precedence as desired.
default_precedence	int unsigned	***variable*** default_precedence The default precedence for an resource that has been created. When two resources have the same precedence, the first resource found has precedence.

Constructors

```
function new(string name = "", string s = "*")
```

Function

new

constructor for uvm_resource_base. The constructor takes two arguments, the name of the resource and a regular expression which represents the set of scopes over which this resource is visible.

Parameters

name (*string*)

s (*string*)

Functions

`virtual function uvm_resource_base get_type_handle()`

Function

get_type_handle

Pure virtual function that returns the type handle of the resource container.

Return type

uvm_resource_base

`function void set_read_only()`

Function

set_read_only

Establishes this resource as a read-only resource. An attempt to call <uvm_resource(T)::write> on the resource will cause an error.

`function void set_read_write()`

Implementation question

Not sure if this function is necessary.

Once a resource is set to read_only no one should be able to change that. If anyone can flip the read_only bit then the resource is not truly read_only.

`function bit is_read_only()`

Function

is_read_only

Returns one if this resource has been set to read-only, zero otherwise

`function void set_scope(string s)`

Function

set_scope

Set the value of the regular expression that identifies the set of scopes over which this resource is visible. If the supplied argument is a glob it will be converted to a regular expression before it is stored.

Parameters

s (*string*)

`function string get_scope()`

Function

get_scope

Retrieve the regular expression string that identifies the set of scopes over which this resource is visible.

`function bit match_scope(string s)`

Function

match_scope

Using the regular expression facility, determine if this resource is visible in a scope. Return one if it is, zero otherwise.

Parameters

s (*string*)

`virtual function void set_priority(uvm_resource_types::priority_e pri)`

Function

set priority

Change the search priority of the resource based on the value of the priority enum argument.

Parameters

pri (*uvm_resource_types::priority_e*)

virtual function string convert2string()

function convert2string

Create a string representation of the resource value. By default we don't know how to do this so we just return a "?". Resource specializations are expected to override this function to produce a proper string representation of the resource value.

virtual function void do_print(uvm_printer printer)

Function

do_print

Implementation of do_print which is called by print().

Parameters

printer (*uvm_printer*)

function void record_read_access(uvm_object accessor = null)

function

record_read_access

Parameters

accessor (*uvm_object*)

function void record_write_access(uvm_object accessor = null)

function

record_write_access

Parameters

accessor (*uvm_object*)

virtual function void print_accessors()

Function

print_accessors

Dump the access records for this resource

function void init_access_record(uvm_resource_types::access_t access_record)

Function

init_access_record

Initialize a new access record

Parameters

access_record (*uvm_resource_types::access_t*)

Tasks

function wait_modified()

Task

wait_modified

This task blocks until the resource has been modified -- that is, a <uvm_resource(T)::write> operation has been performed. When a <uvm_resource(T)::write> is performed the modified bit is set which releases the block. Wait_modified() then clears the modified bit so it can be called repeatedly.

15.1.1.212 Class uvm_pkg::uvm_resource_db

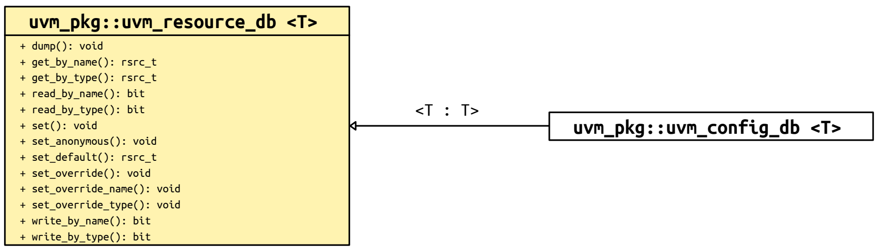


Fig. 76: Inheritance Diagram of uvm_resource_db

class

uvm_resource_db

All of the functions in uvm_resource_db(T) are static, so they must be called using the :: operator. For example:

```
uvm_resource_db#(int)::set("A", "*", 17, this);
```

The parameter value "int" identifies the resource type as uvm_resource(int). Thus, the type of the object in the resource container is int. This maintains the type-safety characteristics of resource operations.

Table 222: Parameters

Name	Default value	Description
T	uvm_object	

Table 223: Typedefs

Name	Actual Type	Description
rsrc_t	*uvm_resource#(T)*	

Functions

`static function rsrc_t get_by_type(string scope)`

function

get_by_type

Get a resource by type. The type is specified in the db class parameter so the only argument to this function is the *scope* .

Parameters
scope (`string`)

Return type
rsrc_t

`static function rsrc_t get_by_name(string scope, string name, bit rpterr = 1)`

function

get_by_name

Imports a resource by *name* . The first argument is the current *scope* of the resource to be retrieved and the second argument is the *name* . The *rpterr* flag indicates whether or not to generate a warning if no matching resource is found.

Parameters
- **scope** (*string*)
- **name** (*string*)
- **rpterr** (*bit*)

Return type
rsrc_t

`static function rsrc_t set_default(string scope, string name)`

function

set_default

add a new item into the resources database. The item will not be written to so it will have its default value. The resource is created using *name* and *scope* as the lookup parameters.

Parameters
- **scope** (*string*)
- **name** (*string*)

Return type
rsrc_t

`static function void set(string scope, string name, uvm_object val, uvm_-object accessor = null)`

function

set

Create a new resource, write a *val* to it, and set it into the database using *name* and *scope* as the lookup parameters. The *accessor* is used for auditing.

Parameters
- **scope** (*string*)
- **name** (*string*)
- **val** (*uvm_object*)
- **accessor** (*uvm_object*)

`static function void set_anonymous(string scope, uvm_object val, uvm_-object accessor = null)`

function

set_anonymous

Create a new resource, write a *val* to it, and set it into the database. The resource has no name and therefore will not be entered into the name map. But is does have a *scope* for lookup purposes. The *accessor* is used for auditing.

Parameters
- **scope** (*string*)
- **val** (*uvm_object*)
- **accessor** (*uvm_object*)

`static function void set_override(string scope, string name, uvm_object val, uvm_-object accessor = null)`

function set_override

Create a new resource, write *val* to it, and set it into the database. Set it at the beginning of the queue in the type map and the name map so that it will be (currently) the highest priority resource with the specified name and type.

Parameters
- **scope** (*string*)
- **name** (*string*)
- **val** (*uvm_object*)
- **accessor** (*uvm_object*)

`static function void set_override_type(string scope, string name, uvm_object val, uvm_object accessor = null)`

function set_override_type

Create a new resource, write *val* to it, and set it into the database. Set it at the beginning of the queue in the type map so that it will be (currently) the highest priority resource with the specified type. It will be normal priority (i.e. at the end of the queue) in the name map.

Parameters

- **scope** (*string*)
- **name** (*string*)
- **val** (*uvm_object*)
- **accessor** (*uvm_object*)

static function void set_override_name(string scope, string name, uvm_object val, uvm_object accessor = null)

function set_override_name

Create a new resource, write *val* to it, and set it into the database. Set it at the beginning of the queue in the name map so that it will be (currently) the highest priority resource with the specified name. It will be normal priority (i.e. at the end of the queue) in the type map.

Parameters

- **scope** (*string*)
- **name** (*string*)
- **val** (*uvm_object*)
- **accessor** (*uvm_object*)

static function bit read_by_name(string scope, string name, uvm_object val, uvm_-object accessor = null)

function

read_by_name

locate a resource by *name* and *scope* and read its value. The value is returned through the output argument *val* . The return value is a bit that indicates whether or not the read was successful. The *accessor* is used for auditing.

Parameters

- **scope** (*string*)
- **name** (*string*)
- **val** (*uvm_object*)
- **accessor** (*uvm_object*)

static function bit read_by_type(string scope, uvm_object val, uvm_-object accessor = null)

function

read_by_type

Read a value by type. The value is returned through the output argument *val* . The *scope* is used for the lookup. The return value is a bit that indicates whether or not the read is successful. The *accessor* is used for auditing.

Parameters

- **scope** (*string*)
- **val** (*uvm_object*)
- **accessor** (*uvm_object*)

static function bit write_by_name(string scope, string name, uvm_object val, uvm_-object accessor = null)

function

write_by_name

write a *val* into the resources database. First, look up the resource by *name* and *scope* . If it is not located then add a new resource to the database and then write its value.

Because the *scope* is matched to a resource which may be a regular expression, and consequently may target other scopes beyond the *scope* argument. Care must be taken with this function. If a *get_by_name* match is found for *name* and *scope* then *val* will be written to that matching resource and thus may impact other scopes which also match the resource.

Parameters

- **scope** (*string*)

`name` (`string`)
`val` (*uvm_object*)
`accessor` (*uvm_object*)

`static function bit write_by_type(string scope, uvm_object val, uvm_-object accessor = null)`

function

write_by_type

write a *val* into the resources database. First, look up the resource by type. If it is not located then add a new resource to the database and then write its value.

Because the *scope* is matched to a resource which may be a regular expression, and consequently may target other scopes beyond the *scope* argument. Care must be taken with this function. If a *get_by_name* match is found for *name* and *scope* then *val* will be written to that matching resource and thus may impact other scopes which also match the resource.

Parameters

`scope` (`string`)
`val` (*uvm_object*)
`accessor` (*uvm_object*)

`static function void dump()`

function

dump

Dump all the resources in the resource pool. This is useful for debugging purposes. This function does not use the parameter T, so it will dump the same thing -- the entire database -- no matter the value of the parameter.

15.1.1.213 Class uvm_pkg::uvm_resource_db_options

Options include:

tracing

on/off

The default for tracing is off.

Functions

`static function void turn_on_tracing()`

Function

turn_on_tracing

Turn tracing on for the resource database. This causes all reads and writes to the database to display information about the accesses. Tracing is off by default.

This method is implicitly called by the +*UVM_RESOURCE_DB_TRACE* .

`static function void turn_off_tracing()`

Function

turn_off_tracing

Turn tracing off for the resource database.

`static function bit is_tracing()`

Function

is_tracing

Returns 1 if the tracing facility is on and 0 if it is off.

15.1.1.214 Class uvm_pkg::uvm_resource_options

Options include:

auditing

on/off

The default for auditing is on. You may wish to turn it off to for performance reasons. With auditing off memory is not consumed for storage of auditing information and time is not spent collecting and storing auditing information. Of course, during the period when auditing is off no audit trail information is available

Functions

`static function void turn_on_auditing()`

Function

turn_on_auditing

Turn auditing on for the resource database. This causes all reads and writes to the database to store information about the accesses. Auditing is turned on by default.

`static function void turn_off_auditing()`

Function

turn_off_auditing

Turn auditing off for the resource database. If auditing is turned off, it is not possible to get extra information about resource database accesses.

`static function bit is_auditing()`

Function

is_auditing

Returns 1 if the auditing facility is on and 0 if it is off.

15.1.1.215 Class uvm_pkg::uvm_resource_pool

Class

uvm_resource_pool

The global (singleton) resource database.

Each resource is stored both by primary name and by type handle. The resource pool contains two associative arrays, one with name as the key and one with the type handle as the key. Each associative array contains a queue of resources. Each resource has a regular expression that represents the set of scopes over which it is visible.

```
+------+------------+                           +------------+------+
| name | rsrc queue |                           | rsrc queue | type |
+------+------------+                           +------------+------+
|      |            |                           |            |      |
+------+------------+                   +-+-+   +------------+------+
|      |            |                   | | |<--+---*        |  T   |
+------+------------+   +-+-+           +-+-+   +------------+------+
|  A   |        *---+-->| | |            |      |            |      |
+------+------------+   +-+-+            |      +------------+------+
|      |            |      |             |      |            |      |
+------+------------+      +--------+  +-+      +------------+------+
|      |            |               |  |        |            |      |
+------+------------+               |  |        +------------+------+
|      |            |               V  V        |            |      |
+------+------------+            +-------+      +------------+------+
|      |            |            |  rsrc |      |            |      |
+------+------------+            +-------+      +------------+------+
```

The above diagrams illustrates how a resource whose name is A and type is T is stored in the pool. The pool contains an entry in the type map for type T and an entry in the name map for name A. The queues in each of the arrays each contain an entry for the resource A whose type is T. The name map can contain in its queue other resources whose name is A which may or may not have the same type as our resource A. Similarly, the type map can contain in its queue other resources whose type is T and whose name may or may not be A.

Resources are added to the pool by calling *set*; they are retrieved from the pool by calling *get_by_name* or *get_by_type*. When an object creates a new resource and calls *set* the resource is made available to be retrieved by other objects outside of itself; an object gets a resource when it wants to access a resource not currently available in its scope.

The scope is stored in the resource itself (not in the pool) so whether you get by name or by type the resource's visibility is the same.

As an auditing capability, the pool contains a history of gets. A record of each get, whether by *get_by_type* or *get_by_name*, is stored in the audit record. Both successful and failed gets are recorded. At the end of simulation, or any time for that matter, you can dump the history list. This will tell which resources were successfully located and which were not. You can use this information to determine if there is some error in name, type, or scope that has caused a resource to not be located or to be incorrectly located (i.e. the wrong resource is located).

Table 224: Variables

Name	Type	Description
rtab	*uvm_resource_-types::rsrc_q_t*	
ttab	*uvm_resource_-types::rsrc_q_t*	

continues on next page

Table 224 – continued from previous page

Name	Type	Description
get_record	*get_t*	history of gets

Functions

static function uvm_resource_pool get()

Function

get

Returns the singleton handle to the resource pool

Return type
uvm_resource_pool

function bit spell_check(string s)

Function

spell_check

Invokes the spell checker for a string s. The universe of correctly spelled strings -- i.e. the dictionary -- is the name map.

Parameters
s (*string*)

function void set(uvm_resource_base rsrc, uvm_resource_types::override_t override = 0)

Function

set

Add a new resource to the resource pool. The resource is inserted into both the name map and type map so it can be located by either.

An object creates a resources and *sets* it into the resource pool. Later, other objects that want to access the resource must *get* it from the pool

Overrides can be specified using this interface. Either a name override, a type override or both can be specified. If an override is specified then the resource is entered at the front of the queue instead of at the back. It is not recommended that users specify the override parameter directly, rather they use the *set_override*, *set_name_override*, or *set_type_override* functions.

Parameters
rsrc (*uvm_resource_base*)
override (*uvm_resource_types::override_t*)

function void set_override(uvm_resource_base rsrc)

Function

set_override

The resource provided as an argument will be entered into the pool and will override both by name and type.

Parameters
rsrc (*uvm_resource_base*)

function void set_name_override(uvm_resource_base rsrc)

Function

set_name_override

The resource provided as an argument will entered into the pool using normal precedence in the type map and will override the name.

Parameters
rsrc (*uvm_resource_base*)

`function void set_type_override(uvm_resource_base rsrc)`

Function

set_type_override

The resource provided as an argument will be entered into the pool using normal precedence in the name map and will override the type.

Parameters

`rsrc` (*uvm_resource_base*)

`function void push_get_record(string name, string scope, uvm_resource_base rsrc)`

function

push_get_record

Insert a new record into the get history list.

Parameters

`name` (`string`)

`scope` (`string`)

`rsrc` (*uvm_resource_base*)

`function void dump_get_records()`

function

dump_get_records

Format and print the get history list.

`function uvm_resource_types::rsrc_q_t lookup_name(string scope = "", string name, uvm_resource_base type_handle = null, bit rpterr = 1)`

Function

lookup_name

Lookup resources by *name* . Returns a queue of resources that match the *name* , *scope* , and *type_handle* . If no resources match the queue is returned empty. If *rpterr* is set then a warning is issued if no matches are found, and the spell checker is invoked on *name* . If *type_handle* is *null* then a type check is not made and resources are returned that match only *name* and *scope* .

Parameters

`scope` (`string`)

`name` (`string`)

`type_handle` (*uvm_resource_base*)

`rpterr` (`bit`)

Return type

uvm_resource_types::rsrc_q_t

`function uvm_resource_base get_highest_precedence(uvm_resource_types::rsrc_q_t q)`

Function

get_highest_precedence

Traverse a queue, *q* , of resources and return the one with the highest precedence. In the case where there exists more than one resource with the highest precedence value, the first one that has that precedence will be the one that is returned.

Parameters

`q` (*uvm_resource_types::rsrc_q_t*)

Return type

uvm_resource_base

`static function void sort_by_precedence(uvm_resource_types::rsrc_q_t q)`

Function

sort_by_precedence

Given a list of resources, obtained for example from *lookup_scope*, sort the resources in precedence order. The highest precedence resource will be first in the list and the lowest precedence will be last. Resources that have the same precedence and the same name will be ordered by most recently set first.

Parameters

`q` (*uvm_resource_types::rsrc_q_t*)

function uvm_resource_base get_by_name(string scope = "", string name, uvm_-resource_base type_handle, bit rpterr = 1)

Function

get_by_name

Lookup a resource by *name* , *scope* , and *type_handle* . Whether the get succeeds or fails, save a record of the get attempt. The *rpterr* flag indicates whether to report errors or not. Essentially, it serves as a verbose flag. If set then the spell checker will be invoked and warnings about multiple resources will be produced.

Parameters

- **scope** (*string*)
- **name** (*string*)
- **type_handle** (*uvm_resource_base*)
- **rpterr** (*bit*)

Return type

uvm_resource_base

function uvm_resource_types::rsrc_q_t lookup_type(string scope = "", uvm_resource_-base type_handle)

Function

lookup_type

Lookup resources by type. Return a queue of resources that match the *type_handle* and *scope* . If no resources match then the returned queue is empty.

Parameters

- **scope** (*string*)
- **type_handle** (*uvm_resource_base*)

Return type

uvm_resource_types::rsrc_q_t

function uvm_resource_base get_by_type(string scope = "", uvm_resource_base type_-handle)

Function

get_by_type

Lookup a resource by *type_handle* and *scope* . Insert a record into the get history list whether or not the get succeeded.

Parameters

- **scope** (*string*)
- **type_handle** (*uvm_resource_base*)

Return type

uvm_resource_base

function uvm_resource_types::rsrc_q_t lookup_regex_names(string scope, string name, uvm_resource_base type_handle = null)

Function

lookup_regex_names

This utility function answers the question, for a given *name* , *scope* , and *type_handle* , what are all of the resources with requested name, a matching scope (where the resource scope may be a regular expression), and a matching type? *name* and *scope* are explicit values.

Parameters

- **scope** (*string*)
- **name** (*string*)
- **type_handle** (*uvm_resource_base*)

Return type

uvm_resource_types::rsrc_q_t

function uvm_resource_types::rsrc_q_t lookup_regex(string re, string scope)

Function

lookup_regex

Looks for all the resources whose name matches the regular expression argument and whose scope matches the current scope.

Parameters
- **re** (*string*)
- **scope** (*string*)

Return type
uvm_resource_types::rsrc_q_t

function uvm_resource_types::rsrc_q_t lookup_scope(string scope)

Function

lookup_scope

This is a utility function that answers the question

For a given

scope , what resources are visible to it? Locate all the resources that are visible to a particular scope. This operation could be quite expensive, as it has to traverse all of the resources in the database.

Parameters
- **scope** (*string*)

Return type
uvm_resource_types::rsrc_q_t

function void set_priority_type(uvm_resource_base rsrc, uvm_resource_-types::priority_e pri)

Function

set_priority_type

Change the priority of the *rsrc* based on the value of *pri* , the priority enum argument. This function changes the priority only in the type map, leaving the name map untouched.

Parameters
- **rsrc** (*uvm_resource_base*)
- **pri** (*uvm_resource_types::priority_e*)

function void set_priority_name(uvm_resource_base rsrc, uvm_resource_-types::priority_e pri)

Function

set_priority_name

Change the priority of the *rsrc* based on the value of *pri* , the priority enum argument. This function changes the priority only in the name map, leaving the type map untouched.

Parameters
- **rsrc** (*uvm_resource_base*)
- **pri** (*uvm_resource_types::priority_e*)

function void set_priority(uvm_resource_base rsrc, uvm_resource_types::priority_-e pri)

Function

set_priority

Change the search priority of the *rsrc* based on the value of *pri* , the priority enum argument. This function changes the priority in both the name and type maps.

Parameters
- **rsrc** (*uvm_resource_base*)
- **pri** (*uvm_resource_types::priority_e*)

function uvm_resource_types::rsrc_q_t find_unused_resources()

Function

find_unused_resources

Locate all the resources that have at least one write and no reads

Return type
uvm_resource_types::rsrc_q_t

`function void print_resources(uvm_resource_types::rsrc_q_t rq, bit audit = 0)`

Function

print_resources

Print the resources that are in a single queue, *rq* . This is a utility function that can be used to print any collection of resources stored in a queue. The *audit* flag determines whether or not the audit trail is printed for each resource along with the name, value, and scope regular expression.

Parameters

`rq` (*uvm_resource_types::rsrc_q_t*)

`audit` (*`bit`*)

`function void dump(bit audit = 0)`

Function

dump

dump the entire resource pool. The resource pool is traversed and each resource is printed. The utility function print_resources() is used to initiate the printing. If the *audit* bit is set then the audit trail is dumped for each resource.

Parameters

`audit` (*`bit`*)

15.1.1.216 Class uvm_pkg::uvm_resource_types

Class

uvm_resource_types

Provides typedefs and enums used throughout the resources facility. This class has no members or methods, only typedefs. It's used in lieu of package-scope types. When needed, other classes can use these types by prefixing their usage with uvm_resource_types::. E.g.

```
uvm_resource_types::rsrc_q_t queue;
```

Table 225: Typedefs

Name	Actual Type	Description
override_t	bit[1:0]	types uses for setting overrides
rsrc_q_t	*uvm_queue#(uvm_resource_base)*	general purpose queue of resourcex

Enums

`override_e`

Enum Items
- TYPE_OVERRIDE = 2'b01
- NAME_OVERRIDE = 2'b10

`priority_e`

enum for setting resource search priority

Enum Items
- PRI_HIGH
- PRI_LOW

Structs

`typedef struct access_t`

access record for resources. A set of these is stored for each resource by accessing object. It's updated for each read/write.

15.1.1.217 Class uvm_pkg::uvm_run_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_task_phase*
↪*uvm_pkg* :: *uvm_run_phase*

Class

uvm_run_phase

Stimulate the DUT.

This *uvm_task_phase* calls the *uvm_component::run_phase* virtual method. This phase runs in parallel to the runtime phases, *uvm_pre_reset_phase* through *uvm_post_shutdown_phase*. All components in the testbench are synchronized with respect to the run phase regardless of the phase domain they belong to.

Upon Entry

Indicates that power has been applied.
There should not have been any active clock edges before entry into this phase (e.g. x->1 transitions via initial blocks).
Current simulation time is still equal to 0 but some "delta cycles" may have occurred.

Typical Uses

Components implement behavior that is exhibited for the entire run-time, across the various run-time phases.
Backward compatibility with OVM.

Exit Criteria

The DUT no longer needs to be simulated, and
The *uvm_post_shutdown_phase* is ready to end

The run phase terminates in one of two ways.

1. All run_phase objections are dropped:

When all objections on the run_phase objection have been dropped, the phase ends and all of its threads are killed. If no component raises a run_phase objection immediately upon entering the phase, the phase ends immediately.

2. Timeout:

The phase ends if the timeout expires before all objections are dropped. By default, the timeout is set to 9200 seconds. You may override this via *uvm_root::set_timeout*.

If a timeout occurs in your simulation, or if simulation never ends despite completion of your test stimulus, then it usually indicates that a component continues to object to the end of a phase.

Table 226: Variables

Name	Type	Description
type_name	string	

Functions

`static function uvm_run_phase get()`

Function

get

Returns the singleton phase handle

Return type
uvm_run_phase

`virtual function string get_type_name()`

Tasks

`virtual function exec_task(uvm_component comp, uvm_phase phase)`

Parameters
- **`comp`** (*uvm_component*)
- **`phase`** (*uvm_phase*)

15.1.1.218 Class uvm_pkg::uvm_scope_stack

CLASS- uvm_scope_stack

Functions

`function int depth()`

depth

`function string get()`

scope

`function string get_arg()`

scope_arg

`function void set(string s)`

set_scope

Parameters

`s` (*`string`*)

`function void down(string s)`

down

Parameters

`s` (*`string`*)

`function void down_element(int element)`

down_element

Parameters

`element` (*`int`*)

`function void up_element()`

up_element

`function void up(byte separator = ".")`

15.2 up

Parameters

`separator` (*`byte`*)

`function void set_arg(string arg)`

set_arg

Parameters

`arg` (*`string`*)

`function void set_arg_element(string arg, int ele)`

set_arg_element

Parameters

`arg` (*`string`*)

`ele` (*`int`*)

`function void unset_arg(string arg)`

unset_arg

Parameters

`arg` (*`string`*)

15.2.0.1 Class uvm_pkg::uvm_scoreboard

uvm_pkg :: *uvm_void*
 ↪*uvm_pkg* :: *uvm_object*
  ↪*uvm_pkg* :: *uvm_report_object*
   ↪*uvm_pkg* :: *uvm_component*
    ↪*uvm_pkg* :: *uvm_scoreboard*

CLASS

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.

Table 227: Variables

Name	Type	Description
type_name	string	

Constructors

`function new(string name, uvm_component parent)`

Function

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.

Parameters

`name` (*`string`*)
`parent` (*uvm_component*)

Functions

`virtual function string get_type_name()`

15.2.0.2 Class uvm_pkg::uvm_seed_map

Class- uvm_seed_map

This map is a seed map that can be used to update seeds. The update is done automatically by the seed hashing routine. The seed_table_lookup uses an instance name lookup and the seed_table inside a given map uses a type name for the lookup.

Table 228: Variables

Name	Type	Description
seed_table	int unsigned	
count	int unsigned	

15.2.0.3 Class uvm_pkg::uvm_seq_item_pull_export

uvm_pkg :: *uvm_sqr_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_seq_item_pull_export*

Class

uvm_seq_item_pull_export (REQ, RSP)

This export type is used in sequencer-driver communication. It has the standard constructor for exports.

Table 229: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.2.0.4 Class uvm_pkg::uvm_seq_item_pull_imp

uvm_pkg :: *uvm_sqr_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_seq_item_pull_imp*

Class

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.

Table 230: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **`name`** (*`string`*)
- **`imp`** (*`int`*)

15.2.0.5 Class uvm_pkg::uvm_seq_item_pull_port

uvm_pkg :: *uvm_sqr_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_seq_item_pull_port*

Class

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.

Table 231: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Table 232: Variables

Name	Type	Description
print_enabled	bit	

Constructors

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

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.2.0.6 Class uvm_pkg::uvm_sequence

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*

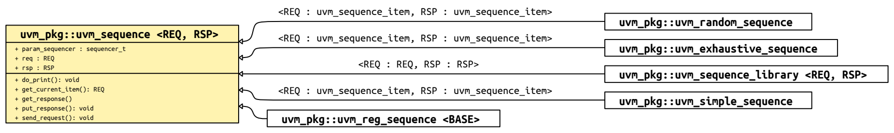

uvm_pkg::uvm_sequence <REQ, RSP>
+ param_sequencer : sequencer_t
+ req : REQ
+ rsp : RSP
+ do_print(): void
+ get_current_item(): REQ
+ get_response()
+ put_response(): void
+ send_request(): void
<REQ : uvm_sequence_item, RSP : uvm_sequence_item>
uvm_pkg::uvm_random_sequence
<REQ : uvm_sequence_item, RSP : uvm_sequence_item>
uvm_pkg::uvm_exhaustive_sequence
<REQ : REQ, RSP : RSP>
uvm_pkg::uvm_sequence_library <REQ, RSP>
<REQ : uvm_sequence_item, RSP : uvm_sequence_item>
uvm_pkg::uvm_simple_sequence
uvm_pkg::uvm_reg_sequence <BASE>

Fig. 77: Inheritance Diagram of uvm_sequence

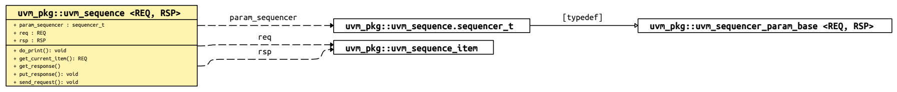

uvm_pkg::uvm_sequence <REQ, RSP>
+ param_sequencer : sequencer_t
+ req : REQ
+ rsp : RSP
+ do_print(): void
+ get_current_item(): REQ
+ get_response()
+ put_response(): void
+ send_request(): void
param_sequencer
uvm_pkg::uvm_sequence.sequencer_t
[typedef]
uvm_pkg::uvm_sequencer_param_base <REQ, RSP>
req
rsp
uvm_pkg::uvm_sequence_item

Fig. 78: Collaboration Diagram of uvm_sequence

CLASS

uvm_sequence (REQ, RSP)

The uvm_sequence class provides the interfaces necessary in order to create streams of sequence items and/or other sequences.

Table 233: Parameters

Name	Default value	Description
REQ	uvm_sequence_item	
RSP	REQ	

Table 234: Variables

Name	Type	Description
param_sequencer	*sequencer_t*	
req	*uvm_sequence_item*	***Variable*** req The sequence contains a field of the request type called req. The user can use this field, if desired, or create another field to use. The default *do_print* will print this field.

continues on next page

Table 234 – continued from previous page

Name	Type	Description
rsp	*uvm_sequence_item*	***Variable*** rsp The sequence contains a field of the response type called rsp. The user can use this field, if desired, or create another field to use. The default *do_print* will print this field.

Table 235: Typedefs

Name	Actual Type	Description
sequencer_t	*uvm_sequencer_param_-base#(REQ, RSP)*	

Constructors

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

Function

new

Creates and initializes a new sequence object.

Parameters

`name` (`string`)

Functions

`virtual function void send_request(uvm_sequence_item request, bit rerandomize = 0)`

Function

send_request

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.

Parameters

`request` (*uvm_sequence_item*)

`rerandomize` (`bit`)

`function REQ get_current_item()`

Function

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.

Return type

REQ

`virtual function void put_response(uvm_sequence_item response_item)`

Function- put_response

Internal method.

Parameters

`response_item` (*uvm_sequence_item*)

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

Function- do_print

Parameters

printer (*uvm_printer*)

Tasks

```
virtual  function  get_response(uvm_sequence_item response, int transaction_id = -1)
```

Task

get_response

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 is 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.

Parameters

response (*uvm_sequence_item*)
transaction_id (*int*)

15.2.0.7 Class uvm_pkg::uvm_sequence_base

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*

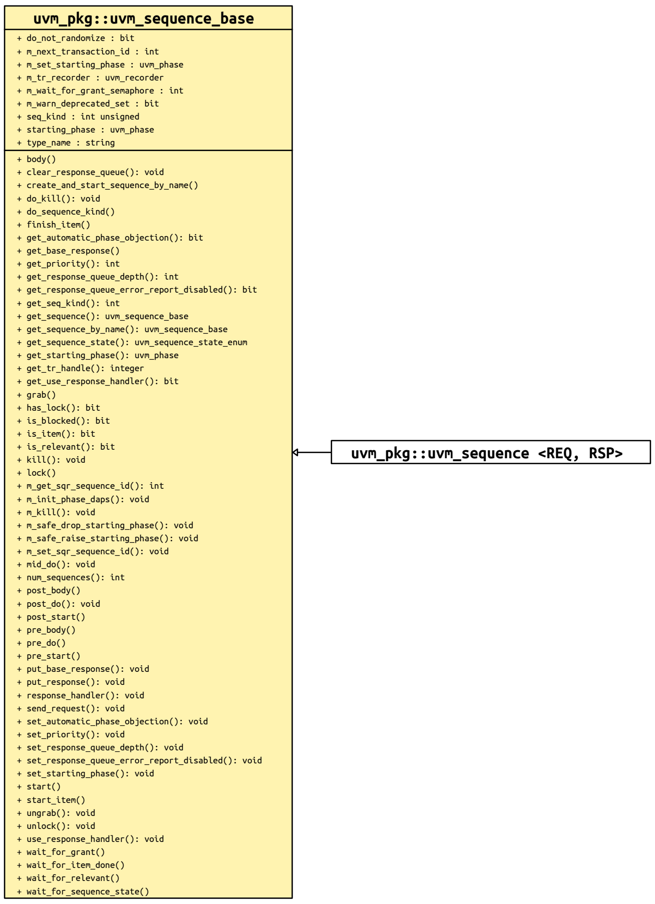

uvm_pkg::uvm_sequence_base
+ do_not_randomize : bit
+ m_next_transaction_id : int
+ m_set_starting_phase : uvm_phase
+ m_tr_recorder : uvm_recorder
+ m_wait_for_grant_semaphore : int
+ m_warn_deprecated_set : bit
+ seq_kind : int unsigned
+ starting_phase : uvm_phase
+ type_name : string
+ body()
+ clear_response_queue(): void
+ create_and_start_sequence_by_name()
+ do_kill(): void
+ do_sequence_kind()
+ finish_item()
+ get_automatic_phase_objection(): bit
+ get_base_response()
+ get_priority(): int
+ get_response_queue_depth(): int
+ get_response_queue_error_report_disabled(): bit
+ get_seq_kind(): int
+ get_sequence(): uvm_sequence_base
+ get_sequence_by_name(): uvm_sequence_base
+ get_sequence_state(): uvm_sequence_state_enum
+ get_starting_phase(): uvm_phase
+ get_tr_handle(): integer
+ get_use_response_handler(): bit
+ grab()
+ has_lock(): bit
+ is_blocked(): bit
+ is_item(): bit
+ is_relevant(): bit
+ kill(): void
+ lock()
+ m_get_sqr_sequence_id(): int
+ m_init_phase_daps(): void
+ m_kill(): void
+ m_safe_drop_starting_phase(): void
+ m_safe_raise_starting_phase(): void
+ m_set_sqr_sequence_id(): void
+ mid_do(): void
+ num_sequences(): int
+ post_body()
+ post_do(): void
+ post_start()
+ pre_body()
+ pre_do()
+ pre_start()
+ put_base_response(): void
+ put_response(): void
+ response_handler(): void
+ send_request(): void
+ set_automatic_phase_objection(): void
+ set_priority(): void
+ set_response_queue_depth(): void
+ set_response_queue_error_report_disabled(): void
+ set_starting_phase(): void
+ start()
+ start_item()
+ ungrab(): void
+ unlock(): void
+ use_response_handler(): void
+ wait_for_grant()
+ wait_for_item_done()
+ wait_for_relevant()
+ wait_for_sequence_state()
uvm_pkg::uvm_sequence <REQ, RSP>

Fig. 79: Inheritance Diagram of uvm_sequence_base

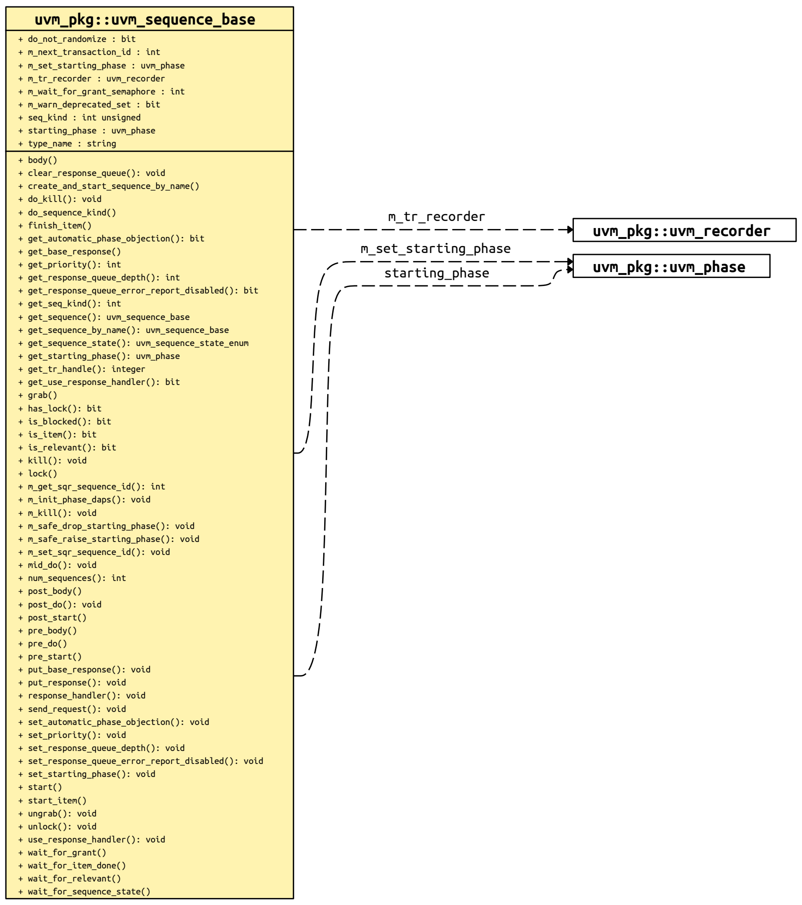

uvm_pkg::uvm_sequence_base
+ do_not_randomize : bit
+ m_next_transaction_id : int
+ m_set_starting_phase : uvm_phase
+ m_tr_recorder : uvm_recorder
+ m_wait_for_grant_semaphore : int
+ m_warn_deprecated_set : bit
+ seq_kind : int unsigned
+ starting_phase : uvm_phase
+ type_name : string
+ body()
+ clear_response_queue(): void
+ create_and_start_sequence_by_name()
+ do_kill(): void
+ do_sequence_kind()
+ finish_item()
+ get_automatic_phase_objection(): bit
+ get_base_response()
+ get_priority(): int
+ get_response_queue_depth(): int
+ get_response_queue_error_report_disabled(): bit
+ get_seq_kind(): int
+ get_sequence(): uvm_sequence_base
+ get_sequence_by_name(): uvm_sequence_base
+ get_sequence_state(): uvm_sequence_state_enum
+ get_starting_phase(): uvm_phase
+ get_tr_handle(): integer
+ get_use_response_handler(): bit
+ grab()
+ has_lock(): bit
+ is_blocked(): bit
+ is_item(): bit
+ is_relevant(): bit
+ kill(): void
+ lock()
+ m_get_sqr_sequence_id(): int
+ m_init_phase_daps(): void
+ m_kill(): void
+ m_safe_drop_starting_phase(): void
+ m_safe_raise_starting_phase(): void
+ m_set_sqr_sequence_id(): void
+ mid_do(): void
+ num_sequences(): int
+ post_body()
+ post_do(): void
+ post_start()
+ pre_body()
+ pre_do()
+ pre_start()
+ put_base_response(): void
+ put_response(): void
+ response_handler(): void
+ send_request(): void
+ set_automatic_phase_objection(): void
+ set_priority(): void
+ set_response_queue_depth(): void
+ set_response_queue_error_report_disabled(): void
+ set_starting_phase(): void
+ start()
+ start_item()
+ ungrab(): void
+ unlock(): void
+ use_response_handler(): void
+ wait_for_grant()
+ wait_for_item_done()
+ wait_for_relevant()
+ wait_for_sequence_state()
m_tr_recorder
uvm_pkg::uvm_recorder
m_set_starting_phase
starting_phase
uvm_pkg::uvm_phase

Fig. 80: Collaboration Diagram of uvm_sequence_base

CLASS

uvm_sequence_base

The uvm_sequence_base class provides the interfaces needed to create streams of sequence items and/or other sequences.

A sequence is executed by calling its *start* method, either directly or invocation of any of the &96;uvm_do_* macros.

Executing sequences via *start*:

A sequence's *start* method has a *parent_sequence* argument that controls whether *pre_do*, *mid_do*, and *post_do* are called **in the parent** sequence. It also has a *call_pre_post* argument that controls whether its *pre_body* and *post_body* methods are called. In all cases, its *pre_start* and *post_start* methods are always called.

When *start* is called directly, you can provide the appropriate arguments according to your application.

The sequence execution flow looks like this

User code

```
sub_seq.randomize(...); // optional
sub_seq.start(seqr, parent_seq, priority, call_pre_post)
```

The following methods are called, in order

|

```
sub_seq.pre_start()         (task)
sub_seq.pre_body()          (task)  if call_pre_post==1
  parent_seq.pre_do(0)      (task)  if parent_sequence!=null
  parent_seq.mid_do(this)   (func)  if parent_sequence!=null
sub_seq.body                (task)  YOUR STIMULUS CODE
  parent_seq.post_do(this)  (func)  if parent_sequence!=null
sub_seq.post_body()         (task)  if call_pre_post==1
sub_seq.post_start()        (task)
```

Executing sub-sequences via &96;uvm_do macros:

A sequence can also be indirectly started as a child in the *body* of a parent sequence. The child sequence's *start* method is called indirectly by invoking any of the &96;uvm_do macros. In these cases, *start* is called with *call_pre_post* set to 0, preventing the started sequence's *pre_body* and *post_body* methods from being called. During execution of the child sequence, the parent's *pre_do*, *mid_do*, and *post_do* methods are called.

The sub-sequence execution flow looks like

User code

|

```
`uvm_do_with_prior(seq_seq, { constraints }, priority)
```

The following methods are called, in order

|

```
sub_seq.pre_start()          (task)
parent_seq.pre_do(0)         (task)
parent_req.mid_do(sub_seq)   (func)
  sub_seq.body()             (task)
parent_seq.post_do(sub_seq)  (func)
sub_seq.post_start()         (task)
```

Remember, it is the **parent** sequence's pre|mid|post_do that are called, not the sequence being executed.

Executing sequence items via *start_item*/*finish_item* or &96;uvm_do macros:

Items are started in the *body* of a parent sequence via calls to *start_item*/*finish_item* or invocations of any of the &96;uvm_do macros. The *pre_do*, *mid_do*, and *post_do* methods of the parent sequence will be called as the item is executed.

The sequence-item execution flow looks like

User code

```
parent_seq.start_item(item, priority);
item.randomize(...) [with {constraints}];
parent_seq.finish_item(item);

or

`uvm_do_with_prior(item, constraints, priority)
```

The following methods are called, in order

|

```
sequencer.wait_for_grant(prior) (task) \ start_item  \
parent_seq.pre_do(1)            (task) /              \
                                                 `uvm_do* macros
parent_seq.mid_do(item)         (func) \              /
sequencer.send_request(item)    (func)  \finish_item /
sequencer.wait_for_item_done()  (task)  /
parent_seq.post_do(item)        (func) /
```

Attempting to execute a sequence via *start_item*/*finish_item* will produce a run-time error.

Table 236: Variables

Name	Type	Description
do_not_randomize	bit	***Variable*** do_not_randomize If set, prevents the sequence from being randomized before being executed by the &96;uvm_do*() and &96;uvm_rand_send*() macros, or as a default sequence.
type_name	string	
starting_phase	*uvm_phase*	**DEPRECATED** !! Use get/set_starting_phase accessors instead!
seq_kind	int unsigned	Variable- seq_kind Used as an identifier in constraints for a specific sequence type.

Table 237: Constraints

Name	Description
pick_sequence	For user random selection. This excludes the exhaustive and random sequences.

Constructors

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

Function

new

The constructor for uvm_sequence_base.

Parameters

name (*string*)

Functions

`virtual function bit is_item()`

Function

is_item

Returns 1 on items and 0 on sequences. As this object is a sequence, *is_item* will always return 0.

