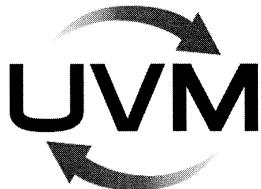


UVM

GOLDEN

REFERENCE GUIDE

A concise guide to the Universal Verification Methodology



UVM Golden Reference Guide

Second Edition, December 2013

Copyright © 2013 by Doulos Ltd. All rights reserved.

The information contained herein is the property of Doulos Ltd and is supplied without liability for errors or omissions. No part may be used, stored, transmitted or reproduced in any form or medium without the written permission of Doulos Ltd.

Doulos® is a registered trademark of Doulos Ltd.

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

All other trademarks are acknowledged as the property of their respective holders.

First published by Doulos 2011.

Doulos
Church Hatch
22 Market Place
Ringwood
Hampshire
BH24 1AW
UK
Tel +44 (0) 1425 471223
Fax +44 (0) 1425 471573
Email: info@doulos.com

Doulos
2055 Gateway Place
Suite 220
San Jose
CA 95110
USA
+1-888-GO DOULOS
+1-408-762-2246
info.usa@doulos.com

Web: <http://www.doulos.com>

Contents

<i>Section</i>	<i>Page</i>
Preface	4
Using This Guide	5
A Brief Introduction To UVM	6
Finding What You Need in this Guide	8
Alphabetical Reference	14
Index	297

Preface

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

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

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

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

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

Using This Guide

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

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

Organization

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

Except in the index, the alphabetical ordering ignores the prefix `uvm_`. So you will find Field Macros between the articles `uvm_factory` and `uvm_heartbeat`.

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

The Index

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

Methods and Members

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

A Brief Introduction to UVM

Background

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

The SystemC TLM-1 and TLM-2.0 libraries define a transport layer for transaction level models. Mentor Graphics' Advanced Verification Methodology (AVM) used SystemVerilog equivalents of the TLM-1 ports, channels and interfaces to communicate between verification components. Support for TLM-2.0 was introduced in UVM.

The Verification Methodology Manual (VMM) was co-authored by Synopsys and ARM and enables the creation of robust, reusable and scalable verification environments using SystemVerilog.

URM and AVM were joined together to form the Open Verification Methodology (OVM). OVM combines the classes from AVM and URM. It is backwards compatible with both.

UVM was developed by Accellera in collaboration with Cadence, Mentor Graphics and Synopsys. Version 1.0 and, subsequently, version 1.1 were released in 2011, although an early access version (UVM 1.0EA) was made available in 2010. UVM is based on OVM 2.1.1 and adds a Register Layer based on the Register Abstraction Layer (RAL) from VMM and TLM-2.0 interfaces, based on the SystemC standard. UVM versions 1.1a through 1.1c provide bug fixes with only a few minor changes.

Transaction-level Modeling

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

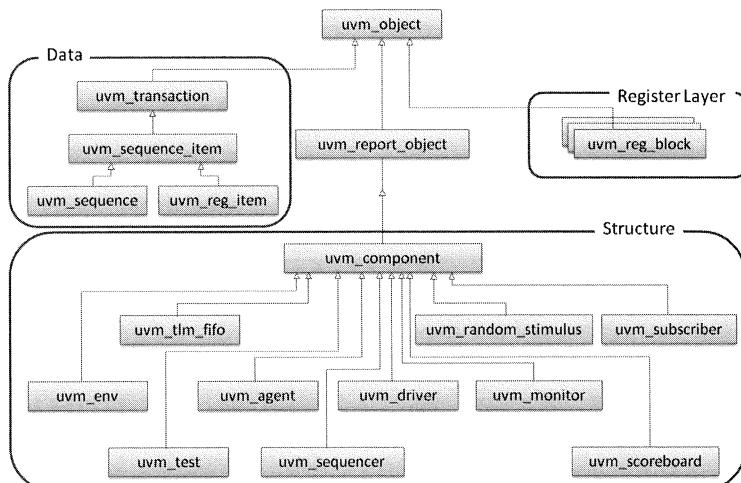
Transaction level modeling is a means of accelerating simulation by abstracting the way communication is modeled. Whereas an HDL simulator models communication by having a separate event for each pin wiggle, a transaction level model works by replacing a bunch of related pin wiggles by a single transaction.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

UVM

UVM is implemented entirely in SystemVerilog so it should work on any simulator that supports the full IEEE 1800-2009 standard.

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

UVM Class Hierarchy



The main UVM classes form a hierarchy as shown here. The `uvm_object` class is the base class for all other UVM classes.

User defined transaction classes should be derived from `uvm_sequence_item`.

TLM channels such as `uvm_tlm_fifo` are derived from `uvm_report_object` so include the ability to print their state. There are also implementations of the SystemC TLM-1 and TLM-2.0 interface classes (not shown here) that are inherited by TLM channels.

The `uvm_component` class, which also derives from `uvm_report_object`, is for user-defined verification components. It has a `run_phase` task that is automatically invoked at the start of a simulation.

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

Finding What You Need in This Guide

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

Designing Transaction Data

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

Creating Problem-Specific Testbench Components

The core of a testbench is the set of *testbench components* that will manipulate the transaction data. Some of these components need a direct connection to signals in HDL modules representing the device-under-test (DUT) and its supporting structures. Other components operate at a higher level of abstraction and work only with transaction data. All the components you create should be represented by classes derived from **uvm_component**.

Components need to pass data to and from other components. In UVM this is achieved by providing the components with suitable **ports and exports** (TLM-1) or **sockets** (TLM-2.0) through which they can communicate with one another. (Note that all the standard components described below use TLM-1 interfaces; TLM-2.0 interfaces are provided for communication between SystemVerilog and SystemC components that have TLM-2.0 interfaces.) Components never need to know about neighboring components to which they are connected; instead, components send and receive data by means of calls to methods in their ports. The set of methods implemented by ports is known as the **TLM-1 Interfaces** (**uvm_tlm_if** in TLM-2.0). This is the fundamental principle of transaction level modeling: one component calls a TLM interface method in its *port* and, thanks to the connection mechanism, the corresponding method is automatically called in a different component's *export*.

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

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

Register Model

UVM includes a **Register Layer**, which enables an abstract model of the memory-mapped registers and memories of a device to be created. A register model is often created using a **Memory Generator**. The register model includes a rich API for accessing a device's registers and memories. Both frontdoor (using the normal bus interface) and backdoor access is supported.

The Register Layer also includes a number of pre-built **Register and Memory Sequences**, so that a device's registers and memories can be tested with minimal new code having to be written.

Choosing and Customizing Built-in UVM Components

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

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

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

Constructing the Testbench: Phases and the Factory

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

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

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

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

Structured Random Stimulus

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

Writing and Executing a Test

Having designed a test environment it is necessary to run it. UVM provides the **uvm_test** class as a base class for user-defined top-level tests, defining the specific usage of a test environment for a single test run or family of runs. **uvm_root** provides a mechanism for encapsulating the whole of your UVM testbench and top-level test.

UVM provides an objections mechanism using **uvm_objection** to manage the **End of Test**.