`function uvm_sequence_state_enum get_sequence_state()`

Function

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() & (UVM_STOPPED|UVM_FINISHED));
```

Return type

uvm_sequence_state_enum

`function integer get_tr_handle()`

Function

get_tr_handle

Returns the integral recording transaction handle for this sequence. Can be used to associate sub-sequences and sequence items as child transactions when calling *uvm_component::begin_child_tr*.

`virtual function void mid_do(uvm_sequence_item this_item)`

Function

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. This method should not be called directly by the user.

Parameters

`this_item` (*uvm_sequence_item*)

`virtual function void post_do(uvm_sequence_item this_item)`

Function

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. This method should not be called directly by the user.

Parameters

`this_item` (*uvm_sequence_item*)

`function uvm_phase get_starting_phase()`

Function

get_starting_phase

Returns the 'starting phase'.

If non- *null* , the starting phase specifies the phase in which this sequence was started. The starting phase is set automatically when this sequence is started as the default sequence on a sequencer. See *uvm_sequencer_base::start_phase_sequence* for more information.

Internally, the *uvm_sequence_base* uses an *uvm_get_to_lock_dap* to protect the starting phase value from being modified after the reference has been read. Once the sequence has ended its execution (either via natural termination, or being killed), then the starting phase value can be modified again.

Return type

uvm_phase

`function void set_starting_phase(uvm_phase phase)`

Function

set_starting_phase

Sets the 'starting phase'.

Internally, the *uvm_sequence_base* uses a *uvm_get_to_lock_dap* to protect the starting phase value from being modified after the reference has been read. Once the sequence has ended its execution (either via natural termination, or being killed), then the starting phase value can be modified again.

Parameters

`phase` (*uvm_phase*)

`function void set_automatic_phase_objection(bit value)`

Function

set_automatic_phase_objection

Sets the 'automatically object to starting phase' bit.

The most common interaction with the starting phase within a sequence is to simply *raise* the phase's objection prior to executing the sequence, and *drop* the objection after ending the sequence (either naturally, or via a call to *kill*). In order to simplify this interaction for the user, the UVM provides the ability to perform this functionality automatically.

For example:

```
function my_sequence::new(string name="unnamed");
  super.new(name);
  set_automatic_phase_objection(1);
endfunction : new
```

From a timeline point of view, the automatic phase objection looks like:

```
start() is executed
  --! Objection is raised !--
  pre_start() is executed
  pre_body() is optionally executed
  body() is executed
  post_body() is optionally executed
  post_start() is executed
  --! Objection is dropped !--
start() unblocks
```

This functionality can also be enabled in sequences which were not written with UVM Run-Time Phasing in mind:

```
my_legacy_seq_type seq = new("seq");
seq.set_automatic_phase_objection(1);
seq.start(my_sequencer);
```

Internally, the *uvm_sequence_base* uses a *uvm_get_to_lock_dap* to protect the *automatic_phase_objection* value from being modified after the reference has been read. Once the sequence has ended its execution (either via natural termination, or being killed), then the *automatic_phase_objection* value can be modified again.

NEVER set the automatic phase objection bit to 1 if your sequence runs with a forever loop inside of the body, as the objection will never get dropped!

Parameters

`value` (*`bit`*)

`function bit get_automatic_phase_objection()`

Function

get_automatic_phase_objection

Returns (and locks) the value of the 'automatically object to starting phase' bit.

If 1, then the sequence will automatically raise an objection to the starting phase (if the starting phase is not *null*) immediately prior to *pre_start* being called. The objection will be dropped after *post_start* has executed, or *kill* has been called.

`function void set_priority(int value)`

Function

set_priority

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.

Parameters

`value` (*`int`*)

`function int get_priority()`

Function

get_priority

This function returns the current priority of the sequence.

`virtual function bit is_relevant()`

Function

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.

`function void unlock(uvm_sequencer_base sequencer = null)`

Function

unlock

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.

Parameters

`sequencer` (*uvm_sequencer_base*)

`function void ungrab(uvm_sequencer_base sequencer = null)`

Function

ungrab

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.

Parameters

`sequencer` (*uvm_sequencer_base*)

`function bit is_blocked()`

Function

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.

`function bit has_lock()`

Function

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.

`function void kill()`

Function

kill

This function will kill the sequence, and cause all current locks and requests in the sequence's default sequencer to be removed. The sequence state will change to UVM_STOPPED, and the post_body() and post_start() callback methods will not be executed.

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.

`virtual  function void do_kill()`

Function

do_kill

This function is a user hook that is called whenever a sequence is terminated by using either sequence.kill() or sequencer.stop_sequences() (which effectively calls sequence.kill()).

`virtual  function void send_request(uvm_sequence_item request, bit rerandomize = 0)`

Function

send_request

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.

Parameters

`request` (*uvm_sequence_item*)
`rerandomize` (*`bit`*)

`function void use_response_handler(bit enable)`

Function

use_response_handler

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.

Parameters

`enable` (*`bit`*)

`function bit get_use_response_handler()`

Function

get_use_response_handler

Returns the state of the use_response_handler bit.

`virtual  function void response_handler(uvm_sequence_item response)`

Function

response_handler

When the use_response_handler bit is set to 1, this virtual task is called by the sequencer for each response that arrives for this sequence.

Parameters

`response` (*uvm_sequence_item*)

`function void set_response_queue_error_report_disabled(bit value)`

Function

set_response_queue_error_report_disabled

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.

Parameters

`value` (*`bit`*)

`function bit get_response_queue_error_report_disabled()`

Function

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.

`function void set_response_queue_depth(int value)`

Function

set_response_queue_depth

The default maximum depth of the response queue is 8. These method is used to examine or change the maximum depth of the response queue.

Setting the response_queue_depth to -1 indicates an arbitrarily deep response queue. No checking is done.

Parameters

`value` (*`int`*)

function int get_response_queue_depth()

Function

get_response_queue_depth

Returns the current depth setting for the response queue.

virtual function void clear_response_queue()

Function

clear_response_queue

Empties the response queue for this sequence.

virtual function void put_base_response(uvm_sequence_item response)

Parameters

response (*uvm_sequence_item*)

virtual function void put_response(uvm_sequence_item response_item)

Function- put_response

Internal method.

Parameters

response_item (*uvm_sequence_item*)

function int num_sequences()

Function- num_sequences

Returns the number of sequences in the sequencer's sequence library.

function int get_seq_kind(string type_name)

Function- get_seq_kind

This function returns an int representing the sequence kind that has been registerd with the sequencer. The return value may be used with the *get_sequence* or *do_sequence_kind* methods.

Parameters

type_name (*string*)

function uvm_sequence_base get_sequence(int unsigned req_kind)

Function- get_sequence

This function returns a reference to a sequence specified by *req_kind* , which can be obtained using the *get_seq_kind* method.

Parameters

req_kind (*int unsigned*)

Return type

uvm_sequence_base

function uvm_sequence_base get_sequence_by_name(string seq_name)

Function- get_sequence_by_name

Internal method.

Parameters

seq_name (*string*)

Return type

uvm_sequence_base

Tasks

function wait_for_sequence_state(int unsigned state_mask)

Task

wait_for_sequence_state

Waits until the sequence reaches one of the given *state* . If the sequence is already in one of the state, this method returns immediately.

```
wait_for_sequence_state(UVM_STOPPED|UVM_FINISHED);
```

Parameters

state_mask (*int unsigned*)

virtual function start(uvm_sequencer_base sequencer, uvm_sequence_base parent_-sequence = null, int this_priority = -1, bit call_pre_post = 1)

Task

start

Executes this sequence, returning when the sequence has completed.

The *sequencer* argument specifies the sequencer on which to run this sequence. The sequencer must be compatible with the sequence.

If *parent_sequence* is *null* , then this sequence is a root parent, otherwise it is a child of *parent_sequence* . The *parent_sequence* 's pre_do, mid_do, and post_do methods will be called during the execution of this sequence.

By default, the *priority* of a sequence is the priority of its parent sequence. If it is a root sequence, its default priority is 100. A different priority may be specified by *this_priority* . Higher numbers indicate higher priority.

If *call_pre_post* is set to 1 (default), then the *pre_body* and *post_body* tasks will be called before and after the sequence *body* is called.

Parameters

sequencer (*uvm_sequencer_base*)
parent_sequence (*uvm_sequence_base*)
this_priority (*int*)
call_pre_post (*bit*)

virtual function pre_start()

Task

pre_start

This task is a user-definable callback that is called before the optional execution of *pre_body*. This method should not be called directly by the user.

virtual function pre_body()

Task

pre_body

This task is a user-definable callback that is called before the execution of *body* *only* when the sequence is started with *start*. If *start* is called with *call_pre_post* set to 0, *pre_body* is not called. This method should not be called directly by the user.

virtual function pre_do(bit is_item)

Task

pre_do

This task is a user-definable callback task that is called ~on the parent sequence~, if any sequence has issued a wait_for_grant() call and after the sequencer has selected this sequence, and before the item is randomized.

Although pre_do is a task, consuming simulation cycles may result in unexpected behavior on the driver.

This method should not be called directly by the user.

Parameters

is_item (*bit*)

virtual function body()

Task

body

This is the user-defined task where the main sequence code resides. This method should not be called directly by the user.

virtual function post_body()

Task

post_body

This task is a user-definable callback task that is called after the execution of *body* *only* when the sequence is started with *start*. If *start* is called with *call_pre_post* set to 0, *post_body* is not called. This task is a user-definable callback task that is called after the execution of the body, unless the sequence is started with call_pre_post = 0. This method should not be called directly by the user.

virtual function post_start()

Task

post_start

This task is a user-definable callback that is called after the optional execution of *post_body*. This method should not be called directly by the user.

virtual function wait_for_relevant()

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.

function lock(uvm_sequencer_base sequencer = null)

Task

lock

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.

Parameters

sequencer (*uvm_sequencer_base*)

function grab(uvm_sequencer_base sequencer = null)

Task

grab

Requests a lock on the specified sequencer. If no argument is supplied, the lock will be requested on the current default sequencer.

A grab request is put in front of the arbitration queue. It will be arbitrated before any other requests. A grab is granted when no other grabs or locks are blocking this sequence.

The grab call will return when the grab has been granted.

Parameters

sequencer (*uvm_sequencer_base*)

virtual function start_item(uvm_sequence_item item, int set_priority = -1, uvm_-sequencer_base sequencer = null)

Function

start_item

start_item and *finish_item* together will initiate operation of a sequence item. If the item has not already been initialized using create_item, then it will be initialized here to use the default sequencer specified by m_sequencer. Randomization may be done between start_item and finish_item to ensure late generation

Parameters

item (*uvm_sequence_item*)

set_priority (`int`)

sequencer (*uvm_sequencer_base*)

virtual function finish_item(uvm_sequence_item item, int set_priority = -1)

Function

finish_item

finish_item, together with start_item together will initiate operation of a sequence_item. 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.

Parameters

item (*uvm_sequence_item*)

set_priority (*int*)

virtual function wait_for_grant(int item_priority = -1, bit lock_request = 0)

Task

wait_for_grant

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.

Parameters

item_priority (*int*)

lock_request (*bit*)

virtual function wait_for_item_done(int transaction_id = -1)

Parameters

transaction_id (*int*)

virtual function get_base_response(uvm_sequence_item response, int transaction_-id = -1)

Function- get_base_response

Parameters

response (*uvm_sequence_item*)

transaction_id (*int*)

function do_sequence_kind(int unsigned req_kind)

Task- do_sequence_kind

This task will start a sequence of kind specified by *req_kind* , which can be obtained using the *get_seq_kind* method.

Parameters

req_kind (*int unsigned*)

function create_and_start_sequence_by_name(string seq_name)

Task- create_and_start_sequence_by_name

Internal method.

Parameters

seq_name (*string*)

15.2.0.8 Class uvm_pkg::uvm_sequence_item

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*

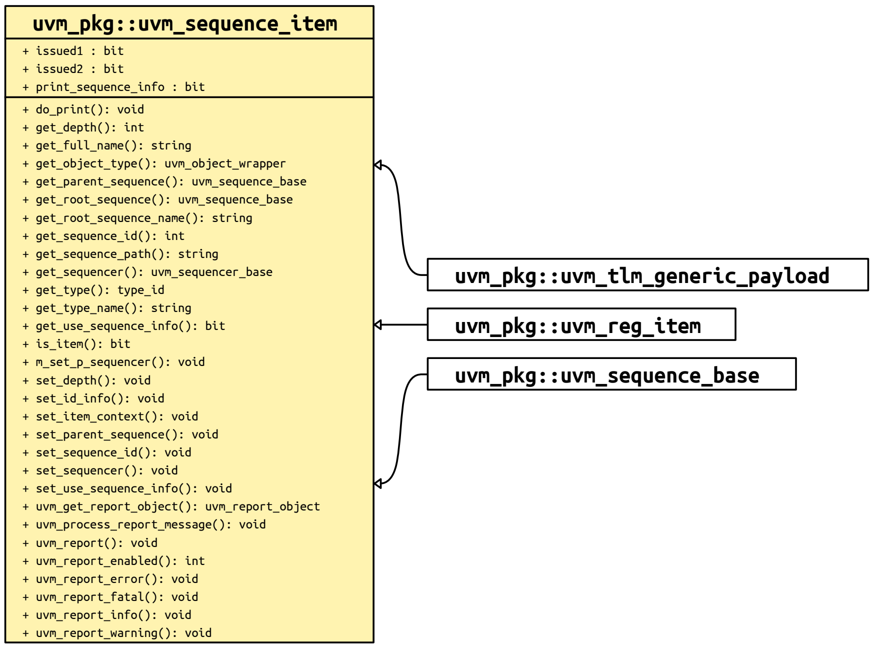

uvm_pkg::uvm_sequence_item
+ issued1 : bit
+ issued2 : bit
+ print_sequence_info : bit
+ do_print(): void
+ get_depth(): int
+ get_full_name(): string
+ get_object_type(): uvm_object_wrapper
+ get_parent_sequence(): uvm_sequence_base
+ get_root_sequence(): uvm_sequence_base
+ get_root_sequence_name(): string
+ get_sequence_id(): int
+ get_sequence_path(): string
+ get_sequencer(): uvm_sequencer_base
+ get_type(): type_id
+ get_type_name(): string
+ get_use_sequence_info(): bit
+ is_item(): bit
+ m_set_p_sequencer(): void
+ set_depth(): void
+ set_id_info(): void
+ set_item_context(): void
+ set_parent_sequence(): void
+ set_sequence_id(): void
+ set_sequencer(): void
+ set_use_sequence_info(): void
+ uvm_get_report_object(): uvm_report_object
+ uvm_process_report_message(): void
+ uvm_report(): void
+ uvm_report_enabled(): int
+ uvm_report_error(): void
+ uvm_report_fatal(): void
+ uvm_report_info(): void
+ uvm_report_warning(): void
uvm_pkg::uvm_tlm_generic_payload
uvm_pkg::uvm_reg_item
uvm_pkg::uvm_sequence_base

Fig. 81: Inheritance Diagram of uvm_sequence_item

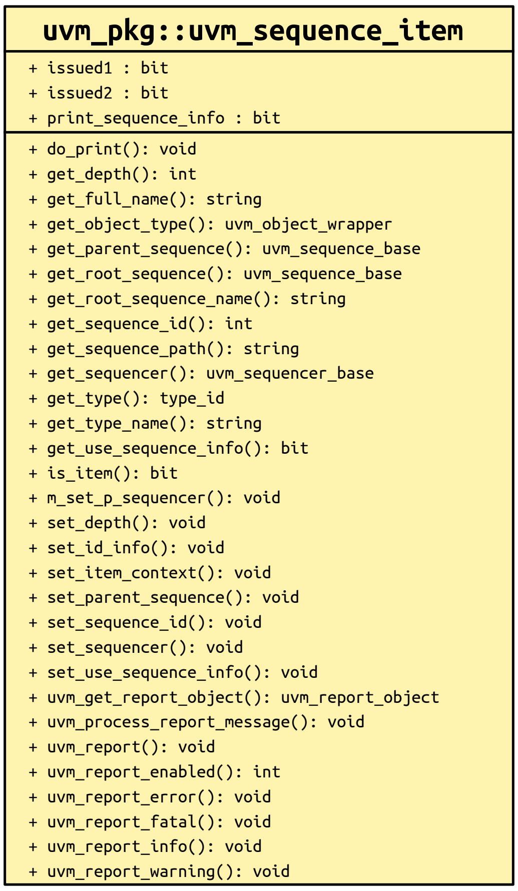

uvm_pkg::uvm_sequence_item
+ issued1 : bit
+ issued2 : bit
+ print_sequence_info : bit
+ do_print(): void
+ get_depth(): int
+ get_full_name(): string
+ get_object_type(): uvm_object_wrapper
+ get_parent_sequence(): uvm_sequence_base
+ get_root_sequence(): uvm_sequence_base
+ get_root_sequence_name(): string
+ get_sequence_id(): int
+ get_sequence_path(): string
+ get_sequencer(): uvm_sequencer_base
+ get_type(): type_id
+ get_type_name(): string
+ get_use_sequence_info(): bit
+ is_item(): bit
+ m_set_p_sequencer(): void
+ set_depth(): void
+ set_id_info(): void
+ set_item_context(): void
+ set_parent_sequence(): void
+ set_sequence_id(): void
+ set_sequencer(): void
+ set_use_sequence_info(): void
+ uvm_get_report_object(): uvm_report_object
+ uvm_process_report_message(): void
+ uvm_report(): void
+ uvm_report_enabled(): int
+ uvm_report_error(): void
+ uvm_report_fatal(): void
+ uvm_report_info(): void
+ uvm_report_warning(): void

Fig. 82: Collaboration Diagram of uvm_sequence_item

CLASS

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.

Table 238: Variables

Name	Type	Description
issued1	bit	
issued2	bit	
print_sequence_info	bit	

Constructors

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

Function

new

The constructor method for uvm_sequence_item.
Parameters
`name` (*`string`*)

Functions

`virtual function string get_type_name()`

`function void set_sequence_id(int id)`

Function- set_sequence_id
Parameters
`id` (*`int`*)

`function int get_sequence_id()`

Function

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. &96;uvm_do_*, 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.

`function void set_item_context(uvm_sequence_base parent_seq, uvm_sequencer_-base sequencer = null)`

Function

set_item_context

Set the sequence and sequencer execution context for a sequence item
Parameters
`parent_seq` (*uvm_sequence_base*)
`sequencer` (*uvm_sequencer_base*)

`function void set_use_sequence_info(bit value)`

Function

set_use_sequence_info
Parameters
`value` (*`bit`*)

`function bit get_use_sequence_info()`

Function

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.

`function void set_id_info(uvm_sequence_item item)`

Function

set_id_info

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.

Parameters

`item` (*uvm_sequence_item*)

`virtual  function void set_sequencer(uvm_sequencer_base sequencer)`

Function

set_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.

Parameters

`sequencer` (*uvm_sequencer_base*)

`function uvm_sequencer_base get_sequencer()`

Function

get_sequencer

Returns a reference to the default sequencer used by this sequence.

Return type

uvm_sequencer_base

`function void set_parent_sequence(uvm_sequence_base parent)`

Function

set_parent_sequence

Sets the parent sequence of this sequence_item. This is used to identify the source sequence of a sequence_item.

Parameters

`parent` (*uvm_sequence_base*)

`function uvm_sequence_base get_parent_sequence()`

Function

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* .

Return type

uvm_sequence_base

`function void set_depth(int value)`

Function

set_depth

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.

Parameters

`value` (`int`)

`function int get_depth()`

Function

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.

`virtual function bit is_item()`

> ***Function***
>
> 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).

`virtual function string get_full_name()`

> Function- get_full_name
>
> Internal method; overrides must follow same naming convention

`function string get_root_sequence_name()`

> ***Function***
>
> get_root_sequence_name
>
> Provides the name of the root sequence (the top-most parent sequence).

`function uvm_sequence_base get_root_sequence()`

> ***Function***
>
> get_root_sequence
>
> Provides a reference to the root sequence (the top-most parent sequence).
>
> Return type
>
> *uvm_sequence_base*

`function string get_sequence_path()`

> ***Function***
>
> 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.

`virtual function uvm_report_object uvm_get_report_object()`

> ***Group***
>
> Reporting Interface
>
> Sequence items and sequences will use the sequencer which they are associated with for reporting messages. If no sequencer has been set for the item/sequence using *set_sequencer* or indirectly via *uvm_sequence_base::start_item* or *uvm_sequence_base::start*), then the global reporter will be used.
>
> Return type
>
> *uvm_report_object*

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

> Parameters
>
> - **`verbosity`** (*`int`*)
> - **`severity`** (*uvm_severity*)
> - **`id`** (*`string`*)

`virtual function void uvm_report(uvm_severity severity, string id, string message, int verbosity = (severity==uvm_severity'(UVM_ERROR))?UVM_LOW:(severity==uvm_severity'(UVM_FATAL))?UVM_NONE:UVM_MEDIUM, string filename = "", int line = 0, string context_name = "", bit report_enabled_checked = 0)`

> ***Function***
>
> uvm_report
>
> Parameters
>
> - **`severity`** (*uvm_severity*)
> - **`id`** (*`string`*)
> - **`message`** (*`string`*)
> - **`verbosity`** (*`int`*)
> - **`filename`** (*`string`*)
> - **`line`** (*`int`*)
> - **`context_name`** (*`string`*)
> - **`report_enabled_checked`** (*`bit`*)

virtual function void uvm_report_info(string id, string message, int verbosity = UVM_MEDIUM, string filename = "", int line = 0, string context_-name = "", bit report_enabled_checked = 0)

Function

uvm_report_info

Parameters

- **id** (*string*)
- **message** (*string*)
- **verbosity** (*int*)
- **filename** (*string*)
- **line** (*int*)
- **context_name** (*string*)
- **report_enabled_checked** (*bit*)

virtual function void uvm_report_warning(string id, string message, int verbosity = UVM_MEDIUM, string filename = "", int line = 0, string context_-name = "", bit report_enabled_checked = 0)

Function

uvm_report_warning

Parameters

- **id** (*string*)
- **message** (*string*)
- **verbosity** (*int*)
- **filename** (*string*)
- **line** (*int*)
- **context_name** (*string*)
- **report_enabled_checked** (*bit*)

virtual function void uvm_report_error(string id, string message, int verbosity = UVM_LOW, string filename = "", int line = 0, string context_-name = "", bit report_enabled_checked = 0)

Function

uvm_report_error

Parameters

- **id** (*string*)
- **message** (*string*)
- **verbosity** (*int*)
- **filename** (*string*)
- **line** (*int*)
- **context_name** (*string*)
- **report_enabled_checked** (*bit*)

virtual function void uvm_report_fatal(string id, string message, int verbosity = UVM_NONE, string filename = "", int line = 0, string context_-name = "", bit report_enabled_checked = 0)

Function

uvm_report_fatal

These are the primary reporting methods in the UVM. uvm_sequence_item derived types delegate these functions to their associated sequencer if they have one, or to the global reporter. See <uvm_report_object::Reporting> for details on the messaging functions.

Parameters

- **id** (*string*)
- **message** (*string*)
- **verbosity** (*int*)
- **filename** (*string*)
- **line** (*int*)
- **context_name** (*string*)
- **report_enabled_checked** (*bit*)

`virtual function void uvm_process_report_message(uvm_report_message report_message)`

Parameters

`report_message` (*uvm_report_message*)

`virtual function void do_print(uvm_printer printer)`

Function- do_print

Internal method

Parameters

`printer` (*uvm_printer*)

15.2.0.9 Class uvm_pkg::uvm_sequence_library

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*
↪*uvm_pkg* :: *uvm_sequence_library*

CLASS

uvm_sequence_library

The *uvm_sequence_library* is a sequence that contains a list of registered sequence types. It can be configured to create and execute these sequences any number of times using one of several modes of operation, including a user-defined mode.

When started (as any other sequence), the sequence library will randomly select and execute a sequence from its *sequences* queue. If in <UVM_SEQ_LIB_RAND> mode, its *select_rand* property is randomized and used as an index into *sequences* . When in <UVM_SEQ_LIB_RANDC> mode, the *select_randc* property is used. When in <UVM_SEQ_LIB_ITEM> mode, only sequence items of the *REQ* type are generated and executed--no sequences are executed. Finally, when in <UVM_SEQ_LIB_USER> mode, the *select_sequence* method is called to obtain the index for selecting the next sequence to start. Users can override this method in subtypes to implement custom selection algorithms.

Creating a subtype of a sequence library requires invocation of the *`uvm_sequence_library_utils* macro in its declaration and calling the *init_sequence_library* method in its constructor. The macro and function are needed to populate the sequence library with any sequences that were statically registered with it or any of its base classes.

```
class my_seq_lib extends uvm_sequence_library #(my_item);
  `uvm_object_utils(my_seq_lib)
  `uvm_sequence_library_utils(my_seq_lib)
   function new(string name="");
     super.new(name);
     init_sequence_library();
   endfunction
   ...
endclass
```

Table 239: Parameters

Name	Default value	Description
REQ	uvm_sequence_item	
RSP	REQ	

Table 240: Variables

Name	Type	Description

selection_mode	*uvm_sequence_lib_mode*	***Variable***
		 selection_mode
		 Specifies the mode used to select sequences for execution
		 If you do not have access to an instance of the library, use the configuration resource interface.
		 The following example sets the *config_seq_lib* as the default sequence for the 'main' phase on the sequencer to be located at "env.agent.sequencer" and set the selection mode to <UVM_SEQ_LIB_RANDC>. If the settings are being done from within a component, the first argument must be *this* and the second argument a path relative to that component.
		 `uvm_config_db #(uvm_object_wrapper)::set(null, "env.agent.sequencer.main_phase", "default_sequence", main_seq_lib::get_type());`
		 `uvm_config_db #(uvm_sequence_lib_mode)::set(null, "env.agent.sequencer.main_phase", "default_sequence.selection_mode", UVM_SEQ_LIB_RANDC);`
		 Alternatively, you may create an instance of the sequence library a priori, initialize all its parameters, randomize it, then set it to run as-is on the sequencer.
		 `main_seq_lib my_seq_lib;`
		 `my_seq_lib = new("my_seq_lib");`
		 `my_seq_lib.selection_mode = UVM_SEQ_LIB_RANDC;`
		 `my_seq_lib.min_random_count = 500;`
		 `my_seq_lib.max_random_count = 1000;`
		 `void'(my_seq_lib.randomize());`
		 `uvm_config_db #(uvm_sequence_base)::set(null, "env.agent.sequencer.main_phase", "default_sequence", my_seq_lib);`

continues on next page

Table 240 – continued from previous page

Name	Type	Description
min_random_count	int unsigned	***Variable*** min_random_count Sets the minimum number of items to execute. Use the configuration mechanism to set. See *selection_mode* for an example.
max_random_count	int unsigned	***Variable*** max_random_count Sets the maximum number of items to execute. Use the configuration mechanism to set. See *selection_mode* for an example.
sequence_count	int unsigned	***Variable*** sequence_count Specifies the number of sequences to execute when this sequence library is started. If in <UVM_SEQ_LIB_ITEM> mode, specifies the number of sequence items that will be generated.
select_rand	int unsigned	***Variable*** select_rand The index variable that is randomized to select the next sequence to execute when in UVM_SEQ_LIB_RAND mode Extensions may place additional constraints on this variable.
select_randc	bit[15:0]	***Variable*** select_randc The index variable that is randomized to select the next sequence to execute when in UVM_SEQ_LIB_RANDC mode Extensions may place additional constraints on this variable.
type_name	string	

Table 241: Constraints

Name	Description
valid_rand_selection	Constraint: valid_rand_selection Constrains <select_rand> to be a valid index into the ~sequences~ array
valid_randc_selection	Constraint: valid_randc_selection Constrains <select_randc> to be a valid index into the ~sequences~ array
valid_sequence_count	Constraint: valid_sequence_count Constrains <sequence_count> to lie within the range defined by <min_random_count> and <max_random_count>.

Table 242: Typedefs

Name	Actual Type	Description
this_type	*uvm_sequence_library#(REQ, RSP)*	

Constructors

`function new(string name = "")`

Function

new

Create a new instance of this class. New

Parameters

`name` (*`string`*)

Functions

`virtual function string get_type_name()`

Function

get_type_name

Get the type name of this class. Get_type_name

`virtual function int unsigned select_sequence(int unsigned max)`

Function

select_sequence

Generates an index used to select the next sequence to execute. Overrides must return a value between 0 and *max* , inclusive. Used only for <UVM_SEQ_LIB_USER> selection mode. The default implementation returns 0, incrementing on successive calls, wrapping back to 0 when reaching *max* . Select_sequence

Parameters

`max` (*`int unsigned`*)

`static function void add_typewide_sequence(uvm_object_wrapper seq_type)`

Function

add_typewide_sequence

Registers the provided sequence type with this sequence library type. The sequence type will be available for selection by all instances of this class. Sequence types already registered are silently ignored. Add_typewide_sequence

Parameters

`seq_type` (*uvm_object_wrapper*)

`static function void add_typewide_sequences(uvm_object_wrapper seq_types)`

Function

add_typewide_sequences

Registers the provided sequence types with this sequence library type. The sequence types will be available for selection by all instances of this class. Sequence types already registered are silently ignored. Add_typewide_sequences

Parameters

`seq_types` (*uvm_object_wrapper*)

`function void add_sequence(uvm_object_wrapper seq_type)`

Function

add_sequence

Registers the provided sequence type with this sequence library instance. Sequence types already registered are silently ignored. Add_sequence

Parameters

`seq_type` (*uvm_object_wrapper*)

virtual function void add_sequences(uvm_object_wrapper seq_types)

Function

add_sequences

Registers the provided sequence types with this sequence library instance. Sequence types already registered are silently ignored. Add_sequences

Parameters

seq_types (*uvm_object_wrapper*)

virtual function void remove_sequence(uvm_object_wrapper seq_type)

Function

remove_sequence

Removes the given sequence type from this sequence library instance. If the type was registered statically, the sequence queues of all instances of this library will be updated accordingly. A warning is issued if the sequence is not registered. Remove_sequence

Parameters

seq_type (*uvm_object_wrapper*)

virtual function void get_sequences(uvm_object_wrapper seq_types)

Function

get_sequences

Append to the provided *seq_types* array the list of registered *sequences* . Get_sequences

Parameters

seq_types (*uvm_object_wrapper*)

function void init_sequence_library()

Function

init_sequence_library

All subtypes of this class must call init_sequence_library in its constructor. Init_sequence_library

virtual function void do_print(uvm_printer printer)

Do_print

Parameters

printer (*uvm_printer*)

Tasks

virtual function execute(uvm_object_wrapper wrap)

Execute

Parameters

wrap (*uvm_object_wrapper*)

virtual function body()

Body

15.2.0.10 Class uvm_pkg::uvm_sequence_library_cfg

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_sequence_library_cfg*

Class

uvm_sequence_library_cfg

A convenient container class for configuring all the sequence library parameters using a single *set* command.

```
uvm_sequence_library_cfg cfg;
cfg = new("seqlib_cfg", UVM_SEQ_LIB_RANDC, 1000, 2000);

uvm_config_db #(uvm_sequence_library_cfg)::set(null,
                                   "env.agent.sequencer.main_ph",
                                   "default_sequence.config",
                                   cfg);
```

Table 243: Variables

Name	Type	Description
selection_mode	*uvm_sequence_lib_mode*	
min_random_count	int unsigned	
max_random_count	int unsigned	

Constructors

`function new(string name = "", uvm_sequence_lib_mode mode = UVM_SEQ_LIB_RAND, int unsigned min = 1, int unsigned max = 10)`

Parameters
- **`name`** (*`string`*)
- **`mode`** (*uvm_sequence_lib_mode*)
- **`min`** (*`int unsigned`*)
- **`max`** (*`int unsigned`*)

15.2.0.11 Class uvm_pkg::uvm_sequence_process_wrapper

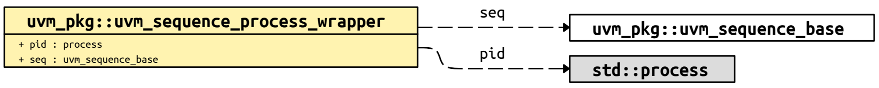

uvm_pkg::uvm_sequence_process_wrapper
+ pid : process
+ seq : uvm_sequence_base
seq
pid
uvm_pkg::uvm_sequence_base
std::process

Fig. 83: Collaboration Diagram of uvm_sequence_process_wrapper

Utility class for tracking default_sequences

Table 244: Variables

Name	Type	Description
pid	process	
seq	*uvm_sequence_base*	

15.2.0.12 Class uvm_pkg::uvm_sequence_request

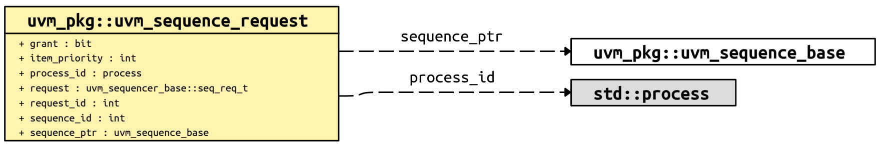

uvm_pkg::uvm_sequence_request
+ grant : bit
+ item_priority : int
+ process_id : process
+ request : uvm_sequencer_base::seq_req_t
+ request_id : int
+ sequence_id : int
+ sequence_ptr : uvm_sequence_base
sequence_ptr
process_id
uvm_pkg::uvm_sequence_base
std::process

Fig. 84: Collaboration Diagram of uvm_sequence_request

Class- uvm_sequence_request

Table 245: Variables

Name	Type	Description
grant	bit	
sequence_id	int	
request_id	int	
item_priority	int	
process_id	process	
request	*uvm_sequencer_-base::seq_req_t*	
sequence_ptr	*uvm_sequence_base*	

15.2.0.13 Class uvm_pkg::uvm_sequencer

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_sequencer_base*
↪*uvm_pkg* :: *uvm_sequencer_param_base*
↪*uvm_pkg* :: *uvm_sequencer*

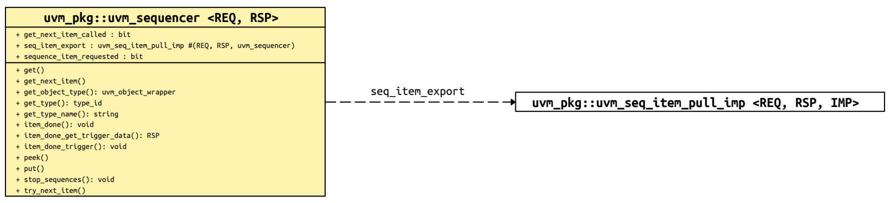

uvm_pkg::uvm_sequencer <REQ, RSP>
+ get_next_item_called : bit
+ seq_item_export : uvm_seq_item_pull_imp #(REQ, RSP, uvm_sequencer)
+ sequence_item_requested : bit
+ get()
+ get_next_item()
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string
+ item_done(): void
+ item_done_get_trigger_data(): RSP
+ item_done_trigger(): void
+ peek()
+ put()
+ stop_sequences(): void
+ try_next_item()
seq_item_export
uvm_pkg::uvm_seq_item_pull_imp <REQ, RSP, IMP>

Fig. 85: Collaboration Diagram of uvm_sequencer

CLASS

uvm_sequencer (REQ, RSP)

Table 246: Parameters

Name	Default value	Description
REQ	uvm_sequence_item	
RSP	REQ	

Table 247: Variables

Name	Type	Description
sequence_item_requested	bit	
get_next_item_called	bit	
seq_item_export	*uvm_seq_item_pull_imp#(uvm_sequence_item, uvm_sequence_item, uvm_sequencer#(uvm_sequence_item, uvm_sequence_item))*	***Variable*** seq_item_export This export provides access to this sequencer's implementation of the sequencer interface.

Table 248: Typedefs

Name	Actual Type	Description
this_type	*uvm_sequencer#(REQ, RSP)*	

Constructors

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

Function

new

Standard component constructor that creates an instance of this class using the given *name* and *parent* , if any. IMPLEMENTATION

Parameters
- **`name`** (`string`)
- **`parent`** (*uvm_component*)

Functions

`virtual function void stop_sequences()`

Function

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. Function- 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.

`virtual function string get_type_name()`

`virtual function void item_done(uvm_sequence_item item = null)`

Function

item_done

Indicates that the request is completed. Item_done

Parameters
- **`item`** (*uvm_sequence_item*)

`function void item_done_trigger(uvm_sequence_item item = null)`

Internal Methods

Do not use directly, not part of standard. Item_done_trigger

Parameters
- **`item`** (*uvm_sequence_item*)

`function RSP item_done_get_trigger_data()`

Return type
- *RSP*

Tasks

`virtual function get_next_item(uvm_sequence_item t)`

Task

get_next_item

Retrieves the next available item from a sequence. Get_next_item

Parameters
- **`t`** (*uvm_sequence_item*)

`virtual function try_next_item(uvm_sequence_item t)`

Task

try_next_item

Retrieves the next available item from a sequence if one is available. Try_next_item

Parameters
- **`t`** (*uvm_sequence_item*)

`virtual function put(uvm_sequence_item t)`

Task

put

Sends a response back to the sequence that issued the request. Put

Parameters

`t` (*uvm_sequence_item*)

`function get(uvm_sequence_item t)`

Task

get

Retrieves the next available item from a sequence. Get

Parameters

`t` (*uvm_sequence_item*)

`function peek(uvm_sequence_item t)`

Task

peek

Returns the current request item if one is in the FIFO. Peek

Parameters

`t` (*uvm_sequence_item*)

15.2.0.14 Class uvm_pkg::uvm_sequencer_analysis_fifo

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_tlm_fifo_base*
↪*uvm_pkg* :: *uvm_tlm_fifo*
↪*uvm_pkg* :: *uvm_sequencer_analysis_fifo*

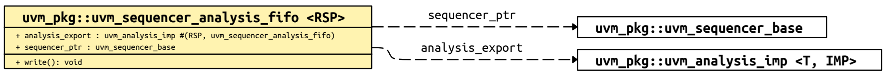

uvm_pkg::uvm_sequencer_analysis_fifo <RSP>
+ analysis_export : uvm_analysis_imp #(RSP, uvm_sequencer_analysis_fifo)
+ sequencer_ptr : uvm_sequencer_base
+ write(): void
sequencer_ptr
analysis_export
uvm_pkg::uvm_sequencer_base
uvm_pkg::uvm_analysis_imp <T, IMP>

Fig. 86: Collaboration Diagram of uvm_sequencer_analysis_fifo

Table 249: Parameters

Name	Default value	Description
RSP	uvm_sequence_item	

Table 250: Variables

Name	Type	Description
analysis_export	*uvm_analysis_imp#(uvm_sequence_item, uvm_sequencer_analysis_fifo#(uvm_sequence_item))*	
sequencer_ptr	*uvm_sequencer_base*	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)

Functions

`function void write(uvm_sequence_item t)`

Parameters
- **`t`** (*uvm_sequence_item*)

15.2.0.15 Class uvm_pkg::uvm_sequencer_base

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_sequencer_base*

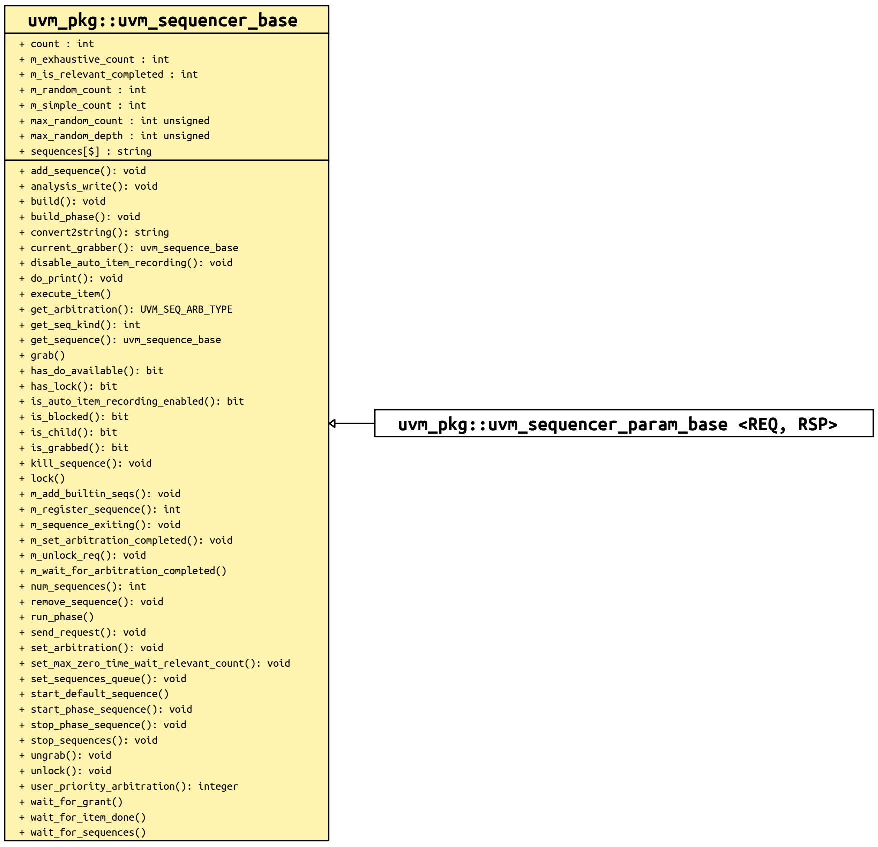

uvm_pkg::uvm_sequencer_base
+ count : int
+ m_exhaustive_count : int
+ m_is_relevant_completed : int
+ m_random_count : int
+ m_simple_count : int
+ max_random_count : int unsigned
+ max_random_depth : int unsigned
+ sequences[$] : string
+ add_sequence(): void
+ analysis_write(): void
+ build(): void
+ build_phase(): void
+ convert2string(): string
+ current_grabber(): uvm_sequence_base
+ disable_auto_item_recording(): void
+ do_print(): void
+ execute_item()
+ get_arbitration(): UVM_SEQ_ARB_TYPE
+ get_seq_kind(): int
+ get_sequence(): uvm_sequence_base
+ grab()
+ has_do_available(): bit
+ has_lock(): bit
+ is_auto_item_recording_enabled(): bit
+ is_blocked(): bit
+ is_child(): bit
+ is_grabbed(): bit
+ kill_sequence(): void
+ lock()
+ m_add_builtin_seqs(): void
+ m_register_sequence(): int
+ m_sequence_exiting(): void
+ m_set_arbitration_completed(): void
+ m_unlock_req(): void
+ m_wait_for_arbitration_completed()
+ num_sequences(): int
+ remove_sequence(): void
+ run_phase()
+ send_request(): void
+ set_arbitration(): void
+ set_max_zero_time_wait_relevant_count(): void
+ set_sequences_queue(): void
+ start_default_sequence()
+ start_phase_sequence(): void
+ stop_phase_sequence(): void
+ stop_sequences(): void
+ ungrab(): void
+ unlock(): void
+ user_priority_arbitration(): integer
+ wait_for_grant()
+ wait_for_item_done()
+ wait_for_sequences()
uvm_pkg::uvm_sequencer_param_base <REQ, RSP>

Fig. 87: Inheritance Diagram of uvm_sequencer_base

CLASS

uvm_sequencer_base

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

Table 251: Variables

Name	Type	Description
count	int	Variable- count Sets the number of items to execute. Supercedes the max_random_count variable for uvm_random_sequence class for backward compatibility.

continues on next page

Table 251 – continued from previous page

Name	Type	Description
max_random_count	int unsigned	
max_random_depth	int unsigned	
sequences	string	

Constructors

`function new(string name, uvm_component parent)`

Function

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. New

Parameters

- **`name`** (`string`)
- **`parent`** (*uvm_component*)

Enums

`seq_req_t`

Enum Items

- SEQ_TYPE_REQ
- SEQ_TYPE_LOCK
- SEQ_TYPE_GRAB
  FIXME SEQ_TYPE_GRAB is unused

Functions

`function bit is_child(uvm_sequence_base parent, uvm_sequence_base child)`

Function

is_child

Returns 1 if the child sequence is a child of the parent sequence, 0 otherwise. Is_child

Parameters

- **`parent`** (*uvm_sequence_base*)
- **`child`** (*uvm_sequence_base*)

`virtual function integer user_priority_arbitration(integer avail_sequences)`

Function

user_priority_arbitration

When the sequencer arbitration mode is set to UVM_SEQ_ARB_USER (via the *set_arbitration* method), the sequencer will call this function each time that it needs to arbitrate among sequences.

Derived sequencers may override this method to perform a custom arbitration policy. The override must return one of the entries from the avail_sequences queue, which are indexes into an internal queue, arb_sequence_q.

The default implementation behaves like UVM_SEQ_ARB_FIFO, which returns the entry at avail_sequences[0]. User_priority_arbitration

Parameters

- **`avail_sequences`** (`integer`)

`virtual function void start_phase_sequence(uvm_phase phase)`

Start_phase_sequence

Parameters

- **`phase`** (*uvm_phase*)

`virtual function void stop_phase_sequence(uvm_phase phase)`

Function

stop_phase_sequence

Stop the default sequence for this phase, if any exists, and it is still executing. Stop_phase_sequence
Parameters
`phase` (*uvm_phase*)

`function bit is_blocked(uvm_sequence_base sequence_ptr)`

Function

is_blocked

Returns 1 if the sequence referred to by sequence_ptr is currently locked out of the sequencer. It will return 0 if the sequence is currently allowed to issue operations.

Note that even when a sequence is not blocked, it is possible for another sequence to issue a lock before this sequence is able to issue a request or lock. Is_blocked
Parameters
`sequence_ptr` (*uvm_sequence_base*)

`function bit has_lock(uvm_sequence_base sequence_ptr)`

Function

has_lock

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. Has_lock
Parameters
`sequence_ptr` (*uvm_sequence_base*)

`virtual function void unlock(uvm_sequence_base sequence_ptr)`

Function

unlock

Removes any locks and grabs obtained by the specified sequence_ptr. Unlock
Parameters
`sequence_ptr` (*uvm_sequence_base*)

`virtual function void ungrab(uvm_sequence_base sequence_ptr)`

Function

ungrab

Removes any locks and grabs obtained by the specified sequence_ptr. Ungrab
Parameters
`sequence_ptr` (*uvm_sequence_base*)

`virtual function void stop_sequences()`

Function

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. Stop_sequences

`virtual function bit is_grabbed()`

Function

is_grabbed

Returns 1 if any sequence currently has a lock or grab on this sequencer, 0 otherwise. Is_grabbed

`virtual function uvm_sequence_base current_grabber()`

Function

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. Current_grabber

Return type

uvm_sequence_base

`virtual function bit has_do_available()`

Function

has_do_available

Returns 1 if any sequence running on this sequencer is ready to supply a transaction, 0 otherwise. A sequence is ready if it is not blocked (via *grab* or *lock* and *is_relevant* returns 1. Has_do_available

`function void set_arbitration(UVM_SEQ_ARB_TYPE val)`

Function

set_arbitration

Specifies the arbitration mode for the sequencer. It is one of

UVM_SEQ_ARB_FIFO

Requests are granted in FIFO order (default)

UVM_SEQ_ARB_WEIGHTED

Requests are granted randomly by weight

UVM_SEQ_ARB_RANDOM

Requests are granted randomly

UVM_SEQ_ARB_STRICT_FIFO

Requests at highest priority granted in FIFO order

UVM_SEQ_ARB_STRICT_RANDOM

Requests at highest priority granted in randomly

UVM_SEQ_ARB_USER

Arbitration is delegated to the user-defined function, user_priority_arbitration. That function will specify the next sequence to grant.

The default user function specifies FIFO order. Set_arbitration

Parameters

`val` (*UVM_SEQ_ARB_TYPE*)

`function UVM_SEQ_ARB_TYPE get_arbitration()`

Function

get_arbitration

Return the current arbitration mode set for this sequencer. See *set_arbitration* for a list of possible modes. Get_arbitration

Return type

UVM_SEQ_ARB_TYPE

`virtual function void send_request(uvm_sequence_base sequence_ptr, uvm_sequence_-item t, bit rerandomize = 0)`

Function

send_request

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. Send_request

Parameters

`sequence_ptr` (*uvm_sequence_base*)

`t` (*uvm_sequence_item*)

`rerandomize` (`bit`)

`virtual function void set_max_zero_time_wait_relevant_count(int new_val)`

Function

set_max_zero_time_wait_relevant_count

Can be called at any time to change the maximum number of times wait_for_relevant() can be called by the sequencer in zero time before an error is declared. The default maximum is 10. Set_max_zero_time_wait_relevant_count

Parameters

`new_val` (*`int`*)

`function void kill_sequence(uvm_sequence_base sequence_ptr)`

Kill_sequence

Parameters

`sequence_ptr` (*uvm_sequence_base*)

`virtual function void analysis_write(uvm_sequence_item t)`

Analysis_write

Parameters

`t` (*uvm_sequence_item*)

`virtual function void build()`

`virtual function void build_phase(uvm_phase phase)`

Build_phase

Parameters

`phase` (*uvm_phase*)

`virtual function void do_print(uvm_printer printer)`

Do_print

Parameters

`printer` (*uvm_printer*)

`virtual function string convert2string()`

Convert2string

`virtual function void disable_auto_item_recording()`

Access to following internal methods provided via seq_item_export

`virtual function bit is_auto_item_recording_enabled()`

`function void add_sequence(string type_name)`

Add_sequence

Adds a sequence of type specified in the type_name paramter to the sequencer's sequence library.

Parameters

`type_name` (*`string`*)

`function void remove_sequence(string type_name)`

Remove_sequence

Parameters

`type_name` (*`string`*)

`function void set_sequences_queue(string sequencer_sequence_lib)`

Set_sequences_queue

Parameters

`sequencer_sequence_lib` (*`string`*)

`function int get_seq_kind(string type_name)`

Get_seq_kind

Returns an int seq_kind correlating to the sequence of type type_name in the sequencers sequence library. If the named sequence is not registered a SEQNF warning is issued and -1 is returned.

Parameters

`type_name` (*`string`*)

`function uvm_sequence_base get_sequence(int req_kind)`

Get_sequence

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.

Parameters
req_kind (*int*)
Return type
uvm_sequence_base

function int num_sequences()

Num_sequences

Tasks

virtual function execute_item(uvm_sequence_item item)

Task

execute_item

Executes the given transaction *item* directly on this sequencer. A temporary parent sequence is automatically created for the *item* . There is no capability to retrieve responses. If the driver returns responses, they will accumulate in the sequencer, eventually causing response overflow unless *uvm_sequence_base::set_response_queue_error_report_disabled* is called. Execute_item

Parameters
item (*uvm_sequence_item*)

virtual function wait_for_grant(uvm_sequence_base sequence_ptr, int item_priority = -1, bit lock_request = 0)

Task

wait_for_grant

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_grant

Parameters
sequence_ptr (*uvm_sequence_base*)
item_priority (*int*)
lock_request (*bit*)

virtual function wait_for_item_done(uvm_sequence_base sequence_ptr, int transaction_id)

Task

wait_for_item_done

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. Wait_for_item_done

Parameters
sequence_ptr (*uvm_sequence_base*)
transaction_id (*int*)

virtual function lock(uvm_sequence_base sequence_ptr)

Task

lock

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. Lock

Parameters

`sequence_ptr` (*uvm_sequence_base*)

`virtual function grab(uvm_sequence_base sequence_ptr)`

Task

grab

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. Grab

Parameters

`sequence_ptr` (*uvm_sequence_base*)

`virtual function wait_for_sequences()`

Task

wait_for_sequences

Waits for a sequence to have a new item available. Uses *uvm_wait_for_nba_region* to give a sequence as much time as possible to deliver an item before advancing time. Wait_for_sequences

`virtual function start_default_sequence()`

Start_default_sequence

Called when the run phase begins, this method starts the default sequence, as specified by the default_sequence member variable.

`virtual function run_phase(uvm_phase phase)`

Run_phase

Parameters

`phase` (*uvm_phase*)

15.2.0.16 Class uvm_pkg::uvm_sequencer_param_base

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_sequencer_base*
↪*uvm_pkg* :: *uvm_sequencer_param_base*

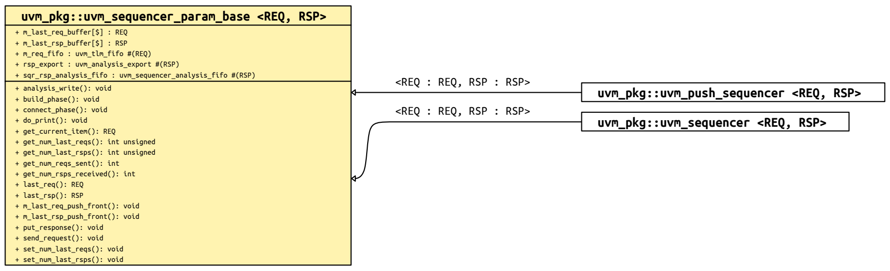

uvm_pkg::uvm_sequencer_param_base <REQ, RSP>
+ m_last_req_buffer[$] : REQ
+ m_last_rsp_buffer[$] : RSP
+ m_req_fifo : uvm_tlm_fifo #(REQ)
+ rsp_export : uvm_analysis_export #(RSP)
+ sqr_rsp_analysis_fifo : uvm_sequencer_analysis_fifo #(RSP)
+ analysis_write(): void
+ build_phase(): void
+ connect_phase(): void
+ do_print(): void
+ get_current_item(): REQ
+ get_num_last_reqs(): int unsigned
+ get_num_last_rsps(): int unsigned
+ get_num_reqs_sent(): int
+ get_num_rsps_received(): int
+ last_req(): REQ
+ last_rsp(): RSP
+ m_last_req_push_front(): void
+ m_last_rsp_push_front(): void
+ put_response(): void
+ send_request(): void
+ set_num_last_reqs(): void
+ set_num_last_rsps(): void
<REQ : REQ, RSP : RSP>
uvm_pkg::uvm_push_sequencer <REQ, RSP>
<REQ : REQ, RSP : RSP>
uvm_pkg::uvm_sequencer <REQ, RSP>

Fig. 88: Inheritance Diagram of uvm_sequencer_param_base

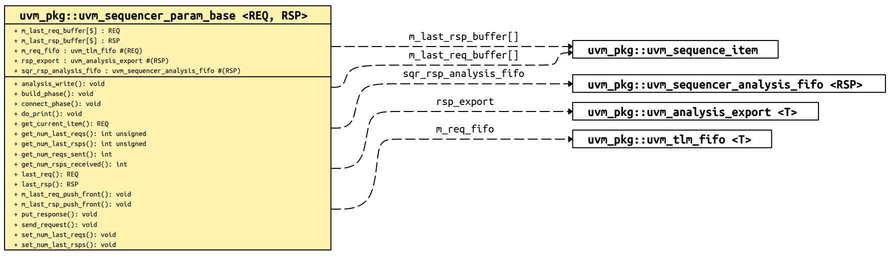

uvm_pkg::uvm_sequencer_param_base <REQ, RSP>
+ m_last_req_buffer[$] : REQ
+ m_last_rsp_buffer[$] : RSP
+ m_req_fifo : uvm_tlm_fifo #(REQ)
+ rsp_export : uvm_analysis_export #(RSP)
+ sqr_rsp_analysis_fifo : uvm_sequencer_analysis_fifo #(RSP)
+ analysis_write(): void
+ build_phase(): void
+ connect_phase(): void
+ do_print(): void
+ get_current_item(): REQ
+ get_num_last_reqs(): int unsigned
+ get_num_last_rsps(): int unsigned
+ get_num_reqs_sent(): int
+ get_num_rsps_received(): int
+ last_req(): REQ
+ last_rsp(): RSP
+ m_last_req_push_front(): void
+ m_last_rsp_push_front(): void
+ put_response(): void
+ send_request(): void
+ set_num_last_reqs(): void
+ set_num_last_rsps(): void
m_last_rsp_buffer[]
m_last_req_buffer[]
uvm_pkg::uvm_sequence_item
sqr_rsp_analysis_fifo
uvm_pkg::uvm_sequencer_analysis_fifo <RSP>
rsp_export
uvm_pkg::uvm_analysis_export <T>
m_req_fifo
uvm_pkg::uvm_tlm_fifo <T>

Fig. 89: Collaboration Diagram of uvm_sequencer_param_base

CLASS

uvm_sequencer_param_base (REQ, RSP)

Extends *uvm_sequencer_base* with an API depending on specific request (REQ) and response (RSP) types.

Table 252: Parameters

Name	Default value	Description
REQ	uvm_sequence_item	
RSP	REQ	

Table 253: Variables

Name	Type	Description
sqr_rsp_analysis_fifo	*uvm_sequencer_analysis_fifo#(uvm_sequence_item)*	

continues on next page

Table 253 – continued from previous page

Name	Type	Description

rsp_export	*uvm_analysis_export#(uvm_sequence_item)*	***Port***
		
		rsp_export
		
		Drivers or monitors can connect to this port to send responses to the sequencer. Alternatively, a driver can send responses via its seq_item_port.
		
		`seq_item_port.item_done(response)`
		`seq_item_port.put(response)`
		`rsp_port.write(response)    <--- via this export`
		
		The rsp_port in the driver and/or monitor must be connected to the rsp_export in this sequencer in order to send responses through the response analysis port.

Table 254: Typedefs

Name	Actual Type	Description
this_type	*uvm_sequencer_param_base#(REQ, RSP)*	
req_type	*REQ*	
rsp_type	*RSP*	

Constructors

function new(string name, uvm_component parent)

Function

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. New

Parameters

name (*string*)
parent (*uvm_component*)

Functions

virtual function void send_request(uvm_sequence_base sequence_ptr, uvm_sequence_item t, bit rerandomize = 0)

Function

send_request

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. Send_request

Parameters

sequence_ptr (*uvm_sequence_base*)
t (*uvm_sequence_item*)
rerandomize (*bit*)

function REQ get_current_item()

Function

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.

Return type
    REQ

`function int get_num_reqs_sent()`

Function

get_num_reqs_sent

Returns the number of requests that have been sent by this sequencer. Get_num_reqs_sent

`function void set_num_last_reqs(int unsigned max)`

Function

set_num_last_reqs

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. Set_num_last_reqs

Parameters
    max (*int unsigned*)

`function int unsigned get_num_last_reqs()`

Function

get_num_last_reqs

Returns the size of the last requests buffer, as set by set_num_last_reqs. Get_num_last_reqs

`function REQ last_req(int unsigned n = 0)`

Function

last_req

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* .

Parameters
    n (*int unsigned*)

Return type
    REQ

`function int get_num_rsps_received()`

Function

get_num_rsps_received

Returns the number of responses received thus far by this sequencer. Get_num_rsps_received

`function void set_num_last_rsps(int unsigned max)`

Function

set_num_last_rsps

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. Set_num_last_rsps

Parameters
    max (*int unsigned*)

`function int unsigned get_num_last_rsps()`

Function

get_num_last_rsps

Returns the max size of the last responses buffer, as set by set_num_last_rsps. Get_num_last_rsps

`function RSP last_rsp(int unsigned n = 0)`

> ***Function***
>
> last_rsp
>
> 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* .
>
> Parameters
>
> **`n`** (*`int unsigned`*)
>
> Return type
>
> *RSP*

`function void put_response(uvm_sequence_item t)`

> Put_response
>
> Parameters
>
> **`t`** (*uvm_sequence_item*) -- local

`virtual  function void build_phase(uvm_phase phase)`

> Build_phase
>
> Parameters
>
> **`phase`** (*uvm_phase*) -- local

`virtual  function void connect_phase(uvm_phase phase)`

> Connect_phase
>
> Parameters
>
> **`phase`** (*uvm_phase*) -- local

`virtual  function void do_print(uvm_printer printer)`

> Do_print
>
> Parameters
>
> **`printer`** (*uvm_printer*) -- local

`virtual  function void analysis_write(uvm_sequence_item t)`

> Analysis_write
>
> Parameters
>
> **`t`** (*uvm_sequence_item*) -- local

15.2.0.17 Class uvm_pkg::uvm_set_before_get_dap

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_set_get_dap_base*
↪*uvm_pkg* :: *uvm_set_before_get_dap*

Class

uvm_set_before_get_dap

Provides a 'Set Before Get' Data Access Policy.

The 'Set Before Get' Data Access Policy enforces that the value must be written at *least* once before it is read. This DAP can be used to pass shared information to multiple components during standard configuration, even if that information hasn't yet been determined.

Such DAP objects can be useful for passing a 'placeholder' reference, before the information is actually available. A good example of this would be the virtual sequencer:

```
typedef uvm_set_before_get_dap#(uvm_sequencer_base) seqr_dap_t;
virtual_seqeuncer_type virtual_sequencer;
agent_type my_agent;
seqr_dap_t seqr_dap;

function void my_env::build_phase(uvm_phase phase);
  seqr_dap = seqr_dap_t::type_id::create("seqr_dap");
  // Pass the DAP, because we don't have a reference to the
  // real sequencer yet...
  uvm_config_db#(seqr_dap_t)::set(this, "virtual_sequencer", "seqr_dap", seqr_
↪dap);

  // Create the virtual sequencer
  virtual_sequencer = virtual_sequencer_type::type_id::create("virtual_sequencer
↪", this);

  // Create the agent
  agent = agent_type::type_id::create("agent", this);
endfunction

function void my_env::connect_phase(uvm_phase phase);
  // Now that we know the value is good, we can set it
  seqr_dap.set(agent.sequencer);
endfunction
```

In the example above, the environment didn't have a reference to the agent's sequencer yet, because the agent hadn't executed its *build_phase* . The environment needed to give the virtual sequencer a "Set before get" DAP so that the virtual sequencer (and any sequences one it), could *eventually* see the agent's sequencer, when the reference was finally available. If the virtual sequencer (or any sequences on it) attempted to 'get' the reference to the agent's sequencer *prior* to the environment assigning it, an error would have been reported.

Table 255: Parameters

Name	Default value	Description
T	int	

Table 256: Typedefs

Name	Actual Type	Description
this_type	*uvm_set_before_get_-dap#(T)*	Used for self-references

Constructors

function new(string name = "unnamed-uvm_set_before_get_dap#(T)")

Function

new

Constructor

Parameters

name (*string*)

Functions

virtual function void set(int value)

Function

set

Updates the value stored within the DAP.

Parameters

value (*int*)

virtual function bit try_set(int value)

Function

try_set

Attempts to update the value stored within the DAP.

try_set will always return a 1.

Parameters

value (*int*)

virtual function T get()

Function

get

Returns the current value stored within the DAP.

If 'get' is called before a call to *set* or *try_set*, then an error will be reported.

virtual function bit try_get(int value)

Function

try_get

Attempts to retrieve the current value stored within the DAP

If the value has not been 'set', then try_get will return a 0, otherwise it will return a 1, and set *value* to the current value stored within the DAP.

Parameters

value (*int*)

virtual function void do_copy(uvm_object rhs)

Group

Introspection

The *uvm_set_before_get_dap* cannot support the standard UVM instrumentation methods (*copy* , *clone* , *pack* and *unpack*), due to the fact that they would potentially violate the access policy.

A call to any of these methods will result in an error.

Parameters

rhs (*uvm_object*)

`virtual function void do_pack(uvm_packer packer)`

Parameters

`packer` (*uvm_packer*)

`virtual function void do_unpack(uvm_packer packer)`

Parameters

`packer` (*uvm_packer*)

`virtual function string convert2string()`

Function- convert2string

`virtual function void do_print(uvm_printer printer)`

Function- do_print

Parameters

`printer` (*uvm_printer*)

15.2.0.18 Class uvm_pkg::uvm_set_get_dap_base

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_set_get_dap_base*

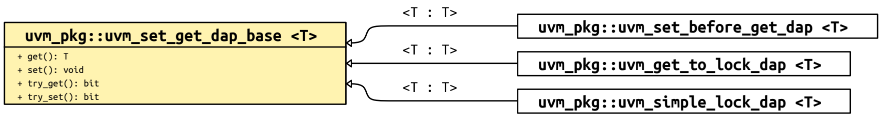

uvm_pkg::uvm_set_get_dap_base <T>
+ get(): T
+ set(): void
+ try_get(): bit
+ try_set(): bit
<T : T>
uvm_pkg::uvm_set_before_get_dap <T>
<T : T>
uvm_pkg::uvm_get_to_lock_dap <T>
<T : T>
uvm_pkg::uvm_simple_lock_dap <T>

Fig. 90: Inheritance Diagram of uvm_set_get_dap_base

Class

uvm_set_get_dap_base

Provides the 'set' and 'get' interface for Data Access Policies (DAPs)

The 'Set/Get' base class simply provides a common interface for the various DAPs to implement. This provides a mechanism for consistent implementations of similar DAPs.

Table 257: Parameters

Name	Default value	Description
T	int	

Table 258: Typedefs

Name	Actual Type	Description
this_type	*uvm_set_get_dap_-base#(T)*	Used for self references

Constructors

`function new(string name = "unnamed-uvm_set_get_dap_base#(T)")`

Function

new

Constructor

Parameters

`name` (*`string`*)

Functions

`virtual function void set(int value)`

Function

set

Sets the value contained within the resource.

Depending on the DAP policies, an error may be reported if it is illegal to 'set' the value at this time.

Parameters

`value` (*`int`*)

`virtual function bit try_set(int value)`

Function

try_set

Attempts to set the value contained within the resource.

If the DAP policies forbid setting at this time, then the method will return 0, however no errors will be reported. Otherwise, the method will return 1, and will be treated like a standard *set* call.

Parameters

`value` (*`int`*)

`virtual function T get()`

Function

get

Retrieves the value contained within the resource.

Depending on the DAP policies, an error may be reported if it is illegal to 'get' the value at this time.

`virtual function bit try_get(int value)`

Function

try_get

Attempts to retrieve the value contained within the resource.

If the DAP policies forbid retrieving at this time, then the method will return 0, however no errors will be reported. Otherwise, the method will return 1, and will be treated like a standard *get* call.

Parameters

`value` (*`int`*)

15.2.0.19 Class uvm_pkg::uvm_shutdown_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_task_phase*
↪*uvm_pkg* :: *uvm_shutdown_phase*

Class

uvm_shutdown_phase

Letting things settle down.

uvm_task_phase that calls the *uvm_component::shutdown_phase* method.

Upon Entry

- None.

Typical Uses

Wait for all data to be drained out of the DUT.
Extract data still buffered in the DUT, usually through read/write operations or sequences.

Exit Criteria

All data has been drained or extracted from the DUT.
All interfaces are idle.

Table 259: Variables

Name	Type	Description
type_name	string	

Functions

`static function uvm_shutdown_phase get()`

Function

get

Returns the singleton phase handle
Return type
uvm_shutdown_phase

`virtual function string get_type_name()`

Tasks

`virtual function exec_task(uvm_component comp, uvm_phase phase)`

Parameters
`comp` (*uvm_component*)
`phase` (*uvm_phase*)

15.2.0.20 Class uvm_pkg::uvm_simple_lock_dap

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_set_get_dap_base*
↪*uvm_pkg* :: *uvm_simple_lock_dap*

Class

uvm_simple_lock_dap

Provides a 'Simple Lock' Data Access Policy.

The 'Simple Lock' Data Access Policy allows for any number of 'sets', so long as the value is not 'locked'. The value can be retrieved using 'get' at any time.

The UVM uses this policy to protect the *file name* value in the *uvm_text_tr_database*.

Table 260: Parameters

Name	Default value	Description
T	int	

Table 261: Typedefs

Name	Actual Type	Description
this_type	*uvm_simple_lock_-dap#(T)*	Used for self-references

Constructors

`function new(string name = "unnamed-uvm_simple_lock_dap#(T)")`

Function

new

Constructor
Parameters
`name` (`string`)

Functions

`virtual function void set(int value)`

Function

set

Updates the value stored within the DAP.

set will result in an error if the DAP has been locked.
Parameters
`value` (`int`)

`virtual function bit try_set(int value)`

Function

try_set

Attempts to update the value stored within the DAP.

try_set will return a 1 if the value was successfully updated, or a 0 if the value can not be updated due to the DAP being locked. No errors will be reported if *try_set* fails.

Parameters

value (*int*)

virtual function T get()

Function

get

Returns the current value stored within the DAP

virtual function bit try_get(int value)

Function

try_get

Retrieves the current value stored within the DAP

try_get will always return 1.

Parameters

value (*int*)

function void lock()

Function

lock

Locks the data value

The data value cannot be updated via *set* or *try_set* while locked.

function void unlock()

Function

unlock

Unlocks the data value

function bit is_locked()

Function

is_locked

Returns the state of the lock.

Returns

1

The value is locked

0

The value is unlocked

virtual function void do_copy(uvm_object rhs)

Group

Introspection

The *uvm_simple_lock_dap* cannot support the standard UVM instrumentation methods (*copy* , *clone* , *pack* and *unpack*), due to the fact that they would potentially violate the access policy.

A call to any of these methods will result in an error.

Parameters

rhs (*uvm_object*)

virtual function void do_pack(uvm_packer packer)

Parameters

packer (*uvm_packer*)

virtual function void do_unpack(uvm_packer packer)

Parameters

packer (*uvm_packer*)

`virtual function string convert2string()`

Function- convert2string

`virtual function void do_print(uvm_printer printer)`

Function- do_print

Parameters

`printer` (*uvm_printer*)

15.2.0.21 Class uvm_pkg::uvm_simple_sequence

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_sequence_base*
↪*uvm_pkg* :: *uvm_sequence*
↪*uvm_pkg* :: *uvm_simple_sequence*

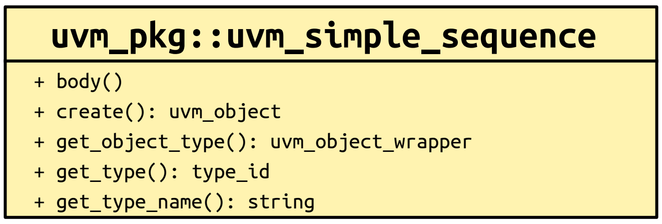

uvm_pkg::uvm_simple_sequence
+ body()
+ create(): uvm_object
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string

Fig. 91: Collaboration Diagram of uvm_simple_sequence

CLASS- 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.

Constructors

function new(string name = "uvm_simple_sequence")

new

Parameters

name (*string*)

Functions

virtual function uvm_object create(string name = "")

Parameters

name (*string*)

Return type

uvm_object

virtual function string get_type_name()

Tasks

virtual function body()

body

15.2.0.22 Class uvm_pkg::uvm_slave_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_slave_export*

Table 262: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters

name (*string*)
parent (*uvm_component*)
min_size (*int*)
max_size (*int*)

15.2.0.23 Class uvm_pkg::uvm_slave_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_slave_imp*

Table 263: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	
IMP	int	
REQ_IMP	IMP	
RSP_IMP	IMP	

Table 264: Typedefs

Name	Actual Type	Description
this_imp_type	IMP	
this_req_type	REQ_IMP	
this_rsp_type	RSP_IMP	

Constructors

`function new(string name, this_imp_type imp, this_req_type req_imp = null, this_rsp_type rsp_imp = null)`

Parameters
- **`name`** (`string`)
- **`imp`** (*this_imp_type*)
- **`req_imp`** (*this_req_type*)
- **`rsp_imp`** (*this_rsp_type*)

15.2.0.24 Class uvm_pkg::uvm_slave_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_slave_port*

Table 265: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.2.0.25 Class uvm_pkg::uvm_spell_chkr

```
class uvm_spell_chkr
```

Table 266: Parameters

Name	Default value	Description
T	int	

Table 267: Variables

Name	Type	Description
max	int unsigned	

Table 268: Typedefs

Name	Actual Type	Description
tab_t	*T*	

Functions

`static function bit check(tab_t strtab, string s)`

check

primary interface to the spell checker. The function takes two arguments, a table of strings and a string to check. The table is organized as an associative array of type T. E.g.

T strtab[string]

It doesn't matter what T is since we are only concerned with the string keys. However, we need T in order to make argument types match.

First, we do the simple thing and see if the string already is in the string table by calling the *exists()* method. If it does exist then there is a match and we're done. If the string doesn't exist in the table then we invoke the spell checker algorithm to see if our string is a misspelled variation on a string that does exist in the table.

The main loop traverses the string table computing the levenshtein distance between each string and the string we are checking. The strings in the table with the minimum distance are considered possible alternatives. There may be more than one string in the table with a minimum distance. So all the alternatives are stored in a queue.

Note

This is not a particularly efficient algorithm. It requires

computing the levenshtein distance for every string in the string table. If that list were very large the run time could be long. For the resources application in UVM probably the size of the string table is not excessive and run times will be fast enough. If, on average, that proves to be an invalid assumption then we'll have to find ways to optimize this algorithm.

note

strtab should not be modified inside check()

Parameters

- **strtab** (*tab_t*) -- const
- **s** (*string*)

15.2.0.26 Class uvm_pkg::uvm_sqr_if_base

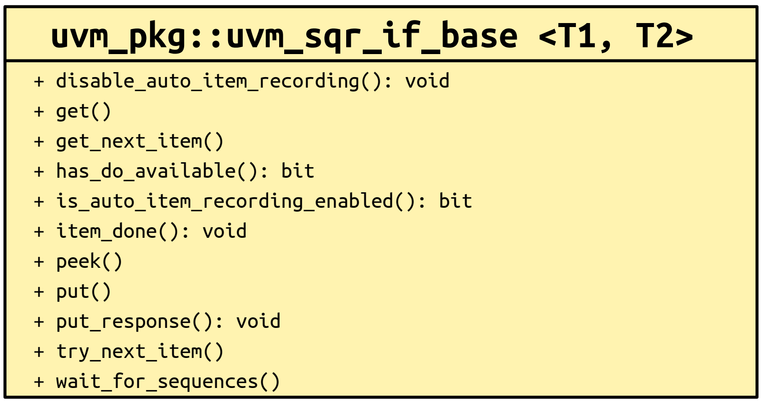

uvm_pkg::uvm_sqr_if_base <T1, T2>
+ disable_auto_item_recording(): void
+ get()
+ get_next_item()
+ has_do_available(): bit
+ is_auto_item_recording_enabled(): bit
+ item_done(): void
+ peek()
+ put()
+ put_response(): void
+ try_next_item()
+ wait_for_sequences()

Fig. 92: Inheritance Diagram of uvm_sqr_if_base

CLASS

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.

Table 269: Parameters

Name	Default value	Description
T1	uvm_object	
T2	T1	

Functions

virtual function void item_done(uvm_object t = null)

Function

item_done

Indicates that the request is completed to the sequencer. Any *uvm_sequence_base::wait_for_item_done* calls made by a sequence for this item will return.

The current item is removed from the sequencer FIFO.

If a response item is provided, then it will be sent back to the requesting sequence. The response item must have its sequence ID and transaction ID set correctly, using the *uvm_sequence_item::set_id_info* method:

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

Before *item_done* is called, any calls to peek will retrieve the current item that was obtained by *get_next_item*. After *item_done* is called, peek will cause the sequencer to arbitrate for a new item.

Parameters

t (*uvm_object*)

virtual function bit has_do_available()

Function

has_do_available

Indicates whether a sequence item is available for immediate processing. Implementations should return 1 if an item is available, 0 otherwise.

`virtual function void put_response(uvm_object t)`

Function

put_response

Sends a response back to the sequence that issued the request. Before the response is put, it must have its sequence ID and transaction ID set to match the request. This can be done using the *uvm_sequence_item::set_id_info* call:

rsp.set_id_info(req);

Parameters

`t` (*uvm_object*)

`virtual function void disable_auto_item_recording()`

Function

disable_auto_item_recording

By default, item recording is performed automatically when get_next_item() and item_done() are called. However, this works only for simple, in-order, blocking transaction execution. For pipelined and out-of-order transaction execution, the driver must turn off this automatic recording and call *uvm_transaction::accept_tr*, *uvm_transaction::begin_tr* and *uvm_transaction::end_tr* explicitly at appropriate points in time.

This methods be called at the beginning of the driver's *run_phase()* method. Once disabled, automatic recording cannot be re-enabled.

For backward-compatibility, automatic item recording can be globally turned off at compile time by defining UVM_DISABLE_AUTO_ITEM_RECORDING

`virtual function bit is_auto_item_recording_enabled()`

Function

is_auto_item_recording_enabled

Return TRUE if automatic item recording is enabled for this port instance.

Tasks

`virtual function get_next_item(uvm_object t)`

Task

get_next_item

Retrieves the next available item from a sequence. The call will block until an item is available. The following steps occur on this call:

1

Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, wait for a requesting unlocked relevant sequence, then re-arbitrate.

2

The chosen sequence will return from wait_for_grant

3

The chosen sequence *uvm_sequence_base::pre_do* is called

4

The chosen sequence item is randomized

5

The chosen sequence *uvm_sequence_base::post_do* is called

6

Return with a reference to the item

Once *get_next_item* is called, *item_done* must be called to indicate the completion of the request to the sequencer. This will remove the request item from the sequencer FIFO.

Parameters

`t` (*uvm_object*)

`virtual function try_next_item(uvm_object t)`

Task

try_next_item

Retrieves the next available item from a sequence if one is available. Otherwise, the function returns immediately with request set to *null* . The following steps occur on this call:

1

Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, return *null* .

2

The chosen sequence will return from wait_for_grant

3

The chosen sequence *uvm_sequence_base::pre_do* is called

4

The chosen sequence item is randomized

5

The chosen sequence *uvm_sequence_base::post_do* is called

6

Return with a reference to the item

Once *try_next_item* is called, *item_done* must be called to indicate the completion of the request to the sequencer. This will remove the request item from the sequencer FIFO.

Parameters

`t` (*uvm_object*)

`virtual function wait_for_sequences()`

Task

wait_for_sequences

Waits for a sequence to have a new item available. The default implementation in the sequencer calls *uvm_wait_for_nba_region*. User-derived sequencers may override its *wait_for_sequences* implementation to perform some other application-specific implementation.

`virtual function get(uvm_object t)`

Task

get

Retrieves the next available item from a sequence. The call blocks until an item is available. The following steps occur on this call:

1

Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, wait for a requesting unlocked relevant sequence, then re-arbitrate.

2

The chosen sequence will return from *uvm_sequence_base::wait_for_grant*

3

The chosen sequence *uvm_sequence_base::pre_do* is called

4

The chosen sequence item is randomized

5

The chosen sequence *uvm_sequence_base::post_do* is called

6

Indicate *item_done* to the sequencer

7

Return with a reference to the item

When get is called, *item_done* may not be called. A new item can be obtained by calling get again, or a response may be sent using either *put*, or uvm_driver::rsp_port.write().

Parameters

`t` (*uvm_object*)

```
virtual function peek(uvm_object t)
```

Task

peek

Returns the current request item if one is in the sequencer FIFO. If no item is in the FIFO, then the call will block until the sequencer has a new request. The following steps will occur if the sequencer FIFO is empty:

1

Arbitrate among requesting, unlocked, relevant sequences - choose the highest priority sequence based on the current sequencer arbitration mode. If no sequence is available, wait for a requesting unlocked relevant sequence, then re-arbitrate.

2

The chosen sequence will return from *uvm_sequence_base::wait_for_grant*

3

The chosen sequence *uvm_sequence_base::pre_do* is called

4

The chosen sequence item is randomized

5

The chosen sequence *uvm_sequence_base::post_do* is called

Once a request item has been retrieved and is in the sequencer FIFO, subsequent calls to peek will return the same item. The item will stay in the FIFO until either get or *item_done* is called.

Parameters

`t` (*uvm_object*)

```
virtual function put(uvm_object t)
```

Parameters

`t` (*uvm_object*)

15.2.0.27 Class uvm_pkg::uvm_start_of_simulation_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_bottomup_phase*
↪*uvm_pkg* :: *uvm_start_of_simulation_phase*

Table 270: Variables

Name	Type	Description
type_name	string	

Functions

`virtual function void exec_func(uvm_component comp, uvm_phase phase)`

Parameters
- **`comp`** (*uvm_component*)
- **`phase`** (*uvm_phase*)

`static function uvm_start_of_simulation_phase get()`

Function

get

Returns the singleton phase handle

Return type
uvm_start_of_simulation_phase

`virtual function string get_type_name()`

15.2.0.28 Class uvm_pkg::uvm_status_container

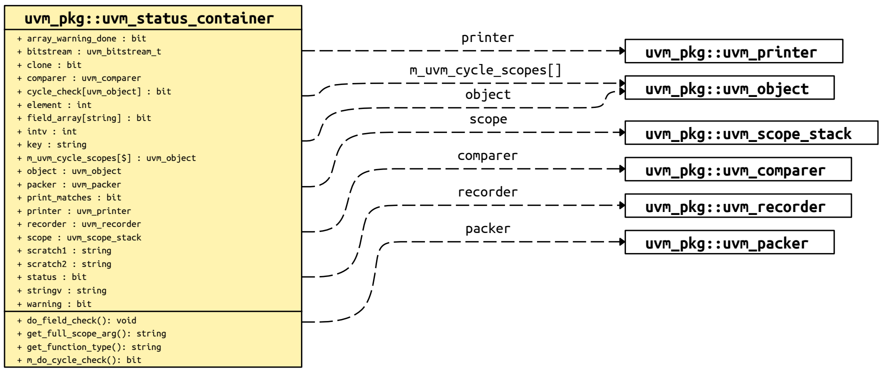


Fig. 93: Collaboration Diagram of uvm_status_container

CLASS- uvm_status_container

Internal class to contain status information for automation methods.

Table 271: Variables

Name	Type	Description
clone	bit	The clone setting is used by the set/get config to know if cloning is on.
warning	bit	Information variables used by the macro functions for storage.
status	bit	
bitstream	*uvm_bitstream_t*	
intv	int	
element	int	
stringv	string	
scratch1	string	
scratch2	string	
key	string	
object	*uvm_object*	
array_warning_done	bit	
field_array	bit	
print_matches	bit	

continues on next page

Table 271 – continued from previous page

Name	Type	Description
scope	*uvm_scope_stack*	The scope stack is used for messages that are emitted by policy classes.
cycle_check	bit	Used for checking cycles. When a data function is entered, if the depth is non-zero, then then the existeance of the object in the map means that a cycle has occured and the function should immediately exit. When the function exits, it should reset the cycle map so that there is no memory leak.
comparer	*uvm_comparer*	These are the policy objects currently in use. The policy object gets set when a function starts up. The macros use this.
packer	*uvm_packer*	
recorder	*uvm_recorder*	
printer	*uvm_printer*	

Functions

`function void do_field_check(string field, uvm_object obj)`

Parameters
- **`field`** (*`string`*)
- **`obj`** (*uvm_object*)

`function string get_function_type(int what)`

Parameters
- **`what`** (*`int`*)

`function string get_full_scope_arg()`

15.2.0.29 Class uvm_pkg::uvm_string_rsrc

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_resource_base*
↪*uvm_pkg* :: *uvm_resource*
↪*uvm_pkg* :: *uvm_string_rsrc*

uvm_string_rsrc

specialization of uvm_resource (T) for T = string

Table 272: Typedefs

Name	Actual Type	Description
this_subtype	*uvm_string_rsrc*	

Constructors

```
function  new(string name, string s = "*")
```

Parameters
name (*string*)
s (*string*)

Functions

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

15.2.0.30 Class uvm_pkg::uvm_structure_proxy

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_structure_proxy*

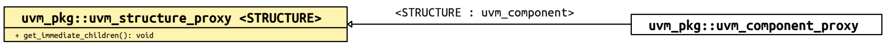

uvm_pkg::uvm_structure_proxy <STRUCTURE>
+ get_immediate_children(): void
<STRUCTURE : uvm_component>
uvm_pkg::uvm_component_proxy

Fig. 94: Inheritance Diagram of uvm_structure_proxy

CLASS

uvm_structure_proxy (STRUCTURE)

The uvm_structure_proxy is a wrapper and provides a set of elements of the STRUCTURE to the caller on demand. This is to decouple the retrieval of the STRUCTUREs subelements from the actual function being invoked on STRUCTURE

Table 273: Parameters

Name	Default value	Description
STRUCTURE	uvm_component	

Constructors

`function new(string name = "")`

Parameters
- **`name`** (*`string`*)

Functions

`virtual function void get_immediate_children(uvm_component s, uvm_-component children)`

Function

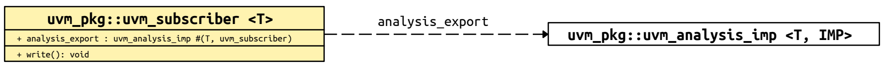
get_immediate_children

This method will be return in *children* a set of the direct subelements of *s*

Parameters
- **`s`** (*uvm_component*)
- **`children`** (*uvm_component*)

15.2.0.31 Class uvm_pkg::uvm_subscriber

uvm_pkg :: *uvm_void*
  ↪*uvm_pkg* :: *uvm_object*
    ↪*uvm_pkg* :: *uvm_report_object*
      ↪*uvm_pkg* :: *uvm_component*
        ↪*uvm_pkg* :: *uvm_subscriber*



uvm_pkg::uvm_subscriber <T>
+ analysis_export : uvm_analysis_imp #(T, uvm_subscriber)
+ write(): void
analysis_export
uvm_pkg::uvm_analysis_imp <T, IMP>

Fig. 95: Collaboration Diagram of uvm_subscriber

CLASS

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.

Table 274: Parameters

Name	Default value	Description
T	int	

Table 275: Variables

Name	Type	Description
analysis_export	*uvm_analysis_imp#(int, uvm_subscriber#(int))*	***Port*** analysis_export This export provides access to the write method, which derived subscribers must implement.

Table 276: Typedefs

Name	Actual Type	Description
this_type	*uvm_subscriber#(T)*	

Constructors

`function new(string name, uvm_component parent)`

Function

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.

Parameters

`name` (*`string`*)

parent (*uvm_component*)

Functions

`virtual function void write(int t)`

Function

write

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.

Parameters

t (*int*)

15.2.0.32 Class uvm_pkg::uvm_table_printer

uvm_pkg :: *uvm_printer*
↪*uvm_pkg* :: *uvm_table_printer*

Class

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
---------------------------------------------------
```

Constructors

`function  new()`

Variable

new

Creates a new instance of *uvm_table_printer* . New

Functions

`virtual  function string emit()`

Function

emit

Formats the collected information from prior calls to *print_** into table format. Emit

`function void calculate_max_widths()`

Calculate_max_widths

15.2.0.33 Class uvm_pkg::uvm_task_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_task_phase*

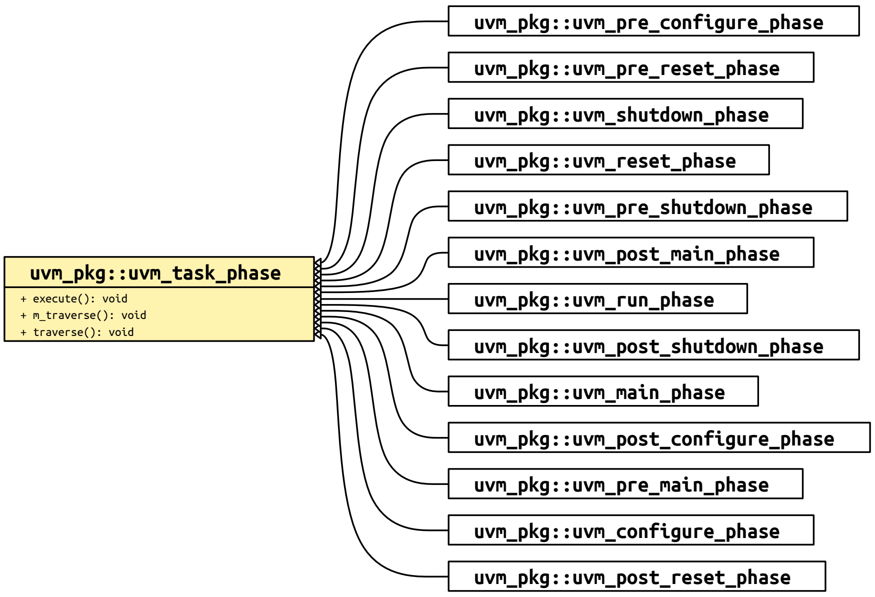

uvm_pkg::uvm_task_phase
+ execute(): void
+ m_traverse(): void
+ traverse(): void
uvm_pkg::uvm_pre_configure_phase
uvm_pkg::uvm_pre_reset_phase
uvm_pkg::uvm_shutdown_phase
uvm_pkg::uvm_reset_phase
uvm_pkg::uvm_pre_shutdown_phase
uvm_pkg::uvm_post_main_phase
uvm_pkg::uvm_run_phase
uvm_pkg::uvm_post_shutdown_phase
uvm_pkg::uvm_main_phase
uvm_pkg::uvm_post_configure_phase
uvm_pkg::uvm_pre_main_phase
uvm_pkg::uvm_configure_phase
uvm_pkg::uvm_post_reset_phase

Fig. 96: Inheritance Diagram of uvm_task_phase

Class

uvm_task_phase

Base class for all task phases. It forks a call to *uvm_phase::exec_task()* for each component in the hierarchy.

The completion of the task does not imply, nor is it required for, the end of phase. Once the phase completes, any remaining forked *uvm_phase::exec_task()* threads are forcibly and immediately killed.

By default, the way for a task phase to extend over time is if there is at least one component that raises an objection.

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

There is however one scenario wherein time advances within a task-based phase without any objections to the phase being raised. If two (or more) phases share a common successor, such as the *uvm_run_phase* and the *uvm_post_shutdown_phase* sharing the *uvm_extract_phase* as a successor, then phase advancement is delayed until all predecessors of the common successor are ready to proceed. Because of this, it is possible for time to advance between *uvm_component::phase_started* and *uvm_component::phase_ended* of a task phase without any participants in the phase raising an objection.

Constructors

`function new(string name)`

Function

new

Create a new instance of a task-based phase

Parameters

`name` (*`string`*)

Functions

`virtual function void traverse(uvm_component comp, uvm_phase phase, uvm_phase_-state state)`

Function

traverse

Traverses the component tree in bottom-up order, calling *execute* for each component. The actual order for task-based phases doesn't really matter, as each component task is executed in a separate process whose starting order is not deterministic.

Parameters

`comp` (*uvm_component*)
`phase` (*uvm_phase*)
`state` (*uvm_phase_state*)

`virtual function void execute(uvm_component comp, uvm_phase phase)`

Function

execute

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

Parameters

`comp` (*uvm_component*)
`phase` (*uvm_phase*)

15.2.0.34 Class uvm_pkg::uvm_test

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_test*

CLASS

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.

Table 277: Variables

Name	Type	Description
type_name	string	

Constructors

`function new(string name, uvm_component parent)`

Function

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.