Configuration

When a test begins, there is an opportunity for user code in the test class to provide configuration information that will control operation of various parts of the testbench. The mechanisms that allow a test to set this configuration information and, later, allow each part of the testbench to retrieve relevant configuration data, is known as **Configuration**. For configuration to work

correctly it is necessary for user-written components to respect a set of conventions concerning the definition of data members in each class. Data members that can be configured are known as *fields*, and must be set up using **Field Macros** provided as part of UVM.

One of the most commonly used and important aspects of configuration is to choose which HDL instances and signals are to be accessed from each part of the testbench. Although UVM does not provide any specific mechanisms for this, the conventional approach is to use the configuration mechanism.

Reporting and Text Output

As it runs, a testbench will generate large amounts of textual information. To allow for flexible control and redirection of this text output, a comprehensive set of reporting facilities is provided. These reporting features are most easily accessed through four reporting macros (See **Reporting**). Detailed control over text output formatting is achieved using **uvm_printer** and **uvm_printer_knobs**.

Alphabetical Reference Section

vuvm_agent

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

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

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

Declaration

```
class uvm_agent extends uvm_component;  
typedef enum bit { UVM_PASSIVE=0, UVM_ACTIVE=1 }  
    uvm_active_passive_enum;
```

Methods

<code>function new(string name, uvm_component parent);</code>	Constructor; mirrors the superclass constructor in <code>uvm_component</code>
<code>virtual function uvm_active_passive_enum get_is_active();</code>	Returns <code>UVM_ACTIVE</code> or <code>UVM_PASSIVE</code>

Members

<code>uvm_active_passive_enum is_active = UVM_ACTIVE;</code>	Controls whether this instance has an active driver.
<code>+uvm_analysis_port #(transaction_class) monitor_ap;</code>	Exposes the monitor sub- component's analysis output to users of the agent.

[†]**Note:** This field is not defined in `uvm_agent`, but should almost always be provided as part of any user extension.

Example

```
class example_agent extends uvm_agent;
example_sequencer #(example_transaction) m_sequencer;
example_driver m_driver;
example_monitor m_monitor;
uvm_analysis_port #(example_transaction) monitor_ap;
virtual dut_if v_dut_if;

virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
m_monitor = example_monitor::type_id::create(
    "m_monitor", this);
monitor_ap = new("monitor_ap", this);
if (get_is_active() == UVM_ACTIVE) begin
    m_sequencer = example_sequencer::type_id::create(
        "m_sequencer", this);
    m_driver = example_driver ::type_id::create(
        "m_driver", this);
end
endfunction: build_phase

virtual function void connect_phase(uvm_phase phase);
m_monitor.monitor_ap.connect(monitor_ap);
... // code to connect monitor's virtual interface

if (get_is_active() == UVM_ACTIVE) begin
    m_driver.seq_item_port.connect(
        m_sequencer.seq_item_export);
    ... // code to connect driver's virtual interface
end
endfunction: connect_phase
...
endclass: example_agent
```

Tips

- An agent should represent a block of “verification IP”, a re-usable component that can be connected to a given DUT interface and then used as part of the UVM test environment. It should be possible to specify a single virtual-interface connection for the agent, and then have the agent’s `build_phase` or `connect_phase` method automatically provide that connection (or appropriate modports of it) to any relevant sub-components such as its driver and monitor.

uvm_agent

- Every active agent should have a sequencer sub-component capable of generating randomized stimulus for its driver sub-component.
- An agent's monitor should collect coverage information.
- Provide your agent with an analysis port that is directly connected to the analysis port of its monitor sub-component. In this way, users of the agent can get analysis output from it without needing to know about its internal structure.
- You will need to create appropriate sequences that the sequencer can use to generate useful stimulus.
- The UVM User Guide recommends two flags to be declared within an agent: `checks_enable` and `coverage_enable`. These flags provide a standard way of configuring the checking and coverage collection.
- The `is_active` property can be set, for example in the test or environment, using:

```
uvm_config_db  
#(uvm_bitstream_t)::set("<path_to_agent>", "",  
"is_active", UVM_ACTIVE);
```

By default, the `is_active` flag is set to `UVM_ACTIVE`.

Gotchas

- `uvm_agent` has no methods or data members of its own, apart from its constructor, its `is_active` flag and what it inherits from `uvm_component`. However, to build a properly-formed agent requires you to follow various methodology guidelines, including the recommendations in this article. In particular, you should always use the `is_active` flag, a configurable means to connect the agent to its target physical interface, and the three standard sub-components (driver, monitor and sequencer). An agent is in effect a piece of verification IP that can be deployed on many different projects; as such it should be designed to be completely self-contained and portable.
- An agent's `is_active` flag must be set using the `uvm_bitstream_t` data type instead of its defined type of `uvm_active_passive_enum`. The agent's build phase uses `get_config_int` to retrieve `is_active`, and `get_config_int` uses `uvm_bitstream_t`.

See also

`uvm_component`; `uvm_driver`; `uvm_monitor`; `uvm_sequencer`

uvm_algorithmic_comparator

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

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

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

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

Declarations

```
class uvm_algorithmic_comparator
  #( type BEFORE = int, type AFTER = int,
    type TRANSFORMER = int )
  extends uvm_component;
```

Methods

<code>function new(</code> <code>string name,</code> <code>uvm_component parent=null,</code> <code>TRANSFORMER transformer=null);</code>	Constructor – "transformer" is the reference-model object whose <code>transform</code> method will be used to predict DUT output values
---	---

Members

<code>uvm_analysis_imp #(BEFORE,...)</code> <code>before_export;</code>	Connect to first transaction stream analysis port, typically monitored from a DUT's input
<code>uvm_analysis_export #(AFTER)</code> <code>after_export;</code>	Connect to second transaction stream analysis port, typically monitored from a DUT's output

uvm_algorithmic_comparator

Code pattern for any user-defined transformer class:

```
class example_transformer;
  ... // member variables to represent internal
       // state of the reference model
  function new(any appropriate arguments);
    ... // initialize the state of the reference
         // model based on the constructor arguments
  endfunction
  function after_class_type transform (
    before_class_type b );
    ... // maintain the state of the reference model
        // based on the new input transaction, and
        // compute and return the expected result
  endfunction
endclass
```

Example

Using `uvm_algorithmic_comparator` within a scoreboard component

```
class cpu_scoreboard extends uvm_scoreboard;

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

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

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

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

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

virtual function void report_phase(uvm_phase phase);
    string txt;
    $sformat(txt, "#matches = %d, #mismatches = %d",
m_comp.m_matches, m_comp.m_mismatches);
    `uvm_info("", txt,UVM_NONE)
endfunction: report_phase
`uvm_component_utils(cpu_scoreboard)
endclass: cpu_scoreboard
```

Tips

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

Gotchas

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

uvm_algorithmic_comparator

See also

[uvm_analysis_port](#); [uvm_analysis_export](#); [uvm_in_order_*_comparator](#)

uvm_analysis_export

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

Declaration

```
class uvm_analysis_export #(type T = int)
  extends uvm_port_base #(uvm_tlm_if_base #(T,T));
```

Methods

function new (string name, uvm_component parent);	Constructor
function void write (input T t);	Called implicitly by connected analysis port's write method, forwards call to connected exports or implementation(s)
virtual function string get_type_name ();	Returns "uvm_analysis_export"
virtual function void connectt (port_type provider);	Connects the analysis export to another analysis export, or to an analysis imp, that implements a subscriber's write functionality

⁷Inherited from *uvm_port_base*

Example

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

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

uvm_analysis_export

```
class custom_subscr_2
  extends uvm_subscriber #(example_transaction);
  ... // code for second custom subscriber

class example_double_subscriber extends uvm_component;
  custom_subscr_1 subscr1;
  custom_subscr_2 subscr2;
  uvm_analysis_export #(example_transaction)
    analysis_export;

  function void build_phase(uvm_phase phase);
    subscr1 = custom_subscr_1::type_id::create(
      "subscr1", this );
    subscr2 = custom_subscr_2::type_id::create(
      "subscr2", this );
    analysis_export = new ( "analysis_export", this );
  endfunction

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

endclass
```

Tips

- Every analysis export must ultimately be connected to a `uvm_analysis_imp` implementation that provides a `write` method. It is possible to connect an analysis port directly to a `uvm_analysis_imp`, but user-written components more commonly have a `uvm_analysis_export` that in its turn is connected either to one or more `uvm_analysis_imp`, or to a `uvm_analysis_export` on a sub-component that is a member of the component.
- An especially useful and common idiom is for a subscriber component to have a `uvm_tlm_analysis_fifo`. The component's `uvm_analysis_export` is then connected to the analysis FIFO's `analysis_export`. In this way, the user has no need to code a `uvm_analysis_imp` explicitly. Transactions from a producer's analysis port are written into the analysis FIFO without blocking. A thread in the user-written component can take transactions from the analysis FIFO's `get` port at its leisure. (Note that `uvm_tlm_analysis_fifo` does not make a copy of transactions written to it. Where it is required to buffer multiple transactions before they are read, the initiator should make and write a

copy of the transaction, otherwise all references in the fifo will point to the same transaction.)

- `uvm_subscriber` provides a convenient base class for user-written subscriber components that observe transactions from exactly one analysis port. In `uvm_subscriber` the necessary arrangement of analysis export and implementation has already been coded, and it is only necessary for the user to override the base class's `write` method in their class derived from `uvm_subscriber`.
- The overall pattern of connection of analysis ports, exports and imps is:
 - A producer of analysis data should write that data to an analysis port.
 - An analysis port can be connected to any number of subscribers (including zero). Each subscriber can be another analysis port on a parent component, or an analysis export or analysis imp on a sibling component.
 - An analysis export can be connected to any number of subscribers. Each subscriber can be an analysis export or an analysis imp on a child component.

Gotchas

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

See also

`uvm_subscriber`; `uvm_analysis_port`; `uvm_tlm_analysis_fifo`

uvm_analysis_port

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

Declaration

```
class uvm_analysis_port #(type T = int)
  extends uvm_port_base #(uvm_tlm_if_base #(T,T));
```

Methods

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

[†]Inherited from *uvm_port_base*

Example

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

Tips

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

block. It has the effect of calling the `write` method in every connected subscriber. If there is no subscriber connected, the method has no effect.

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

Gotchas

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

```
my_transaction_t temp;
$cast(temp, tr.clone());
my_ap.write(temp);
```

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

See also

`uvm_monitor`; `uvm_subscriber`; `uvm_analysis_export`; `uvm_tlm_fifo`

uvm_barrier

A uvm_barrier provides a mechanism for synchronizing between multiple processes. When a process is ready to synchronize, it waits on the barrier, which causes it to block until all other processes are ready. Each barrier has a threshold that determines when enough processes are waiting, and it is time to proceed. A barrier's default threshold is set when calling the constructor. Barriers are useful in a testbench for creating synchronization points between components.

Declaration

```
class uvm_barrier extends uvm_object;
```

Methods

<code>function new(string name = "", int threshold = 0);</code>	Constructor. Default threshold count is 0.
<code>virtual function void cancel();</code>	Removes the wait on the barrier when a process is killed.
<code>virtual function uvm_object create(string name = "");</code>	Creates and returns a barrier object.
<code>virtual function int get_num_waiters();</code>	Returns the number of processes waiting on the barrier.
<code>virtual function int get_threshold();</code>	Returns the barrier's threshold.
<code>virtual function string get_type_name();</code>	Returns the type name of the barrier object.
<code>virtual function void reset(bit wakeup = 1);</code>	Resets the barrier. Waiting processes are awakened if the wakeup bit is set.
<code>virtual function void set_auto_reset(bit value = 1);</code>	When <code>value = 1</code> , the barrier is reset when the threshold is reached. When <code>value = 0</code> , new processes will not block if the threshold is already reached.

virtual function void set_threshold (int threshold);	Sets the threshold for the number of processes that must be waiting until the barrier is reached.
virtual task wait_for() ;	Waits on the barrier until the threshold number of processes is reached.

Also inherited are the standard methods from `uvm_object`.

Example

```
class usb_driver extends uvm_driver#(usb_trans);
  uvm_barrier m_barrier;
  ...
  task post_reset_phase(uvm_phase phase);
    // Wait for the barrier before proceeding
    m_barrier.wait_for();
  endtask
endclass

class spi_driver extends uvm_driver#(spi_trans);
  uvm_barrier m_barrier;
  ...
  task pre_main_phase(uvm_phase phase);
    // Wait for the barrier before proceeding
    m_barrier.wait_for();
  endtask
endclass

class top_env extends uvm_env;
  uvm_barrier m_barrier;
  usb_driver  m_usb_drv;
  spi_driver  m_spi_drv;

  function void build_phase(uvm_phase phase);
    super.build_phase();

    // Create a barrier for 2 processes
    m_barrier = new("m_barrier", 2);

    m_usb_drv =
      usb_driver::type_id::create("m_usb_drv",this);
    m_spi_drv =
      spi_driver::type_id::create("m_spi_drv",this);
  endfunction
endclass
```

uvm_barrier

```
// Pass the barrier object to the drivers
m_usb_drv.m_barrier = m_barrier;
m_spi_drv.m_barrier = m_barrier;
endfunction
endclass
```

Tips

- `uvm_barrier` provides a custom `copy` for copying barriers.
- `uvm_barrier` provides a custom `print` method for displaying the contents of a `uvm_barrier`.
- If a barrier needs canceled, consider using `reset(1)` instead of `cancel` since it clears the barrier and awakens any sleeping processes.

Gotchas

- When setting the threshold, if a smaller threshold value is used than the current threshold, then the barrier is reset and all waiting processes are awakened.
- Calling `cancel` on a barrier only cancels the barrier count and does not awaken sleeping processes.

See also

`uvm_event`; `uvm_objection`; `uvm_heartbeat`

uvm_callback is the base class for user-defined callback classes. (It is also used as the base class for callback classes in UVM, for example uvm_objection_callback.) Typically, the component developer defines an application-specific callback class that extends from this class. The extended class will include one or more virtual methods that represent the hooks available for users to override.

Declaration

```
class uvm_callback extends uvm_object;
```

Methods

function new(string name = "uvm_callback");	Constructor.
function bit callback_mode(int on = -1);	Enable or disable callbacks.
function bit is_enabled();	Returns 1 if the callback is enabled, or 0 if not.
virtual function string get_type_name();	Returns the type name of the callback object.

Example

See Callbacks.

Tips

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.

See also

Callbacks; uvm_callbacks

uvm_callback_iter

The `uvm_callback_iter` class is an iterator class for iterating over callback queues of a specific callback type. The callback iteration macros, ``uvm_do_callbacks` and ``uvm_do_callbacks_exit_on` provide a simple mechanism for iterating through callbacks and executing the callback methods.