Parameters

- **`name`** (`string`)
- **`parent`** (*uvm_component*)

Functions

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

15.2.0.35 Class uvm_pkg::uvm_test_done_objection

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_objection*
↪*uvm_pkg* :: *uvm_test_done_objection*

Class- uvm_test_done_objection **DEPRECATED**

Provides built-in end-of-test coordination

Table 278: Variables

Name	Type	Description
stop_timeout	time	Variable- stop_timeout **DEPRECATED** These set watchdog timers for task-based phases and stop tasks. You cannot 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.

Table 279: Typedefs

Name	Actual Type	Description
type_id	*uvm_object_registry#(uvm_test_done_objection, "uvm_test_done")*	Below are basic data operations needed for all uvm_objects for factory registration, printing, comparing, etc.

Constructors

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

Function- new **DEPRECATED**

Creates the singleton test_done objection. Users must not call this method directly.

Parameters

`name` (*`string`*)

Functions

`virtual function void qualify(uvm_object obj = null, bit is_raise, string description)`

Function- qualify **DEPRECATED**

Checks that the given *object* is derived from either *uvm_component* or *uvm_sequence_base*.

Parameters

`obj` (*uvm_object*)
`is_raise` (*`bit`*)
`description` (*`string`*)

`function void stop_request()`

Function- stop_request **DEPRECATED**

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.

`virtual function void raise_objection(uvm_object obj = null, string description = "", int count = 1)`

Function- raise_objection **DEPRECATED**

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.

Parameters

- **`obj`** (*uvm_object*)
- **`description`** (*`string`*)
- **`count`** (*`int`*)

`virtual function void drop_objection(uvm_object obj = null, string description = "", int count = 1)`

Function- drop_objection **DEPRECATED**

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.

Parameters

- **`obj`** (*uvm_object*)
- **`description`** (*`string`*)
- **`count`** (*`int`*)

`static function type_id get_type()`

Return type

type_id

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

Parameters

- **`name`** (*`string`*)

Return type

uvm_object

`virtual function string get_type_name()`

`static function uvm_test_done_objection get()`

Return type

uvm_test_done_objection

Tasks

`virtual function all_dropped(uvm_object obj, uvm_object source_obj, string description, int count)`

Task- all_dropped **DEPRECATED**

This callback is called when the given *object's* objection count reaches zero; if the *object* is the implicit top-level, *uvm_root* then it means there are no more objections raised for the *uvm_test_done* objection. Thus, after calling *uvm_objection::all_dropped*, this method will call *global_stop_request* to stop the current task-based phase (e.g. run).

Parameters

- **`obj`** (*uvm_object*)
- **`source_obj`** (*uvm_object*)
- **`description`** (*`string`*)
- **`count`** (*`int`*)

`virtual function force_stop(uvm_object obj = null)`

Task- force_stop **DEPRECATED**

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.

Parameters

- **`obj`** (*uvm_object*)

15.2.0.36 Class uvm_pkg::uvm_text_recorder

uvm_pkg :: *uvm_void*
  ↪*uvm_pkg* :: *uvm_object*
    ↪*uvm_pkg* :: *uvm_recorder*
      ↪*uvm_pkg* :: *uvm_text_recorder*

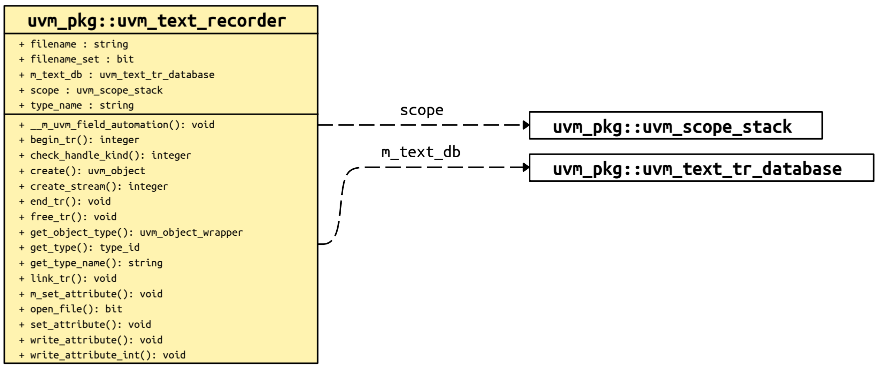

uvm_pkg::uvm_text_recorder
+ filename : string
+ filename_set : bit
+ m_text_db : uvm_text_tr_database
+ scope : uvm_scope_stack
+ type_name : string
+ __m_uvm_field_automation(): void
+ begin_tr(): integer
+ check_handle_kind(): integer
+ create(): uvm_object
+ create_stream(): integer
+ end_tr(): void
+ free_tr(): void
+ get_object_type(): uvm_object_wrapper
+ get_type(): type_id
+ get_type_name(): string
+ link_tr(): void
+ m_set_attribute(): void
+ open_file(): bit
+ set_attribute(): void
+ write_attribute(): void
+ write_attribute_int(): void
scope
m_text_db
uvm_pkg::uvm_scope_stack
uvm_pkg::uvm_text_tr_database

Fig. 97: Collaboration Diagram of uvm_text_recorder

CLASS

uvm_text_recorder

The *uvm_text_recorder* is the default recorder implementation for the *uvm_text_tr_database*.

Table 280: Variables

Name	Type	Description
scope	*uvm_scope_stack*	Variable- scope Imeplementation detail
filename	string	UVM provides only a text-based default implementation. Vendors provide subtype implementations and overwrite the <uvm_default_recorder> handle.
filename_set	bit	

Constructors

`function new(string name = "unnamed-uvm_text_recorder")`

Function

new

Constructor

Parameters

name

Instance name

Parameters

`name` (*`string`*)

Functions

`function void write_attribute(string nm, uvm_bitstream_t value, uvm_radix_-enum radix, integer numbits = $bits(uvm_bitstream_t))`

Function

write_attribute

Outputs an integral attribute to the textual log

Parameters

nm

Name of the attribute

value

Value

radix

Radix of the output

numbits

number of valid bits

Parameters

- **`nm`** (*`string`*)
- **`value`** (*uvm_bitstream_t*)
- **`radix`** (*uvm_radix_enum*)
- **`numbits`** (*`integer`*)

`function void write_attribute_int(string nm, uvm_integral_t value, uvm_radix_-enum radix, integer numbits = $bits(uvm_bitstream_t))`

Function

write_attribute_int

Outputs an integral attribute to the textual log

Parameters

nm

Name of the attribute

value

Value

radix

Radix of the output

numbits

number of valid bits

Parameters

- **`nm`** (*`string`*)
- **`value`** (*uvm_integral_t*)
- **`radix`** (*uvm_radix_enum*)
- **`numbits`** (*`integer`*)

`virtual  function bit open_file()`

Function- open_file

Opens the file in the *filename* property and assigns to the file descriptor <file>.

`virtual  function integer create_stream(string name, string t, string scope)`

Function- create_stream

Parameters

- **`name`** (*`string`*)
- **`t`** (*`string`*)
- **`scope`** (*`string`*)

virtual function void set_attribute(integer txh, string nm, logic[1023:0] value, uvm_radix_enum radix, integer numbits = 1024)

Function- set_attribute

Parameters

- **txh** (*integer*)
- **nm** (*string*)
- **value** (*logic[1023:0]*)
- **radix** (*uvm_radix_enum*)
- **numbits** (*integer*)

virtual function integer check_handle_kind(string htype, integer handle)

Function- check_handle_kind

Parameters

- **htype** (*string*)
- **handle** (*integer*)

virtual function integer begin_tr(string txtype, integer stream, string nm, string label = "", string desc = "", time begin_time = 0)

Function- begin_tr

Parameters

- **txtype** (*string*)
- **stream** (*integer*)
- **nm** (*string*)
- **label** (*string*)
- **desc** (*string*)
- **begin_time** (*time*)

virtual function void end_tr(integer handle, time end_time = 0)

Function- end_tr

Parameters

- **handle** (*integer*)
- **end_time** (*time*)

virtual function void link_tr(integer h1, integer h2, string relation = "")

Function- link_tr

Parameters

- **h1** (*integer*)
- **h2** (*integer*)
- **relation** (*string*)

virtual function void free_tr(integer handle)

Function- free_tr

Parameters

- **handle** (*integer*)

15.2.0.37 Class uvm_pkg::uvm_text_tr_database

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_tr_database*
↪*uvm_pkg* :: *uvm_text_tr_database*

CLASS

uvm_text_tr_database

The *uvm_text_tr_database* is the default implementation for the *uvm_tr_database*. It provides the ability to store recording information into a textual log file.

Constructors

`function  new(string name = "unnamed-uvm_text_tr_database")`

Function

new

Constructor

Parameters

name

Instance name
Parameters
`name` (*string*)

Functions

`function void set_file_name(string filename)`

Function

set_file_name

Sets the file name which will be used for output.

The *set_file_name* method can only be called prior to *open_db* .

By default, the database will use a file named "tr_db.log".
Parameters
`filename` (*string*)

15.2.0.38 Class uvm_pkg::uvm_text_tr_stream

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_tr_stream*
↪*uvm_pkg* :: *uvm_text_tr_stream*

CLASS

uvm_text_tr_stream

The *uvm_text_tr_stream* is the default stream implementation for the *uvm_text_tr_database*.

Constructors

```
function  new(string name = "unnamed-uvm_text_tr_stream")
```

Function

new

Constructor

Parameters

name

Instance name

Parameters

name (*string*)

15.2.0.39 Class uvm_pkg::uvm_tlm_analysis_fifo

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_tlm_fifo_base*
↪*uvm_pkg* :: *uvm_tlm_fifo*
↪*uvm_pkg* :: *uvm_tlm_analysis_fifo*

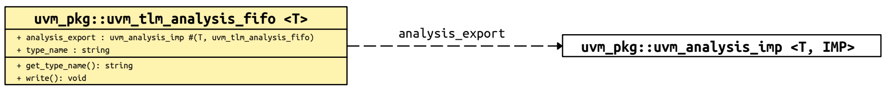

uvm_pkg::uvm_tlm_analysis_fifo <T>
+ analysis_export : uvm_analysis_imp #(T, uvm_tlm_analysis_fifo)
+ type_name : string
+ get_type_name(): string
+ write(): void
analysis_export
uvm_pkg::uvm_analysis_imp <T, IMP>

Fig. 98: Collaboration Diagram of uvm_tlm_analysis_fifo

Class

uvm_tlm_analysis_fifo(T)

An analysis_fifo is a <uvm_tlm_fifo(T)> with an unbounded size and a write interface. It can be used any place a *uvm_analysis_imp* is used. Typical usage is as a buffer between a *uvm_analysis_port* in an initiator component and TLM1 target component.

Table 281: Parameters

Name	Default value	Description
T	int	

Table 282: Variables

Name	Type	Description
analysis_export	*uvm_analysis_imp#(int, uvm_tlm_analysis_-fifo#(int))*	***Port*** analysis_export (T) The analysis_export provides the write method to all connected analysis ports and parent exports: `function void write (T t)` Access via ports bound to this export is the normal mechanism for writing to an analysis FIFO. See write method of <uvm_tlm_if_base (T1, T2)> for more information.
type_name	string	

Constructors

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

Function

new

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.

Parameters

- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)

Functions

`virtual  function string get_type_name()`

`function void write(int t)`

Parameters

- **`t`** (*`int`*)

15.2.0.40 Class uvm_pkg::uvm_tlm_b_initiator_socket

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_b_initiator_socket_base*
↪*uvm_pkg* :: *uvm_tlm_b_initiator_socket*

Class

uvm_tlm_b_initiator_socket

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

Table 283: Parameters

Name	Default value	Description
T	uvm_tlm_generic_pay-load	

Constructors

`function new(string name, uvm_component parent)`

Function

new

Construct a new instance of this socket
Parameters
`name` (`string`)
`parent` (*uvm_component*)

Functions

`virtual function void connect(this_type provider)`

Function

Connect

Connect this socket to the specified *uvm_tlm_b_target_socket*
Parameters
`provider` (*this_type*)

15.2.0.41 Class uvm_pkg::uvm_tlm_b_initiator_socket_base

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_b_initiator_socket_base*

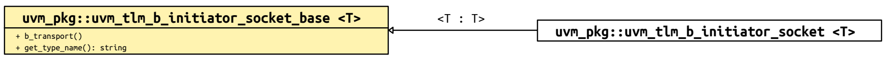

uvm_pkg::uvm_tlm_b_initiator_socket_base <T>
+ b_transport()
+ get_type_name(): string
<T : T>
uvm_pkg::uvm_tlm_b_initiator_socket <T>

Fig. 99: Inheritance Diagram of uvm_tlm_b_initiator_socket_base

Class

uvm_tlm_b_initiator_socket_base

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

Table 284: Parameters

Name	Default value	Description
T	uvm_tlm_generic_payload	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.2.0.42 Class uvm_pkg::uvm_tlm_b_passthrough_initiator_socket

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_b_passthrough_initiator_socket_base*
↪*uvm_pkg* :: *uvm_tlm_b_passthrough_initiator_socket*

Class

uvm_tlm_b_passthrough_initiator_socket

IS-A forward port;

Table 285: Parameters

Name	Default value	Description
T	uvm_tlm_generic_payload	

Constructors

`function new(string name, uvm_component parent)`

Parameters
- **`name`** (`string`)
- **`parent`** (*uvm_component*)

Functions

`virtual function void connect(this_type provider)`

Function

connect

Connect this socket to the specified *uvm_tlm_b_target_socket*

Parameters
- **`provider`** (*this_type*)

15.2.0.43 Class uvm_pkg::uvm_tlm_b_passthrough_initiator_socket_base

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_b_passthrough_initiator_socket_base*

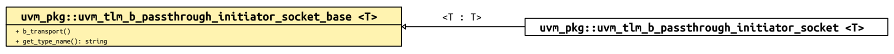

uvm_pkg::uvm_tlm_b_passthrough_initiator_socket_base <T>
+ b_transport()
+ get_type_name(): string
<T : T>
uvm_pkg::uvm_tlm_b_passthrough_initiator_socket <T>

Fig. 100: Inheritance Diagram of uvm_tlm_b_passthrough_initiator_socket_base

Class

uvm_tlm_b_passthrough_initiator_socket_base

IS-A forward port

Table 286: Parameters

Name	Default value	Description
T	uvm_tlm_generic_payload	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.2.0.44 Class uvm_pkg::uvm_tlm_b_passthrough_target_socket

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_b_passthrough_target_socket_base*
↪*uvm_pkg* :: *uvm_tlm_b_passthrough_target_socket*

Class

uvm_tlm_b_passthrough_target_socket

IS-A forward export;

Table 287: Parameters

Name	Default value	Description
T	uvm_tlm_generic_payload	

Constructors

`function new(string name, uvm_component parent)`

Parameters
`name` (`string`)
`parent` (*uvm_component*)

Functions

`virtual function void connect(this_type provider)`

Function

connect

Connect this socket to the specified *uvm_tlm_b_initiator_socket*

Parameters
`provider` (*this_type*)

15.2.0.45 Class uvm_pkg::uvm_tlm_b_passthrough_target_socket_base

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_b_passthrough_target_socket_base*

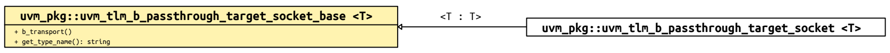

uvm_pkg::uvm_tlm_b_passthrough_target_socket_base <T>
+ b_transport()
+ get_type_name(): string
<T : T>
uvm_pkg::uvm_tlm_b_passthrough_target_socket <T>

Fig. 101: Inheritance Diagram of uvm_tlm_b_passthrough_target_socket_base

Class

uvm_tlm_b_passthrough_target_socket_base

IS-A forward export

Table 288: Parameters

Name	Default value	Description
T	uvm_tlm_generic_payload	

Constructors

function new(string name, uvm_component parent, int min_size = 1, int max_size = 1)

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.2.0.46 Class uvm_pkg::uvm_tlm_b_target_socket

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_b_target_socket_base*
↪*uvm_pkg* :: *uvm_tlm_b_target_socket*

Class

uvm_tlm_b_target_socket

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

The component instantiating this socket must implement a b_transport() method with the following signature

```
task b_transport(T t, uvm_tlm_time delay);
```

Table 289: Parameters

Name	Default value	Description
IMP	int	
T	uvm_tlm_generic_pay-load	

Constructors

`function new(string name, uvm_component parent, int imp = null)`

Function

new

Construct a new instance of this socket *imp* is a reference to the class implementing the b_transport() method. If not specified, it is assume to be the same as *parent* .

Parameters
- **`name`** (`string`)
- **`parent`** (*uvm_component*)
- **`imp`** (`int`)

Functions

`virtual function void connect(this_type provider)`

Function

Connect

Connect this socket to the specified *uvm_tlm_b_initiator_socket*

Parameters
- **`provider`** (*this_type*)

15.2.0.47 Class uvm_pkg::uvm_tlm_b_target_socket_base

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_b_target_socket_base*

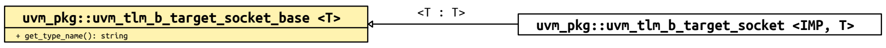

uvm_pkg::uvm_tlm_b_target_socket_base <T>
+ get_type_name(): string
<T : T>
uvm_pkg::uvm_tlm_b_target_socket <IMP, T>

Fig. 102: Inheritance Diagram of uvm_tlm_b_target_socket_base

Class

uvm_tlm_b_target_socket_base

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

Table 290: Parameters

Name	Default value	Description
T	uvm_tlm_generic_pay-load	

Constructors

`function new(string name, uvm_component parent)`

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)

15.2.0.48 Class uvm_pkg::uvm_tlm_b_transport_export

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_b_transport_export*

Class

uvm_tlm_b_transport_export

Blocking transport export class.

Table 291: Parameters

Name	Default value	Description
T	uvm_tlm_generic_pay-load	

Constructors

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

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.2.0.49 Class uvm_pkg::uvm_tlm_b_transport_imp

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_b_transport_imp*

Class

uvm_tlm_b_transport_imp

Used like exports, except an additional class parameter specifies the type of the implementation object. When the imp is instantiated the implementation object is bound.

Table 292: Parameters

Name	Default value	Description
T	uvm_tlm_generic_pay-load	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **`name`** (*`string`*)
- **`imp`** (*`int`*)

15.2.0.50 Class uvm_pkg::uvm_tlm_b_transport_port

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_b_transport_port*

class

uvm_tlm_b_transport_port

Class providing the blocking transport port. The port can be bound to one export. There is no backward path for the blocking transport.

Table 293: Parameters

Name	Default value	Description
T	uvm_tlm_generic_pay-load	

Constructors

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

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.2.0.51 Class uvm_pkg::uvm_tlm_event

Events

`trigger`

15.2.0.52 Class uvm_pkg::uvm_tlm_extension

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_tlm_extension_base*
↪*uvm_pkg* :: *uvm_tlm_extension*

Class

uvm_tlm_extension

TLM extension class. The class is parameterized with arbitrary type which represents the type of the extension. An instance of the generic payload can contain one extension object of each type; it cannot contain two instances of the same extension type.

The extension type can be identified using the *ID()* method.

To implement a generic payload extension, simply derive a new class from this class and specify the name of the derived class as the extension parameter.

|

```
class my_ID extends uvm_tlm_extension#(my_ID);
  int ID;

  `uvm_object_utils_begin(my_ID)
     `uvm_field_int(ID, UVM_ALL_ON)
  `uvm_object_utils_end

  function new(string name = "my_ID");
     super.new(name);
  endfunction
endclass
```

Table 294: Parameters

Name	Default value	Description
T	int	

Table 295: Typedefs

Name	Actual Type	Description
this_type	*uvm_tlm_extension#(T)*	

Constructors

function new(string name = "")

Function

new

creates a new extension object.

Parameters

name (*string*)

Functions

`static function this_type ID()`

> ***Function***
>
> ID()
>
> Return the unique ID of this TLM extension type. This method is used to identify the type of the extension to retrieve from a *uvm_tlm_generic_payload* instance, using the *uvm_tlm_generic_payload::get_extension()* method.
>
> Return type
>
> *this_type*

`virtual function uvm_tlm_extension_base get_type_handle()`

> Return type
>
> *uvm_tlm_extension_base*

`virtual function string get_type_handle_name()`

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

> Parameters
>
> **`name`** (`string`)
>
> Return type
>
> *uvm_object*

15.2.0.53 Class uvm_pkg::uvm_tlm_extension_base

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_tlm_extension_base*

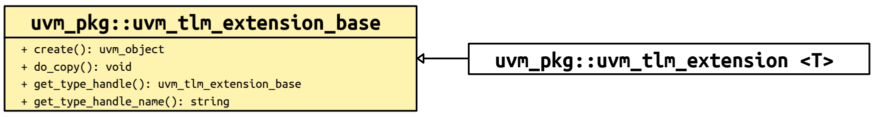

uvm_pkg::uvm_tlm_extension_base
+ create(): uvm_object
+ do_copy(): void
+ get_type_handle(): uvm_tlm_extension_base
+ get_type_handle_name(): string
uvm_pkg::uvm_tlm_extension <T>

Fig. 103: Inheritance Diagram of uvm_tlm_extension_base

Class

uvm_tlm_extension_base

The class uvm_tlm_extension_base is the non-parameterized base class for all generic payload extensions. It includes the utility do_copy() and create(). The pure virtual function get_type_handle() allows you to get a unique handle that represents the derived type. This is implemented in derived classes.

This class is never used directly by users. The *uvm_tlm_extension* class is used instead.

Constructors

`function new(string name = "")`

Function

new

Parameters
name (*string*)

Functions

`virtual function uvm_tlm_extension_base get_type_handle()`

Function

get_type_handle

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

Return type
uvm_tlm_extension_base

`virtual function string get_type_handle_name()`

Function

get_type_handle_name

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

`virtual function void do_copy(uvm_object rhs)`

Parameters
rhs (*uvm_object*)

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

Function

create

Parameters
name (*string*)

Return type
uvm_object

15.2.0.54 Class uvm_pkg::uvm_tlm_fifo

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_tlm_fifo_base*
↪*uvm_pkg* :: *uvm_tlm_fifo*

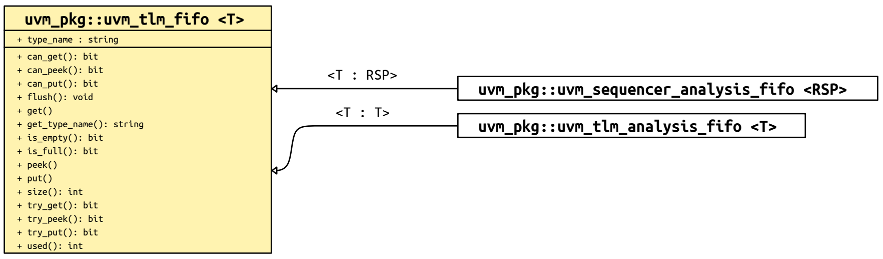


Fig. 104: Inheritance Diagram of uvm_tlm_fifo

Class

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.

Table 296: Parameters

Name	Default value	Description
T	int	

Table 297: Variables

Name	Type	Description
type_name	string	

Constructors

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

Function

new

The *name* and *parent* are the normal uvm_component constructor arguments. The *parent* should be *null* if the <uvm_tlm_fifo(T)> 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.

Parameters

- **name** (*string*)
- **parent** (*uvm_component*)
- **size** (*int*)

Functions

`virtual function string get_type_name()`

`virtual function int size()`

> ***Function***
>
> 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.

`virtual function int used()`

> ***Function***
>
> used
>
> Returns the number of entries put into the FIFO.

`virtual function bit is_empty()`

> ***Function***
>
> is_empty
>
> Returns 1 when there are no entries in the FIFO, 0 otherwise.

`virtual function bit is_full()`

> ***Function***
>
> is_full
>
> Returns 1 when the number of entries in the FIFO is equal to its *size*, 0 otherwise.

`virtual function bit try_get(int t)`

> Parameters
>
> **t** (*int*)

`virtual function bit try_peek(int t)`

> Parameters
>
> **t** (*int*)

`virtual function bit try_put(int t)`

> Parameters
>
> **t** (*int*)

`virtual function bit can_put()`

`virtual function bit can_get()`

`virtual function bit can_peek()`

`virtual function void flush()`

> ***Function***
>
> flush
>
> Removes all entries from the FIFO, after which *used* returns 0 and *is_empty* returns 1.

Tasks

`virtual function put(int t)`

> Parameters
>
> **t** (*int*)

`virtual function get(int t)`

> Parameters
>
> **t** (*int*)

`virtual function peek(int t)`

> Parameters
>
> **t** (*int*)

15.2.0.55 Class uvm_pkg::uvm_tlm_fifo_base

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_tlm_fifo_base*

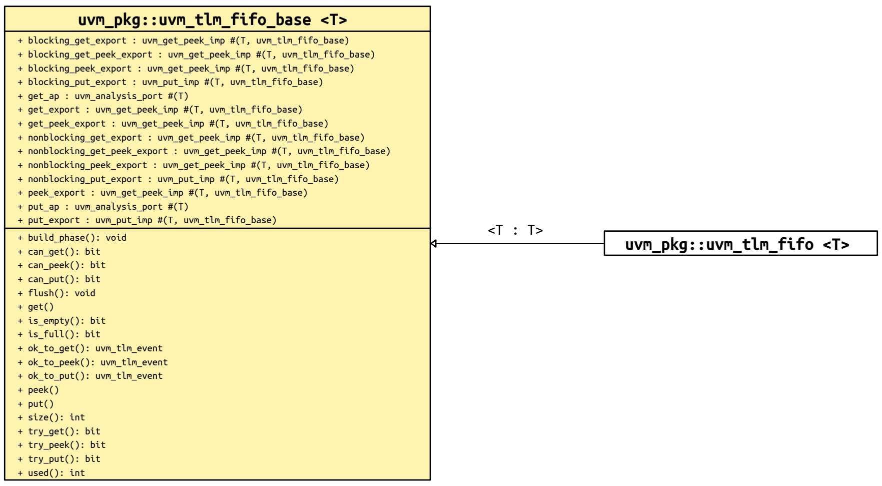

uvm_pkg::uvm_tlm_fifo_base <T>
+ blocking_get_export : uvm_get_peek_imp #(T, uvm_tlm_fifo_base)
+ blocking_get_peek_export : uvm_get_peek_imp #(T, uvm_tlm_fifo_base)
+ blocking_peek_export : uvm_get_peek_imp #(T, uvm_tlm_fifo_base)
+ blocking_put_export : uvm_put_imp #(T, uvm_tlm_fifo_base)
+ get_ap : uvm_analysis_port #(T)
+ get_export : uvm_get_peek_imp #(T, uvm_tlm_fifo_base)
+ get_peek_export : uvm_get_peek_imp #(T, uvm_tlm_fifo_base)
+ nonblocking_get_export : uvm_get_peek_imp #(T, uvm_tlm_fifo_base)
+ nonblocking_get_peek_export : uvm_get_peek_imp #(T, uvm_tlm_fifo_base)
+ nonblocking_peek_export : uvm_get_peek_imp #(T, uvm_tlm_fifo_base)
+ nonblocking_put_export : uvm_put_imp #(T, uvm_tlm_fifo_base)
+ peek_export : uvm_get_peek_imp #(T, uvm_tlm_fifo_base)
+ put_ap : uvm_analysis_port #(T)
+ put_export : uvm_put_imp #(T, uvm_tlm_fifo_base)
+ build_phase(): void
+ can_get(): bit
+ can_peek(): bit
+ can_put(): bit
+ flush(): void
+ get()
+ is_empty(): bit
+ is_full(): bit
+ ok_to_get(): uvm_tlm_event
+ ok_to_peek(): uvm_tlm_event
+ ok_to_put(): uvm_tlm_event
+ peek()
+ put()
+ size(): int
+ try_get(): bit
+ try_peek(): bit
+ try_put(): bit
+ used(): int
<T : T>
uvm_pkg::uvm_tlm_fifo <T>

Fig. 105: Inheritance Diagram of uvm_tlm_fifo_base

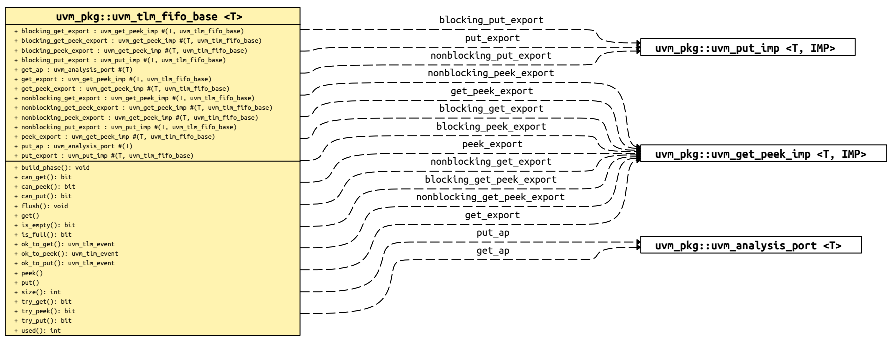


Fig. 106: Collaboration Diagram of uvm_tlm_fifo_base

CLASS

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.

Table 298: Parameters

Name	Default value	Description
T	int	

Table 299: Variables

Name	Type	Description

put_export	*uvm_put_imp#(int, uvm_tlm_fifo_base#(int))*	***Port***
		
		 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	*uvm_get_peek_imp#(int, uvm_tlm_fifo_base#(int))*	***Port***
		
		 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	*uvm_analysis_port#(int)*	***Port*** 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.

continues on next page

Table 299 – continued from previous page

Name	Type	Description
get_ap	*uvm_analysis_port#(int)*	***Port*** 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.
blocking_put_export	*uvm_put_imp#(int, uvm_-tlm_fifo_base#(int))*	The following are aliases to the above put_export.
nonblocking_put_export	*uvm_put_imp#(int, uvm_-tlm_fifo_base#(int))*	
blocking_get_export	*uvm_get_peek_imp#(int, uvm_tlm_fifo_base#(int))*	The following are all aliased to the above get_peek_ex-port, which provides the superset of these interfaces.
nonblocking_get_export	*uvm_get_peek_imp#(int, uvm_tlm_fifo_base#(int))*	
get_export	*uvm_get_peek_imp#(int, uvm_tlm_fifo_base#(int))*	
blocking_peek_export	*uvm_get_peek_imp#(int, uvm_tlm_fifo_base#(int))*	
nonblocking_peek_ex-port	*uvm_get_peek_imp#(int, uvm_tlm_fifo_base#(int))*	
peek_export	*uvm_get_peek_imp#(int, uvm_tlm_fifo_base#(int))*	
blocking_get_peek_ex-port	*uvm_get_peek_imp#(int, uvm_tlm_fifo_base#(int))*	
nonblocking_get_peek_-export	*uvm_get_peek_imp#(int, uvm_tlm_fifo_base#(int))*	

Table 300: Typedefs

Name	Actual Type	Description
this_type	*uvm_tlm_fifo_base#(T)*	

Constructors

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

Function

new

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.

Parameters

`name` (`string`)
`parent` (*uvm_component*)

Functions

`virtual function void build_phase(uvm_phase phase)`

Turn off auto config

Parameters
: **`phase`** (*uvm_phase*)

`virtual function void flush()`

`virtual function int size()`

`virtual function bit try_put(int t)`

Parameters
: **`t`** (*`int`*)

`virtual function bit try_get(int t)`

Parameters
: **`t`** (*`int`*)

`virtual function bit try_peek(int t)`

Parameters
: **`t`** (*`int`*)

`virtual function bit can_put()`

`virtual function bit can_get()`

`virtual function bit can_peek()`

`virtual function uvm_tlm_event ok_to_put()`

Return type
: *uvm_tlm_event*

`virtual function uvm_tlm_event ok_to_get()`

Return type
: *uvm_tlm_event*

`virtual function uvm_tlm_event ok_to_peek()`

Return type
: *uvm_tlm_event*

`virtual function bit is_empty()`

`virtual function bit is_full()`

`virtual function int used()`

Tasks

`virtual function put(int t)`

Parameters
: **`t`** (*`int`*)

`virtual function get(int t)`

Parameters
: **`t`** (*`int`*)

`virtual function peek(int t)`

Parameters
: **`t`** (*`int`*)

15.2.0.56 Class uvm_pkg::uvm_tlm_generic_payload

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_transaction*
↪*uvm_pkg* :: *uvm_sequence_item*
↪*uvm_pkg* :: *uvm_tlm_generic_payload*

Class

uvm_tlm_generic_payload

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

Constructors

`function new(string name = "")`

Function

new

Create a new instance of the generic payload. Initialize all the members to their default values.

Parameters

`name` (*`string`*)

Functions

`virtual function void do_print(uvm_printer printer)`

Function- do_print

Parameters

`printer` (*uvm_printer*)

`virtual function void do_copy(uvm_object rhs)`

Function- do_copy

Parameters

`rhs` (*uvm_object*)

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

Function- do_compare

Parameters

`rhs` (*uvm_object*)
`comparer` (*uvm_comparer*)

`virtual function void do_pack(uvm_packer packer)`

Function- do_pack

We only pack m_length bytes of the m_data array, even if m_data is larger than m_length. Same treatment for the byte-enable array. We do not pack the extensions, if any, as we will be unable to unpack them.

Parameters

`packer` (*uvm_packer*)

`virtual function void do_unpack(uvm_packer packer)`

Function- do_unpack

We only reallocate m_data/m_byte_enable if the new size is greater than their current size. We do not unpack extensions because we do not know what object types to allocate before we unpack into them. Extensions must be handled by user code.

Parameters

`packer` (*uvm_packer*)

`virtual function void do_record(uvm_recorder recorder)`

Function- do_record

Parameters

`recorder` (*uvm_recorder*)

`virtual function string convert2string()`

Function- convert2string

`virtual function uvm_tlm_command_e get_command()`

Function

get_command

Get the value of the m_command variable

Return type

uvm_tlm_command_e

`virtual function void set_command(uvm_tlm_command_e command)`

Function

set_command

Set the value of the m_command variable

Parameters

`command` (*uvm_tlm_command_e*)

`virtual function bit is_read()`

Function

is_read

Returns true if the current value of the m_command variable is *UVM_TLM_READ_COMMAND* .

`virtual function void set_read()`

Function

set_read

Set the current value of the m_command variable to *UVM_TLM_READ_COMMAND* .

`virtual function bit is_write()`

Function

is_write

Returns true if the current value of the m_command variable is *UVM_TLM_WRITE_COMMAND* .

`virtual function void set_write()`

Function

set_write

Set the current value of the m_command variable to *UVM_TLM_WRITE_COMMAND* .

`virtual function void set_address(bit[63:0] addr)`

Function

set_address

Set the value of the m_address variable

Parameters

`addr` (*`bit[63:0]`*)

`virtual function bit[63:0] get_address()`

Function

get_address

Get the value of the m_address variable

`virtual function void get_data(byte unsigned p)`

Function

get_data

Return the value of the m_data array

Parameters
`p` (*`byte unsigned`*)

`virtual function void set_data(byte unsigned p)`

Function

set_data

Set the value of the m_data array
Parameters
`p` (*`byte unsigned`*)

`virtual function int unsigned get_data_length()`

Function

get_data_length

Return the current size of the m_data array

`virtual function void set_data_length(int unsigned length)`

Function

set_data_length

Set the value of the m_length
Parameters
`length` (*`int unsigned`*)

`virtual function int unsigned get_streaming_width()`

Function

get_streaming_width

Get the value of the m_streaming_width array

`virtual function void set_streaming_width(int unsigned width)`

Function

set_streaming_width

Set the value of the m_streaming_width array
Parameters
`width` (*`int unsigned`*)

`virtual function void get_byte_enable(byte unsigned p)`

Function

get_byte_enable

Return the value of the m_byte_enable array
Parameters
`p` (*`byte unsigned`*)

`virtual function void set_byte_enable(byte unsigned p)`

Function

set_byte_enable

Set the value of the m_byte_enable array
Parameters
`p` (*`byte unsigned`*)

`virtual function int unsigned get_byte_enable_length()`

Function

get_byte_enable_length

Return the current size of the m_byte_enable array

`virtual function void set_byte_enable_length(int unsigned length)`

Function

set_byte_enable_length

Set the size m_byte_enable_length of the m_byte_enable array i.e. m_byte_enable.size()
Parameters
`length` (*`int unsigned`*)

`virtual function void set_dmi_allowed(bit dmi)`

Function

set_dmi_allowed

DMI hint. Set the internal flag m_dmi to allow dmi access

Parameters

`dmi` (*`bit`*)

`virtual function bit is_dmi_allowed()`

Function

is_dmi_allowed

DMI hint. Query the internal flag m_dmi if allowed dmi access

`virtual function uvm_tlm_response_status_e get_response_status()`

Function

get_response_status

Return the current value of the m_response_status variable

Return type

uvm_tlm_response_status_e

`virtual function void set_response_status(uvm_tlm_response_status_e status)`

Function

set_response_status

Set the current value of the m_response_status variable

Parameters

`status` (*uvm_tlm_response_status_e*)

`virtual function bit is_response_ok()`

Function

is_response_ok

Return TRUE if the current value of the m_response_status variable is *UVM_TLM_OK_RESPONSE*

`virtual function bit is_response_error()`

Function

is_response_error

Return TRUE if the current value of the m_response_status variable is not *UVM_TLM_OK_RESPONSE*

`virtual function string get_response_string()`

Function

get_response_string

Return the current value of the m_response_status variable as a string

`function uvm_tlm_extension_base set_extension(uvm_tlm_extension_base ext)`

Function

set_extension

Add an instance-specific extension. Only one instance of any given extension type is allowed. If there is an existing extension instance of the type of *ext* , *ext* replaces it and its handle is returned. Otherwise, *null* is returned.

Parameters

`ext` (*uvm_tlm_extension_base*)

Return type

uvm_tlm_extension_base

`function int get_num_extensions()`

Function

get_num_extensions

Return the current number of instance specific extensions.

`function uvm_tlm_extension_base get_extension(uvm_tlm_extension_base ext_handle)`

Function

get_extension

Return the instance specific extension bound under the specified key. If no extension is bound under that key, *null* is returned.

Parameters
: **`ext_handle`** (*uvm_tlm_extension_base*)

Return type
: *uvm_tlm_extension_base*

`function void clear_extension(uvm_tlm_extension_base ext_handle)`

Function

clear_extension

Remove the instance-specific extension bound under the specified key.

Parameters
: **`ext_handle`** (*uvm_tlm_extension_base*)

`function void clear_extensions()`

Function

clear_extensions

Remove all instance-specific extensions

15.2.0.57 Class uvm_pkg::uvm_tlm_if

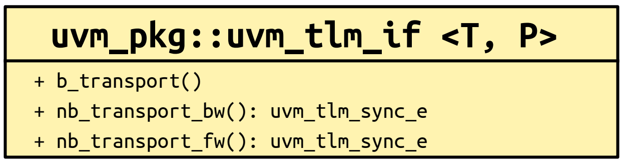

uvm_pkg::uvm_tlm_if <T, P>
+ b_transport()
+ nb_transport_bw(): uvm_tlm_sync_e
+ nb_transport_fw(): uvm_tlm_sync_e

Fig. 107: Inheritance Diagram of uvm_tlm_if

Class

uvm_tlm_if

Base class type to define the transport functions.

nb_transport_fw
nb_transport_bw
b_transport

Table 301: Parameters

Name	Default value	Description
T	uvm_tlm_generic_payload	
P	uvm_tlm_phase_e	

Functions

`virtual function uvm_tlm_sync_e nb_transport_fw(uvm_tlm_generic_payload t, uvm_tlm_phase_e p, uvm_tlm_time delay)`

Function

nb_transport_fw

Forward path call. The first call to this method for a transaction marks the initial timing point. Every call to this method may mark a timing point in the execution of the transaction. The timing annotation argument allows the timing points to be offset from the simulation times at which the forward path is used. The final timing point of a transaction may be marked by a call to *nb_transport_bw* or a return from this or subsequent call to nb_transport_fw.

See <TLM2 Interfaces, Ports, Exports and Transport Interfaces Subset> for more details on the semantics and rules of the nonblocking transport interface.

Parameters
- **t** (*uvm_tlm_generic_payload*)
- **p** (*uvm_tlm_phase_e*)
- **delay** (*uvm_tlm_time*)

Return type
uvm_tlm_sync_e

`virtual function uvm_tlm_sync_e nb_transport_bw(uvm_tlm_generic_payload t, uvm_tlm_phase_e p, uvm_tlm_time delay)`

Function

nb_transport_bw

Implementation of the backward path. This function MUST be implemented in the INITIATOR component class.

Every call to this method may mark a timing point, including the final timing point, in the execution of the transaction. The timing annotation argument allows the timing point to be offset from the simulation times at which the backward path is used. The final timing point of a transaction may be marked by a call to *nb_transport_fw* or a return from this or subsequent call to nb_transport_bw.

See <TLM2 Interfaces, Ports, Exports and Transport Interfaces Subset> for more details on the semantics and rules of the nonblocking transport interface.

Example:

```
class master extends uvm_component;
```

```
uvm\_tlm\_nb\_initiator\_socket (trans, uvm\_tlm\_phase\_e, this\_t) initiator\_socket;
```

```
...
function void build_phase(uvm_phase phase);
```

```
    initiator\_socket = new("initiator\_socket", this, this);
```

```
   endfunction

   function uvm_tlm_sync_e nb_transport_bw(ref trans t,
                                  ref uvm_tlm_phase_e p,
                                  input uvm_tlm_time delay);
       transaction = t;
       state = p;
       return UVM_TLM_ACCEPTED;
   endfunction

   ...
endclass
```

Parameters
- **t** (*uvm_tlm_generic_payload*)
- **p** (*uvm_tlm_phase_e*)
- **delay** (*uvm_tlm_time*)

Return type
- *uvm_tlm_sync_e*

Tasks

virtual function b_transport(uvm_tlm_generic_payload t, uvm_tlm_time delay)

Function

b_transport

Execute a blocking transaction. Once this method returns, the transaction is assumed to have been executed. Whether that execution is successful or not must be indicated by the transaction itself.

The callee may modify or update the transaction object, subject to any constraints imposed by the transaction class. The initiator may re-use a transaction object from one call to the next and across calls to b_transport().

The call to b_transport shall mark the first timing point of the transaction. The return from b_transport shall mark the final timing point of the transaction. The timing annotation argument allows the timing points to be offset from the simulation times at which the task call and return are executed.

Parameters
- **t** (*uvm_tlm_generic_payload*)
- **delay** (*uvm_tlm_time*)

15.2.0.58 Class uvm_pkg::uvm_tlm_if_base

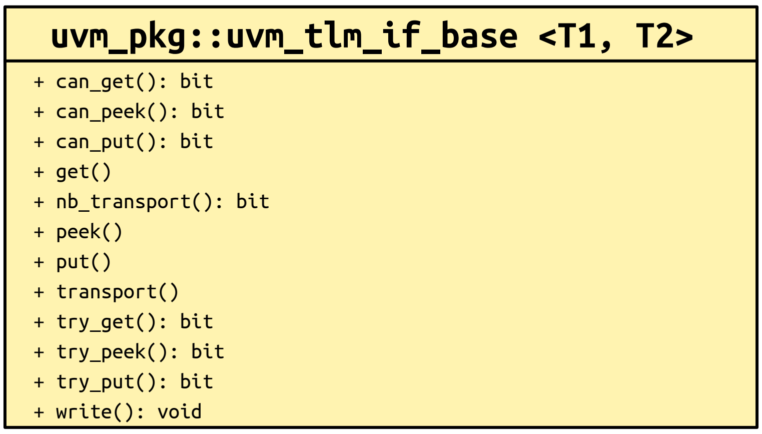


Fig. 108: Inheritance Diagram of uvm_tlm_if_base

CLASS

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.

Table 302: Parameters

Name	Default value	Description
T1	int	
T2	int	

Functions

virtual function bit try_put(int t)

Function

try_put

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.

Parameters
t (*int*)

`virtual function bit can_put()`

Function

can_put

Returns 1 if the component is ready to accept the transaction; 0 otherwise.

`virtual function bit try_get(int t)`

Function

try_get

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.

Parameters

`t` (*`int`*)

`virtual function bit can_get()`

Function

can_get

Returns 1 if a new transaction can be provided immediately upon request, 0 otherwise.

`virtual function bit try_peek(int t)`

Function

try_peek

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.

Parameters

`t` (*`int`*)

`virtual function bit can_peek()`

Function

can_peek

Returns 1 if a new transaction is available; 0 otherwise.