Declaration

```
class uvm_callback_iter #(type T = uvm_object,  
                      type CB = uvm_callback);
```

Parameters

<code>type T = uvm_object</code>	The base type of the object with which the callback objects will be registered.
<code>type CB = uvm_callback</code>	The base callback type that will be managed by the interface.

Methods

<code>function new(T obj);</code>	Constructor
<code>function CB first();</code>	Returns the first enabled callback, or <code>null</code> if there are none.
<code>function CB last();</code>	Returns the last enabled callback, or <code>null</code> if there are none.
<code>function CB next();</code>	Returns the next enabled callback, or <code>null</code> if there are none.
<code>function CB prev();</code>	Returns the previous enabled callback, or <code>null</code> if there are none.
<code>function CB get_cb();</code>	Returns the last callback returned by one of the above calls.

Example

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();
```

Tips

Use this class rather than the equivalent iterator methods provided by `uvm_callbacks`.

See also

[Callbacks](#); [uvm_callbacks](#)

Callbacks

Extending functionality of testbench components in an environment is most easily and ideally done in UVM using the built-in factory mechanism. However, UVM also provides another way to change functionality: callbacks.

A *callback* is a method call placed inside a testbench component that can be overridden by other components—typically, a user test case. This provides an easy mechanism for test cases to change the behavior of the verification environment without actually modifying the testbench source code.

A number of UVM classes have built-in callbacks. Component developers can also add callbacks to their own classes. Callbacks can be used to change a transaction's randomization, the generation of data, add additional coverage terms, define custom scoreboard compare functions, inject errors, or perform any test specific modifications needed.

Adding Callbacks to a Component

This consists of four steps. The first three are usually performed by the component developer. the fourth step is usually performed by the user of the component.

Step 1 (component developer) – The first step to adding callbacks into a component is to create an abstract (virtual) base class. In this base class, a skeleton definition of any callback methods need to be defined. The names of the callback functions or tasks are not important, nor is their functionality, since this will be defined later— typically in a test case. Likewise, the arguments to the callback methods are up to the component developer since these will be passed into the callback when the method is invoked.

Step 2 (component developer) – The second step is to register and place the callbacks in the component. Using one of the UVM callback macros, the callback(s) can easily be inserted inside the testbench components.

Step 3 (component developer) – The third step is to create a user defined abstract type for the callbacks by parameterizing `uvm_callbacks` with the component that includes the callbacks (the one that was modified in step 2) and the abstract callback class defined in the first step.

Step 4 (component user) – In the fourth and final step, the specific callback functions can be defined. The ideal place for this is in a test case. Once the callbacks have been defined, the test case can register them with the testbench component and the component will automatically invoke them at the appropriate times, using the callback macros that were included in the component in step 2.

Example

In this example, callbacks are included in the component class `my_driver`, and implemented in a test, `error_injector_test`.

Step 1 – Create an abstract base class with callback functions

```
virtual class driver_cbs extends uvm_callback;

// Constructor
function new(string name = "driver_cbs");
    super.new(name);
endfunction

// Callback method
virtual task trans_received(my_driver drv, trans tr);
    drv.uvm_report_info("callback", "In trans_received()");
endtask

// Callback method
virtual task trans_executed(my_driver drv, trans tr);
endtask
endclass : driver_cbs
```

Step 2 – Register the callback class, add and call the callbacks in the component

```
class my_driver extends uvm_driver#(trans);
    `uvm_register_cb(my_driver,driver_cbs) // Register callbacks
    `uvm_component_utils(my_driver)
    ...

// Callback tasks – these simply call a callback macro
virtual task trans_received(my_driver drv, trans tr);
    `uvm_do_callbacks(driver_cbs,my_driver,
                      trans_received(this,tr ))
endtask : trans_received

virtual task trans_executed(my_driver drv, trans tr);
    `uvm_do_callbacks(driver_cbs,my_driver,
                      trans_executed(this,tr ))
endtask : trans_executed

task run_phase(uvm_phase phase);
    forever begin
        seq_item_port.get_next_item( tr );
        // Call callback task
        trans_received(this,tr );
        ... // Drive the transaction
        // Call callback task
        trans_executed(this,tr );
        seq_item_port.item_done( tr );
    end
endtask
```

Callbacks

```
end
endtask : run_phase
endclass : my_driver

Step 3 – Create a user defined type for the callback

typedef uvm_callbacks#(my_driver,driver_cbs) driver_cbs_t;

Step 4 – Define the functionality of the callback tasks

class error_cb extends driver_cbs;
  `uvm_object_utils(error_cb)

  ... // Constructor

  task trans_received(my_driver drv, trans tr);
    bit [2:0] i;
    // Twiddle up to 8 bits
    repeat ($urandom_range(8))
      assert(std::randomize(i)) begin
        tr.data[i] = ~tr.data[i]; // Corrupt the data
      end
      else
        drv.uvm_report_warning("Error Callback",
                               "Unable to select bit to twiddle! ");
    endtask

  task trans_executed(my_driver drv, trans tr);
    // Implement the trans_executed callback
  endtask
endclass : error_cb
```

Finally, Register the callback in the test case.

```
class error_injector_test extends uvm_test;
  ...
  function void start_of_simulation_phase(uvm_phase phase);
    my_driver drv;           // Driver in the environment
    error_cb e_cb;          // User defined callbacks class
    e_cb = new( "e_cb" );   // Create the error injecting callbacks
    // Find the driver where the callbacks will be installed
    $cast(drv, uvm_top.find( "*.m_drv" ));
    // Install the callbacks in the driver
    driver_cbs_t::add(drv, e_cb);
  endfunction : start_of_simulation_phase
endclass : error_injector_test
```

Gotcha

There are two classes with very similar names: `uvm_callback` and `uvm_callbacks`. Make sure you use the correct one: `uvm_callback` is the base class for user-defined callback classes. `uvm_callbacks` is used for registering callbacks with a component (See Example).

See also

`uvm_callback`; `uvm_callbacks`

For further information on this and other topics, please see the UVM tutorials at <http://www.doulos.com/knowhow/sysverilog/uvm/>.

uvm_callbacks

The `uvm_callbacks` class is a base class for implementing callbacks. 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.

Declaration

```
class uvm_callbacks #(type T = uvm_object,  
                     type CB = uvm_callback)  
extends uvm_typed_callbacks#(T);
```

Parameters

<code>type T = uvm_object</code>	The base type of the object with which the callback objects will be registered.
<code>type CB = uvm_callback</code>	The base callback type that will be managed by the interface.

Methods

<code>static function void add(T obj, uvm_callback cb, uvm_appprepend ordering = UVM_APPEND);</code>	Registers the given callback object, cb, with the given obj handle, which may be null.
<code>static function void add_by_name(string name, uvm_callback cb, uvm_component root, uvm_appprepend ordering = UVM_APPEND);</code>	Registers the given callback object, cb, with one or more uvm_components.root specifies the location in the component hierarchy to start the search for name.
<code>static function void delete(T obj, uvm_callback cb);</code>	Deletes the given callback object from the queue associated with the given object handle (possibly null).

<pre>static function void delete_by_name(string name, uvm_callback cb, uvm_component root);</pre>	Removes the given callback by name.
<pre>static function void display(T obj = null);</pre>	Displays callback information for obj used for debugging.

Rules

- The parameters T and CB must be derived from `uvm_object` and `uvm_callback` respectively.
- If a `null` object handle is used with `add`, `add_by_name`, `delete` or `delete_by_name` methods, then this refers to callbacks without an object context.

Example

See Callbacks.

Tips

`uvm_callbacks` also includes iterator methods, which are not shown here; a facade class, `uvm_callback_iter`, is generally the preferred way to iterate over callback queues.

See also

Callbacks; `uvm_callback`; `uvm_callback_iter`

uvm_callbacks_objection

An extended version of `uvm_objection` that includes callback hooks for when the objection is raised or dropped. The user can define the callbacks by extending the `uvm_objection_callback` class.

`uvm_heartbeat` is an example of a class that extends `uvm_callbacks_objection`.

Declaration