`virtual function bit nb_transport(int req, int rsp)`

Task

nb_transport

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.

Parameters

`req` (*`int`*)
`rsp` (*`int`*)

`virtual function void write(int t)`

Function

write

Broadcasts a user-defined transaction of type T to any number of listeners. The operation must complete without blocking.

Parameters

`t` (*`int`*)

Tasks

`virtual function put(int t)`

Task

put

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.

Parameters

`t` (*`int`*)

`virtual function get(int t)`

Task

get

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.

Parameters

`t` (*`int`*)

`virtual function peek(int t)`

Task

peek

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.

Parameters

`t` (*`int`*)

`virtual function transport(int req, int rsp)`

Task

transport

Executes the given request and returns the response in the given output argument. The calling thread may block until the operation is complete.

Parameters

`req` (*`int`*)

`rsp` (*`int`*)

15.2.0.59 Class uvm_pkg::uvm_tlm_nb_initiator_socket

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_nb_initiator_socket_base*
↪*uvm_pkg* :: *uvm_tlm_nb_initiator_socket*

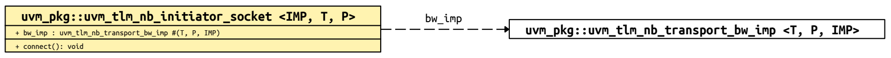

uvm_pkg::uvm_tlm_nb_initiator_socket <IMP, T, P>
+ bw_imp : uvm_tlm_nb_transport_bw_imp #(T, P, IMP)
+ connect(): void
bw_imp
uvm_pkg::uvm_tlm_nb_transport_bw_imp <T, P, IMP>

Fig. 109: Collaboration Diagram of uvm_tlm_nb_initiator_socket

Class

uvm_tlm_nb_initiator_socket

IS-A forward port; HAS-A backward imp

The component instantiating this socket must implement a nb_transport_bw() method with the following signature

```
function uvm_tlm_sync_e nb_transport_bw(T t, ref P p, input uvm_tlm_time delay);
```

Table 303: Parameters

Name	Default value	Description
IMP	int	
T	uvm_tlm_generic_pay-load	
P	uvm_tlm_phase_e	

Table 304: Variables

Name	Type	Description
bw_imp	*uvm_tlm_nb_trans-port_bw_imp#(uvm_-tlm_generic_payload, uvm_tlm_phase_e, int)*	

Constructors

`function new(string name, uvm_component parent, int imp = null)`

Function

new

Construct a new instance of this socket *imp* is a reference to the class implementing the nb_transport_bw() method. If not specified, it is assume to be the same as *parent* .

Parameters

- **`name`** (`string`)
- **`parent`** (*uvm_component*)
- **`imp`** (`int`)

Functions

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

Function

Connect

Connect this socket to the specified *uvm_tlm_nb_target_socket*

Parameters

`provider` (*this_type*)

15.2.0.60 Class uvm_pkg::uvm_tlm_nb_initiator_socket_base

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_nb_initiator_socket_base*

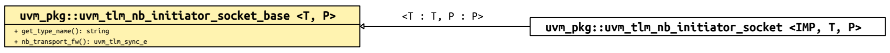

uvm_pkg::uvm_tlm_nb_initiator_socket_base <T, P>
+ get_type_name(): string
+ nb_transport_fw(): uvm_tlm_sync_e
<T : T, P : P>
uvm_pkg::uvm_tlm_nb_initiator_socket <IMP, T, P>

Fig. 110: Inheritance Diagram of uvm_tlm_nb_initiator_socket_base

Class

uvm_tlm_nb_initiator_socket_base

IS-A forward port; HAS-A backward imp

Table 305: Parameters

Name	Default value	Description
T	uvm_tlm_generic_pay-load	
P	uvm_tlm_phase_e	

Constructors

`function new(string name, uvm_component parent)`

Parameters
- **`name`** (`string`)
- **`parent`** (*uvm_component*)

15.2.0.61 Class uvm_pkg::uvm_tlm_nb_passthrough_initiator_socket

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_nb_passthrough_initiator_socket_base*
↪*uvm_pkg* :: *uvm_tlm_nb_passthrough_initiator_socket*

Class

uvm_tlm_nb_passthrough_initiator_socket

IS-A forward port; HAS-A backward export

Table 306: Parameters

Name	Default value	Description
T	uvm_tlm_generic_pay-load	
P	uvm_tlm_phase_e	

Constructors

`function new(string name, uvm_component parent)`

Parameters
- **`name`** (`string`)
- **`parent`** (*uvm_component*)

Functions

`virtual function void connect(this_type provider)`

Function

connect

Connect this socket to the specified *uvm_tlm_nb_target_socket*

Parameters
- **`provider`** (*this_type*)

15.2.0.62 Class uvm_pkg::uvm_tlm_nb_passthrough_initiator_socket_base

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_nb_passthrough_initiator_socket_base*

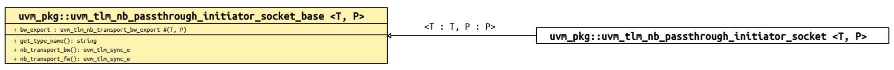

uvm_pkg::uvm_tlm_nb_passthrough_initiator_socket_base <T, P>
+ bw_export : uvm_tlm_nb_transport_bw_export #(T, P)
+ get_type_name(): string
+ nb_transport_bw(): uvm_tlm_sync_e
+ nb_transport_fw(): uvm_tlm_sync_e
<T : T, P : P>
uvm_pkg::uvm_tlm_nb_passthrough_initiator_socket <T, P>

Fig. 111: Inheritance Diagram of uvm_tlm_nb_passthrough_initiator_socket_base

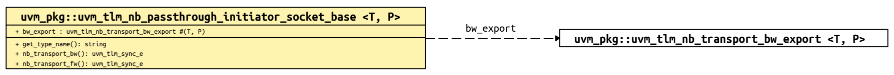

uvm_pkg::uvm_tlm_nb_passthrough_initiator_socket_base <T, P>
+ bw_export : uvm_tlm_nb_transport_bw_export #(T, P)
+ get_type_name(): string
+ nb_transport_bw(): uvm_tlm_sync_e
+ nb_transport_fw(): uvm_tlm_sync_e
bw_export
uvm_pkg::uvm_tlm_nb_transport_bw_export <T, P>

Fig. 112: Collaboration Diagram of uvm_tlm_nb_passthrough_initiator_socket_base

Class

uvm_tlm_nb_passthrough_initiator_socket_base

IS-A forward port; HAS-A backward export

Table 307: Parameters

Name	Default value	Description
T	uvm_tlm_generic_pay-load	
P	uvm_tlm_phase_e	

Table 308: Variables

Name	Type	Description
bw_export	*uvm_tlm_nb_trans-port_bw_export#(uvm_-tlm_generic_payload, uvm_tlm_phase_e)*	

Constructors

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

Parameters

name (*string*)
parent (*uvm_component*)
min_size (*int*)
max_size (*int*)

15.2.0.63 Class uvm_pkg::uvm_tlm_nb_passthrough_target_socket

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_nb_passthrough_target_socket_base*
↪*uvm_pkg* :: *uvm_tlm_nb_passthrough_target_socket*

Class

uvm_tlm_nb_passthrough_target_socket

IS-A forward export; HAS-A backward port

Table 309: Parameters

Name	Default value	Description
T	uvm_tlm_generic_payload	
P	uvm_tlm_phase_e	

Constructors

`function new(string name, uvm_component parent)`

Parameters
- **`name`** (`string`)
- **`parent`** (*uvm_component*)

Functions

`virtual function void connect(this_type provider)`

Function

connect

Connect this socket to the specified *uvm_tlm_nb_initiator_socket*

Parameters
- **`provider`** (*this_type*)

15.2.0.64 Class uvm_pkg::uvm_tlm_nb_passthrough_target_socket_base

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_nb_passthrough_target_socket_base*

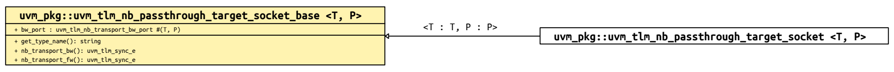

uvm_pkg::uvm_tlm_nb_passthrough_target_socket_base <T, P>
+ bw_port : uvm_tlm_nb_transport_bw_port #(T, P)
+ get_type_name(): string
+ nb_transport_bw(): uvm_tlm_sync_e
+ nb_transport_fw(): uvm_tlm_sync_e
<T : T, P : P>
uvm_pkg::uvm_tlm_nb_passthrough_target_socket <T, P>

Fig. 113: Inheritance Diagram of uvm_tlm_nb_passthrough_target_socket_base

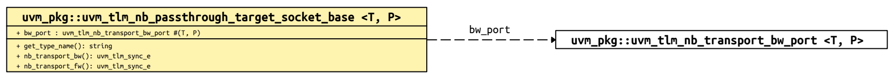

uvm_pkg::uvm_tlm_nb_passthrough_target_socket_base <T, P>
+ bw_port : uvm_tlm_nb_transport_bw_port #(T, P)
+ get_type_name(): string
+ nb_transport_bw(): uvm_tlm_sync_e
+ nb_transport_fw(): uvm_tlm_sync_e
bw_port
uvm_pkg::uvm_tlm_nb_transport_bw_port <T, P>

Fig. 114: Collaboration Diagram of uvm_tlm_nb_passthrough_target_socket_base

Class

uvm_tlm_nb_passthrough_target_socket_base

IS-A forward export; HAS-A backward port

Table 310: Parameters

Name	Default value	Description
T	uvm_tlm_generic_payload	
P	uvm_tlm_phase_e	

Table 311: Variables

Name	Type	Description
bw_port	*uvm_tlm_nb_transport_bw_port#(uvm_tlm_generic_payload, uvm_tlm_phase_e)*	

Constructors

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

Parameters
- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.2.0.65 Class uvm_pkg::uvm_tlm_nb_target_socket

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_nb_target_socket_base*
↪*uvm_pkg* :: *uvm_tlm_nb_target_socket*

Class

uvm_tlm_nb_target_socket

IS-A forward imp; HAS-A backward port

The component instantiating this socket must implement a nb_transport_fw() method with the following signature

```
function uvm_tlm_sync_e nb_transport_fw(T t, ref P p, input uvm_tlm_time delay);
```

Table 312: Parameters

Name	Default value	Description
IMP	int	
T	uvm_tlm_generic_pay-load	
P	uvm_tlm_phase_e	

Constructors

`function new(string name, uvm_component parent, int imp = null)`

Function

new

Construct a new instance of this socket *imp* is a reference to the class implementing the nb_transport_fw() method. If not specified, it is assume to be the same as *parent* .

Parameters
- **`name`** (`string`)
- **`parent`** (*uvm_component*)
- **`imp`** (`int`)

Functions

`virtual function void connect(this_type provider)`

Function

connect

Connect this socket to the specified *uvm_tlm_nb_initiator_socket*

Parameters
- **`provider`** (*this_type*)

15.2.0.66 Class uvm_pkg::uvm_tlm_nb_target_socket_base

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_nb_target_socket_base*

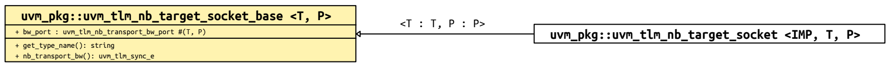

uvm_pkg::uvm_tlm_nb_target_socket_base <T, P>
+ bw_port : uvm_tlm_nb_transport_bw_port #(T, P)
+ get_type_name(): string
+ nb_transport_bw(): uvm_tlm_sync_e
<T : T, P : P>
uvm_pkg::uvm_tlm_nb_target_socket <IMP, T, P>

Fig. 115: Inheritance Diagram of uvm_tlm_nb_target_socket_base

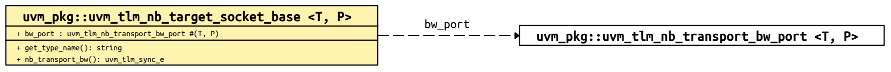

uvm_pkg::uvm_tlm_nb_target_socket_base <T, P>
+ bw_port : uvm_tlm_nb_transport_bw_port #(T, P)
+ get_type_name(): string
+ nb_transport_bw(): uvm_tlm_sync_e
bw_port
uvm_pkg::uvm_tlm_nb_transport_bw_port <T, P>

Fig. 116: Collaboration Diagram of uvm_tlm_nb_target_socket_base

Class

uvm_tlm_nb_target_socket_base

IS-A forward imp; HAS-A backward port

Table 313: Parameters

Name	Default value	Description
T	uvm_tlm_generic_pay-load	
P	uvm_tlm_phase_e	

Table 314: Variables

Name	Type	Description
bw_port	*uvm_tlm_nb_trans-port_bw_port#(uvm_-tlm_generic_payload, uvm_tlm_phase_e)*	

Constructors

`function new(string name, uvm_component parent)`

Parameters

- **`name`** (`string`)
- **`parent`** (*uvm_component*)

15.2.0.67 Class uvm_pkg::uvm_tlm_nb_transport_bw_export

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_nb_transport_bw_export*

Class

uvm_tlm_nb_transport_bw_export

Non-blocking backward transport export class

Table 315: Parameters

Name	Default value	Description
T	uvm_tlm_generic_pay-load	
P	uvm_tlm_phase_e	

Constructors

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

Function

new

Parameters

- **name** (*string*)
- **parent** (*uvm_component*)
- **min_size** (*int*)
- **max_size** (*int*)

15.2.0.68 Class uvm_pkg::uvm_tlm_nb_transport_bw_imp

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_nb_transport_bw_imp*

Class

uvm_tlm_nb_transport_bw_imp

Used like exports, except an additional class parameter specifies the type of the implementation object. When the imp is instantiated the implementation object is bound.

Table 316: Parameters

Name	Default value	Description
T	uvm_tlm_generic_pay-load	
P	uvm_tlm_phase_e	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **`name`** (*`string`*)
- **`imp`** (*`int`*)

15.2.0.69 Class uvm_pkg::uvm_tlm_nb_transport_bw_port

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_nb_transport_bw_port*

class

uvm_tlm_nb_transport_bw_port

Class providing the non-blocking backward transport port. Transactions received from the producer, on the forward path, are sent back to the producer on the backward path using this non-blocking transport port The port can be bound to one export.

Table 317: Parameters

Name	Default value	Description
T	uvm_tlm_generic_payload	
P	uvm_tlm_phase_e	

Constructors

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

Function

new

Parameters

- **`name`** (`string`)
- **`parent`** (*uvm_component*)
- **`min_size`** (`int`)
- **`max_size`** (`int`)

15.2.0.70 Class uvm_pkg::uvm_tlm_nb_transport_fw_export

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_nb_transport_fw_export*

Class

uvm_tlm_nb_transport_fw_export

Non-blocking forward transport export class

Table 318: Parameters

Name	Default value	Description
T	uvm_tlm_generic_pay-load	
P	uvm_tlm_phase_e	

Constructors

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

Parameters
- **`name`** (`string`)
- **`parent`** (*uvm_component*)
- **`min_size`** (`int`)
- **`max_size`** (`int`)

15.2.0.71 Class uvm_pkg::uvm_tlm_nb_transport_fw_imp

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_nb_transport_fw_imp*

Class

uvm_tlm_nb_transport_fw_imp

Used like exports, except an additional class parameter specifies the type of the implementation object. When the imp is instantiated the implementation object is bound.

Table 319: Parameters

Name	Default value	Description
T	uvm_tlm_generic_pay-load	
P	uvm_tlm_phase_e	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **name** (*string*)
- **imp** (*int*)

15.2.0.72 Class uvm_pkg::uvm_tlm_nb_transport_fw_port

uvm_pkg :: *uvm_tlm_if*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_tlm_nb_transport_fw_port*

class

uvm_tlm_nb_transport_fw_port

Class providing the non-blocking backward transport port. Transactions received from the producer, on the forward path, are sent back to the producer on the backward path using this non-blocking transport port. The port can be bound to one export.

Table 320: Parameters

Name	Default value	Description
T	uvm_tlm_generic_pay-load	
P	uvm_tlm_phase_e	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.2.0.73 Class uvm_pkg::uvm_tlm_req_rsp_channel

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_tlm_req_rsp_channel*

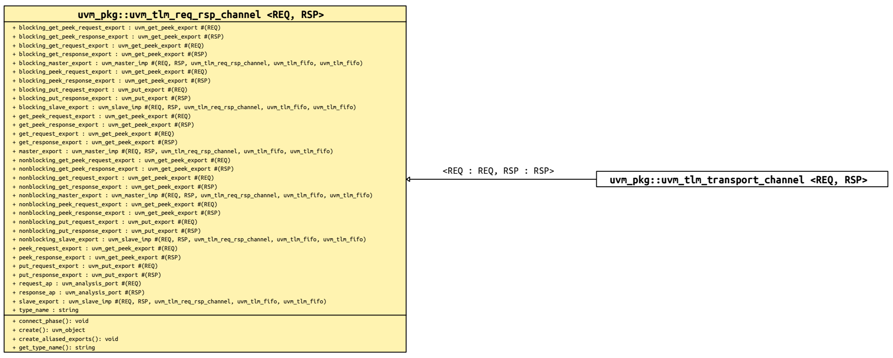

uvm_pkg::uvm_tlm_req_rsp_channel <REQ, RSP>
+ blocking_get_peek_request_export : uvm_get_peek_export #(REQ)
+ blocking_get_peek_response_export : uvm_get_peek_export #(RSP)
+ blocking_get_request_export : uvm_get_peek_export #(REQ)
+ blocking_get_response_export : uvm_get_peek_export #(RSP)
+ blocking_master_export : uvm_master_imp #(REQ, RSP, uvm_tlm_req_rsp_channel, uvm_tlm_fifo, uvm_tlm_fifo)
+ blocking_peek_request_export : uvm_get_peek_export #(REQ)
+ blocking_peek_response_export : uvm_get_peek_export #(RSP)
+ blocking_put_request_export : uvm_put_export #(REQ)
+ blocking_put_response_export : uvm_put_export #(RSP)
+ blocking_slave_export : uvm_slave_imp #(REQ, RSP, uvm_tlm_req_rsp_channel, uvm_tlm_fifo, uvm_tlm_fifo)
+ get_peek_request_export : uvm_get_peek_export #(REQ)
+ get_peek_response_export : uvm_get_peek_export #(RSP)
+ get_request_export : uvm_get_peek_export #(REQ)
+ get_response_export : uvm_get_peek_export #(RSP)
+ master_export : uvm_master_imp #(REQ, RSP, uvm_tlm_req_rsp_channel, uvm_tlm_fifo, uvm_tlm_fifo)
+ nonblocking_get_peek_request_export : uvm_get_peek_export #(REQ)
+ nonblocking_get_peek_response_export : uvm_get_peek_export #(RSP)
+ nonblocking_get_request_export : uvm_get_peek_export #(REQ)
+ nonblocking_get_response_export : uvm_get_peek_export #(RSP)
+ nonblocking_master_export : uvm_master_imp #(REQ, RSP, uvm_tlm_req_rsp_channel, uvm_tlm_fifo, uvm_tlm_fifo)
+ nonblocking_peek_request_export : uvm_get_peek_export #(REQ)
+ nonblocking_peek_response_export : uvm_get_peek_export #(RSP)
+ nonblocking_put_request_export : uvm_put_export #(REQ)
+ nonblocking_put_response_export : uvm_put_export #(RSP)
+ nonblocking_slave_export : uvm_slave_imp #(REQ, RSP, uvm_tlm_req_rsp_channel, uvm_tlm_fifo, uvm_tlm_fifo)
+ peek_request_export : uvm_get_peek_export #(REQ)
+ peek_response_export : uvm_get_peek_export #(RSP)
+ put_request_export : uvm_put_export #(REQ)
+ put_response_export : uvm_put_export #(RSP)
+ request_ap : uvm_analysis_port #(REQ)
+ response_ap : uvm_analysis_port #(RSP)
+ slave_export : uvm_slave_imp #(REQ, RSP, uvm_tlm_req_rsp_channel, uvm_tlm_fifo, uvm_tlm_fifo)
+ type_name : string
+ connect_phase(): void
+ create(): uvm_object
+ create_aliased_exports(): void
+ get_type_name(): string
<REQ : REQ, RSP : RSP>
uvm_pkg::uvm_tlm_transport_channel <REQ, RSP>

Fig. 117: Inheritance Diagram of uvm_tlm_req_rsp_channel

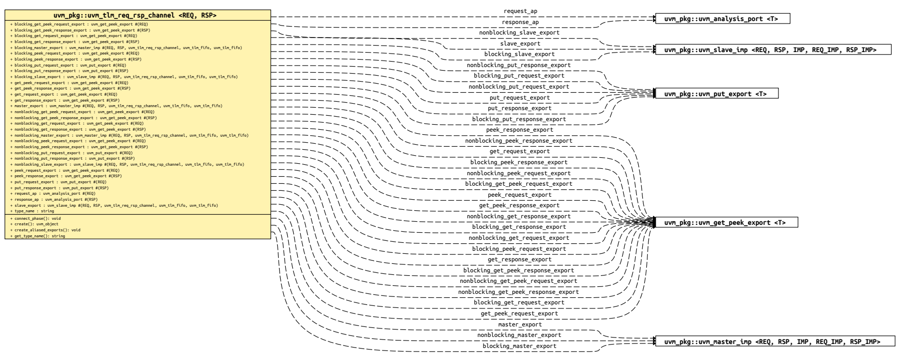

uvm_pkg::uvm_tlm_req_rsp_channel <REQ, RSP>
request_ap
response_ap
uvm_pkg::uvm_analysis_port <T>
nonblocking_slave_export
slave_export
blocking_slave_export
uvm_pkg::uvm_slave_imp <REQ, RSP, IMP, REQ_IMP, RSP_IMP>
nonblocking_put_response_export
blocking_put_request_export
nonblocking_put_request_export
put_request_export
put_response_export
blocking_put_response_export
uvm_pkg::uvm_put_export <T>
peek_response_export
nonblocking_peek_response_export
get_request_export
blocking_peek_response_export
nonblocking_peek_request_export
blocking_get_peek_request_export
peek_request_export
get_peek_response_export
nonblocking_get_response_export
blocking_get_response_export
nonblocking_get_request_export
blocking_peek_request_export
get_response_export
blocking_get_peek_response_export
nonblocking_get_peek_request_export
nonblocking_get_peek_response_export
blocking_get_request_export
get_peek_request_export
uvm_pkg::uvm_get_peek_export <T>
master_export
nonblocking_master_export
blocking_master_export
uvm_pkg::uvm_master_imp <REQ, RSP, IMP, REQ_IMP, RSP_IMP>

Fig. 118: Collaboration Diagram of uvm_tlm_req_rsp_channel

CLASS

uvm_tlm_req_rsp_channel (REQ, RSP)

The uvm_tlm_req_rsp_channel contains a request FIFO of type *REQ* and a response FIFO of type *RSP* . These FIFOs can be of any size. This channel is particularly useful for dealing with pipelined protocols where the request and response are not tightly coupled.

Type parameters:

REQ

Type of the request transactions conveyed by this channel.

RSP

Type of the response transactions conveyed by this channel.

Table 321: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Table 322: Variables

Name	Type	Description
type_name	string	

put_request_export	*uvm_put_export#(int)*	***Port***
		
		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	*uvm_get_peek_export#(int)*	***Port***
		
		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	*uvm_get_peek_export#(int)*	

put_response_export	*uvm_put_export#(int)*	***Port***
		
		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.

continues on next page

Table 322 – continued from previous page

Name	Type	Description
request_ap	*uvm_analysis_port#(int)*	***Port*** 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	*uvm_analysis_port#(int)*	***Port*** 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	*uvm_master_imp#(int, int, uvm_tlm_req_-rsp_channel#(int, int), uvm_tlm_fifo#(int), uvm_tlm_fifo#(int))*	***Port*** 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	*uvm_slave_imp#(int, int, uvm_tlm_req_-rsp_channel#(int, int), uvm_tlm_fifo#(int), uvm_tlm_fifo#(int))*	***Port*** 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.
blocking_put_request_-export	*uvm_put_export#(int)*	port aliases for backward compatibility
nonblocking_put_request_export	*uvm_put_export#(int)*	
get_request_export	*uvm_get_peek_export#(int)*	
blocking_get_request_-export	*uvm_get_peek_export#(int)*	
nonblocking_get_request_export	*uvm_get_peek_export#(int)*	
peek_request_export	*uvm_get_peek_export#(int)*	
blocking_peek_request_-export	*uvm_get_peek_export#(int)*	

continues on next page

Table 322 – continued from previous page

Name	Type	Description
nonblocking_peek_request_export	*uvm_get_peek_export#(int)*	
blocking_get_peek_request_export	*uvm_get_peek_export#(int)*	
nonblocking_get_peek_request_export	*uvm_get_peek_export#(int)*	
blocking_put_response_export	*uvm_put_export#(int)*	
nonblocking_put_response_export	*uvm_put_export#(int)*	
get_response_export	*uvm_get_peek_export#(int)*	
blocking_get_response_export	*uvm_get_peek_export#(int)*	
nonblocking_get_response_export	*uvm_get_peek_export#(int)*	
peek_response_export	*uvm_get_peek_export#(int)*	
blocking_peek_response_export	*uvm_get_peek_export#(int)*	
nonblocking_peek_response_export	*uvm_get_peek_export#(int)*	
blocking_get_peek_response_export	*uvm_get_peek_export#(int)*	
nonblocking_get_peek_response_export	*uvm_get_peek_export#(int)*	
blocking_master_export	*uvm_master_imp#(int, int, uvm_tlm_req_rsp_channel#(int, int), uvm_tlm_fifo#(int), uvm_tlm_fifo#(int))*	
nonblocking_master_export	*uvm_master_imp#(int, int, uvm_tlm_req_rsp_channel#(int, int), uvm_tlm_fifo#(int), uvm_tlm_fifo#(int))*	
blocking_slave_export	*uvm_slave_imp#(int, int, uvm_tlm_req_rsp_channel#(int, int), uvm_tlm_fifo#(int), uvm_tlm_fifo#(int))*	

continues on next page

Table 322 – continued from previous page

Name	Type	Description
nonblocking_slave_export	*uvm_slave_imp#(int, int, uvm_tlm_req_rsp_channel#(int, int), uvm_tlm_fifo#(int), uvm_tlm_fifo#(int))*	

Table 323: Typedefs

Name	Actual Type	Description
this_type	*uvm_tlm_req_rsp_channel#(REQ, RSP)*	

Constructors

`function new(string name, uvm_component parent = null, int request_fifo_size = 1, int response_fifo_size = 1)`

Function

new

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.

Parameters

- **name** (*string*)
- **parent** (*uvm_component*)
- **request_fifo_size** (*int*)
- **response_fifo_size** (*int*)

Functions

`virtual function void connect_phase(uvm_phase phase)`

Parameters

- **phase** (*uvm_phase*)

`function void create_aliased_exports()`

`virtual function string get_type_name()`

get_type_name

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

create

Parameters

- **name** (*string*)

Return type

uvm_object

15.2.0.74 Class uvm_pkg::uvm_tlm_time

CLASS

uvm_tlm_time

Canonical time type that can be used in different timescales

This time type is used to represent time values in a canonical form that can bridge initiators and targets located in different timescales and time precisions.

For a detailed explanation of the purpose for this class, see <Why is this necessary>.

Constructors

`function new(string name = "uvm_tlm_time", real res = 0)`

Function

new

Create a new canonical time value.

The new value is initialized to 0. If a resolution is not specified, the default resolution, as specified by *set_time_resolution()*, is used.

Parameters

`name` (*`string`*)

`res` (*`real`*)

Functions

`static function void set_time_resolution(real res)`

Function

set_time_resolution

Set the default canonical time resolution.

Must be a power of 10. When co-simulating with SystemC, it is recommended that default canonical time resolution be set to the SystemC time resolution.

By default, the default resolution is 1.0e-12 (ps)

Parameters

`res` (*`real`*)

`function string get_name()`

Function

get_name

Return the name of this instance

`function void reset()`

Function

reset

Reset the value to 0

`function real get_realtime(time scaled, real secs = 1.0e-9)`

Function

get_realtime

Return the current canonical time value, scaled for the caller's timescale

scaled must be a time literal value that corresponds to the number of seconds specified in *secs* (1ns by default). It must be a time literal value that is greater or equal to the current timescale.

```
#(delay.get_realtime(1ns));
#(delay.get_realtime(1fs, 1.0e-15));
```

Parameters

scaled (*time*)

secs (*real*)

function void incr(real t, time scaled, real secs = 1.0e-9)

Function

incr

Increment the time value by the specified number of scaled time unit

t is a time value expressed in the scale and precision of the caller. *scaled* must be a time literal value that corresponds to the number of seconds specified in *secs* (1ns by default). It must be a time literal value that is greater or equal to the current timescale.

```
delay.incr(1.5ns, 1ns);
delay.incr(1.5ns, 1ps, 1.0e-12);
```

Parameters

t (*real*)

scaled (*time*)

secs (*real*)

function void decr(real t, time scaled, real secs)

Function

decr

Decrement the time value by the specified number of scaled time unit

t is a time value expressed in the scale and precision of the caller. *scaled* must be a time literal value that corresponds to the number of seconds specified in *secs* (1ns by default). It must be a time literal value that is greater or equal to the current timescale.

```
delay.decr(200ps, 1ns);
```

Parameters

t (*real*)

scaled (*time*)

secs (*real*)

function real get_abstime(real secs)

Function

get_abstime

Return the current canonical time value, in the number of specified time unit, regardless of the current timescale of the caller.

secs is the number of seconds in the desired time unit e.g. 1e-9 for nanoseconds.

```
$write("%.3f ps\n", delay.get_abstime(1e-12));
```

Parameters

secs (*real*)

function void set_abstime(real t, real secs)

Function

set_abstime

Set the current canonical time value, to the number of specified time unit, regardless of the current timescale of the caller.

secs is the number of seconds in the time unit in the value *t* e.g. 1e-9 for nanoseconds.

```
delay.set_abstime(1.5, 1e-12));
```

Parameters

- **t** (*real*)
- **secs** (*real*)

15.2.0.75 Class uvm_pkg::uvm_tlm_transport_channel

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_report_object*
↪*uvm_pkg* :: *uvm_component*
↪*uvm_pkg* :: *uvm_tlm_req_rsp_channel*
↪*uvm_pkg* :: *uvm_tlm_transport_channel*

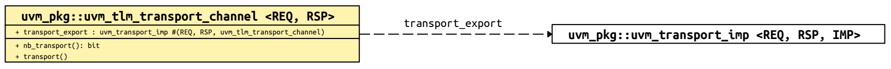

uvm_pkg::uvm_tlm_transport_channel <REQ, RSP>
+ transport_export : uvm_transport_imp #(REQ, RSP, uvm_tlm_transport_channel)
+ nb_transport(): bit
+ transport()
transport_export
uvm_pkg::uvm_transport_imp <REQ, RSP, IMP>

Fig. 119: Collaboration Diagram of uvm_tlm_transport_channel

CLASS

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.

Table 324: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Table 325: Variables

Name	Type	Description

transport_export	*uvm_transport_imp#(int, int, uvm_tlm_transport_channel#(int, int))*	***Port***
		 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.

Table 326: Typedefs

Name	Actual Type	Description
this_type	*uvm_tlm_transport_channel#(REQ, RSP)*	

Constructors

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

Function

new

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.

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)

Functions

`function bit nb_transport(int req, int rsp)`

Parameters
- **`req`** (*`int`*)
- **`rsp`** (*`int`*)

Tasks

`function transport(int request, int response)`

Parameters
- **`request`** (*`int`*)
- **`response`** (*`int`*)

15.2.0.76 Class uvm_pkg::uvm_top_down_visitor_adapter

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_visitor_adapter*
↪*uvm_pkg* :: *uvm_top_down_visitor_adapter*

CLASS

uvm_top_down_visitor_adapter

This uvm_top_down_visitor_adapter traverses the STRUCTURE *s* (and will invoke the visitor) in a hierarchical fashion. During traversal *s* will be visited before all subnodes of *s* will be visited.

Table 327: Parameters

Name	Default value	Description
STRUCTURE	uvm_component	
VISITOR	uvm_visitor	

Constructors

`function new(string name = "")`

Parameters
name (*string*)

Functions

`virtual function void accept(uvm_component s, uvm_visitor#(uvm_component) v, uvm_-structure_proxy#(uvm_component) p, bit invoke_begin_end = 1)`

Parameters
s (*uvm_component*)
v (*uvm_visitor#(uvm_component)*)
p (*uvm_structure_proxy#(uvm_component)*)
invoke_begin_end (*bit*)

15.2.0.77 Class uvm_pkg::uvm_topdown_phase

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_phase*
↪*uvm_pkg* :: *uvm_topdown_phase*

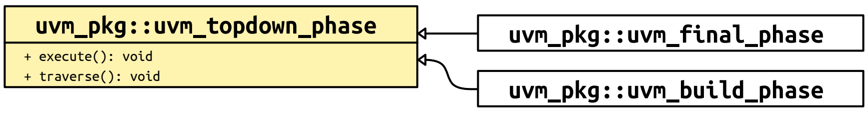


Fig. 120: Inheritance Diagram of uvm_topdown_phase

Class

uvm_topdown_phase

Virtual base class for function phases that operate top-down. The pure virtual function execute() is called for each component.

A top-down function phase completes when the *execute()* method has been called and returned on all applicable components in the hierarchy.

Constructors

`function new(string name)`

Function

new

Create a new instance of a top-down phase
Parameters
`name` (`string`)

Functions

`virtual function void traverse(uvm_component comp, uvm_phase phase, uvm_phase_-state state)`

Function

traverse

Traverses the component tree in top-down order, calling *execute* for each component.
Parameters
`comp` (*uvm_component*)
`phase` (*uvm_phase*)
`state` (*uvm_phase_state*)

`virtual function void execute(uvm_component comp, uvm_phase phase)`

Function

execute

Executes the top-down phase *phase* for the component *comp* .
Parameters
`comp` (*uvm_component*)
`phase` (*uvm_phase*)

15.2.0.78 Class uvm_pkg::uvm_transport_export

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_transport_export*

Table 328: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters

name (*string*)
parent (*uvm_component*)
min_size (*int*)
max_size (*int*)

15.2.0.79 Class uvm_pkg::uvm_transport_imp

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_transport_imp*

Table 329: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	
IMP	int	

Constructors

`function new(string name, int imp)`

Parameters
- **`name`** (*`string`*)
- **`imp`** (*`int`*)

15.2.0.80 Class uvm_pkg::uvm_transport_port

uvm_pkg :: *uvm_tlm_if_base*
↪*uvm_pkg* :: *uvm_port_base*
↪*uvm_pkg* :: *uvm_transport_port*

Table 330: Parameters

Name	Default value	Description
REQ	int	
RSP	REQ	

Constructors

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

Parameters
- **`name`** (*`string`*)
- **`parent`** (*uvm_component*)
- **`min_size`** (*`int`*)
- **`max_size`** (*`int`*)

15.2.0.81 Class uvm_pkg::uvm_tree_printer

uvm_pkg :: *uvm_printer*
↪*uvm_pkg* :: *uvm_tree_printer*

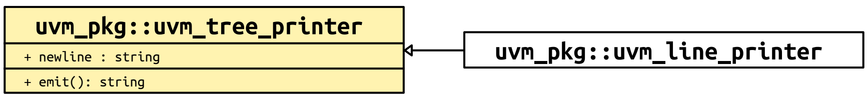


Fig. 121: Inheritance Diagram of uvm_tree_printer

Class

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
}
```

Table 331: Variables

Name	Type	Description
newline	string	

Constructors

`function new()`

Variable

new

Creates a new instance of *uvm_tree_printer* . New

Functions

`virtual function string emit()`

Function

emit

Formats the collected information from prior calls to *print_** into hierarchical tree format. Emit

15.2.0.82 Class uvm_pkg::uvm_typed_callbacks

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_callbacks_base*
↪*uvm_pkg* :: *uvm_typed_callbacks*

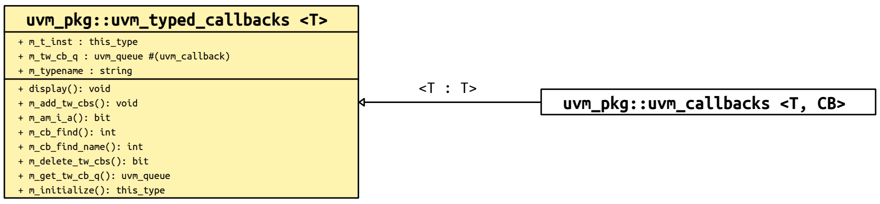


Fig. 122: Inheritance Diagram of uvm_typed_callbacks

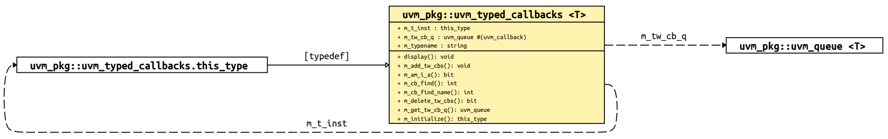


Fig. 123: Collaboration Diagram of uvm_typed_callbacks

Class

uvm_typed_callbacks(T)

Another internal class. This contains the queue of typewide callbacks. It also contains some of the public interface methods, but those methods are accessed via the uvm_callbacks() class so they are documented in that class even though the implementation is in this class.

The <add>, <delete>, and *display* methods are implemented in this class.

Table 332: Parameters

Name	Default value	Description
T	uvm_object	

Table 333: Typedefs

Name	Actual Type	Description
this_type	*uvm_typed_callbacks#(T)*	
super_type	*uvm_callbacks_base*	

Functions

`static function void display(uvm_object obj = null)`

Parameters
- **`obj`** (*uvm_object*)

15.2.0.83 Class uvm_pkg::uvm_typeid

uvm_pkg :: *uvm_typeid_base*
↪*uvm_pkg* :: *uvm_typeid*

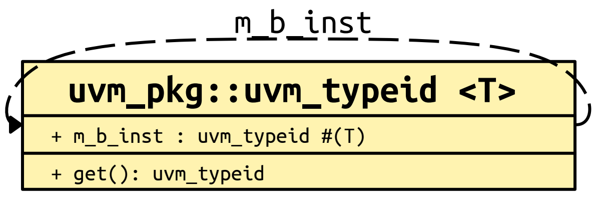


Fig. 124: Collaboration Diagram of uvm_typeid

Class

uvm_typeid(T)

Table 334: Parameters

Name	Default value	Description
T	uvm_object	

Functions

```
static function uvm_typeid get()
```

Return type
uvm_typeid

15.2.0.84 Class uvm_pkg::uvm_typeid_base

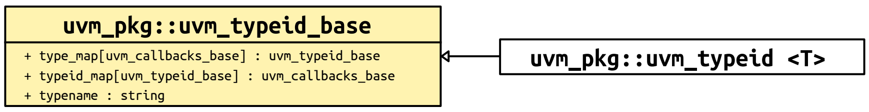

uvm_pkg::uvm_typeid_base
+ type_map[uvm_callbacks_base] : uvm_typeid_base
+ typeid_map[uvm_typeid_base] : uvm_callbacks_base
+ typename : string
uvm_pkg::uvm_typeid <T>

Fig. 125: Inheritance Diagram of uvm_typeid_base

Class

uvm_typeid_base

Simple typeid interface. Need this to set up the base-super mapping. This is similar to the factory, but much simpler. The idea of this interface is that each object type T has a typeid that can be used for mapping type relationships. This is not a user visible class.

Table 335: Variables

Name	Type	Description
typename	string	
typeid_map	*uvm_callbacks_base*	
type_map	*uvm_typeid_base*	

15.2.0.85 Class uvm_pkg::uvm_utils

CLASS

uvm_utils (TYPE, FIELD)

This class contains useful template functions.

Table 336: Parameters

Name	Default value	Description
TYPE	int	
FIELD	"config"	

Table 337: Typedefs

Name	Actual Type	Description
types_t	*TYPE*	

Functions

`static function types_t find_all(uvm_component start)`

Function

find_all

Recursively finds all component instances of the parameter type *TYPE* , starting with the component given by *start* . Uses *uvm_root::find_all*.

Parameters

`start` (*uvm_component*)

Return type

types_t

`static function TYPE find(uvm_component start)`

Parameters

`start` (*uvm_component*)

`static function TYPE create_type_by_name(string type_name, string contxt)`

Parameters

`type_name` (`string`)

`contxt` (`string`)

`static function TYPE get_config(uvm_component comp, bit is_fatal)`

Function

get_config

This method gets the object config of type *TYPE* associated with component *comp* . We check for the two kinds of error which may occur with this kind of operation.

Parameters

`comp` (*uvm_component*)

`is_fatal` (`bit`)

15.2.0.86 Class uvm_pkg::uvm_visitor

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_visitor*

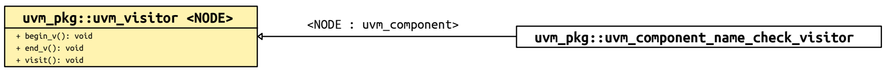


Fig. 126: Inheritance Diagram of uvm_visitor

CLASS

uvm_visitor (NODE)

The uvm_visitor class provides an abstract base class for a visitor. The visitor visits instances of type NODE. For general information regarding the visitor pattern see http://en.wikipedia.org/wiki/Visitor_pattern

Table 338: Parameters

Name	Default value	Description
NODE	uvm_component	

Constructors

`function new(string name = "")`

Parameters
name (*string*)

Functions

`virtual function void begin_v()`

Function

begin_v

This method will be invoked by the visitor before the first NODE is visited

`virtual function void end_v()`

Function

end_v

This method will be invoked by the visitor after the last NODE is visited

`virtual function void visit(uvm_component node)`

Function

visit

This method will be invoked by the visitor for every visited *node* of the provided structure. The user is expected to provide the own functionality in this function.