```
class uvm_callbacks_objection extends uvm_objection;
```

Methods

<code>function new(string name = "");</code>	Constructor.
<code>virtual function void raised (</code> <code>uvm_object obj,</code> <code>uvm_object source_obj,</code> <code>string description,</code> <code>int count);</code>	Calls the <code>raised</code> method in the user callback class when an objection is raised.
<code>virtual function void dropped (</code> <code>uvm_object obj,</code> <code>uvm_object source_obj,</code> <code>string description,</code> <code>int count);</code>	Calls the <code>dropped</code> method in the user callback class when an objection is dropped.
<code>virtual task all_dropped (</code> <code>uvm_object obj,</code> <code>uvm_object source_obj,</code> <code>string description,</code> <code>int count);</code>	Calls the <code>all_dropped</code> method in the user callback class when the last objection is dropped.

Tips

Use this class in place of `uvm_objection` if you want to implement the callbacks.

Gotcha

The naming convention – `uvm_callbacks_objection` versus `uvm_objection_callback` – can be confusing.

See also

Callbacks; `uvm_heartbeat`; `uvm_objection`; `uvm_objection_callback`

uvm_cmdline_processor

`uvm_cmdline_processor` provides an interface to the command line arguments that were used to invoke the simulator. The command line processor also provides support for setting various UVM variables from the command line such as components' verbosities and configuration settings.

A global variable called `uvm_cmdline_proc` is created at initialization time and may be used to access command line information.

Declaration

```
class uvm_cmdline_processor extends uvm_report_object;  
const uvm_cmdline_processor uvm_cmdline_proc  
    = uvm_cmdline_processor::get_inst();
```

Methods

static function uvm_cmdline_processor get_inst() ;	Obtain the unique instance of uvm_cmdline_processor .
function void get_args (output string args[\$]);	Returns a queue with all of the command line arguments. Element 0 of the array will always be the name of the executable which started the simulation.
function void get_plusargs (output string args[\$]);	Returns a queue with all of the plus arguments.
function void get_uvm_args (output string args[\$]);	Returns a queue with all of the +/-uvm*/UVM* arguments.
function int get_arg_matches (string match, ref string args[\$]);	Loads a queue with all of the arguments that match the input expression and returns the number of items that matched.
function int get_arg_value (string match, ref string value);	Find the first argument which matches <code>match</code> and returns the suffix of the argument. The function return is the total number of matches.

uvm_cmdline_processor

<pre>function int get_arg_values (string match, ref string values[\$]);</pre>	Finds all the arguments which match <code>match</code> and returns their suffixes. The function return is the total number of matches.
<pre>function string get_tool_name ();</pre>	Returns the name of the invoking simulator.
<pre>function string get_tool_version ();</pre>	Returns the version number of the invoking simulator.

Built-in Command Line Arguments

<code>+UVM_CONFIG_DB_TRACE</code>	Turns on tracing of configuration database access
<code>+UVM_DUMP_CMDLINE_ARGS</code>	For debugging. dumps all command line arguments in a tree format.
<code>+UVM_MAX_QUIT_COUNT=count,can_override</code>	Change the maximum quit count for the report server. <code>can_override</code> is YES or NO.
<code>+UVM_OBJECTION_TRACE</code>	Turns on tracing of objections.
<code>+UVM_PHASE_TRACE</code>	Turns on tracing of phase executions.
<code>+UVM_RESOURCE_DB_TRACE</code>	Turns on tracing of resource database access
<code>+uvm_set_action =comp,id,severity,action</code>	Sets reporting actions. Equivalent to <code>set_report_*_action</code> .
<code>+uvm_set_config_int =comp,field,value</code>	As procedural equivalent
<code>+uvm_set_config_string =comp,field,value</code>	As procedural equivalent
<code>+uvm_set_inst_override =type,override_type,path</code>	As factory <code>set_inst_override_by_name</code>
<code>+uvm_set_severity =comp,id,old_sev,new_sev</code>	Sets reporting severity. Equivalent to <code>set_report_*_severity_override</code>

+uvm_set_type_override =type,override_type[,replace]	As factory set_type_override_by_name If replace 1 (default), existing overrides are replaced
+uvm_set_verbosity =comp,id,verbosity,phase +uvm_set_verbosity =comp,id,verbosity,time,time	Sets the reporting verbosity for specific components, message Ids and phases or times. See Examples below.
+UVM_TESTNAME=test	Specifies which test to run.
+UVM_TIMEOUT=timeout,can_override	Change the global timeout. can_override is YES or NO.
+UVM_VERBOSITY=verbosity	Sets the initial reporting verbosity. Verbosity can be any valid verbosity enumeration value such as UVM_HIGH.

Rules

- Command line arguments that are in UPPERCASE should only have one setting per invocation. Command line arguments in lowercase can have multiple settings per invocation
- For `get_arg_matches`, 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.
- An `id` argument of `_ALL_` for `+uvm_set_verbosity`, `+uvm_set_action` or `+uvm_set_severity` and a `severity` argument of `_ALL_` for `+uvm-set_severity` matches all ids or all severities respectively.

Examples

Retrieve all the command line arguments and all the UVM command line arguments

```
string all_cmdline_args[$];
string all_uvm_cmdline_args[$];

uvm_cmdline_proc.get_args(all_cmdline_args);
uvm_cmdline_proc.get_uvm_args(all_uvm_cmdline_args);

`uvm_info("", $sformatf(
    {"There were %0d command-line arguments, ",
     "including %0d UVM ones"},
```

uvm_cmdline_processor

```
    all_cmdline_args.size(),
    all_uvm_cmdline_args.size()),
UVM_NONE)
```

Set verbosity in the build phase to UVM_FULL for all messages generated by `uvm_test_top.m_env.m_driver`:

```
simulator
+uvm_set_verbosity=uvm_test_top.m_env.m_driver,_ALL_,UVM_FULL,build ...
```

Set verbosity in the run phase to UVM_FULL for all messages generated in `uvm_test_top` and below, starting at time 200

```
simulator
+uvm_set_verbosity=uvm_test_top.*,_ALL_,UVM_FULL,time,200
...
```

Tips

`uvm_split_string` can be used to split a string returned by `get_arg_values`.

Gotchas

- The command line processor requires some DPI functions. If you are using the standard UVM distribution you will need to compile the file `$UVM_HOME/src/dpi/uvm_dpi.cc` and/or specify the location of the resulting UVM DPI shared library to your simulator. Your simulator may include a pre-compiled UVM library, in which case the DPI functions may also be pre-compiled. Please refer to the documentation from your simulator vendor for details.
- Don't use the constructor, `new`; the class is used as a singleton. Use `uvm_cmdline_processor::get_inst` to obtain the unique instance. Alternatively, use the global variable `uvm_cmdline_proc`.
- `get_arg_value` only returns the first match. To return all the matches for a given argument, use `get_arg_values`.
- There is no command-line equivalent for `set_config_object()`.
- Wildcarding is not supported for message IDs.

Compilation Directives

The UVM source code has a number of conditional compilation directives, which change or set its default behavior. These are just standard Verilog `ifdef statements that are set by providing the appropriate +define+ argument on the UVM compilation command line. For example, if linking in the DPI libraries produces a compiler error, then the libraries can be removed by adding the +define+UVM_NO_DPI compilation argument. Use these directives judiciously since the UVM defaults are typically adequate. Note, deprecated directives are not included.

Usage

```
<compiler> +define+<Compilation Directive>
```

Compiler Arguments

+define+UVM_CB_TRACE_ON
Print out informational messages as callbacks are executed.
Disabled by default.

+define+UVM_CMDLINE_NO_DPI
Disable the DPI regular expression parser for command line options.
Enabled by default.

+define+UVM_DISABLE_AUTO_ITEM_RECORDING
Disable automatic transaction recording for sequence items.
Enabled by default.

+define+UVM_EMPTY_MACROS
Creates empty definitions for the field, object, and component macros.
Disabled by default.

+define+UVM_ENABLE_FIELD_CHECKS
Report errors if multiple field macros are used on the same field.
Disabled by default.

Compilation Directives

+define+UVM_HDL_MAX_WIDTH

Maximum vector width for the HDL backdoor access methods.

Default is 1024.

+define+UVM_HDL_NO_DPI

Disables the DPI HDL backdoor access.

Enabled by default.

+define+UVM_LINE_WIDTH

Sets the default string length.

Default is 120 characters.

+define+UVM_MAX_STREAMBITS

Sets the maximum bit vector size for integral values.

Default is 4096.

+define+UVM_NO_DEPRECATED

Disables the use of deprecated features.

Enabled by default.

+define+UVM_NO_DPI

Disables the DPI HDL backdoor access, DPI regular expression processing, and the DPI command line regular expression parser.

Enabled by default.

+define+UVM_NUM_LINES

Sets the number of lines for strings.

Default is 120.

+define+UVM_PACKER_MAX_BYTES

Sets the maximum bytes to allocate for packing objects using `uvm_packer`.

Default is 4096.

```
+define+UVM_REGEX_NO_DPI
```

Disables using the DPI regular expression parser for command line options.

Enabled by default.

```
+define+UVM_REG_ADDR_WIDTH
```

Sets the maximum address bit width for `uvm_reg_addr_t`.

Default is 64.

```
+define+UVM_REG_BYTENABLE_WIDTH
```

Sets the maximum number of byte enables for `uvm_reg_byte_en_t`.

Default is (``UVM_REG_DATA_WIDTH-1)/8+1`).

```
+define+UVM_REG_CVR_WIDTH
```

Sets the maximum number of bits for a `uvm_reg_cvr_t` coverage model set.

Default is 32.

```
+define+UVM_REG_DATA_WIDTH
```

Sets the maximum data bit width for `uvm_reg_data_t`.

Default is 64.

```
+define+UVM_REG_NO_INDIVIDUAL_FIELD_ACCESS
```

Disables register field access.

Enabled by default.

```
+define+UVM_REPORT_DISABLE_FILE
```

Disables printing file information (`__FILE`) when using the report macros.

Enabled by default.

Compilation Directives

```
+define+UVM_REPORT_DISABLE_FILE_LINE
```

Disables printing file and line information (`__FILE and `__LINE) when using the report macros.

Enabled by default.

```
+define+UVM_REPORT_DISABLE_LINE
```

Disables printing the line information (`__LINE) when using the report macros.

Enabled by default.

```
+define+UVM_USE_CALLBACKS_OBJECTION_FOR_TEST_DONE
```

Switches the test done objection object from uvm_objection to uvm_callbacks_objection.

Default is uvm_objection.

Examples

Disable the use of deprecated features:

```
compiler +define+UVM_NO_DEPRECATED ...
```

Disable the use of the DPI methods:

```
compiler +define+UVM_NO_DPI ...
```

Enable callback tracing:

```
compiler +define+UVM_CB_TRACE_ON ...
```

See also

[uvm_cmdline_processor](#)

uvm_component

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

Declaration