```
class count_nodes_visitor#(type T=uvm_component) extends uvm_visitor#(T);
        function new (string name = "");
                super.new(name);
    endfunction
        local int cnt;
```

(continues on next page)

(continued from previous page)

```
   virtual function void begin_v(); cnt = 0; endfunction
      virtual function void end_v(); `uvm_info("TEXT",$sformatf("%d elements",
↪cnt),UVM_NONE) endfunction
      virtual function void visit(T node); cnt++; endfunction
      endclass
```

Parameters
  node (*uvm_component*)

15.2.0.87 Class uvm_pkg::uvm_visitor_adapter

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_visitor_adapter*

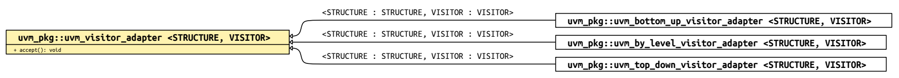

uvm_pkg::uvm_visitor_adapter <STRUCTURE, VISITOR>
+ accept(): void
<STRUCTURE : STRUCTURE, VISITOR : VISITOR>
<STRUCTURE : STRUCTURE, VISITOR : VISITOR>
<STRUCTURE : STRUCTURE, VISITOR : VISITOR>
uvm_pkg::uvm_bottom_up_visitor_adapter <STRUCTURE, VISITOR>
uvm_pkg::uvm_by_level_visitor_adapter <STRUCTURE, VISITOR>
uvm_pkg::uvm_top_down_visitor_adapter <STRUCTURE, VISITOR>

Fig. 127: Inheritance Diagram of uvm_visitor_adapter

CLASS

uvm_visitor_adapter (STRUCTURE, uvm_visitor(STRUCTURE))

The visitor adaptor traverses all nodes of the STRUCTURE and will invoke visitor.visit() on every node.

Table 339: Parameters

Name	Default value	Description
STRUCTURE	uvm_component	
VISITOR	uvm_visitor	

Constructors

`function new(string name = "")`

Parameters
- **`name`** (*`string`*)

Functions

`virtual function void accept(uvm_component s, uvm_visitor#(uvm_component) v, uvm_structure_proxy#(uvm_component) p, bit invoke_begin_end = 1)`

Function

accept()

Calling this function will traverse through *s* (and every subnode of *s*). For each node found *v* .visit(node) will be invoked. The children of *s* are recursively determined by invoking *p* .get_immediate_children(). *invoke_begin_end* determines whether the visitors begin/end functions should be invoked prior to traversal.

Parameters
- **`s`** (*uvm_component*)
- **`v`** (*uvm_visitor#(uvm_component)*)
- **`p`** (*uvm_structure_proxy#(uvm_component)*)
- **`invoke_begin_end`** (*`bit`*)

15.2.0.88 Class uvm_pkg::uvm_vreg

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_vreg*

Class

uvm_vreg

Virtual register abstraction base class

A virtual register represents a set of fields that are logically implemented in consecutive memory locations.

All virtual register accesses eventually turn into memory accesses.

A virtual register array may be implemented on top of any memory abstraction class and possibly dynamically resized and/or relocated.

Constructors

`function new(string name, int unsigned n_bits)`

FUNCTION

new

Create a new instance and type-specific configuration

Creates an instance of a virtual register abstraction class with the specified name.

n_bits specifies the total number of bits in a virtual register. Not all bits need to be mapped to a virtual field. This value is usually a multiple of 8. IMPLEMENTATION

Parameters

`name` (*`string`*)
`n_bits` (*`int unsigned`*)

Functions

`function void configure(uvm_reg_block parent, uvm_mem mem = null, longint unsigned size = 0, uvm_reg_addr_t offset = 0, int unsigned incr = 0)`

Function

configure

Instance-specific configuration

Specify the *parent* block of this virtual register array. If one of the other parameters are specified, the virtual register is assumed to be dynamic and can be later (re-)implemented using the *uvm_vreg::implement()* method.

If *mem* is specified, then the virtual register array is assumed to be statically implemented in the memory corresponding to the specified memory abstraction class and *size* , *offset* and *incr* must also be specified. Static virtual register arrays cannot be re-implemented.

Parameters

`parent` (*uvm_reg_block*)
`mem` (*uvm_mem*)
`size` (*`longint unsigned`*)
`offset` (*uvm_reg_addr_t*)
`incr` (*`int unsigned`*)

`virtual function bit implement(longint unsigned n, uvm_mem mem = null, uvm_reg_addr_t offset = 0, int unsigned incr = 0)`

FUNCTION

implement

Dynamically implement, resize or relocate a virtual register array

Implement an array of virtual registers of the specified *size* , in the specified memory and *offset* . If an offset increment is specified, each virtual register is implemented at the specified offset increment from the previous one. If an offset increment of 0 is specified, virtual registers are packed as closely as possible in the memory.

If no memory is specified, the virtual register array is in the same memory, at the same base offset using the same offset increment as originally implemented. Only the number of virtual registers in the virtual register array is modified.

The initial value of the newly-implemented or relocated set of virtual registers is whatever values are currently stored in the memory now implementing them.

Returns TRUE if the memory can implement the number of virtual registers at the specified base offset and offset increment. Returns FALSE otherwise.

The memory region used to implement a virtual register array is reserved in the memory allocation manager associated with the memory to prevent it from being allocated for another purpose.

Parameters

- **`n`** (`longint unsigned`)
- **`mem`** (*uvm_mem*)
- **`offset`** (*uvm_reg_addr_t*)
- **`incr`** (`int unsigned`)

`virtual function uvm_mem_region allocate(longint unsigned n, uvm_mem_mam mam, uvm_mem_mam_policy alloc = null)`

FUNCTION

allocate

Randomly implement, resize or relocate a virtual register array

Implement a virtual register array of the specified size in a randomly allocated region of the appropriate size in the address space managed by the specified memory allocation manager. If a memory allocation policy is specified, it is passed to the uvm_mem_mam::request_region() method.

The initial value of the newly-implemented or relocated set of virtual registers is whatever values are currently stored in the memory region now implementing them.

Returns a reference to a *uvm_mem_region* memory region descriptor if the memory allocation manager was able to allocate a region that can implement the virtual register array with the specified allocation policy. Returns *null* otherwise.

A region implementing a virtual register array must not be released using the *uvm_mem_mam::release_region()* method. It must be released using the *uvm_vreg::release_region()* method.

Parameters

- **`n`** (`longint unsigned`)
- **`mam`** (*uvm_mem_mam*)
- **`alloc`** (*uvm_mem_mam_policy*)

Return type

uvm_mem_region

`virtual function uvm_mem_region get_region()`

FUNCTION

get_region

Get the region where the virtual register array is implemented

Returns a reference to the *uvm_mem_region* memory region descriptor that implements the virtual register array.

Returns *null* if the virtual registers array is not currently implemented. A region implementing a virtual register array must not be released using the *uvm_mem_mam::release_region()* method. It must be released using the *uvm_vreg::release_region()* method.

Return type

uvm_mem_region

`virtual function void release_region()`

FUNCTION

release_region

Dynamically un-implement a virtual register array

Release the memory region used to implement a virtual register array and return it to the pool of available memory that can be allocated by the memory's default allocation manager. The virtual register array is subsequently considered as unimplemented and can no longer be accessed.

Statically-implemented virtual registers cannot be released.

`virtual function void set_parent(uvm_reg_block parent)`

Parameters

`parent` (*uvm_reg_block*) -- Local

`function void Xlock_modelX()`

Local

`function void add_field(uvm_vreg_field field)`

Parameters

`field` (*uvm_vreg_field*) -- Local

`virtual function string get_full_name()`

Function

get_full_name

Get the hierarchical name

Return the hierarchal name of this register. The base of the hierarchical name is the root block.

`virtual function uvm_reg_block get_parent()`

FUNCTION

get_parent

Get the parent block

Return type

uvm_reg_block

`virtual function uvm_reg_block get_block()`

Return type

uvm_reg_block

`virtual function uvm_mem get_memory()`

FUNCTION

get_memory

Get the memory where the virtual register array is implemented

Return type

uvm_mem

`virtual function int get_n_maps()`

Function

get_n_maps

Returns the number of address maps this virtual register array is mapped in

`function bit is_in_map(uvm_reg_map map)`

Function

is_in_map

Return TRUE if this virtual register array is in the specified address *map*

Parameters

`map` (*uvm_reg_map*)

`virtual function void get_maps(uvm_reg_map maps)`

Function

get_maps

Returns all of the address *maps* where this virtual register array is mapped

Parameters

`maps` (*uvm_reg_map*)

`virtual function string get_rights(uvm_reg_map map = null)`

FUNCTION

get_rights

Returns the access rights of this virtual register array

Returns "RW", "RO" or "WO". The access rights of a virtual register array is always "RW", unless it is implemented in a shared memory with access restriction in a particular address map.

If no address map is specified and the memory is mapped in only one address map, that address map is used. If the memory is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the memory is not mapped in the specified address map, an error message is issued and "RW" is returned.

Parameters

`map` (*uvm_reg_map*)

`virtual function string get_access(uvm_reg_map map = null)`

FUNCTION

get_access

Returns the access policy of the virtual register array when written and read via an address map.

If the memory implementing the virtual register array is mapped in more than one address map, an address *map* must be specified. If access restrictions are present when accessing a memory through the specified address map, the access mode returned takes the access restrictions into account. For example, a read-write memory accessed through an address map with read-only restrictions would return "RO".

Parameters

`map` (*uvm_reg_map*)

`virtual function int unsigned get_size()`

FUNCTION

get_size

Returns the size of the virtual register array.

`virtual function int unsigned get_n_bytes()`

FUNCTION

get_n_bytes

Returns the width, in bytes, of a virtual register.

The width of a virtual register is always a multiple of the width of the memory locations used to implement it. For example, a virtual register containing two 1-byte fields implemented in a memory with 4-bytes memory locations is 4-byte wide.

`virtual function int unsigned get_n_memlocs()`

FUNCTION

get_n_memlocs

Returns the number of memory locations used by a single virtual register.

`virtual function int unsigned get_incr()`

FUNCTION

get_incr

Returns the number of memory locations between two individual virtual registers in the same array.

`virtual function void get_fields(uvm_vreg_field fields)`

FUNCTION

get_fields

Return the virtual fields in this virtual register

Fills the specified array with the abstraction class for all of the virtual fields contained in this virtual register. Fields are ordered from least-significant position to most-significant position within the register.

Parameters

`fields` (*uvm_vreg_field*)

`virtual function uvm_vreg_field get_field_by_name(string name)`

FUNCTION

get_field_by_name

Return the named virtual field in this virtual register

Finds a virtual field with the specified name in this virtual register and returns its abstraction class. If no fields are found, returns *null* .

Parameters

`name` (*`string`*)

Return type

uvm_vreg_field

`virtual function uvm_reg_addr_t get_offset_in_memory(longint unsigned idx)`

FUNCTION

get_offset_in_memory

Returns the offset of a virtual register

Returns the base offset of the specified virtual register, in the overall address space of the memory that implements the virtual register array.

Parameters

`idx` (*`longint unsigned`*)

Return type

uvm_reg_addr_t

**`virtual function uvm_reg_addr_t get_address(longint unsigned idx, uvm_reg_-
map map = null)`**

FUNCTION

get_address

Returns the base external physical address of a virtual register

Returns the base external physical address of the specified virtual register if accessed through the specified address *map* .

If no address map is specified and the memory implementing the virtual register array is mapped in only one address map, that address map is used. If the memory is mapped in more than one address map, the default address map of the parent block is used.

If an address map is specified and the memory is not mapped in the specified address map, an error message is issued.

Parameters

`idx` (*`longint unsigned`*)

`map` (*uvm_reg_map*)

Return type

uvm_reg_addr_t

`function void reset(string kind = "HARD")`

Function

reset

Reset the access semaphore

Reset the semaphore that prevents concurrent access to the virtual register. This semaphore must be explicitly reset if a thread accessing this virtual register array was killed in before the access was completed

Parameters
: **kind** (`string`)

`virtual function void do_print(uvm_printer printer)`

Parameters
: **printer** (*uvm_printer*)

`virtual function string convert2string()`

`virtual function uvm_object clone()`

TODO

add fatal messages

Return type
: *uvm_object*

`virtual function void do_copy(uvm_object rhs)`

Parameters
: **rhs** (*uvm_object*)

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

Parameters
: **rhs** (*uvm_object*)
: **comparer** (*uvm_comparer*)

`virtual function void do_pack(uvm_packer packer)`

Parameters
: **packer** (*uvm_packer*)

`virtual function void do_unpack(uvm_packer packer)`

Parameters
: **packer** (*uvm_packer*)

Tasks

`function XatomicX(bit on)`

Parameters
: **on** (`bit`) -- Local

`virtual function write(longint unsigned idx, uvm_status_e status, uvm_reg_data_t value, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_base parent = null, uvm_object extension = null, string fname = "", int lineno = 0)`

TASK

write

Write the specified value in a virtual register

Write *value* in the DUT memory location(s) that implements the virtual register array that corresponds to this abstraction class instance using the specified access *path* .

If the memory implementing the virtual register array is mapped in more than one address map, an address *map* must be specified if a physical access is used (front-door access).

The operation is eventually mapped into set of memory-write operations at the location where the virtual register specified by *idx* in the virtual register array is implemented.

Parameters
: **idx** (`longint unsigned`)
: **status** (*uvm_status_e*)
: **value** (*uvm_reg_data_t*)
: **path** (*uvm_path_e*)
: **map** (*uvm_reg_map*)
: **parent** (*uvm_sequence_base*)
: **extension** (*uvm_object*)
: **fname** (`string`)
: **lineno** (`int`)

virtual function read(longint unsigned idx, uvm_status_e status, uvm_reg_data_t value, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_base parent = null, uvm_object extension = null, string fname = "", int lineno = 0)

TASK

read

Read the current value from a virtual register

Read from the DUT memory location(s) that implements the virtual register array that corresponds to this abstraction class instance using the specified access *path* and return the readback *value* .

If the memory implementing the virtual register array is mapped in more than one address map, an address *map* must be specified if a physical access is used (front-door access).

The operation is eventually mapped into set of memory-read operations at the location where the virtual register specified by *idx* in the virtual register array is implemented.

Parameters

- **idx** (*longint unsigned*)
- **status** (*uvm_status_e*)
- **value** (*uvm_reg_data_t*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **parent** (*uvm_sequence_base*)
- **extension** (*uvm_object*)
- **fname** (*string*)
- **lineno** (*int*)

virtual function poke(longint unsigned idx, uvm_status_e status, uvm_reg_data_t value, uvm_sequence_base parent = null, uvm_object extension = null, string fname = "", int lineno = 0)

TASK

poke

Deposit the specified value in a virtual register

Deposit *value* in the DUT memory location(s) that implements the virtual register array that corresponds to this abstraction class instance using the memory backdoor access.

The operation is eventually mapped into set of memory-poke operations at the location where the virtual register specified by *idx* in the virtual register array is implemented.

Parameters

- **idx** (*longint unsigned*)
- **status** (*uvm_status_e*)
- **value** (*uvm_reg_data_t*)
- **parent** (*uvm_sequence_base*)
- **extension** (*uvm_object*)
- **fname** (*string*)
- **lineno** (*int*)

virtual function peek(longint unsigned idx, uvm_status_e status, uvm_reg_data_t value, uvm_sequence_base parent = null, uvm_object extension = null, string fname = "", int lineno = 0)

TASK

peek

Sample the current value in a virtual register

Sample the DUT memory location(s) that implements the virtual register array that corresponds to this abstraction class instance using the memory backdoor access, and return the sampled *value* .

The operation is eventually mapped into set of memory-peek operations at the location where the virtual register specified by *idx* in the virtual register array is implemented.

Parameters

idx (*longint unsigned*)
status (*uvm_status_e*)
value (*uvm_reg_data_t*)
parent (*uvm_sequence_base*)
extension (*uvm_object*)
fname (*string*)
lineno (*int*)

virtual function pre_write(longint unsigned idx, uvm_reg_data_t wdat, uvm_path_-e path, uvm_reg_map map)

TASK

pre_write

Called before virtual register write.

If the specified data value, access *path* or address *map* are modified, the updated data value, access path or address map will be used to perform the virtual register operation.

The registered callback methods are invoked after the invocation of this method. All register callbacks are executed after the corresponding field callbacks The pre-write virtual register and field callbacks are executed before the corresponding pre-write memory callbacks

Parameters

idx (*longint unsigned*)
wdat (*uvm_reg_data_t*)
path (*uvm_path_e*)
map (*uvm_reg_map*)

virtual function post_write(longint unsigned idx, uvm_reg_data_t wdat, uvm_path_-e path, uvm_reg_map map, uvm_status_e status)

TASK

post_write

Called after virtual register write.

If the specified *status* is modified, the updated status will be returned by the virtual register operation.

The registered callback methods are invoked before the invocation of this method. All register callbacks are executed before the corresponding field callbacks The post-write virtual register and field callbacks are executed after the corresponding post-write memory callbacks

Parameters

idx (*longint unsigned*)
wdat (*uvm_reg_data_t*)
path (*uvm_path_e*)
map (*uvm_reg_map*)
status (*uvm_status_e*)

virtual function pre_read(longint unsigned idx, uvm_path_e path, uvm_reg_map map)

TASK

pre_read

Called before virtual register read.

If the specified access *path* or address *map* are modified, the updated access path or address map will be used to perform the register operation.

The registered callback methods are invoked after the invocation of this method. All register callbacks are executed after the corresponding field callbacks The pre-read virtual register and field callbacks are executed before the corresponding pre-read memory callbacks

Parameters

idx (*longint unsigned*)
path (*uvm_path_e*)
map (*uvm_reg_map*)

`virtual function post_read(longint unsigned idx, uvm_reg_data_t rdat, uvm_path_-e path, uvm_reg_map map, uvm_status_e status)`

TASK

post_read

Called after virtual register read.

If the specified readback data or *status* is modified, the updated readback data or status will be returned by the register operation.

The registered callback methods are invoked before the invocation of this method. All register callbacks are executed before the corresponding field callbacks The post-read virtual register and field callbacks are executed after the corresponding post-read memory callbacks

Parameters

`idx` (*`longint unsigned`*)
`rdat` (*uvm_reg_data_t*)
`path` (*uvm_path_e*)
`map` (*uvm_reg_map*)
`status` (*uvm_status_e*)

15.2.0.89 Class uvm_pkg::uvm_vreg_cbs

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_callback*
↪*uvm_pkg* :: *uvm_vreg_cbs*

Class

uvm_vreg_cbs

Pre/post read/write callback facade class

Table 340: Variables

Name	Type	Description
fname	string	
lineno	int	

Constructors

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

Parameters
`name` (`string`)

Tasks

`virtual function pre_write(uvm_vreg rg, longint unsigned idx, uvm_reg_data_t wdat, uvm_path_e path, uvm_reg_map map)`

Task

pre_write

Callback called before a write operation.

The registered callback methods are invoked after the invocation of the *uvm_vreg::pre_write()* method. All virtual register callbacks are executed after the corresponding virtual field callbacks The pre-write virtual register and field callbacks are executed before the corresponding pre-write memory callbacks

The written value *wdat* , access *path* and address *map* , if modified, modifies the actual value, access path or address map used in the virtual register operation.

Parameters
`rg` (*uvm_vreg*)
`idx` (`longint unsigned`)
`wdat` (*uvm_reg_data_t*)
`path` (*uvm_path_e*)
`map` (*uvm_reg_map*)

`virtual function post_write(uvm_vreg rg, longint unsigned idx, uvm_reg_data_t wdat, uvm_path_e path, uvm_reg_map map, uvm_status_e status)`

TASK

post_write

Called after register write.

The registered callback methods are invoked before the invocation of the *uvm_reg::post_write()* method. All register callbacks are executed before the corresponding virtual field callbacks The post-write virtual register and field callbacks are executed after the corresponding post-write memory callbacks

The *status* of the operation, if modified, modifies the actual returned status.

Parameters

- **rg** (*uvm_vreg*)
- **idx** (*longint unsigned*)
- **wdat** (*uvm_reg_data_t*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **status** (*uvm_status_e*)

virtual function pre_read(uvm_vreg rg, longint unsigned idx, uvm_path_e path, uvm_reg_map map)

TASK

pre_read

Called before register read.

The registered callback methods are invoked after the invocation of the *uvm_reg::pre_read()* method. All register callbacks are executed after the corresponding virtual field callbacks The pre-read virtual register and field callbacks are executed before the corresponding pre-read memory callbacks

The access *path* and address *map* , if modified, modifies the actual access path or address map used in the register operation.

Parameters

- **rg** (*uvm_vreg*)
- **idx** (*longint unsigned*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)

virtual function post_read(uvm_vreg rg, longint unsigned idx, uvm_reg_data_t rdat, uvm_path_e path, uvm_reg_map map, uvm_status_e status)

TASK

post_read

Called after register read.

The registered callback methods are invoked before the invocation of the *uvm_reg::post_read()* method. All register callbacks are executed before the corresponding virtual field callbacks The post-read virtual register and field callbacks are executed after the corresponding post-read memory callbacks

The readback value *rdat* and the *status* of the operation, if modified, modifies the actual returned readback value and status.

Parameters

- **rg** (*uvm_vreg*)
- **idx** (*longint unsigned*)
- **rdat** (*uvm_reg_data_t*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **status** (*uvm_status_e*)

15.2.0.90 Class uvm_pkg::uvm_vreg_field

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_vreg_field*

Class

uvm_vreg_field

Virtual field abstraction class

A virtual field represents a set of adjacent bits that are logically implemented in consecutive memory locations.

Constructors

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

Function

new

Create a new virtual field instance

This method should not be used directly. The uvm_vreg_field::type_id::create() method should be used instead.

Parameters

`name` (`string`)

Functions

`function void configure(uvm_vreg parent, int unsigned size, int unsigned lsb_pos)`

Function

configure

Instance-specific configuration

Specify the *parent* virtual register of this virtual field, its *size* in bits, and the position of its least-significant bit within the virtual register relative to the least-significant bit of the virtual register.

Parameters

`parent` (*uvm_vreg*)
`size` (`int unsigned`)
`lsb_pos` (`int unsigned`)

`virtual function string get_full_name()`

Function

get_full_name

Get the hierarchical name

Return the hierarchal name of this virtual field The base of the hierarchical name is the root block.

`virtual function uvm_vreg get_parent()`

FUNCTION

get_parent

Get the parent virtual register

Return type

uvm_vreg

`virtual function uvm_vreg get_register()`

Return type

uvm_vreg

`virtual function int unsigned get_lsb_pos_in_register()`

FUNCTION

get_lsb_pos_in_register

Return the position of the virtual field / Returns the index of the least significant bit of the virtual field in the virtual register that instantiates it. An offset of 0 indicates a field that is aligned with the least-significant bit of the register.

`virtual function int unsigned get_n_bits()`

FUNCTION

get_n_bits

Returns the width, in bits, of the virtual field.

`virtual function string get_access(uvm_reg_map map = null)`

FUNCTION

get_access

Returns the access policy of the virtual field register when written and read via an address map.

If the memory implementing the virtual field is mapped in more than one address map, an address *map* must be specified. If access restrictions are present when accessing a memory through the specified address map, the access mode returned takes the access restrictions into account. For example, a read-write memory accessed through an address map with read-only restrictions would return "RO".

Parameters

`map` (*uvm_reg_map*)

`virtual function void do_print(uvm_printer printer)`

Parameters

`printer` (*uvm_printer*)

`virtual function string convert2string()`

`virtual function uvm_object clone()`

TODO

add fatal messages

Return type

uvm_object

`virtual function void do_copy(uvm_object rhs)`

Parameters

`rhs` (*uvm_object*)

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

Parameters

`rhs` (*uvm_object*)

`comparer` (*uvm_comparer*)

`virtual function void do_pack(uvm_packer packer)`

Parameters

`packer` (*uvm_packer*)

`virtual function void do_unpack(uvm_packer packer)`

Parameters

`packer` (*uvm_packer*)

Tasks

`virtual function write(longint unsigned idx, uvm_status_e status, uvm_reg_data_t value, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_base parent = null, uvm_object extension = null, string fname = "", int lineno = 0)`

TASK

write

Write the specified value in a virtual field

Write *value* in the DUT memory location(s) that implements the virtual field that corresponds to this abstraction class instance using the specified access *path* .

If the memory implementing the virtual register array containing this virtual field is mapped in more than one address map, an address *map* must be specified if a physical access is used (front-door access).

The operation is eventually mapped into memory read-modify-write operations at the location where the virtual register specified by *idx* in the virtual register array is implemented. If a backdoor is available for the memory implementing the virtual field, it will be used for the memory-read operation.

Parameters

- **idx** (*longint unsigned*)
- **status** (*uvm_status_e*)
- **value** (*uvm_reg_data_t*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **parent** (*uvm_sequence_base*)
- **extension** (*uvm_object*)
- **fname** (*string*)
- **lineno** (*int*)

virtual function read(longint unsigned idx, uvm_status_e status, uvm_reg_data_t value, uvm_path_e path = UVM_DEFAULT_PATH, uvm_reg_map map = null, uvm_sequence_base parent = null, uvm_object extension = null, string fname = "", int lineno = 0)

TASK

read

Read the current value from a virtual field

Read from the DUT memory location(s) that implements the virtual field that corresponds to this abstraction class instance using the specified access *path* , and return the readback *value* .

If the memory implementing the virtual register array containing this virtual field is mapped in more than one address map, an address *map* must be specified if a physical access is used (front-door access).

The operation is eventually mapped into memory read operations at the location(s) where the virtual register specified by *idx* in the virtual register array is implemented.

Parameters

- **idx** (*longint unsigned*)
- **status** (*uvm_status_e*)
- **value** (*uvm_reg_data_t*)
- **path** (*uvm_path_e*)
- **map** (*uvm_reg_map*)
- **parent** (*uvm_sequence_base*)
- **extension** (*uvm_object*)
- **fname** (*string*)
- **lineno** (*int*)

virtual function poke(longint unsigned idx, uvm_status_e status, uvm_reg_data_t value, uvm_sequence_base parent = null, uvm_object extension = null, string fname = "", int lineno = 0)

TASK

poke

Deposit the specified value in a virtual field

Deposit *value* in the DUT memory location(s) that implements the virtual field that corresponds to this abstraction class instance using the specified access *path* .

The operation is eventually mapped into memory peek-modify-poke operations at the location where the virtual register specified by *idx* in the virtual register array is implemented.

Parameters

- **idx** (*longint unsigned*)
- **status** (*uvm_status_e*)

value (*uvm_reg_data_t*)
parent (*uvm_sequence_base*)
extension (*uvm_object*)
fname (`string`)
lineno (`int`)

`virtual function peek(longint unsigned idx, uvm_status_e status, uvm_reg_data_t value, uvm_sequence_base parent = null, uvm_object extension = null, string fname = "", int lineno = 0)`

TASK

peek

Sample the current value from a virtual field

Sample from the DUT memory location(s) that implements the virtual field that corresponds to this abstraction class instance using the specified access *path* , and return the readback *value* .

If the memory implementing the virtual register array containing this virtual field is mapped in more than one address map, an address *map* must be specified if a physical access is used (front-door access).

The operation is eventually mapped into memory peek operations at the location(s) where the virtual register specified by *idx* in the virtual register array is implemented.

Parameters

idx (`longint unsigned`)
status (*uvm_status_e*)
value (*uvm_reg_data_t*)
parent (*uvm_sequence_base*)
extension (*uvm_object*)
fname (`string`)
lineno (`int`)

`virtual function pre_write(longint unsigned idx, uvm_reg_data_t wdat, uvm_path_e path, uvm_reg_map map)`

TASK

pre_write

Called before virtual field write.

If the specified data value, access *path* or address *map* are modified, the updated data value, access path or address map will be used to perform the virtual register operation.

The virtual field callback methods are invoked before the callback methods on the containing virtual register. The registered callback methods are invoked after the invocation of this method. The pre-write virtual register and field callbacks are executed before the corresponding pre-write memory callbacks

Parameters

idx (`longint unsigned`)
wdat (*uvm_reg_data_t*)
path (*uvm_path_e*)
map (*uvm_reg_map*)

`virtual function post_write(longint unsigned idx, uvm_reg_data_t wdat, uvm_path_e path, uvm_reg_map map, uvm_status_e status)`

TASK

post_write

Called after virtual field write

If the specified *status* is modified, the updated status will be returned by the virtual register operation.

The virtual field callback methods are invoked after the callback methods on the containing virtual register. The registered callback methods are invoked before the invocation of this method. The post-write virtual register and field callbacks are executed after the corresponding post-write memory callbacks

Parameters

idx (`longint unsigned`)

wdat (*uvm_reg_data_t*)
path (*uvm_path_e*)
map (*uvm_reg_map*)
status (*uvm_status_e*)

virtual function pre_read(longint unsigned idx, uvm_path_e path, uvm_reg_map map)

TASK

pre_read

Called before virtual field read.

If the specified access *path* or address *map* are modified, the updated access path or address map will be used to perform the virtual register operation.

The virtual field callback methods are invoked after the callback methods on the containing virtual register. The registered callback methods are invoked after the invocation of this method. The pre-read virtual register and field callbacks are executed before the corresponding pre-read memory callbacks

Parameters

idx (`longint unsigned`)
path (*uvm_path_e*)
map (*uvm_reg_map*)

virtual function post_read(longint unsigned idx, uvm_reg_data_t rdat, uvm_path_-e path, uvm_reg_map map, uvm_status_e status)

TASK

post_read

Called after virtual field read.

If the specified readback data *rdat* or *status* is modified, the updated readback data or status will be returned by the virtual register operation.

The virtual field callback methods are invoked after the callback methods on the containing virtual register. The registered callback methods are invoked before the invocation of this method. The post-read virtual register and field callbacks are executed after the corresponding post-read memory callbacks

Parameters

idx (`longint unsigned`)
rdat (*uvm_reg_data_t*)
path (*uvm_path_e*)
map (*uvm_reg_map*)
status (*uvm_status_e*)

15.2.0.91 Class uvm_pkg::uvm_vreg_field_cbs

uvm_pkg :: *uvm_void*
↪*uvm_pkg* :: *uvm_object*
↪*uvm_pkg* :: *uvm_callback*
↪*uvm_pkg* :: *uvm_vreg_field_cbs*

Class

uvm_vreg_field_cbs

Pre/post read/write callback facade class

Table 341: Variables

Name	Type	Description
fname	string	
lineno	int	

Constructors

function new(string name = "uvm_vreg_field_cbs")

Parameters
name (`string`)

Tasks

virtual function pre_write(uvm_vreg_field field, longint unsigned idx, uvm_reg_-data_t wdat, uvm_path_e path, uvm_reg_map map)

Task

pre_write

Callback called before a write operation.

The registered callback methods are invoked before the invocation of the virtual register pre-write callbacks and after the invocation of the *uvm_vreg_field::pre_write()* method.

The written value *wdat* , access *path* and address *map* , if modified, modifies the actual value, access path or address map used in the register operation.

Parameters
field (*uvm_vreg_field*)
idx (`longint unsigned`)
wdat (*uvm_reg_data_t*)
path (*uvm_path_e*)
map (*uvm_reg_map*)

virtual function post_write(uvm_vreg_field field, longint unsigned idx, uvm_reg_-data_t wdat, uvm_path_e path, uvm_reg_map map, uvm_status_e status)

TASK

post_write

Called after a write operation

The registered callback methods are invoked after the invocation of the virtual register post-write callbacks and before the invocation of the *uvm_vreg_field::post_write()* method.

The *status* of the operation, if modified, modifies the actual returned status.

Parameters

- **`field`** (*uvm_vreg_field*)
- **`idx`** (`longint unsigned`)
- **`wdat`** (*uvm_reg_data_t*)
- **`path`** (*uvm_path_e*)
- **`map`** (*uvm_reg_map*)
- **`status`** (*uvm_status_e*)

`virtual function pre_read(uvm_vreg_field field, longint unsigned idx, uvm_path_e path, uvm_reg_map map)`

TASK

pre_read

Called before a virtual field read.

The registered callback methods are invoked after the invocation of the virtual register pre-read callbacks and after the invocation of the *uvm_vreg_field::pre_read()* method.

The access *path* and address *map* , if modified, modifies the actual access path or address map used in the register operation.

Parameters

- **`field`** (*uvm_vreg_field*)
- **`idx`** (`longint unsigned`)
- **`path`** (*uvm_path_e*)
- **`map`** (*uvm_reg_map*)

`virtual function post_read(uvm_vreg_field field, longint unsigned idx, uvm_reg_data_t rdat, uvm_path_e path, uvm_reg_map map, uvm_status_e status)`

TASK

post_read

Called after a virtual field read.

The registered callback methods are invoked after the invocation of the virtual register post-read callbacks and before the invocation of the *uvm_vreg_field::post_read()* method.

The readback value *rdat* and the *status* of the operation, if modified, modifies the actual returned readback value and status.

Parameters

- **`field`** (*uvm_vreg_field*)
- **`idx`** (`longint unsigned`)
- **`rdat`** (*uvm_reg_data_t*)
- **`path`** (*uvm_path_e*)
- **`map`** (*uvm_reg_map*)
- **`status`** (*uvm_status_e*)

Table 342: Typedefs

Name	Actual Type	Description
UVM_SEQ_ARB_-TYPE	*uvm_sequencer_arb_-mode*	backward compat
uvm_action	int	
uvm_barrier_pool	*uvm_object_string_-pool#(uvm_barrier)*	
uvm_bitstream_t	logic signed[UVM_-STREAMBITS-1:0]	Type: uvm_bitstream_t The bitstream type is used as a argument type for passing integral values in such methods as <uvm_object::set_int_local>, <uvm_-config_int>, <uvm_printer::print_field>, <uvm_-recorder::record_field>, <uvm_packer::pack_field> and <uvm_packer::unpack_field>.
uvm_callbacks_objection	*uvm_objection*	Typedef - Exists for backwards compat
uvm_config_int	*uvm_config_db#(uvm_-bitstream_t)*	-- Topic: uvm_config_int Convenience type for uvm_-config_db#(uvm_bitstream_t) | typedef uvm_config_-db#(uvm_bitstream_t) uvm_config_int;
uvm_config_object	*uvm_config_db#(uvm_-object)*	-- Topic: uvm_config_object Convenience type for uvm_config_db#(uvm_object) | typedef uvm_config_-db#(uvm_object) uvm_config_object;
uvm_config_seq	*uvm_config_db#(uvm_se-quence_base)*	-- Copyright 2007-2011 Mentor Graphics Corporation Copyright 2007-2011 Cadence Design Systems, Inc. Copyright 2010-2011 Synopsys, Inc. Copy-right 2013-2014 NVIDIA Corporation All Rights Reserved Worldwide Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the Li-cense. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language gov-erning permissions and limitations under the License. --
uvm_config_string	*uvm_config_db#(string)*	-- Topic: uvm_config_string Convenience type for uvm_-config_db#(string) | typedef uvm_config_db#(string) uvm_config_string;

continues on next page

Table 342 – continued from previous page

Name	Actual Type	Description
uvm_config_wrapper	*uvm_config_db#(uvm_-object_wrapper)*	-- Topic: uvm_config_wrapper Convenience type for uvm_config_db#(uvm_object_wrapper) | typedef uvm_config_db#(uvm_object_wrapper) uvm_config_-wrapper;
uvm_default_driver_type	*uvm_driver#(uvm_se-quence_item, uvm_se-quence_item)*	
uvm_default_sequence_-type	*uvm_sequence#(uvm_-sequence_item, uvm_se-quence_item)*	
uvm_default_se-quencer_param_type	*uvm_sequencer_param_-base#(uvm_sequence_-item, uvm_sequence_item)*	
uvm_default_se-quencer_type	*uvm_sequencer#(uvm_-sequence_item, uvm_se-quence_item)*	
uvm_event_pool	*uvm_object_string_-pool#(uvm_event#(uvm_-object))*	
uvm_hdl_data_t	logic[UVM_HDL_-MAX_WIDTH-1:0]	
uvm_heartbeat_cbs_t	*uvm_callbacks#(uvm_ob-jection, uvm_heartbeat_-callback)*	
uvm_id_actions_array	*uvm_pool#(string, uvm_-action)*	
uvm_id_file_array	*uvm_pool#(string, UVM_FILE)*	
uvm_id_verbosities_ar-ray	*uvm_pool#(string, int)*	
uvm_integral_t	logic signed[63:0]	Type: uvm_integral_t The integral type is used as a argument type for passing integral values of 64 bits or less in such methods as <uvm_printer::print_-field_int>, <uvm_recorder::record_field_int>, <uvm_-packer::pack_field_int> and <uvm_packer::unpack_-field_int>.
uvm_mem_cb	*uvm_callbacks#(uvm_-mem, uvm_reg_cbs)*	Type: uvm_mem_cb Convenience callback type dec-laration for memories Use this declaration to register memory callbacks rather than the more verbose param-eterized class
uvm_mem_cb_iter	*uvm_callback_-iter#(uvm_mem, uvm_-reg_cbs)*	Type: uvm_mem_cb_iter Convenience callback itera-tor type declaration for memories Use this declaration to iterate over registered memory callbacks rather than the more verbose parameterized class
uvm_objection_cbs_t	*uvm_callbacks#(uvm_ob-jection, uvm_objection_-callback)*	

continues on next page

Table 342 – continued from previous page

Name	Actual Type	Description
uvm_pack_bitstream_t	bit signed[(4096*8)-1:0]	--- CLASS: 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). ---
uvm_phase_cb_pool	*uvm_callbacks#(uvm_phase, uvm_phase_cb)*	--- Class: uvm_phase_cb_pool --- Convenience type for the uvm_callbacks#(uvm_phase, uvm_phase_cb) class.
uvm_port_list	*uvm_port_component_base*	
uvm_reg_addr_logic_t	logic unsigned[64-1:0]	Type: uvm_reg_addr_logic_t 4-state address value with <UVM_REG_ADDR_WIDTH> bits
uvm_reg_addr_t	bit unsigned[64-1:0]	Type: uvm_reg_addr_t 2-state address value with <UVM_REG_ADDR_WIDTH> bits
uvm_reg_bd_cb	*uvm_callbacks#(uvm_reg_backdoor, uvm_reg_cbs)*	Type: uvm_reg_bd_cb Convenience callback type declaration for backdoor Use this declaration to register register backdoor callbacks rather than the more verbose parameterized class
uvm_reg_bd_cb_iter	*uvm_callback_iter#(uvm_reg_backdoor, uvm_reg_cbs)*	Type: uvm_reg_bd_cb_iter Convenience callback iterator type declaration for backdoor Use this declaration to iterate over registered register backdoor callbacks rather than the more verbose parameterized class
uvm_reg_byte_en_t	bit unsigned[((64-1)/8+1)-1:0]	Type: uvm_reg_byte_en_t 2-state byte_enable value with <UVM_REG_BYTENABLE_WIDTH> bits
uvm_reg_cb	*uvm_callbacks#(uvm_reg, uvm_reg_cbs)*	Type: uvm_reg_cb Convenience callback type declaration for registers Use this declaration to register the register callbacks rather than the more verbose parameterized class
uvm_reg_cb_iter	*uvm_callback_iter#(uvm_reg, uvm_reg_cbs)*	Type: uvm_reg_cb_iter Convenience callback iterator type declaration for registers Use this declaration to iterate over registered register callbacks rather than the more verbose parameterized class
uvm_reg_cvr_rsrc_db	*uvm_resource_db#(uvm_reg_cvr_t)*	

continues on next page

Table 342 – continued from previous page

Name	Actual Type	Description
uvm_reg_cvr_t	bit[32-1:0]	Type: uvm_reg_cvr_t Coverage model value set with <UVM_REG_CVR_WIDTH> bits. Symbolic values for individual coverage models are defined by the <uvm_coverage_model_e> type. The following bits in the set are assigned as follows 0-7 - UVM pre-defined coverage models 8-15 - Coverage models defined by EDA vendors, implemented in a register model generator. 16-23 - User-defined coverage models 24.. - Reserved
uvm_reg_data_logic_t	logic unsigned[64-1:0]	Type: uvm_reg_data_logic_t 4-state data value with <UVM_REG_DATA_WIDTH> bits
uvm_reg_data_t	bit unsigned[64-1:0]	Type: uvm_reg_data_t 2-state data value with <UVM_REG_DATA_WIDTH> bits
uvm_reg_field_cb	*uvm_callbacks#(uvm_reg_field, uvm_reg_cbs)*	Type: uvm_reg_field_cb Convenience callback type declaration for fields Use this declaration to register field callbacks rather than the more verbose parameterized class
uvm_reg_field_cb_iter	*uvm_callback_iter#(uvm_reg_field, uvm_reg_cbs)*	Type: uvm_reg_field_cb_iter Convenience callback iterator type declaration for fields Use this declaration to iterate over registered field callbacks rather than the more verbose parameterized class
uvm_report_cb	*uvm_callbacks#(uvm_report_object, uvm_report_catcher)*	
uvm_report_cb_iter	*uvm_callback_iter#(uvm_report_object, uvm_report_catcher)*	
uvm_sequence_state_enum	*uvm_sequence_state*	backward compat
uvm_sev_override_array	*uvm_pool#(uvm_severity, uvm_severity)*	
uvm_severity_type	*uvm_severity*	
uvm_table_printer_knobs	*uvm_printer_knobs*	
uvm_tlm_gp	*uvm_tlm_generic_payload*	-- Class: uvm_tlm_gp This typedef provides a short, more convenient name for the <uvm_tlm_generic_payload> type. --
uvm_tree_printer_knobs	*uvm_printer_knobs*	
uvm_virtual_sequencer	*uvm_sequencer#(uvm_sequence_item)*	

continues on next page

Table 342 – continued from previous page

Name	Actual Type	Description
uvm_vreg_cb	*uvm_callbacks#(uvm_vreg, uvm_vreg_cbs)*	Type: uvm_vreg_cb Convenience callback type declaration Use this declaration to register virtual register callbacks rather than the more verbose parameterized class
uvm_vreg_cb_iter	*uvm_callback_iter#(uvm_vreg, uvm_vreg_cbs)*	Type: uvm_vreg_cb_iter Convenience callback iterator type declaration Use this declaration to iterate over registered virtual register callbacks rather than the more verbose parameterized class
uvm_vreg_field_cb	*uvm_callbacks#(uvm_vreg_field, uvm_vreg_field_cbs)*	Type: uvm_vreg_field_cb Convenience callback type declaration Use this declaration to register virtual field callbacks rather than the more verbose parameterized class
uvm_vreg_field_cb_iter	*uvm_callback_iter#(uvm_vreg_field, uvm_vreg_field_cbs)*	Type: uvm_vreg_field_cb_iter Convenience callback iterator type declaration Use this declaration to iterate over registered virtual field callbacks rather than the more verbose parameterized class

15.2.1 Enums

uvm_access_e

Enum

uvm_access_e

Type of operation begin performed

UVM_READ

Read operation

UVM_WRITE

Write operation

Enum Items
- UVM_READ
- UVM_WRITE
- UVM_BURST_READ
- UVM_BURST_WRITE

uvm_action_type

Enum Items
- UVM_NO_ACTION = 'b0000000
- UVM_DISPLAY = 'b0000001
- UVM_LOG = 'b0000010
- UVM_COUNT = 'b0000100
- UVM_EXIT = 'b0001000
- UVM_CALL_HOOK = 'b0010000
- UVM_STOP = 'b0100000
- UVM_RM_RECORD = 'b1000000

uvm_active_passive_enum

Enum

uvm_active_passive_enum

Convenience value to define whether a component, usually an agent, is in "active" mode or "passive" mode.

UVM_PASSIVE

"Passive" mode

UVM_ACTIVE

"Active" mode
Enum Items
UVM_PASSIVE = 0
UVM_ACTIVE = 1

`uvm_apprepend`

Append/prepend symbolic values for order-dependent APIs
Enum Items
UVM_APPEND
UVM_PREPEND

`uvm_check_e`

Enum

uvm_check_e

Read-only or read-and-check

UVM_NO_CHECK

Read only

UVM_CHECK

Read and check
Enum Items
UVM_NO_CHECK
UVM_CHECK

`uvm_coverage_model_e`

Enum

uvm_coverage_model_e

Coverage models available or desired. Multiple models may be specified by bitwise OR'ing individual model identifiers.

UVM_NO_COVERAGE

None

UVM_CVR_REG_BITS

Individual register bits

UVM_CVR_ADDR_MAP

Individual register and memory addresses

UVM_CVR_FIELD_VALS

Field values

UVM_CVR_ALL

All coverage models
Enum Items
UVM_NO_COVERAGE = 'h0000
UVM_CVR_REG_BITS = 'h0001
UVM_CVR_ADDR_MAP = 'h0002
UVM_CVR_FIELD_VALS = 'h0004
UVM_CVR_ALL = -1

`uvm_elem_kind_e`

Enum

uvm_elem_kind_e

Type of element being read or written

UVM_REG

Register

UVM_FIELD

Field

UVM_MEM

Memory location

Enum Items

UVM_REG
UVM_FIELD
UVM_MEM

`uvm_endianness_e`

Enum

uvm_endianness_e

Specifies byte ordering

UVM_NO_ENDIAN

Byte ordering not applicable

UVM_LITTLE_ENDIAN

Least-significant bytes first in consecutive addresses

UVM_BIG_ENDIAN

Most-significant bytes first in consecutive addresses

UVM_LITTLE_FIFO

Least-significant bytes first at the same address

UVM_BIG_FIFO

Most-significant bytes first at the same address

Enum Items

UVM_NO_ENDIAN
UVM_LITTLE_ENDIAN
UVM_BIG_ENDIAN
UVM_LITTLE_FIFO
UVM_BIG_FIFO

`uvm_heartbeat_modes`

Enum Items

UVM_ALL_ACTIVE
UVM_ONE_ACTIVE
UVM_ANY_ACTIVE
UVM_NO_HB_MODE

`uvm_hier_e`

Enum

uvm_hier_e

Whether to provide the requested information from a hierarchical context.

UVM_NO_HIER

Provide info from the local context

UVM_HIER

Provide info based on the hierarchical context

Enum Items

UVM_NO_HIER
UVM_HIER

uvm_objection_event

Enum

uvm_objection_event

Enumerated the possible objection events one could wait on. See *uvm_objection::wait_for*.

UVM_RAISED

an objection was raised

UVM_DROPPED

an objection was raised

UVM_ALL_DROPPED

all objections have been dropped

Enum Items

- UVM_RAISED = 'h01
- UVM_DROPPED = 'h02
- UVM_ALL_DROPPED = 'h04

uvm_path_e

Enum

uvm_path_e

Path used for register operation

UVM_FRONTDOOR

Use the front door

UVM_BACKDOOR

Use the back door

UVM_PREDICT

Operation derived from observations by a bus monitor via the *uvm_reg_predictor* class.

UVM_DEFAULT_PATH

Operation specified by the context

Enum Items

- UVM_FRONTDOOR
- UVM_BACKDOOR
- UVM_PREDICT
- UVM_DEFAULT_PATH

uvm_phase_state

Enum

uvm_phase_state

The set of possible states of a phase. This is an attribute of a schedule node in the graph, not of a phase, to maintain independent per-domain state

UVM_PHASE_UNINITIALIZED

The state is uninitialized. This is the default state for phases, and for nodes which have not yet been added to a schedule.

UVM_PHASE_DORMANT

The schedule is not currently operating on the phase node, however it will be scheduled at some point in the future.

UVM_PHASE_SCHEDULED

At least one immediate predecessor has completed. Scheduled phases block until all predecessors complete or until a jump is executed.

UVM_PHASE_SYNCING

All predecessors complete, checking that all synced phases (e.g. across domains) are at or beyond this point

UVM_PHASE_STARTED

phase ready to execute, running phase_started() callback

UVM_PHASE_EXECUTING

An executing phase is one where the phase callbacks are being executed. Its process is tracked by the phaser.

UVM_PHASE_READY_TO_END

no objections remain in this phase or in any predecessors of its successors or in any sync'd phases. This state indicates an opportunity for any phase that needs extra time for a clean exit to raise an objection, thereby causing a return to UVM_PHASE_EXECUTING. If no objection is raised, state will transition to UVM_PHASE_ENDED after a delta cycle. (An example of predecessors of successors: The successor to phase 'run' is 'extract', whose predecessors are 'run' and 'post_shutdown'. Therefore, 'run' will go to this state when both its objections and those of 'post_shutdown' are all dropped.

UVM_PHASE_ENDED

phase completed execution, now running phase_ended() callback

UVM_PHASE_JUMPING

all processes related to phase are being killed and all predecessors are forced into the DONE state.

UVM_PHASE_CLEANUP

all processes related to phase are being killed

UVM_PHASE_DONE

A phase is done after it terminated execution. Becoming done may enable a waiting successor phase to execute.

The state transitions occur as follows:

```
UNINITIALIZED -> DORMANT -> SCHED -> SYNC -> START -> EXEC -> READY -> END -+->
↪ CLEAN -> DONE
                    ^                                                       |
                    |                      <-- jump_to                      |
                    +-------------------------------------------- JUMPING< -+
```

Enum Items

UVM_PHASE_UNINITIALIZED = 0
UVM_PHASE_DORMANT = 1
UVM_PHASE_SCHEDULED = 2
UVM_PHASE_SYNCING = 4
UVM_PHASE_STARTED = 8
UVM_PHASE_EXECUTING = 16
UVM_PHASE_READY_TO_END = 32
UVM_PHASE_ENDED = 64
UVM_PHASE_CLEANUP = 128
UVM_PHASE_DONE = 256
UVM_PHASE_JUMPING = 512

`uvm_phase_type`

Enum

uvm_phase_type

This is an attribute of a *uvm_phase* object which defines the phase type.

UVM_PHASE_IMP

The phase object is used to traverse the component hierarchy and call the component phase method as well as the *phase_started* and *phase_ended* callbacks. These nodes are created by the phase macros, &96;uvm_builtin_task_phase, &96;uvm_builtin_topdown_phase, and &96;uvm_builtin_bottomup_phase. These nodes represent the phase type, i.e. uvm_run_phase, uvm_main_phase.

UVM_PHASE_NODE

The object represents a simple node instance in the graph. These nodes will contain a reference to their corresponding IMP object.

UVM_PHASE_SCHEDULE

The object represents a portion of the phasing graph, typically consisting of several NODE types, in series, parallel, or both.

UVM_PHASE_TERMINAL

This internal object serves as the termination NODE for a SCHEDULE phase object.

UVM_PHASE_DOMAIN

This object represents an entire graph segment that executes in parallel with the 'run' phase. Domains may define any network of NODEs and SCHEDULEs. The built-in domain, *uvm* , consists of a single schedule of all the run-time phases, starting with *pre_reset* and ending with *post_shutdown* .

Enum Items

- UVM_PHASE_IMP
- UVM_PHASE_NODE
- UVM_PHASE_TERMINAL
- UVM_PHASE_SCHEDULE
- UVM_PHASE_DOMAIN
- UVM_PHASE_GLOBAL

`uvm_port_type_e`

Enum

uvm_port_type_e

Specifies the type of port

UVM_PORT

The port requires the interface that is its type parameter.

UVM_EXPORT

The port provides the interface that is its type parameter via a connection to some other export or implementation.

UVM_IMPLEMENTATION

The port provides the interface that is its type parameter, and it is bound to the component that implements the interface.

Enum Items

- UVM_PORT
- UVM_EXPORT
- UVM_IMPLEMENTATION

`uvm_predict_e`

Enum

uvm_predict_e

How the mirror is to be updated

UVM_PREDICT_DIRECT

Predicted value is as-is

UVM_PREDICT_READ

Predict based on the specified value having been read

UVM_PREDICT_WRITE

Predict based on the specified value having been written

Enum Items

- UVM_PREDICT_DIRECT
- UVM_PREDICT_READ
- UVM_PREDICT_WRITE

uvm_radix_enum

Enum

uvm_radix_enum

Specifies the radix to print or record in.

UVM_BIN

Selects binary (%b) format

UVM_DEC

Selects decimal (%d) format

UVM_UNSIGNED

Selects unsigned decimal (%u) format

UVM_UNFORMAT2

Selects unformatted 2 value data (%u) format

UVM_UNFORMAT4

Selects unformatted 4 value data (%z) format

UVM_OCT

Selects octal (%o) format

UVM_HEX

Selects hexadecimal (%h) format

UVM_STRING

Selects string (%s) format

UVM_TIME

Selects time (%t) format

UVM_ENUM

Selects enumeration value (name) format

UVM_REAL

Selects real (%g) in exponential or decimal format, whichever format results in the shorter printed output

UVM_REAL_DEC

Selects real (%f) in decimal format

UVM_REAL_EXP

Selects real (%e) in exponential format

Enum Items

- UVM_BIN = 'h1000000
- UVM_DEC = 'h2000000
- UVM_UNSIGNED = 'h3000000
- UVM_UNFORMAT2 = 'h4000000
- UVM_UNFORMAT4 = 'h5000000
- UVM_OCT = 'h6000000
- UVM_HEX = 'h7000000
- UVM_STRING = 'h8000000
- UVM_TIME = 'h9000000
- UVM_ENUM = 'ha000000
- UVM_REAL = 'hb000000
- UVM_REAL_DEC = 'hc000000
- UVM_REAL_EXP = 'hd000000
- UVM_NORADIX = 0

`uvm_recursion_policy_enum`

Enum

uvm_recursion_policy_enum

Specifies the policy for copying objects.

UVM_DEEP

Objects are deep copied (object must implement *uvm_object::copy* method)

UVM_SHALLOW

Objects are shallow copied using default SV copy.

UVM_REFERENCE

Only object handles are copied.

Enum Items

UVM_DEFAULT_POLICY = 0
UVM_DEEP = 'h400
UVM_SHALLOW = 'h800
UVM_REFERENCE = 'h1000

`uvm_reg_mem_tests_e`

Enum

uvm_reg_mem_tests_e

Select which pre-defined test sequence to execute.

Multiple test sequences may be selected by bitwise OR'ing their respective symbolic values.

UVM_DO_REG_HW_RESET

Run *uvm_reg_hw_reset_seq*

UVM_DO_REG_BIT_BASH

Run *uvm_reg_bit_bash_seq*

UVM_DO_REG_ACCESS

Run *uvm_reg_access_seq*

UVM_DO_MEM_ACCESS

Run *uvm_mem_access_seq*

UVM_DO_SHARED_ACCESS

Run *uvm_reg_mem_shared_access_seq*

UVM_DO_MEM_WALK

Run *uvm_mem_walk_seq*

UVM_DO_ALL_REG_MEM_TESTS

Run all of the above

Test sequences, when selected, are executed in the order in which they are specified above.

Enum Items

UVM_DO_REG_HW_RESET = 64'h0000_0000_0000_0001
UVM_DO_REG_BIT_BASH = 64'h0000_0000_0000_0002
UVM_DO_REG_ACCESS = 64'h0000_0000_0000_0004
UVM_DO_MEM_ACCESS = 64'h0000_0000_0000_0008
UVM_DO_SHARED_ACCESS = 64'h0000_0000_0000_0010
UVM_DO_MEM_WALK = 64'h0000_0000_0000_0020
UVM_DO_ALL_REG_MEM_TESTS = 64'hffff_ffff_ffff_ffff

`uvm_sequence_lib_mode`

Enum

uvm_sequence_lib_mode

Specifies the random selection mode of a sequence library

UVM_SEQ_LIB_RAND

Random sequence selection

UVM_SEQ_LIB_RANDC

Random cyclic sequence selection

UVM_SEQ_LIB_ITEM

Emit only items, no sequence execution

UVM_SEQ_LIB_USER

Apply a user-defined random-selection algorithm

Enum Items
- UVM_SEQ_LIB_RAND
- UVM_SEQ_LIB_RANDC
- UVM_SEQ_LIB_ITEM
- UVM_SEQ_LIB_USER

`uvm_sequence_state`

Enum

uvm_sequence_state_enum

Defines current sequence state

UVM_CREATED

The sequence has been allocated.

UVM_PRE_START

The sequence is started and the *uvm_sequence_base::pre_start()* task is being executed.

UVM_PRE_BODY

The sequence is started and the *uvm_sequence_base::pre_body()* task is being executed.

UVM_BODY

The sequence is started and the *uvm_sequence_base::body()* task is being executed.

UVM_ENDED

The sequence has completed the execution of the *uvm_sequence_base::body()* task.

UVM_POST_BODY

The sequence is started and the *uvm_sequence_base::post_body()* task is being executed.

UVM_POST_START

The sequence is started and the *uvm_sequence_base::post_start()* task is being executed.

UVM_STOPPED

The sequence has been forcibly ended by issuing a *uvm_sequence_base::kill()* on the sequence.

UVM_FINISHED

The sequence is completely finished executing.

Enum Items
- UVM_CREATED = 1
- UVM_PRE_START = 2
- UVM_PRE_BODY = 4

UVM_BODY = 8
UVM_POST_BODY = 16
UVM_POST_START = 32
UVM_ENDED = 64
UVM_STOPPED = 128
UVM_FINISHED = 256

`uvm_sequencer_arb_mode`

Enum Items

UVM_SEQ_ARB_FIFO
UVM_SEQ_ARB_WEIGHTED
UVM_SEQ_ARB_RANDOM
UVM_SEQ_ARB_STRICT_FIFO
UVM_SEQ_ARB_STRICT_RANDOM
UVM_SEQ_ARB_USER

`uvm_severity`

Enum

uvm_severity

Defines all possible values for report severity.

UVM_INFO

Informative message.

UVM_WARNING

Indicates a potential problem.

UVM_ERROR

Indicates a real problem. Simulation continues subject to the configured message action.

UVM_FATAL

Indicates a problem from which simulation cannot recover. Simulation exits via $finish after a 0 delay.

Enum Items

UVM_INFO
UVM_WARNING
UVM_ERROR
UVM_FATAL

`uvm_status_e`

Enum

uvm_status_e

Return status for register operations

UVM_IS_OK

Operation completed successfully

UVM_NOT_OK

Operation completed with error

UVM_HAS_X

Operation completed successfully bit had unknown bits.

Enum Items

UVM_IS_OK
UVM_NOT_OK
UVM_HAS_X

`uvm_tlm_command_e`

Enum

uvm_tlm_command_e

Command attribute type definition

UVM_TLM_READ_COMMAND

Bus read operation

UVM_TLM_WRITE_COMMAND

Bus write operation

UVM_TLM_IGNORE_COMMAND

No bus operation.

Enum Items

- UVM_TLM_READ_COMMAND
- UVM_TLM_WRITE_COMMAND
- UVM_TLM_IGNORE_COMMAND

`uvm_tlm_phase_e`

Enum

uvm_tlm_phase_e

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

UNINITIALIZED_PHASE

Defaults for constructor

BEGIN_REQ

Beginning of request phase

END_REQ

End of request phase

BEGIN_RESP

Beginning of response phase

END_RESP

End of response phase

Enum Items

- UNINITIALIZED_PHASE
- BEGIN_REQ
- END_REQ
- BEGIN_RESP
- END_RESP

`uvm_tlm_response_status_e`

Enum

uvm_tlm_response_status_e

Response status attribute type definition

UVM_TLM_OK_RESPONSE

Bus operation completed successfully

UVM_TLM_INCOMPLETE_RESPONSE

Transaction was not delivered to target

UVM_TLM_GENERIC_ERROR_RESPONSE

Bus operation had an error

UVM_TLM_ADDRESS_ERROR_RESPONSE

Invalid address specified

UVM_TLM_COMMAND_ERROR_RESPONSE

Invalid command specified

UVM_TLM_BURST_ERROR_RESPONSE

Invalid burst specified

UVM_TLM_BYTE_ENABLE_ERROR_RESPONSE

Invalid byte enabling specified

Enum Items

UVM_TLM_OK_RESPONSE = 1
UVM_TLM_INCOMPLETE_RESPONSE = 0
UVM_TLM_GENERIC_ERROR_RESPONSE = -1
UVM_TLM_ADDRESS_ERROR_RESPONSE = -2
UVM_TLM_COMMAND_ERROR_RESPONSE = -3
UVM_TLM_BURST_ERROR_RESPONSE = -4
UVM_TLM_BYTE_ENABLE_ERROR_RESPONSE = -5

`uvm_tlm_sync_e`

Enum

uvm_tlm_sync_e

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

UVM_TLM_ACCEPTED

Transaction has been accepted

UVM_TLM_UPDATED

Transaction has been modified

UVM_TLM_COMPLETED

Execution of transaction is complete

Enum Items

UVM_TLM_ACCEPTED
UVM_TLM_UPDATED
UVM_TLM_COMPLETED

`uvm_verbosity`

Enum

uvm_verbosity

Defines standard verbosity levels for reports.

UVM_NONE

Report is always printed. Verbosity level setting cannot disable it.

UVM_LOW

Report is issued if configured verbosity is set to UVM_LOW or above.

UVM_MEDIUM

Report is issued if configured verbosity is set to UVM_MEDIUM or above.

UVM_HIGH

Report is issued if configured verbosity is set to UVM_HIGH or above.

UVM_FULL

Report is issued if configured verbosity is set to UVM_FULL or above.

Enum Items

UVM_NONE = 0
UVM_LOW = 100
UVM_MEDIUM = 200
UVM_HIGH = 300
UVM_FULL = 400
UVM_DEBUG = 500

`uvm_wait_op`

Enum

uvm_wait_op

Specifies the operand when using methods like *uvm_phase::wait_for_state*.

UVM_EQ

equal

UVM_NE

not equal

UVM_LT

less than

UVM_LTE

less than or equal to

UVM_GT

greater than

UVM_GTE

greater than or equal to

Enum Items

UVM_LT
UVM_LTE
UVM_NE
UVM_EQ
UVM_GT
UVM_GTE

15.2.2 Structs

`typedef struct uvm_hdl_path_slice`

Type

uvm_hdl_path_slice

Slice of an HDL path

Struct that specifies the HDL variable that corresponds to all or a portion of a register.

path

Path to the HDL variable.

offset

Offset of the LSB in the register that this variable implements

size

Number of bits (toward the MSB) that this variable implements

If the HDL variable implements all of the register, *offset* and *size* are specified as -1. For example: |

```
r1.add_hdl_path('{ '{"r1", -1, -1} });
```

`typedef struct uvm_printer_row_info`

`typedef struct uvm_reg_bus_op`

CLASS

uvm_reg_bus_op

Struct that defines a generic bus transaction for register and memory accesses, having *kind* (read or write), *address* , *data* , and *byte enable* information. If the bus is narrower than the register or memory location being accessed, there will be multiple of these bus operations for every abstract *uvm_reg_item* transaction. In this case, *data* represents the portion of *uvm_reg_item::value* being transferred during this bus cycle. If the bus is wide enough to perform the register or memory operation in a single cycle, *data* will be the same as *uvm_reg_item::value*.

`typedef struct uvm_reg_map_addr_range`

15.2.3 Functions

`function void global_stop_request()`

Method- global_stop_request - **DEPRECATED**

Convenience function for uvm_test_done.stop_request(). See *uvm_test_done_objection::stop_request* for more information.

`function void set_config_int(string inst_name, string field_name, uvm_bitstream_-t value)`

Function- set_config_int

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.

Parameters

- **`inst_name`** (*`string`*)
- **`field_name`** (*`string`*)
- **`value`** (*uvm_bitstream_t*)

`function void set_config_object(string inst_name, string field_name, uvm_-object value, bit clone = 1)`

Function- set_config_object

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.

Parameters

- **`inst_name`** (*`string`*)
- **`field_name`** (*`string`*)
- **`value`** (*uvm_object*)
- **`clone`** (*`bit`*)

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

Function- set_config_string

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.

Parameters

- **`inst_name`** (*`string`*)
- **`field_name`** (*`string`*)
- **`value`** (*`string`*)

`function void set_global_stop_timeout(time timeout)`

Function- set_global_stop_timeout - **DEPRECATED**

Convenience function for uvm_test_done.stop_timeout = timeout. See <uvm_uvm_test_done::stop_timeout> for more information.

Parameters

- **`timeout`** (*`time`*)

`function void set_global_timeout(time timeout, bit overridable = 1)`

Parameters

`timeout` (*`time`*)

`overridable` (*`bit`*)

`function string uvm_bits_to_string(logic[UVM_LARGE_STRING:0] str)`

Function

uvm_bits_to_string

Converts an input bit-vector to its string equivalent. Max bit-vector length is approximately 14000 characters.

Parameters

`str` (*`logic[UVM_LARGE_STRING:0]`*)

`function string uvm_bitstream_to_string(uvm_bitstream_t value, int size, uvm_radix_-enum radix = UVM_NORADIX, string radix_str = "")`

Function- uvm_bitstream_to_string

Parameters

`value` (*uvm_bitstream_t*)

`size` (*`int`*)

`radix` (*uvm_radix_enum*)

`radix_str` (*`string`*)

`function int unsigned uvm_create_random_seed(string type_id, string inst_id = "")`

Function- uvm_create_random_seed

Creates a random seed and updates the seed map so that if the same string is used again, a new value will be generated. The inst_id is used to hash by instance name and get a map of type name hashes which the type_id uses for its lookup.

Parameters

`type_id` (*`string`*)

`inst_id` (*`string`*)

`function string uvm_dpi_get_next_arg(int init = 0)`

Parameters

`init` (*`int`*)

`function string uvm_dpi_get_tool_name()`

`function string uvm_dpi_get_tool_version()`

`function int uvm_get_array_index_int(string arg, bit is_wildcard)`

Function- uvm_get_array_index_int

The following functions check to see if a string is representing an array index, and if so, what the index is.

Parameters

`arg` (*`string`*)

`is_wildcard` (*`bit`*)

`function string uvm_get_array_index_string(string arg, bit is_wildcard)`

Function- uvm_get_array_index_string

Parameters

`arg` (*`string`*)

`is_wildcard` (*`bit`*)

`function uvm_report_object uvm_get_report_object()`

Function

uvm_get_report_object

Returns the nearest uvm_report_object when called. For the global version, it returns uvm_root.

Return type

uvm_report_object

`function bit uvm_has_wildcard(string arg)`

Function- uvm_has_wildcard

Parameters

`arg` (*`string`*)

`function string uvm_hdl_concat2string(uvm_hdl_path_concat concat)`

concat2string

Parameters

`concat` (*uvm_hdl_path_concat*)

`function string uvm_instance_scope()`

Function- uvm_instance_scope

A function that returns the scope that the UVM library lives in, either an instance, a module, or a package.

`function string uvm_integral_to_string(uvm_integral_t value, int size, uvm_radix_-enum radix = UVM_NORADIX, string radix_str = "")`

Function- uvm_integral_to_string

Parameters

`value` (*uvm_integral_t*)
`size` (*`int`*)
`radix` (*uvm_radix_enum*)
`radix_str` (*`string`*)

`function bit uvm_is_array(string arg)`

Function- uvm_is_array

Parameters

`arg` (*`string`*)

`function bit uvm_is_match(string expr, string str)`

Function

uvm_is_match

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.

Parameters

`expr` (*`string`*)
`str` (*`string`*)

`function string uvm_leaf_scope(string full_name, byte scope_separator = ".")`

Function- uvm_leaf_scope

Parameters

`full_name` (*`string`*)
`scope_separator` (*`byte`*)

`function string uvm_object_value_str(uvm_object v)`

Function- uvm_object_value_str

Parameters

`v` (*uvm_object*)

`function int unsigned uvm_oneway_hash(string string_in, int unsigned seed = 0)`

Parameters

`string_in` (*`string`*)
`seed` (*`int unsigned`*)

`function void uvm_process_report_message(uvm_report_message report_message)`

Function

uvm_process_report_message

This method, defined in package scope, is a convenience function that delegate to the corresponding component method in *uvm_top* . It 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.

Parameters

`report_message` (*uvm_report_message*)

`function string uvm_radix_to_string(uvm_radix_enum radix)`

Function- uvm_radix_to_string

Parameters

`radix` (*uvm_radix_enum*)

function void uvm_report(uvm_severity severity, string id, string message, int verbosity = (severity==uvm_severity'(UVM_ERROR))?UVM_LOW:(severity==uvm_-severity'(UVM_FATAL))?UVM_NONE:UVM_MEDIUM, string filename = "", int line = 0, string context_name = "", bit report_enabled_checked = 0)

Function

uvm_report

Parameters

- **severity** (*uvm_severity*)
- **id** (*string*)
- **message** (*string*)
- **verbosity** (*int*)
- **filename** (*string*)
- **line** (*int*)
- **context_name** (*string*)
- **report_enabled_checked** (*bit*)

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

Function

uvm_report_enabled

Returns 1 if the configured verbosity in *uvm_top* for this severity/id is greater than or equal to *verbosity* else returns 0.

See also *uvm_report_object::uvm_report_enabled*.

Static methods of an extension of uvm_report_object, e.g. uvm_component-based objects, cannot call *uvm_report_enabled* because the call will resolve to the *uvm_report_object::uvm_report_enabled*, which is non-static. Static methods cannot call non-static methods of the same class.

Parameters

- **verbosity** (*int*)
- **severity** (*uvm_severity*)
- **id** (*string*)

function void uvm_report_error(string id, string message, int verbosity = UVM_LOW, string filename = "", int line = 0, string context_name = "", bit report_enabled_-checked = 0)

Function

uvm_report_error

Parameters

- **id** (*string*)
- **message** (*string*)
- **verbosity** (*int*)
- **filename** (*string*)
- **line** (*int*)
- **context_name** (*string*)
- **report_enabled_checked** (*bit*)

function void uvm_report_fatal(string id, string message, int verbosity = UVM_NONE, string filename = "", int line = 0, string context_name = "", bit report_enabled_-checked = 0)

Function

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.

Note: Verbosity is ignored for warnings, errors, and fatals to ensure users do not inadvertently filter them out. It remains in the methods for backward compatibility.

Parameters

id (*string*)
message (*string*)
verbosity (*int*)
filename (*string*)
line (*int*)
context_name (*string*)
report_enabled_checked (*bit*)

function void uvm_report_info(string id, string message, int verbosity = UVM_MEDIUM, string filename = "", int line = 0, string context_name = "", bit report_enabled_checked = 0)

Function

uvm_report_info

Parameters

id (*string*)
message (*string*)
verbosity (*int*)
filename (*string*)
line (*int*)
context_name (*string*)
report_enabled_checked (*bit*)

function void uvm_report_warning(string id, string message, int verbosity = UVM_MEDIUM, string filename = "", int line = 0, string context_name = "", bit report_enabled_checked = 0)

Function

uvm_report_warning

Parameters

id (*string*)
message (*string*)
verbosity (*int*)
filename (*string*)
line (*int*)
context_name (*string*)
report_enabled_checked (*bit*)

function string uvm_revision_string()

function void uvm_split_string(string str, byte sep, string values)

Function

uvm_split_string

Returns a queue of strings, *values* , that is the result of the *str* split based on the *sep* . For example:

```
uvm_split_string("1,on,false", ",", splits);
```

Results in the 'splits' queue containing the three elements: 1, on and false.

Parameters

str (*string*)
sep (*byte*)
values (*string*)

function bit uvm_string_to_action(string action_str, uvm_action action)

Parameters

action_str (*string*)
action (*uvm_action*)

function logic[UVM_LARGE_STRING:0] uvm_string_to_bits(string str)

Function

uvm_string_to_bits

Converts an input string to its bit-vector equivalent. Max bit-vector length is approximately 14000 characters.

Parameters
- **`str`** (*`string`*)

`function bit uvm_string_to_severity(string sev_str, uvm_severity sev)`

TODO merge with uvm_enum_wrapper(uvm_severity)

Parameters
- **`sev_str`** (*`string`*)
- **`sev`** (*uvm_severity*)

`function string uvm_vector_to_string(uvm_bitstream_t value, int size, uvm_radix_-enum radix = UVM_NORADIX, string radix_str = "")`

Backwards compat

Parameters
- **`value`** (*uvm_bitstream_t*)
- **`size`** (*`int`*)
- **`radix`** (*uvm_radix_enum*)
- **`radix_str`** (*`string`*)

15.2.4 DPI Import Functions

`import function string uvm_dpi_get_next_arg_c(int init)`

Parameters
- **`init`** (*`int`*)

`import function string uvm_dpi_get_tool_name_c()`

`import function string uvm_dpi_get_tool_version_c()`

`import function chandle uvm_dpi_regcomp(string regex)`

Parameters
- **`regex`** (*`string`*)

`import function int uvm_dpi_regexec(chandle preg, string str)`

Parameters
- **`preg`** (*`chandle`*)
- **`str`** (*`string`*)

`import function void uvm_dpi_regfree(chandle preg)`

Parameters
- **`preg`** (*`chandle`*)

`import function void uvm_dump_re_cache()`

`import function string uvm_glob_to_re(string glob)`

Parameters
- **`glob`** (*`string`*)

`import function int uvm_hdl_check_path(string path)`

Function

uvm_hdl_check_path

Checks that the given HDL *path* exists. Returns 0 if NOT found, 1 otherwise.

Parameters
- **`path`** (*`string`*)

`import function int uvm_hdl_deposit(string path, uvm_hdl_data_t value)`

Function

uvm_hdl_deposit

Sets the given HDL *path* to the specified *value* . Returns 1 if the call succeeded, 0 otherwise.

Parameters
- **`path`** (*`string`*)
- **`value`** (*uvm_hdl_data_t*)

`import function int uvm_hdl_force(string path, uvm_hdl_data_t value)`

Function

uvm_hdl_force

Forces the *value* on the given *path* . Returns 1 if the call succeeded, 0 otherwise.

Parameters

path (*string*)

value (*uvm_hdl_data_t*)

import function int uvm_hdl_read(string path, uvm_hdl_data_t value)

Function

uvm_hdl_read()

Gets the value at the given *path* . Returns 1 if the call succeeded, 0 otherwise.

Parameters

path (*string*)

value (*uvm_hdl_data_t*)

import function int uvm_hdl_release(string path)

Function

uvm_hdl_release

Releases a value previously set with *uvm_hdl_force*. Returns 1 if the call succeeded, 0 otherwise.

Parameters

path (*string*)

import function int uvm_hdl_release_and_read(string path, uvm_hdl_data_t value)

Function

uvm_hdl_release_and_read

Releases a value previously set with *uvm_hdl_force*. Returns 1 if the call succeeded, 0 otherwise. *value* is set to the HDL value after the release. For 'reg', the value will still be the forced value until it has been procedurally reassigned. For 'wire', the value will change immediately to the resolved value of its continuous drivers, if any. If none, its value remains as forced until the next direct assignment.

Parameters

path (*string*)

value (*uvm_hdl_data_t*)

import function int uvm_re_match(string re, string str)

Parameters

re (*string*)

str (*string*)

15.2.5 Tasks

function run_test(string test_name = "")

Task

run_test

Convenience function for uvm_top.run_test(). See *uvm_root* for more information.

Parameters

test_name (*string*)

function uvm_hdl_force_time(string path, uvm_hdl_data_t value, time force_time = 0)

Function

uvm_hdl_force_time

Forces the *value* on the given *path* for the specified amount of *force_time* . If *force_time* is 0, *uvm_hdl_deposit* is called. Returns 1 if the call succeeded, 0 otherwise.

Parameters

path (*string*)

value (*uvm_hdl_data_t*)

force_time (*time*)

function uvm_wait_for_nba_region()

Task

uvm_wait_for_nba_region

Callers of this task will not return until the NBA region, thus allowing other processes any number of delta cycles (0) to settle out before continuing. See *uvm_sequencer_base::wait_for_sequences* for example usage.

CHAPTER

SIXTEEN

MACROS

Table 1: Defined Control Defines

Name	Description
UVM_CORESERVICE_TYPE	
UVM_DEPRECATED_STARTING_PHASE	
UVM_HDL_MAX_WIDTH	
UVM_LINE_WIDTH	
UVM_MAX_STREAMBITS	
UVM_NUM_LINES	
UVM_PACKER_MAX_BYTES	
UVM_REG_ADDR_WIDTH	
UVM_REG_BYTENABLE_WIDTH	
UVM_REG_CVR_WIDTH	
UVM_REG_DATA_WIDTH	
uvm_record_attribute(TR_HANDLE, NAME, VALUE)	
uvm_record_int(NAME, VALUE, SIZE, RADIX =UVM_NORADIX)	
uvm_record_real(NAME, VALUE)	
uvm_record_string(NAME, VALUE)	
uvm_record_time(NAME, VALUE)	

Table 2: Undefined Control Defines

Name	Description
UVM_CB_TRACE_ON	The +define+UVM_CB_TRACE_ON setting will instrument the uvm library to emit messages with message id UVMCB_TRC and UVM_NONE verbosity notifing add, delete and execution of uvm callbacks. The instrumentation is off by default.
UVM_CMDLINE_NO_DPI	Import DPI functions used by the interface to generate the lists.
UVM_DEPRECATED_REPORTING	
UVM_DISABLE_AUTO_ITEM_RECORDING	
UVM_EMPTY_MACROS	
UVM_ENABLE_FIELD_CHECKS	
UVM_FIX_REV	***Macro*** UVM_VERSION_STRING Provides a string-ized version of the UVM Library version number. When there is a FIX_REV, the string is "<name>-<major>.<minor><fix>" (such as "UVM-1.1d"). When there is NO FIX_REV, the string is "<name>-<major>.<minor>" (such as "UVM-1.2").

continues on next page

Table 2 – continued from previous page

Name	Description
UVM_HDL_NO_DPI	
UVM_NO_DEPRECATED	Deprecation Control Macros
UVM_NO_DPI	Top-level file for DPI subroutines used by UVM. Tool-specific distribution overlays may be required. To use UVM without any tool-specific overlay, use +defin+UVM_NO_DPI
UVM_NO_WAIT_FOR_NBA	If &96;included directly in a program block, can't use a non-blocking assign, but it isn't needed since program blocks are in a separate region.
UVM_OBJECT_DO_NOT_NEED_CON-STRUCTOR	
UVM_REGEX_NO_DPI	
UVM_REG_NO_INDIVIDUAL_FIELD_AC-CESS	
UVM_REPORT_DISABLE_FILE	
UVM_REPORT_DISABLE_FILE_LINE	&96;ifndef UVM_USE_FILE_LINE &96;define UVM_REPORT_DISABLE_FILE_LINE &96;endif
UVM_REPORT_DISABLE_LINE	
UVM_USE_PROCESS_CONTAINER	
UVM_USE_PROCESS_STATE	
UVM_USE_RESOURCE_CONVERTER	
UVM_USE_STRING_QUEUE_STREAM-ING_PACK	
UVM_USE_SUSPEND_RESUME	

Table 3: Defines

Name	Value	Description
UVM_BLOCKING_GET_-IMP(imp, TYPE, arg)		
UVM_BLOCKING_GET_IMP_-SFX(SFX, imp, TYPE, arg)		
UVM_BLOCKING_GET_PEEK_-IMP(imp, TYPE, arg)		
UVM_BLOCKING_PEEK_-IMP(imp, TYPE, arg)		
UVM_BLOCKING_PEEK_IMP_-SFX(SFX, imp, TYPE, arg)		
UVM_BLOCKING_PUT_-IMP(imp, TYPE, arg)		TLM imp implementations
UVM_BLOCKING_PUT_IMP_-SFX(SFX, imp, TYPE, arg)		These imps are used in uvmport, uvm*export and uvm**imp, using suffixes
UVM_BLOCKING_TRANS-PORT_IMP(imp, REQ, RSP, req_arg, rsp_arg)		
UVM_BLOCKING_TRANS-PORT_IMP_SFX(SFX, imp, REQ, RSP, req_arg, rsp_arg)		

continues on next page

Table 3 – continued from previous page

Name	Value	Description
UVM_DEFAULT_TIMEOUT	9200s	***MACRO*** &96;UVM_DEFAULT_TIMEOUT The default timeout for simulation, if not overridden by <uvm_root::set_timeout> or <uvm_cmdline_processor::+UVM_TIMEOUT>
UVM_EXPORT_COMMON(MASK, TYPE_NAME)		
UVM_FUNCTION_ERROR	"TLM interface function not implemented"	
UVM_GET_IMP(imp, TYPE, arg)		
UVM_GET_PEEK_IMP(imp, TYPE, arg)		
UVM_IMP_COMMON(MASK, TYPE_NAME, IMP)		
UVM_MAJOR_REV	1	***Macro*** UVM_MAJOR_REV Defines the MAJOR revision number. For UVM version 1.2, the MAJOR revision number is '1' `` `define UVM_MAJOR_REV 1 ``
UVM_MAJOR_REV_1		***Macro*** UVM_MAJOR_REV_1 Indicates that the MAJOR version of this release is '1'. `` `define UVM_MAJOR_REV_1 ``
UVM_MAJOR_VERSION_1_2		Undocumented, same thing as UVM_VERSION_1_2
UVM_MINOR_REV	2	***Macro*** UVM_MINOR_REV Defines the MINOR revision number. For UVM version 1.2, the MINOR revision number is '2' `` `define UVM_MINOR_REV 2 ``
UVM_MINOR_REV_2		***Macro*** UVM_MINOR_REV_2 Indicates that the MINOR version of this release is '2'. `` `define UVM_MINOR_REV_2 ``
UVM_MS_IMP_COMMON(MASK, TYPE_NAME)		

continues on next page

Table 3 – continued from previous page

Name	Value	Description
UVM_NAME	UVM	***Macro*** UVM_NAME The name used by the library when displaying the name of the library. `` `define UVM_NAME UVM ``
UVM_NONBLOCKING_GET_-IMP(imp, TYPE, arg)		
UVM_NONBLOCKING_GET_-IMP_SFX(SFX, imp, TYPE, arg)		
UVM_NONBLOCKING_GET_-PEEK_IMP(imp, TYPE, arg)		
UVM_NONBLOCKING_PEEK_-IMP(imp, TYPE, arg)		
UVM_NONBLOCKING_PEEK_-IMP_SFX(SFX, imp, TYPE, arg)		
UVM_NONBLOCKING_PUT_-IMP(imp, TYPE, arg)		
UVM_NONBLOCKING_PUT_-IMP_SFX(SFX, imp, TYPE, arg)		
UVM_NONBLOCKING_TRANS-PORT_IMP(imp, REQ, RSP, req_arg, rsp_arg)		
UVM_NONBLOCKING_TRANS-PORT_IMP_SFX(SFX, imp, REQ, RSP, req_arg, rsp_arg)		
UVM_PEEK_IMP(imp, TYPE, arg)		
UVM_PH_TRACE(ID, MSG, PH, VERB)		
UVM_PORT_COMMON(MASK, TYPE_NAME)		
UVM_POST_VERSION_1_1		***Macro*** UVM_POST_VERSION_1_1 Indicates that this version of the UVM came after the 1.1 versions, including the various 1.1 fix revisions. The first UVM version wherein this macro is defined is 1.2, and the macro will continue to be defined for all future revisions of the UVM library. `` `define UVM_POST_VERSION_1_1 ``
UVM_PUT_IMP(imp, TYPE, arg)		

continues on next page

Table 3 – continued from previous page

Name	Value	Description

UVM_RESOURCE_GET_-FCNS(base_type)		When specicializing resources the get_by_name and get_by_type functions must be redefined. The reason is that the version of these functions in the base class (uvm_resource(T)) returns an object of type uvm_resource(T). In the specializations we must return an object of the type of the specialization. So, we call the base_class implementation of these functions and then downcast to the subtype.
		
		This macro is invokved once in each where a resource specialization is a class defined as:
		`class <resource_specialization>␣ ↪extends uvm_resource#(T)`
		where <resource_specialization> is the name of the derived class. The argument to this macro is T, the type of the uvm_resource(T) specialization. The class in which the macro is defined must supply a typedef of the specialized class of the form:
		`typedef <resource_specialization>␣ ↪this_subtype;`
		where <resource_specialization> is the same as above. The macro generates the get_by_name() and get_by_type() functions for the specialized resource (i.e. resource subtype).

UVM_SEQ_ITEM_FUNCTION_-ERROR	"Sequencer interface function not implemented"	
UVM_SEQ_ITEM_GET_MASK	(1<<7)	
UVM_SEQ_ITEM_GET_NEXT_-ITEM_MASK	(1<<0)	
UVM_SEQ_ITEM_HAS_DO_-AVAILABLE_MASK	(1<<3)	
UVM_SEQ_ITEM_ITEM_-DONE_MASK	(1<<2)	
UVM_SEQ_ITEM_PEEK_MASK	(1<<8)	
UVM_SEQ_ITEM_PULL_-IMP(imp, REQ, RSP, req_arg, rsp_arg)		imp definitions
UVM_SEQ_ITEM_PULL_MASK		
UVM_SEQ_ITEM_PUSH_MASK	(UVM_SEQ_-ITEM_PUT_-MASK)	
UVM_SEQ_ITEM_PUT_MASK	(1<<6)	
UVM_SEQ_ITEM_PUT_RE-SPONSE_MASK	(1<<5)	
UVM_SEQ_ITEM_TASK_ERROR	"Sequencer interface task not implemented"	
UVM_SEQ_ITEM_TRY_NEXT_-ITEM_MASK	(1<<1)	
UVM_SEQ_ITEM_UNI_PULL_-MASK		

continues on next page

Table 3 – continued from previous page

Name	Value	Description
UVM_SEQ_ITEM_WAIT_FOR_-SEQUENCES_MASK	(1<<4)	
UVM_SEQ_PORT(MASK, TYPE_NAME)		
UVM_STRING_QUEUE_-STREAMING_PACK(q)	uvm_pkg::m_-uvm_string_-queue_join(q)	
UVM_TASK_ERROR	"TLM interface task not imple-mented"	Copyright 2007-2011 Mentor Graphics Corpora-tion Copyright 2007-2010 Cadence Design Sys-tems, Inc. Copyright 2010 Synopsys, Inc. All Rights Reserved Worldwide Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.
UVM_TLM_ANALYSIS_MASK	(1<<8)	
UVM_TLM_BLOCKING_GET_-MASK	(1<<1)	
UVM_TLM_BLOCKING_GET_-PEEK_MASK	(UVM_TLM_-BLOCKING_-GET_MASK | UVM_TLM_-BLOCKING_-PEEK_MASK)	
UVM_TLM_BLOCKING_MAS-TER_MASK	(UVM_TLM_-BLOCKING_-PUT_MASK | UVM_TLM_-BLOCKING_-GET_MASK | UVM_TLM_-BLOCKING_-PEEK_MASK | UVM_TLM_-MASTER_BIT_-MASK)	
UVM_TLM_BLOCKING_PEEK_-MASK	(1<<2)	
UVM_TLM_BLOCKING_PUT_-MASK	(1<<0)	primitive interfaces

continues on next page

Table 3 – continued from previous page

Name	Value	Description
UVM_TLM_BLOCKING_-SLAVE_MASK	(UVM_TLM_-BLOCKING_-PUT_MASK | UVM_TLM_-BLOCKING_-GET_MASK | UVM_TLM_-BLOCKING_-PEEK_MASK | UVM_TLM_-SLAVE_BIT_-MASK)	
UVM_TLM_BLOCKING_-TRANSPORT_MASK	(1<<3)	
UVM_TLM_B_MASK	(1<<2)	***MACRO*** &96;UVM_TLM_B_MASK Define blocking mask onehot assignment = 'b100
UVM_TLM_B_TRANSPORT_-IMP(imp, T, t, delay)		***Macro*** &96;UVM_TLM_B_TRANSPORT_IMP The macro wraps the function b_transport() Execute a blocking transaction. Once this method returns, the transaction is assumed to have been executed. Whether that execution is successful or not must be indicated by the transaction itself. The callee may modify or update the transaction object, subject to any constraints imposed by the transaction class. The initiator may re-use a transaction object from one call to the next and across calls to b_transport(). The call to b_transport shall mark the first timing point of the transaction. The return from b_transport() shall mark the final timing point of the transaction. The timing annotation argument allows the timing points to be offset from the simulation times at which the task call and return are executed.
UVM_TLM_FIFO_FUNCTION_-ERROR	"fifo channel function not implemented"	

continues on next page

Table 3 – continued from previous page

Name	Value	Description
UVM_TLM_FIFO_TASK_ERROR	"fifo channel task not implemented"	Copyright 2007-2011 Mentor Graphics Corporation Copyright 2007-2011 Cadence Design Systems, Inc. Copyright 2010 Synopsys, Inc. All Rights Reserved Worldwide Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.
UVM_TLM_FUNCTION_ERROR	"TLM-2 interface function not implemented"	***MACRO*** &96;UVM_TLM_FUNCTION_ERROR Defines Not-Yet-Implemented TLM functions
UVM_TLM_GET_MASK	(UVM_TLM_BLOCKING_GET_MASK | UVM_TLM_NONBLOCKING_GET_MASK)	
UVM_TLM_GET_PEEK_MASK	(UVM_TLM_GET_MASK | UVM_TLM_PEEK_MASK)	
UVM_TLM_GET_TYPE_NAME(NAME)		
UVM_TLM_MASTER_BIT_MASK	(1<<9)	
UVM_TLM_MASTER_MASK	(UVM_TLM_BLOCKING_MASTER_MASK | UVM_TLM_NONBLOCKING_MASTER_MASK)	
UVM_TLM_NB_BW_MASK	(1<<1)	***MACRO*** &96;UVM_TLM_NB_BW_MASK Define Non blocking backward mask onehot assignment = 'b010
UVM_TLM_NB_FW_MASK	(1<<0)	***MACRO*** &96;UVM_TLM_NB_FW_MASK Define Non blocking Forward mask onehot assignment = 'b001

continues on next page

Table 3 – continued from previous page

Name	Value	Description

UVM_TLM_NB_TRANSPORT_-BW_IMP(imp, T, P, t, p, delay)		***Macro***
		
		&96;UVM_TLM_NB_TRANS-PORT_BW_IMP
		
		Implementation of the backward path. The macro wraps the function called nb_trans-port_bw(). This function MUST be implemented in the INITIATOR component class.
		
		Every call to this method may mark a timing point, including the final timing point, in the ex-ecution of the transaction. The timing annotation argument allows the timing point to be offset from the simulation times at which the backward path is used. The final timing point of a transaction may be marked by a call to nb_transport_fw() within \`*UVM_TLM_NB_TRANSPORT_FW_IMP* or a return from this or subsequent call to nb_transport_bw().
		
		See <TLM2 Interfaces, Ports, Exports and Trans-port Interfaces Subset> for more details on the semantics and rules of the nonblocking transport interface.
		
		Example:
		
		`class master extends uvm_component;`
		`   uvm_tlm_nb_initiator_socket`
		`         #(trans, uvm_tlm_phase_e,␣`
		`↪this_t) initiator_socket;`
		
		`   function void build_phase(uvm_`
		`↪phase phase);`
		`      initiator_socket = new(`<br>`↪"initiator_socket", this, this);`
		`   endfunction`
		
		`   function uvm_tlm_sync_e nb_`
		`↪transport_bw(trans t,`
		`␣`
		`↪ref uvm_tlm_phase_e p,`
		`␣`
		`↪input uvm_tlm_time delay);`
		`       transaction = t;`
		`       state = p;`
		`       return UVM_TLM_ACCEPTED;`
		`   endfunction`
		
		`   ...`
		`endclass`

continues on next page

Table 3 – continued from previous page

Name	Value	Description
UVM_TLM_NB_TRANSPORT_-FW_IMP(imp, T, P, t, p, delay)		***Macro*** &96;UVM_TLM_NB_TRANS-PORT_FW_IMP The macro wraps the forward path call function nb_transport_fw() The first call to this method for a transaction marks the initial timing point. Every call to this method may mark a timing point in the execution of the transaction. The timing annotation argument allows the timing points to be offset from the simulation times at which the forward path is used. The final timing point of a transaction may be marked by a call to nb_transport_bw() within *`UVM_TLM_NB_TRANSPORT_BW_IMP* or a return from this or subsequent call to nb_transport_fw(). See <TLM2 Interfaces, Ports, Exports and Transport Interfaces Subset> for more details on the semantics and rules of the nonblocking transport interface.
UVM_TLM_NONBLOCKING_-GET_MASK	(1<<5)	
UVM_TLM_NONBLOCKING_-GET_PEEK_MASK	(UVM_TLM_-NONBLOCK-ING_GET_MASK | UVM_TLM_-NONBLOCK-ING_PEEK_-MASK)	
UVM_TLM_NONBLOCKING_-MASTER_MASK	(UVM_TLM_-NONBLOCK-ING_PUT_MASK | UVM_TLM_-NONBLOCK-ING_GET_MASK | UVM_TLM_-NONBLOCK-ING_PEEK_-MASK | UVM_-TLM_MASTER_-BIT_MASK)	
UVM_TLM_NONBLOCKING_-PEEK_MASK	(1<<6)	
UVM_TLM_NONBLOCKING_-PUT_MASK	(1<<4)	

continues on next page

Table 3 – continued from previous page

Name	Value	Description
UVM_TLM_NONBLOCKING_-SLAVE_MASK	(UVM_TLM_-NONBLOCK-ING_PUT_MASK | UVM_TLM_-NONBLOCK-ING_GET_MASK | UVM_TLM_-NONBLOCK-ING_PEEK_-MASK | UVM_-TLM_SLAVE_-BIT_MASK)	
UVM_TLM_NONBLOCKING_-TRANSPORT_MASK	(1<<7)	
UVM_TLM_PEEK_MASK	(UVM_TLM_-BLOCKING_-PEEK_MASK | UVM_TLM_-NONBLOCK-ING_PEEK_-MASK)	
UVM_TLM_PUT_MASK	(UVM_TLM_-BLOCKING_-PUT_MASK | UVM_TLM_-NONBLOCK-ING_PUT_-MASK)	combination interfaces
UVM_TLM_SLAVE_BIT_MASK	(1<<10)	
UVM_TLM_SLAVE_MASK	(UVM_TLM_-BLOCKING_-SLAVE_MASK | UVM_TLM_-NONBLOCK-ING_SLAVE_-MASK)	
UVM_TLM_TASK_ERROR	"TLM-2 interface task not imple-mented"	***MACRO*** &96;UVM_TLM_TASK_ERROR Defines Not-Yet-Implemented TLM tasks
UVM_TLM_TRANSPORT_MASK	(UVM_TLM_-BLOCKING_-TRANSPORT_-MASK | UVM_-TLM_NON-BLOCKING_-TRANSPORT_-MASK)	
UVM_TRANSPORT_IMP(imp, REQ, RSP, req_arg, rsp_arg)		

continues on next page

Table 3 – continued from previous page

Name	Value	Description
UVM_VERSION_1_2		***Macro*** UVM_VERSION_1_2 Indicates that the version of this release is '1.2'. ``` `define UVM_VERSION_1_2```
UVM_VERSION_STRING	"UVM_-NAME-UVM_-MAJOR_-REV.UVM_-MINOR_REV"	
uvm_add_to_seq_lib(TYPE, LIB-TYPE)		

continues on next page

Table 3 – continued from previous page

Name	Value	Description

uvm_analysis_imp_decl(SFX)		***MACRO***
		
		 &96;uvm_analysis_imp_decl
		
		 `` `uvm_analysis_imp_decl(SFX) ``
		
		 Define the class uvm_analysis_impSFX for providing an analysis implementation. *SFX* is the suffix for the new class type. The analysis implementation is the write function. The &96;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_decl(_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`

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_blocking_get_imp_decl(SFX)		***MACRO*** &96;uvm_blocking_get_imp_decl `` `uvm_blocking_get_imp_decl(SFX) `` Define the class uvm_blocking_get_impSFX for providing blocking get implementations. *SFX* is the suffix for the new class type.
uvm_blocking_get_peek_imp_-decl(SFX)		***MACRO*** &96;uvm_blocking_get_peek_imp_decl `` `uvm_blocking_get_peek_imp_↪decl(SFX) `` Define the class uvm_blocking_get_peek_impSFX for providing the blocking get_peek implementation.
uvm_blocking_master_imp_-decl(SFX)		***MACRO*** &96;uvm_blocking_master_imp_decl `` `uvm_blocking_master_imp_decl(SFX) `` Define the class uvm_blocking_master_impSFX for providing the blocking master implementation.
uvm_blocking_peek_imp_-decl(SFX)		***MACRO*** &96;uvm_blocking_peek_imp_decl `` `uvm_blocking_peek_imp_decl(SFX) `` Define the class uvm_blocking_peek_impSFX for providing blocking peek implementations. *SFX* is the suffix for the new class type.
uvm_blocking_put_imp_decl(SFX)		***MACRO*** &96;uvm_blocking_put_imp_decl `` `uvm_blocking_put_imp_decl(SFX) `` Define the class uvm_blocking_put_impSFX for providing blocking put implementations. *SFX* is the suffix for the new class type.
uvm_blocking_slave_imp_-decl(SFX)		***MACRO*** &96;uvm_blocking_slave_imp_decl `` `uvm_blocking_slave_imp_decl(SFX) `` Define the class uvm_blocking_slave_impSFX for providing the blocking slave implementation.
uvm_blocking_transport_imp_-decl(SFX)		***MACRO*** &96;uvm_blocking_transport_imp_decl `` `uvm_blocking_transport_imp_↪decl(SFX) `` Define the class uvm_blocking_transport_impSFX for providing the blocking transport implementation.
uvm_builtin_bottomup_-phase(PHASE)		

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_builtin_task_phase(PHASE)		PREFIX&96;&96;PHASE&96;&96;*phase PREFIX&96;&96;PHASE&96;&96;*ph = PRE-FIX&96;&96;PHASE&96;&96;_phase::get();
uvm_builtin_topdown_-phase(PHASE)		
uvm_cb_trace(OBJ, CB, OPER)	/* null */	
uvm_cb_trace_noobj(CB, OPER)	/* null */	
uvm_component_param_utils(T)		
uvm_component_param_utils_be-gin(T)		
uvm_component_registry(T, S)		***MACRO*** &96;uvm_component_registry Registers a uvm_component-based class with the factory `` `uvm_component_registry(T,S) `` Registers a uvm_component-based class *T* and lookup string *S* with the factory. *S* typically is the name of the class in quotes. The *`uvm_object_utils* family of macros uses this macro.

continues on next page

Table 3 – continued from previous page

Name	Value	Description

uvm_component_utils(T)		***MACRO***
		
		 &96;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 &96;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 &96;uvm_field** macros can be placed. The block must be terminated by &96;uvm_component_utils_end.

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_component_utils_begin(T)		
uvm_component_utils_end		
uvm_create(SEQ_OR_ITEM)		***MACRO*** &96;uvm_create `` `uvm_create(SEQ_OR_ITEM) `` 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_create_on(SEQ_OR_ITEM, SEQR)		***MACRO*** &96;uvm_create_on `` `uvm_create_on(SEQ_OR_ITEM, SEQR) `` 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 *SEQR* argument.
uvm_create_seq(UVM_SEQ, SEQR_CONS_IF)		

uvm_declare_p_sequencer(SEQUENCER)		***MACRO*** &96;uvm_declare_p_sequencer
		This macro is used to declare a variable *p_sequencer* whose type is specified by *SEQUENCER* .
		`` `uvm_declare_p_sequencer(SEQUENCER) ``
		The example below shows using the &96;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`

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_declare_sequence_lib		**DEPRECATED** * Group- Sequencer Registration Macros The sequencer-specific macros perform the same function as the set of &96;uvm_component_*utils macros except that they also declare the plumbing necessary for creating the sequencer's sequence library.
uvm_delay(TIME)	#(TIME);	

uvm_do(SEQ_OR_ITEM)		***MACRO***
		
		 &96;uvm_do
		
		 `` `uvm_do(SEQ_OR_ITEM) ``
		
		 This macro takes as an argument a uvm_sequence_item variable or object. The argument is created using *`uvm_create* if necessary, then randomized. In the case of an item, it is randomized after the call to <uvm_sequence_base::start_item()> returns. This is called late-randomization. In the case of a sequence, the sub-sequence is started using <uvm_sequence_base::start()> with *call_pre_post* set to 0. In the case of an item, the item is sent to the driver through the associated sequencer.
		
		 For a sequence item, the following are called, in order
		
		 |
		
		 `` `uvm_create(item) ``
		 `sequencer.wait_for_grant(prior) (task)`
		 `this.pre_do(1) (task)`
		 `item.randomize()`
		 `this.mid_do(item) (func)`
		 `sequencer.send_request(item) (func)`
		 `sequencer.wait_for_item_done() (task)`
		 `this.post_do(item) (func)`
		
		 For a sequence, the following are called, in order
		
		 |
		
		 `` `uvm_create(sub_seq) ``
		 `sub_seq.randomize()`
		 `sub_seq.pre_start()         (task)`
		 `this.pre_do(0)              (task)`
		 `this.mid_do(sub_seq)        (func)`
		 `sub_seq.body()              (task)`
		 `this.post_do(sub_seq)       (func)`
		 `sub_seq.post_start()        (task)`

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_do_callbacks(T, CB, METHOD)		
uvm_do_callbacks_exit_on(T, CB, METHOD, VAL)		
uvm_do_obj_callbacks(T, CB, OBJ, METHOD)		***MACRO*** &96;uvm_do_obj_callbacks `` `uvm_do_obj_callbacks(T,CB,OBJ, ↪METHOD) `` 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_obj_callbacks_exit_on(T, CB, OBJ, METHOD, VAL)		***MACRO*** &96;uvm_do_obj_callbacks_exit_on `` `uvm_do_obj_callbacks_exit_on(T,CB, ↪OBJ,METHOD,VAL) `` Calls the given *METHOD* of all callbacks of type *CB* registered with the given object *OBJ* , which must be or be based on type *T* , and returns upon the first callback that returns the bit value given by *VAL* . It is exactly the same as the *`uvm_do_callbacks_exit_on* but has a specific object instance (instead of the implicit this instance) as the third argument. `... // Exit if a callback returns a 1 `` `uvm_do_callbacks_exit_on(mycomp,␣ ↪mycb, seqr, drop_trans(seqr, ↪trans), 1) `` ...` Because this macro calls *return* , its use is restricted to implementations of functions that return a *bit* value, as in the above example.

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_do_on(SEQ_OR_ITEM, SEQR)		***MACRO*** &96;uvm_do_on `` `uvm_do_on(SEQ_OR_ITEM, SEQR) `` 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 *SEQR* argument.

uvm_do_on_pri(SEQ_OR_ITEM, SEQR, PRIORITY)		***MACRO***
		
		&96;uvm_do_on_pri
		
		`` `uvm_do_on_pri(SEQ_OR_ITEM, SEQR,␣ ``
		`` ↪PRIORITY) ``
		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 *SEQR* argument.

uvm_do_on_pri_with(SEQ_OR_-ITEM, SEQR, PRIORITY, CON-STRAINTS)		***MACRO***
		
		&96;uvm_do_on_pri_with
		
		`` `uvm_do_on_pri_with(SEQ_OR_ITEM,␣ ``
		`` ↪SEQR, PRIORITY, CONSTRAINTS) ``
		This is the same as &96;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 *SEQR* argument.

uvm_do_on_with(SEQ_OR_ITEM, SEQR, CONSTRAINTS)		***MACRO***
		
		&96;uvm_do_on_with
		
		`` `uvm_do_on_with(SEQ_OR_ITEM, SEQR,␣ ``
		`` ↪CONSTRAINTS) ``
		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 *SEQR* argument. The user must supply brackets around the constraints.

uvm_do_pri(SEQ_OR_ITEM, PRI-ORITY)		***MACRO*** &96;uvm_do_pri `` `uvm_do_pri(SEQ_OR_ITEM, PRIORITY) `` This is the same as &96;uvm_do except that the sequence item or sequence is executed with the priority specified in the argument

uvm_do_pri_with(SEQ_OR_ITEM, PRIORITY, CONSTRAINTS)		***MACRO***
		
		&96;uvm_do_pri_with
		
		`` `uvm_do_pri_with(SEQ_OR_ITEM,␣ ``
		`` ↪PRIORITY, CONSTRAINTS) ``
		This is the same as &96;uvm_do_pri except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.

uvm_do_seq(UVM_SEQ, SEQR_-CONS_IF)		

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_do_seq_with(UVM_-SEQ, SEQR_CONS_IF, CON-STRAINTS)		
uvm_do_with(SEQ_OR_ITEM, CONSTRAINTS)		***MACRO*** &96;uvm_do_with `` `uvm_do_with(SEQ_OR_ITEM, CONSTRAINTS) `` This is the same as &96;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_end_package		
uvm_error(ID, MSG)		***MACRO*** &96;uvm_error Calls uvm_report_error with a verbosity of UVM_NONE. The message cannot 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_error(ID, MSG) ``
uvm_error_begin(ID, MSG, RM =_-_uvm_msg)		***MACRO*** &96;uvm_error_begin `` `uvm_error_begin(ID, MSG, RM = __uvm_msg) ``
uvm_error_context(ID, MSG, RO)		***MACRO*** &96;uvm_error_context `` `uvm_error_context(ID, MSG, RO) `` Operates identically to &96;uvm_error but requires that the context, or <uvm_report_object> in which the message is printed be explicitly supplied as a macro argument.
uvm_error_context_begin(ID, MSG, RO, RM =__uvm_msg)		***MACRO*** &96;uvm_error_context_begin `` `uvm_error_context_begin(ID, MSG, RO, RM = __uvm_msg) ``
uvm_error_context_end		***MACRO*** &96;uvm_error_context_end `` `uvm_error_context_end `` This macro pair operates identically to *uvm\\_error\\_begin](macro-647aa9df)/[uvm_error_end*, but requires that the context, or <uvm_report_object> in which the message is printed be explicitly supplied as a macro argument.

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_error_end		***MACRO*** &96;uvm_error_end This macro pair operates identically to `uvm\\_info\\_begin](macro-95076a0b)/[`*uvm_info_end* with exception that the message severity is <UVM_ERROR> and has no verbosity threshold. `` `uvm_error_end `` The usage shown in *`uvm_info_end* works identically for this pair.
uvm_fatal(ID, MSG)		***MACRO*** &96;uvm_fatal Calls uvm_report_fatal with a verbosity of UVM_NONE. The message cannot be turned off using the reporter's verbosity setting, but can be turned off by setting the action for the message. *ID* is given as the message tag and *MSG* is given as the message text. The file and line are also sent to the uvm_report_fatal call. `` `uvm_fatal(ID, MSG) ``
uvm_fatal_begin(ID, MSG, RM =__-uvm_msg)		***MACRO*** &96;uvm_fatal_begin `` `uvm_fatal_begin(ID, MSG, RM = __ ↪uvm_msg) ``
uvm_fatal_context(ID, MSG, RO)		***MACRO*** &96;uvm_fatal_context `` `uvm_fatal_context(ID, MSG, RO) `` Operates identically to &96;uvm_fatal but requires that the context, or <uvm_report_object>, in which the message is printed be explicitly supplied as a macro argument.
uvm_fatal_context_begin(ID, MSG, RO, RM =__uvm_msg)		***MACRO*** &96;uvm_fatal_context_begin `` `uvm_fatal_context_begin(ID, MSG,␣ ↪RO, RM = __uvm_msg) ``
uvm_fatal_context_end		***MACRO*** &96;uvm_fatal_context_end `` `uvm_fatal_context_end `` This macro pair operates identically to `uvm\\_fatal\\_begin](macro-fd51dfdb)/[`*uvm_fatal_end*, but requires that the context, or <uvm_report_object> in which the message is printed be explicitly supplied as a macro argument.

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_fatal_end		***MACRO*** &96;uvm_fatal_end This macro pair operates identically to `uvm\\_info\\_begin](macro-95076a0b)/[`*uvm_info_end* with exception that the message severity is <UVM_FATAL> and has no verbosity threshold. `` `uvm_fatal_end `` The usage shown in *`uvm_info_end* works identically for this pair.
uvm_field_aa_int_byte(ARG, FLAG)		***MACRO*** &96;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(ARG, FLAG)		***MACRO***
		
		&96;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_enumkey(KEY, ARG, FLAG)		***MACRO***
		
		&96;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_enumkey(KEY, ARG, ``
		`↪FLAG)`
		
		KEY is the enumeration type of the 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.

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_field_aa_int_int(ARG, FLAG)		***MACRO*** &96;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(ARG, FLAG)		***MACRO*** &96;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(ARG, FLAG)		***MACRO*** &96;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(ARG, FLAG)		***MACRO*** &96;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.

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_field_aa_int_key(KEY, ARG, FLAG)		***MACRO*** `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(KEY,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_longint(ARG, FLAG)		***MACRO*** `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_un-signed(ARG, FLAG)		***MACRO***
		
		 `uvm_field_aa_int_longint_unsigned
		
		 Implements the data operations for an associative array of integral types indexed by the *longint un-signed* data type.
		
		 `` `uvm_field_aa_int_longint_ ``
		`` ↪unsigned(ARG,FLAG) ``
		
		 ARG is the name of a property that is an associa-tive 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_shortint(ARG, FLAG)		***MACRO***
		
		 `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 associa-tive array of integrals with *shortint* key, and *FLAG* is a bitwise OR of one or more flag settings as de-scribed in <Field Macros> above.

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_field_aa_int_shortint_unsigned(ARG, FLAG)		***MACRO*** &96;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_string(ARG, FLAG)		***MACRO*** &96;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_int(ARG, FLAG)		***MACRO*** &96;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_object_string(ARG, FLAG)		***MACRO*** &96;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.

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_field_aa_string_string(ARG, FLAG)		***MACRO*** &96;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_array_enum(T, ARG, FLAG)		***MACRO*** &96;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_array_int(ARG, FLAG)		***MACRO*** &96;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(ARG, FLAG)		***MACRO*** &96;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(ARG, FLAG)		***MACRO*** &96;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.

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_field_enum(T, ARG, FLAG)		***MACRO*** &96;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_event(ARG, FLAG)		***MACRO*** &96;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_int(ARG, FLAG)		***MACRO*** &96;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(ARG, FLAG)		***MACRO*** &96;uvm_field_object Implements the data operations for a <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_queue_enum(T, ARG, FLAG)		***MACRO*** &96;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.

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_field_queue_int(ARG, FLAG)		***MACRO*** &96;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(ARG, FLAG)		***MACRO*** &96;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(ARG, FLAG)		***MACRO*** &96;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_real(ARG, FLAG)		***MACRO*** &96;uvm_field_real Implements the data operations for any real property. `` `uvm_field_real(ARG,FLAG) `` *ARG* is an real property of the class, and *FLAG* is a bitwise OR of one or more flag settings as described in <Field Macros> above.
uvm_field_sarray_enum(T, ARG, FLAG)		***MACRO*** &96;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.

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_field_sarray_int(ARG, FLAG)		***MACRO*** &96;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(ARG, FLAG)		***MACRO*** &96;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(ARG, FLAG)		***MACRO*** &96;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_string(ARG, FLAG)		***MACRO*** &96;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.

continues on next page

Table 3 – continued from previous page

Name	Value	Description

uvm_field_utils_begin(T)		***MACRO***
		
		 &96;uvm_field_utils_end
		
		 These macros form a block in which &96;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 nor implement the *get_type_name* and *create* methods. 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_field_utils_end		
uvm_file	__FILE__	
uvm_get_imp_decl(SFX)		***MACRO*** &96;uvm_get_imp_decl `` `uvm_get_imp_decl(SFX) `` 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_get_peek_imp_decl(SFX)		***MACRO*** &96;uvm_get_peek_imp_decl `` `uvm_get_peek_imp_decl(SFX) `` 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_info(ID, MSG, VERBOSITY)		***MACRO*** &96;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_info(ID, MSG, VERBOSITY) ``

uvm_info_begin(ID, MSG, VERBOSITY, RM =__uvm_msg)		***MACRO***
		
		 &96;uvm_info_begin
		
		 `` `uvm_info_begin(ID, MSG, VERBOSITY, ``
		 `↪ RM = __uvm_msg)`

continues on next page

Table 3 – continued from previous page

Name	Value	Description

uvm_info_context(ID, MSG, VERBOSITY, RO)		***MACRO***
		
		&96;uvm_info_context
		`` `uvm_info_context(ID, MSG,␣ ``
		`↪VERBOSITY, RO)`
		Operates identically to &96;uvm_info but requires that the context, or <uvm_report_object>, in which the message is printed be explicitly supplied as a macro argument.

uvm_info_context_begin(ID, MSG, VERBOSITY, RO, RM =__uvm_msg)		***MACRO***
		
		&96;uvm_info_context_begin
		`` `uvm_info_context_begin(ID, MSG,␣ ``
		`↪UVM_NONE, RO, RM = __uvm_msg)`

uvm_info_context_end		***MACRO*** &96;uvm_info_context_end `` `uvm_info_context_end `` This macro pair operates identically to `uvm\\_info\\_begin](macro-95076a0b)/[`*uvm_info_end*, but requires that the context, or <uvm_report_object> in which the message is printed be explicitly supplied as a macro argument.

uvm_info_end		***MACRO***
		
		&96;uvm_info_end
		
		This macro pair provides the ability to add elements to messages.
		`` `uvm_info_end ``
		Example usage is shown here.
		`...`
		`task my_task();`
		`    ...`
		`` `uvm_info_begin("MY_ID", "This␣ ``
		`↪is my message...", UVM_LOW)`
		`` `uvm_message_add_tag("my_color ``
		`↪", "red")`
		`` `uvm_message_add_int(my_int,␣ ``
		`↪UVM_DEC)`
		`` `uvm_message_add_string(my_ ``
		`↪string)`
		`` `uvm_message_add_object(my_ ``
		`↪obj)`
		`` `uvm_info_end ``
		`    ...`
		`endtask`

uvm_line	__LINE__	

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_master_imp_decl(SFX)		***MACRO*** &96;uvm_master_imp_decl `` `uvm_master_imp_decl(SFX) `` 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_message_add_int(VAR, RADIX, LABEL ="", ACTION =(UVM_LOG|UVM_RM_-RECORD))		***MACRO*** &96;uvm_message_add_int `` `uvm_message_add_int(VAR, RADIX, ␣ ↪LABEL = "", ACTION=(UVM_LOG|UVM_ ↪RM_RECORD)) ``
uvm_message_add_object(VAR, LABEL ="", ACTION =(UVM_-LOG|UVM_RM_RECORD))		***MACRO*** &96;uvm_message_add_object These macros allow the user to provide elements that are associated with <uvm_report_message>s. Separate macros are provided such that the user can supply arbitrary string/string pairs using `uvm\\_message\\_add\\_tag](macro-1b7aa271), integral types along with a radix using` *[uvm_message_add_int*, string using `uvm\\_message\\_add\\_string](macro-c37af9da) and <uvm\_object>s using [`*uvm_message_add_object*. `` `uvm_message_add_object(VAR, LABEL␣ ↪= "", ACTION=(UVM_LOG|UVM_RM_ ↪RECORD)) `` Example usage is shown in *`uvm_info_end*.
uvm_message_add_string(VAR, LABEL ="", ACTION =(UVM_-LOG|UVM_RM_RECORD))		***MACRO*** &96;uvm_message_add_string `` `uvm_message_add_string(VAR, LABEL␣ ↪= "", ACTION=(UVM_LOG|UVM_RM_ ↪RECORD)) ``
uvm_message_add_tag(NAME, VALUE, ACTION =(UVM_-LOG|UVM_RM_RECORD))		***MACRO*** &96;uvm_message_add_tag `` `uvm_message_add_tag(NAME, VALUE,␣ ↪ACTION=(UVM_LOG|UVM_RM_RECORD)) ``
uvm_message_begin(SEVERITY, ID, MSG, VERBOSITY, FILE, LINE, RM)		MACRO- &96;uvm_message_begin Undocumented. Library internal use.
uvm_message_context_begin(SEVERITY, ID, MSG, VERBOSITY, FILE, LINE, RO, RM)		MACRO- &96;uvm_message_context_begin Undocumented. Library internal use.
uvm_message_context_end		MACRO- &96;uvm_message_context_end Undocumented. Library internal use.

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_message_end		MACRO- &96;uvm_message_end Undocumented. Library internal use.
uvm_new_func		uvm_new_func
uvm_non_blocking_transport_imp_-decl(SFX)		
uvm_nonblocking_get_imp_-decl(SFX)		***MACRO*** &96;uvm_nonblocking_get_imp_decl `` `uvm_nonblocking_get_imp_decl(SFX) `` Define the class uvm_nonblocking_get_impSFX for providing non-blocking get implementations. *SFX* is the suffix for the new class type.

uvm_nonblocking_get_peek_imp_-decl(SFX)		***MACRO***
		
		&96;uvm_nonblocking_get_peek_imp_decl
		`` `uvm_nonblocking_get_peek_imp_ ``
		`↪decl(SFX)`
		Define the class uvm_nonblocking_get_peek_impSFX for providing non-blocking get_peek implementation.

uvm_nonblocking_master_imp_-decl(SFX)		***MACRO***
		
		&96;uvm_nonblocking_master_imp_decl
		`` `uvm_nonblocking_master_imp_ ``
		`↪decl(SFX)`
		Define the class uvm_nonblocking_master_impSFX for providing the non-blocking master implementation.

uvm_nonblocking_peek_imp_-decl(SFX)		***MACRO*** &96;uvm_nonblocking_peek_imp_decl `` `uvm_nonblocking_peek_imp_decl(SFX) `` Define the class uvm_nonblocking_peek_impSFX for providing non-blocking peek implementations. *SFX* is the suffix for the new class type.
uvm_nonblocking_put_imp_-decl(SFX)		***MACRO*** &96;uvm_nonblocking_put_imp_decl `` `uvm_nonblocking_put_imp_decl(SFX) `` Define the class uvm_nonblocking_put_impSFX for providing non-blocking put implementations. *SFX* is the suffix for the new class type.

uvm_nonblocking_slave_imp_-decl(SFX)		***MACRO***
		
		&96;uvm_nonblocking_slave_imp_decl
		`` `uvm_nonblocking_slave_imp_ ``
		`↪decl(SFX)`
		Define the class uvm_nonblocking_slave_impSFX for providing the non-blocking slave implementation.

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_nonblocking_transport_imp_-decl(SFX)		***MACRO*** &96;uvm_nonblocking_transport_imp_decl `` `uvm_nonblocking_transport_imp_ ↪decl(SFX) `` Define the class uvm_nonblocking_transport_impSFX for providing the non-blocking transport implementation.
uvm_object_param_utils(T)		
uvm_object_param_utils_begin(T)		
uvm_object_registry(T, S)		***MACRO*** &96;uvm_object_registry Register a uvm_object-based class with the factory `` `uvm_object_registry(T,S) `` Registers a uvm_object-based class *T* and lookup string *S* with the factory. *S* typically is the name of the class in quotes. The *`uvm_object_utils* family of macros uses this macro.

continues on next page

Table 3 – continued from previous page

Name	Value	Description

uvm_object_utils(T)		***MACRO*** &96;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 it 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 &96;uvm_field** macros can be placed. The block must be terminated by &96;uvm_object_utils_end.

uvm_object_utils_begin(T)		
uvm_object_utils_end		

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_pack_array(VAR)		***Macro*** &96;uvm_pack_array Pack a dynamic array without having to also specify the bit size of its elements. Array size must be non-zero. `` `uvm_pack_array(VAR) ``
uvm_pack_arrayN(VAR, SIZE)		***Macro*** &96;uvm_pack_arrayN Pack a dynamic array of integrals. `` `uvm_pack_arrayN(VAR,SIZE) ``
uvm_pack_enum(VAR)		***Macro*** &96;uvm_pack_enum Pack an enumeration value. Packing does not require its type be specified. `` `uvm_pack_enum(VAR) ``
uvm_pack_enumN(VAR, SIZE)		***Macro*** &96;uvm_pack_enumN Pack an integral variable. `` `uvm_pack_enumN(VAR,SIZE) ``
uvm_pack_int(VAR)		***Macro*** &96;uvm_pack_int Pack an integral variable without having to also specify the bit size. `` `uvm_pack_int(VAR) ``
uvm_pack_intN(VAR, SIZE)		***Macro*** &96;uvm_pack_intN Pack an integral variable. `` `uvm_pack_intN(VAR,SIZE) ``
uvm_pack_queue(VAR)		***Macro*** &96;uvm_pack_queue Pack a queue without having to also specify the bit size of its elements. Queue must not be empty. `` `uvm_pack_queue(VAR) ``

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_pack_queueN(VAR, SIZE)		***Macro*** &96;uvm_pack_queueN Pack a queue of integrals. `` `uvm_pack_queueN(VAR,SIZE) ``
uvm_pack_real(VAR)		***Macro*** &96;uvm_pack_real Pack a variable of type real. `` `uvm_pack_real(VAR) ``
uvm_pack_sarray(VAR)		***Macro*** &96;uvm_pack_sarray Pack a static array without having to also specify the bit size of its elements. `` `uvm_pack_sarray(VAR) ``
uvm_pack_sarrayN(VAR, SIZE)		***Macro*** &96;uvm_pack_sarrayN Pack a static array of integrals. `` `uvm_pack_sarray(VAR,SIZE) ``
uvm_pack_string(VAR)		***Macro*** &96;uvm_pack_string Pack a string variable. `` `uvm_pack_string(VAR) ``
uvm_package(PKG)		MACRO- &96;uvm_package Use &96;uvm_package to define the SV package and to create a bogus type to help automate triggering the static initializers of the package. Use uvm_end_package to endpackage.
uvm_peek_imp_decl(SFX)		***MACRO*** &96;uvm_peek_imp_decl `` `uvm_peek_imp_decl(SFX) `` 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_print_aa_int_key4(KEY, F, R, P)		
uvm_print_aa_int_object(F, FLAG)		
uvm_print_aa_int_object3(F, P, FLAG)		
uvm_print_aa_string_int(F)		Associative array printing methods
uvm_print_aa_string_int3(F, R, P)		

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_print_aa_string_object(F, FLAG)		
uvm_print_aa_string_object3(F, P, FLAG)		
uvm_print_aa_string_string(F)		
uvm_print_aa_string_string2(F, P)		
uvm_print_array_int(F, R)		uvm_print_array*
uvm_print_array_int3(F, R, P)		
uvm_print_array_object(F, FLAG)		
uvm_print_array_object3(F, P, FLAG)		
uvm_print_array_string(F)		
uvm_print_array_string2(F, P)		
uvm_print_enum(T, F, NM, P)		uvm_print_enum
uvm_print_int(F, R)		uvm_print_int*
uvm_print_int3(F, R, P)		
uvm_print_int4(F, R, NM, P)		
uvm_print_object(F)		uvm_print_object*
uvm_print_object2(F, P)		
uvm_print_object_qda4(F, P, T, FLAG)		
uvm_print_object_queue(F, FLAG)		
uvm_print_object_queue3(F, P, FLAG)		
uvm_print_qda_enum(F, P, T, ET)		
uvm_print_qda_int4(F, R, P, T)		
uvm_print_queue_int(F, R)		
uvm_print_queue_int3(F, R, P)		
uvm_print_sarray_int3(F, R, P)		uvm_print_sarray*
uvm_print_sarray_object(F, FLAG)		
uvm_print_sarray_object3(F, P, FLAG)		
uvm_print_sarray_string2(F, P)		
uvm_print_string(F)		uvm_print_string*
uvm_print_string2(F, P)		
uvm_print_string_qda3(F, P, T)		
uvm_print_string_queue(F)		
uvm_print_string_queue2(F, P)		
uvm_put_imp_decl(SFX)		***MACRO*** &96;uvm_put_imp_decl `` `uvm_put_imp_decl(SFX) `` 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_rand_send(SEQ_OR_ITEM)		***MACRO*** &96;uvm_rand_send `` `uvm_rand_send(SEQ_OR_ITEM) `` This macro processes the item or sequence that has been already been allocated (possibly with &96;uvm_create). The processing is done with randomization. Essentially, an &96;uvm_do without the create.

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_rand_send_pri(SEQ_OR_-ITEM, PRIORITY)		***MACRO*** &96;uvm_rand_send_pri `` `uvm_rand_send_pri(SEQ_OR_ITEM, PRIORITY) `` This is the same as &96;uvm_rand_send except that the sequence item or sequence is executed with the priority specified in the argument.
uvm_rand_send_pri_with(SEQ_-OR_ITEM, PRIORITY, CON-STRAINTS)		***MACRO*** &96;uvm_rand_send_pri_with `` `uvm_rand_send_pri_with(SEQ_OR_ITEM, PRIORITY, CONSTRAINTS) `` This is the same as &96;uvm_rand_send_pri except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.
uvm_rand_send_with(SEQ_OR_-ITEM, CONSTRAINTS)		***MACRO*** &96;uvm_rand_send_with `` `uvm_rand_send_with(SEQ_OR_ITEM, CONSTRAINTS) `` This is the same as &96;uvm_rand_send except that the given constraint block is applied to the item or sequence in a randomize with statement before execution.

uvm_record_field(NAME, VALUE)		***Macro***
		
		&96;uvm_record_field
		
		Macro for recording arbitrary name-value pairs into a transaction recording database. Requires a valid transaction handle, as provided by the <uvm_transaction::begin_tr> and <uvm_component::begin_tr> methods.
		
		`` `uvm_record_field(NAME, VALUE) ``
		
		The default implementation will pass the name/-value pair to *`uvm_record_attribute* if enabled, otherwise the information will be passed to <uvm_recorder::record_generic>, with the *VALUE* being converted to a string using "%p" notation.
		
		`recorder.record_generic(NAME, $sformatf("%p",VALUE));`

continues on next page

Table 3 – continued from previous page

Name	Value	Description

uvm_register_cb(T, CB)		***MACRO***
		
		&96;uvm_register_cb
		
		`` `uvm_register_cb(T,CB) ``
		
		Registers the given *CB* callback type with the given *T* object type. If a type-callback pair is not registered then a warning is issued if an attempt is made to use the pair (add, delete, etc.).
		
		The registration will typically occur in the component that executes the given type of callback. For instance:
		
		`virtual class mycb extends uvm_`
		`↪callback;`
		`  virtual function void doit();`
		`endclass`
		
		`class my_comp extends uvm_`
		`↪component;`
		`` `uvm_register_cb(my_comp,mycb) ``
		`  ...`
		`  task run_phase(uvm_phase phase);`
		`    ...`
		`` `uvm_do_callbacks(my_comp,␣ ``
		`↪mycb, doit())`
		`  endtask`
		`endclass`

uvm_send(SEQ_OR_ITEM)		***MACRO*** &96;uvm_send `` `uvm_send(SEQ_OR_ITEM) `` This macro processes the item or sequence that has been created using &96;uvm_create. The processing is done without randomization. Essentially, an &96;uvm_do without the create or randomization.

uvm_send_pri(SEQ_OR_ITEM, PRIORITY)		***MACRO***
		
		&96;uvm_send_pri
		
		`` `uvm_send_pri(SEQ_OR_ITEM,␣ ``
		`↪PRIORITY)`
		
		This is the same as &96;uvm_send except that the sequence item or sequence is executed with the priority specified in the argument.

uvm_sequence_library_package(PKG_NAME)		MACRO- &96;uvm_sequence_library_package This macro is used to trigger static initializers in packages. &96;uvm_package creates a bogus type which gets referred to by uvm_sequence_library_package to make a package-based variable of the bogus type.

continues on next page

Table 3 – continued from previous page

Name	Value	Description

uvm_sequence_library_utils(TYPE)		***MACRO***
		
		&96;uvm_sequence_library_utils
		
		`` `uvm_sequence_library_utils(TYPE) ``
		
		Declares the infrastructure needed to define extensions to the <uvm_sequence_library> class. You define new sequence library subtypes to statically specify sequence membership from within sequence definitions. See also <`uvm_add_to_sequence_library> for more information.
		
		`typedef simple_seq_lib uvm_sequence_library #(simple_item);`
		
		`class simple_seq_lib_RST extends simple_seq_lib;`
		
		`` `uvm_object_utils(simple_seq_lib_RST) ``
		
		`` `uvm_sequence_library_utils(simple_seq_lib_RST) ``
		
		`function new(string name="");`
		`super.new(name);`
		`endfunction`
		
		`endclass`
		
		Each library, itself a sequence, can then be started independently on different sequencers or in different phases of the same sequencer. See <uvm_sequencer_base::start_phase_sequence> for information on starting default sequences.

continues on next page

Table 3 – continued from previous page

Name	Value	Description

uvm_sequence_utils(TYPE_NAME, SEQUENCER)		MACRO- &96;uvm_sequence_utils
		
		The sequence macros can be used in non-parameterized <uvm_sequence (REQ, RSP)> extensions to pre-register the sequence with a given <uvm_sequencer (REQ, RSP)> type.
		
		For sequences that do not use any &96;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 &96;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 &96;uvm_sequence_utils[_begin] for non-parameterized classes and &96;uvm_sequence_param_utils[_begin] for parameterized classes.

uvm_sequence_utils_begin(TYPE_-NAME, SEQUENCER)		MACRO- &96;uvm_sequence_utils_begin
uvm_sequence_utils_end		MACRO- &96;uvm_sequence_utils_end
uvm_sequencer_param_-utils(TYPE_NAME)		MACRO- &96;uvm_sequencer_param_utils
uvm_sequencer_param_utils_be-gin(TYPE_NAME)		MACRO- &96;uvm_sequencer_param_utils_begin
uvm_sequencer_utils(TYPE_-NAME)		MACRO- &96;uvm_sequencer_utils
uvm_sequencer_utils_begin(TYPE_-NAME)		MACRO- &96;uvm_sequencer_utils_begin

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_sequencer_utils_end		MACRO- &96;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 &96;uvm_sequencer_utils(SQR_TYPE_NAME) For simple sequencers, with field macros &96;uvm_sequencer_utils_begin(SQR_TYPE_NAME) &96;uvm_field_* macros here &96;uvm_sequencer_utils_end For parameterized sequencers, no field macros &96;uvm_sequencer_param_utils(SQR_TYPE_NAME) For parameterized sequencers, with field macros &96;uvm_sequencer_param_utils_begin(SQR_TYPE_NAME) &96;uvm_field_* macros here &96;uvm_sequencer_utils_end The sequencer-specific macros perform the same function as the set of &96;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 &96;uvm_sequencer_utils[_begin] for non-parameterized classes and &96;uvm_sequencer_param_utils[_begin] for parameterized classes.

continues on next page

Table 3 – continued from previous page

Name	Value	Description

uvm_set_super_type(T, ST)		***MACRO***
		
		 &96;uvm_set_super_type
		
		 `` `uvm_set_super_type(T,ST) ``
		 Defines the super type of *T* to be *ST* . This allows for derived class objects to inherit typewide callbacks that are registered with the base class.
		
		 The registration will typically occur in the component that executes the given type of callback. For instance:
		 `virtual class mycb extend uvm_`
		`↪callback;`
		`  virtual function void doit();`
		`endclass`
		
		`class my_comp extends uvm_`
		`↪component;`
		`` `uvm_register_cb(my_comp,mycb) ``
		`  ...`
		`  task run_phase(uvm_phase phase);`
		`    ...`
		`` `uvm_do_callbacks(my_comp,␣ ``
		`↪mycb, doit())`
		`  endtask`
		`endclass`
		
		`class my_derived_comp extends my_`
		`↪comp;`
		`` `uvm_set_super_type(my_derived_ ``
		`↪comp,my_comp)`
		`  ...`
		`  task run_phase(uvm_phase phase);`
		`    ...`
		`` `uvm_do_callbacks(my_comp,␣ ``
		`↪mycb, doit())`
		`  endtask`
		`endclass`

uvm_slave_imp_decl(SFX)		***MACRO*** &96;uvm_slave_imp_decl `` `uvm_slave_imp_decl(SFX) `` 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_transport_imp_decl(SFX)		***MACRO*** &96;uvm_transport_imp_decl `` `uvm_transport_imp_decl(SFX) `` Define the class uvm_transport_impSFX for providing both blocking and non-blocking transport implementations. *SFX* is the suffix for the new class type.

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_typename(X)	$typename(X)	
uvm_unpack_array(VAR)		***Macro*** &96;uvm_unpack_array Unpack a dynamic array without having to also specify the bit size of its elements. Array size must be non-zero. `` `uvm_unpack_array(VAR) ``
uvm_unpack_arrayN(VAR, SIZE)		***Macro*** &96;uvm_unpack_arrayN Unpack into a dynamic array of integrals. `` `uvm_unpack_arrayN(VAR,SIZE) ``
uvm_unpack_enum(VAR, TYPE)		***Macro*** &96;uvm_unpack_enum Unpack an enumeration value, which requires its type be specified. `` `uvm_unpack_enum(VAR,TYPE) ``
uvm_unpack_enumN(VAR, SIZE, TYPE)		***Macro*** &96;uvm_unpack_enumN Unpack enum of type *TYPE* into *VAR* . &96;uvm_unpack_enumN(VAR, SIZE, TYPE)
uvm_unpack_int(VAR)		***Macro*** &96;uvm_unpack_int Unpack an integral variable without having to also specify the bit size. `` `uvm_unpack_int(VAR) ``
uvm_unpack_intN(VAR, SIZE)		***Macro*** &96;uvm_unpack_intN Unpack into an integral variable. `` `uvm_unpack_intN(VAR,SIZE) ``
uvm_unpack_queue(VAR)		***Macro*** &96;uvm_unpack_queue Unpack a queue without having to also specify the bit size of its elements. Queue must not be empty. `` `uvm_unpack_queue(VAR) ``

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_unpack_queueN(VAR, SIZE)		***Macro*** &96;uvm_unpack_queueN Unpack into a queue of integrals. `` `uvm_unpack_queue(VAR,SIZE) ``
uvm_unpack_real(VAR)		***Macro*** &96;uvm_unpack_real Unpack a variable of type real. `` `uvm_unpack_real(VAR) ``
uvm_unpack_sarray(VAR)		***Macro*** &96;uvm_unpack_sarray Unpack a static array without having to also specify the bit size of its elements. `` `uvm_unpack_sarray(VAR) ``
uvm_unpack_sarrayN(VAR, SIZE)		***Macro*** &96;uvm_unpack_sarrayN Unpack a static (fixed) array of integrals. `` `uvm_unpack_sarrayN(VAR,SIZE) ``
uvm_unpack_string(VAR)		***Macro*** &96;uvm_unpack_string Unpack a string variable. `` `uvm_unpack_string(VAR) ``
uvm_update_sequence_lib		MACRO- &96;uvm_update_sequence_lib This macro populates the instance-specific sequence library for a sequencer. It should be invoked inside the sequencers constructor.
uvm_update_sequence_lib_and_item(USER_ITEM)		MACRO- &96;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_user_bottomup_phase(PHASE, COMP, PREFIX)		
uvm_user_task_phase(PHASE, COMP, PREFIX)		
uvm_user_topdown_phase(PHASE, COMP, PREFIX)		

continues on next page

Table 3 – continued from previous page

Name	Value	Description
uvm_warning(ID, MSG)		***MACRO*** &96;uvm_warning Calls uvm_report_warning with a verbosity of UVM_NONE. The message cannot 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_warning(ID, MSG) ``
uvm_warning_begin(ID, MSG, RM =__uvm_msg)		***MACRO*** &96;uvm_warning_begin `` `uvm_warning_begin(ID, MSG, RM = __ ↪uvm_msg) ``
uvm_warning_context(ID, MSG, RO)		***MACRO*** &96;uvm_warning_context `` `uvm_warning_context(ID, MSG, RO) `` Operates identically to &96;uvm_warning but requires that the context, or <uvm_report_object>, in which the message is printed be explicitly supplied as a macro argument.
uvm_warning_context_begin(ID, MSG, RO, RM =__uvm_msg)		***MACRO*** &96;uvm_warning_context_begin `` `uvm_warning_context_begin(ID, MSG, ↪ RO, RM = __uvm_msg) ``
uvm_warning_context_end		***MACRO*** &96;uvm_warning_context_end `` `uvm_warning_context_end `` This macro pair operates identically to `uvm\\_warning\\_begin](macro-5c5c3133)/[`*uvm_warning_end*, but requires that the context, or <uvm_report_object> in which the message is printed be explicitly supplied as a macro argument.
uvm_warning_end		***MACRO*** &96;uvm_warning_end This macro pair operates identically to `uvm\\_info\\_begin](macro-95076a0b)/[`*uvm_info_end* with exception that the message severity is <UVM_WARNING> and has no verbosity threshold. `` `uvm_warning_end `` The usage shown in *`uvm_info_end* works identically for this pair.