```
virtual class uvm_component extends uvm_report_object;
```

Constructor and interaction with hierarchy

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

Methods

<code>function new(string name, uvm_component parent);</code>	Constructor
<code>function uvm_component get_child(string name);</code>	Returns handle to named child component
<code>function void get_children(ref uvm_component children[\$]);</code>	Returns an array of handles to child components
<code>function int unsigned get_depth();</code>	Returns the component's depth from <code>uvm_top</code>
<code>function int get_first_child(ref string name);</code>	Get the name of the first child
<code>virtual function string get_full_name();</code>	Returns the full hierarchical path name
<code>function string get_name();</code>	Returns the name
<code>function int get_next_child(ref string name);</code>	Get the name of the next child
<code>function int get_num_children();</code>	Return the number of child components
<code>virtual function uvm_component get_parent();</code>	Returns handle to parent component

uvm_component

<code>virtual function string get_type_name();</code>	Returns type name
<code>function int has_child(string name);</code>	True if child exists
<code>function uvm_component lookup(string name);</code>	Search for named component (no wildcards)
<code>function void print(uvm_printer printer = null);</code>	Prints the component [†]

[†]Inherited from `uvm_object`

UVM phases and control

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

Phase Callback Methods

For further details of these, see [Phase](#).

<code>virtual function void build_phase(uvm_phase phase);</code>	Build phase callback
<code>virtual function void connect_phase(uvm_phase phase);</code>	Connect phase callback
<code>virtual function void end_of_elaboration_phase(uvm_phase phase);</code>	End_of_elaboration phase callback
<code>virtual function void start_of_simulation_phase(uvm_phase phase);</code>	Start_of_simulation phase callback
<code>virtual task run_phase(uvm_phase phase);</code>	Run phase callback
<code>virtual function void extract_phase(uvm_phase phase);</code>	Extract phase callback
<code>virtual function void check_phase(uvm_phase phase);</code>	Check phase callback
<code>virtual function void report_phase(uvm_phase phase);</code>	Report phase callback
<code>virtual function void final_phase(uvm_phase phase);</code>	Final (tidy-up) phase

The following implement the pre-defined run-time schedule, which runs concurrently with `run_phase`.

<pre>virtual task pre_reset_phase(uvm_phase phase);</pre>	Pre_reset phase callback
<pre>virtual task reset_phase(uvm_phase phase);</pre>	Reset phase callback
<pre>virtual task post_reset_phase(uvm_phase phase);</pre>	Post_reset phase callback
<pre>virtual task pre_configure_phase(uvm_phase phase);</pre>	Pre-configure phase callback
<pre>virtual task configure_phase(uvm_phase phase);</pre>	Configure phase callback
<pre>virtual task post_configure_phase(uvm_phase phase);</pre>	Post configure phase callback
<pre>virtual task pre_main_phase(uvm_phase phase);</pre>	Pre_main phase callback
<pre>virtual task main_phase(uvm_phase phase);</pre>	Main phase callback
<pre>virtual task post_main_phase(uvm_phase phase);</pre>	Post_main phase callback
<pre>virtual task pre_shutdown_phase(uvm_phase phase);</pre>	Pre_shutdown phase callback
<pre>virtual task shutdown_phase(uvm_phase phase);</pre>	Shutdown phase callback
<pre>virtual task post_shutdown_phase(uvm_phase phase);</pre>	Post_shutdown phase callback
<pre>virtual function void phase_started(uvm_phase phase);</pre>	Invoked at the start of each phase
<pre>virtual function void phase_ended(uvm_phase phase);</pre>	Invoked at the end of each phase

uvm_component

<pre>virtual function void phase_ready_to_end(uvm_phase phase);</pre>	Invoked when all objections have been dropped for the given phase and all sibling phases (i.e., phases with a common ancestor)
--	--

Custom phase schedules and domains

<pre>function void set_domain(uvm_domain domain, int hier = 1);</pre>	Apply a phase domain to this component and, if <i>hier</i> is set, recursively to all its children.
<pre>function uvm_domain get_domain();</pre>	Return handle to the phase schedule graph that applies to this component
<pre>function uvm_phase get_schedule();</pre>	Return handle to the phase schedule graph that applies to this component
<pre>function void set_phase_imp(uvm_phase phase, uvm_phase imp, int hier = 1);</pre>	Override the default implementation for a phase on this component (tree) with a custom one
<pre>virtual task resume();</pre>	User-defined method to resume the phase.
<pre>virtual task suspend();</pre>	User-defined method to suspend the phase.

Phase Support Methods

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

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

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

<pre>virtual function void resolve_bindings();</pre>	Called immediately before end_of_elaboration phase – override to check connections
--	--

<code>function void do_resolve_bindings();</code>	Calls resolve_bindings for current component and recursively for its children
<code>virtual function void flush();</code>	Callback intended for clearing queues
<code>function void do_flush();</code>	Recursively calls flush for all children
<code>virtual task suspend();</code>	Suspend current task
<code>virtual task resume();</code>	Resume current task

Component configuration

Components work with the UVM configuration mechanism to set the value of members using a string-based interface. The `set_config_*`/`get_config_*` interface is for backward compatibility. New designs should use `uvm_config_db::set/get` or `uvm_resource_db::set/get`. See [Configuration](#) for full details.

Methods

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

uvm_component

<pre>virtual function bit get_config_object(string field_name, inout uvm_object value, input bit clone = 1);</pre>	Gets a configuration item as a <code>uvm_object</code> (or <code>null</code>). Updates member and returns <code>1'b1</code> if field name found. By default, the object is cloned.
<pre>function void check_config_usage(bit recurse = 1)</pre>	Check all configuration settings in a component's (and, if <code>recurse</code> is 1, recursively its children's) configuration table to determine if the setting has been used, overridden or not used.
<pre>virtual function void apply_config_settings(bit verbose = 0);</pre>	Searches for configuration items and updates members
<pre>function void print_config_settings(string field="", uvm_component comp = null, bit recurse = 0);</pre>	Prints all configuration information for component.
<pre>function void print_config(bit recurse = 0, bit audit = 0);</pre>	Prints all configuration information for this component and, if <code>recurse</code> is 1, recursively its children
<pre>function void print_config_with_audit(bit recurse = 0);</pre>	As <code>print_config</code> , with <code>audit</code> set to 1

Members

<code>static bit print_config_matches = 0;</code>	For debugging. If set, configuration matches are printed.
---	---

Component Objection Callbacks

See `uvm_objection` for full details.

<pre>virtual function void raised(uvm_objection objection, uvm_object source_obj, string description, int count);</pre>	Called when a descendant (<code>source_obj</code>) of the component instance raises the specified objection
--	---

<pre>virtual function void dropped(uvm_objection objection, uvm_object source_obj, string description, int count);</pre>	Called when a descendant (<i>source_obj</i>) of the component instance drops the specified objection
<pre>virtual task all_dropped(uvm_objection objection, uvm_object source_obj, string description, int count);</pre>	Called when all objections have been dropped by this component or a descendant. <i>source_obj</i> is the last to drop.

The Factory

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

The factory supports both parameterized and non-parameterized components using a proxy class for each component type that is derived from class `uvm_object_wrapper`. The component utility macros register a component with the factory. They also define a nested proxy class named `type_id` and a static function `get_type` that returns the singleton instance of the proxy class for a particular component type. `type_id` provides a very convenient way to use the factory. For example calling `my_component::type_id::create` will use the factory to create an instance of `my_component` by calling the `create` method of the nested `type_id` class.

The `create` and `clone` methods inherited from `uvm_object` are disabled for components. See [uvm_factory](#).

Methods

<pre>static function T type_id::create(string name, uvm_component parent, string context = "")</pre>	Create an instance of the component (type T) using the factory.
<pre>virtual function string type_id::get_type_name();</pre>	Returns the type name of the component as a string.
<pre>static function void type_id::set_type_override(string original_type_name, string override_type_name, bit replace = 1);</pre>	Set a factory override for all instances of this component.

uvm_component

<pre>static function void type_id::set_inst_override(uvm_object_wrapper override_type, string inst_path, uvm_component parent = null);</pre>	Set a factory override for a specific instance or instances of this component.
<pre>function uvm_component create_component(string requested_type_name, string name);</pre>	Creates component as a child of current component (parent set to "this").
<pre>function uvm_object create_object(string requested_type_name, string name="");</pre>	Creates object as a child of current component.
<pre>static function void set_type_override(string original_type_name, string override_type_name, bit replace=1);</pre>	Overrides the type used by the factory for specified type.
<pre>static function void set_type_override_by_type(uvm_object_wrapper original_type, uvm_object_wrapper override_type, bit replace=1);</pre>	Overrides the type used by the factory for specified type.
<pre>function void set_inst_override(string relative_inst_path, string original_type_name, string override_type_name);</pre>	Overrides the type used by the factory for the specified instance only.
<pre>function void set_inst_override_by_type(string relative_inst_path, uvm_object_wrapper original_type, uvm_object_wrapper override_type);</pre>	Overrides the type used by the factory for the specified instance only.
<pre>function void print_override_info(string requested_type_name, string name="");</pre>	Prints details about the type of object that would be created for the given arguments.
<pre>static function type_id get_type();</pre>	Returns proxy (wrapper) for class type required by factory methods.

^{*}Created by utility macros

Hierarchical configuration of component report handler

Components provide methods to configure the UVM report handler for a particular component and recursively for all of its children. The methods can apply to all reports of a particular severity, all reports with a matching id or all

reports whose severity and id both match those specified. Where there are overlapping conditions, matching both severity and id takes precedence over matching only id which takes precedence over matching only severity.

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

See [uvm_report_object](#).

Methods

<pre>function void set_report_id_verbosity_hier(string id, int verbosity);</pre>	Recursively set the action for reports with matching id and verbosity.
<pre>function void set_report_severity_id_verbosity_hier(uvm_severity severity, string id, int verbosity);</pre>	Recursively set the action hierarchically for reports with matching severity, id, and verbosity.
<pre>function void set_report_severity_action_hier(uvm_severity s, uvm_action a);</pre>	Recursively set the action for reports with severity s.
<pre>function void set_report_id_action_hier(string id, uvm_action a);</pre>	Recursively set the action for reports with matching id.
<pre>function void set_report_severity_id_action_hier(uvm_severity s, string id, uvm_action a);</pre>	Recursively set the action for reports with both severity s AND matching id.
<pre>function void set_report_default_file_hier(UVM_FILE f);</pre>	Recursively set the default file written by action <code>UVM_LOG</code> .
<pre>function void set_report_severity_file_hier(uvm_severity s, UVM_FILE f);</pre>	Recursively set the file written by action <code>UVM_LOG</code> for reports of severity s.
<pre>function void set_report_id_file_hier(string id, UVM_FILE f);</pre>	Recursively set the file written by action <code>UVM_LOG</code> for reports with matching id.
<pre>function void set_report_severity_id_file_hier(uvm_severity s, string id, UVM_FILE f);</pre>	Recursively set the file written by action <code>UVM_LOG</code> for reports with both severity s AND matching id.

uvm_component

<pre>function void set_report_verbosity_level_hier(int v);</pre>	Recursively set verbosity threshold – only messages with lower verbosity written.
<pre>virtual function void pre_abort();</pre>	This callback is executed when the message system is executing a UVM_EXIT action.

Types

<code>typedef int UVM_FILE;</code>	File descriptor
------------------------------------	-----------------

Recording component transactions

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

As of UVM 1.1, the API is not fully defined and is subject to change.

See [uvm_transaction](#).

Methods

<pre>function void accept_tr(uvm_transaction tr, time accept_time = 0);</pre>	Call transaction's <code>accept_tr</code> function and trigger <code>accept_tr</code> event
<pre>function integer begin_tr(uvm_transaction tr, string stream_name = "main", string label = "", string desc = "", time begin_time = 0, integer parent_handle = 0);</pre>	Call transaction's <code>begin_tr</code> function, trigger <code>begin_tr</code> event and write transaction details to stream. Return transaction handle.
<pre>function integer begin_child_tr(uvm_transaction tr, integer parent_handle = 0, string stream_name = "main", string label = "", string desc = "", time begin_time = 0);</pre>	Call transaction's <code>begin_child_tr</code> function, trigger <code>begin_tr</code> event and write transaction details to stream. Return transaction handle.

<pre>function void end_tr(uvm_transaction tr, time end_time = 0, bit free_handle = 1);</pre>	Call transaction's end_tr function, trigger end_tr event and write transaction details to stream.
<pre>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);</pre>	Records error in transaction stream.
<pre>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);</pre>	Records "event" in transaction stream.
<pre>virtual protected function void do_accept_tr(uvm_transaction tr);</pre>	Callback from accept_tr (by default does nothing).
<pre>virtual protected function void do_begin_tr(uvm_transaction tr, string stream_name, integer tr_handle);</pre>	Callback from begin_tr (by default does nothing).
<pre>virtual protected function void do_end_tr(uvm_transaction tr, integer tr_handle);</pre>	Callback from end_tr (by default does nothing).

Members

<pre>protected uvm_event_pool event_pool;</pre>	Events for transaction accept, begin and end.
<pre>bit print_enabled = 1;</pre>	Determines if component should be printed. Default is true.

General

Macros

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

```
`uvm_component_utils(TYPE)
```

uvm_component

or

```
`uvm_component_utils_begin(TYPE)
 `uvm_field_* (ARG, FLAG)
 ...
`uvm_component_utils_end
```

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

Parameterized components should use the

```
`uvm_component_param_utils(TYPE#(T))
```

or

```
`uvm_component_param_utils_begin(TYPE#(T))
 `uvm_field_* (ARG, FLAG)
 ...
`uvm_component_utils_end
```

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

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

```
`uvm_field_utils_begin(TYPE)
 `uvm_field_* (ARG, FLAG)
 ...
`uvm_field_utils_end
```

Rules

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

Example

Using `uvm_component` for a simple parameterized testbench class

```
class lookup_table #(WIDTH=10) extends uvm_component;
```

```

uvm_blocking_get_imp #(int,lookup_table#(WIDTH))  

get_export;  
  

int lut [WIDTH];  

int index = 0;  
  

function new (string name, uvm_component parent);  

    super.new(name,parent);  

endfunction : new  
  

function void build_phase(uvm_phase phase);  

    super.build_phase(phase);  

    foreach (lut[i]) lut[i] = i * 10;  

    get_export = new("get_export",this);  

endfunction: build_phase  
  

task get (output int val);  

    #10 val = lut[index++];  

    if (index > WIDTH-1) index = 0;  

endtask: get  
  

`uvm_component_param_utils_begin(lookup_table#(WIDTH))  

`uvm_field_sarray_int(lut,UVM_ALL_ON + UVM_DEC)  

`uvm_component_utils_end
endclass: lookup_table

```

Tips

- UVM defines virtual base classes for various common testbench components (e.g. `uvm_monitor`) that are themselves derived from `uvm_component`. These should be used as base classes for testbench components in preference to `uvm_component` where appropriate.
- Use `class_name::type_id::create` or `create_component` to create new component instances in the build phase rather than `new` or `uvm_factory::create_component`.
- Use the field automation macros for any fields that need to be configured automatically. These also enable the fields of one component instance to be copied or compared to those of another.
- Set the required reporting options by calling the hierarchical functions (`set_report_*_hier`) for a top-level component, for example `uvm_top`, since these settings are applied recursively to all child components.
- Use `objection` to stop the simulation rather than `kill`. It gives components the opportunity to complete their current actions before halting.

uvm_component

(Use of `global_stop_request` or `uvm_top.stop_request` is deprecated).

Gotchas

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

See also

`Configuration`; `uvm_factory`; `uvm_driver`; `uvm_monitor`; `uvm_scoreboard`; `uvm_agent`; `uvm_env`; `uvm_test`; `uvm_root`

The `uvm_config_db` parameterized class provides a convenience interface on top of `uvm_resource_db` to simplify the basic interface that is used for reading and writing into the resource database.

The `set` and `get` methods provide the same interface and semantics as the `set/get_config_*` functions in `uvm_component`.

UVM's configuration facility provides an alternative to using the factory to configure a verification environment.

Declaration

```
class uvm_config_db#(type T = int)
  extends uvm_resource_db#(T);
```

Methods

<pre>static function bit get(uvm_component cntxt, string inst_name, string field_name, inout T value);</pre>	Get the value of <code>field_name</code> in <code>inst_name</code> , using component <code>cntxt</code> as the starting search point.
<pre>static function void set(uvm_component cntxt, string inst_name, string field_name, T value);</pre>	Set the value of <code>field_name</code> in <code>inst_name</code> , using component <code>cntxt</code> as the starting search point. if the setting exists it will be updated; otherwise it will be created.
<pre>static function bit exists(uvm_component cntxt, string inst_name, string field_name, bit spell_chk = 0);</pre>	Check if a value for <code>field_name</code> is available in <code>inst_name</code> , using component <code>cntxt</code> as the starting search point. returns 1 if the resource exists, 0 if not. With <code>spell_chk</code> set to 1, a warning is issued if the resource isn't found.
<pre>static task wait_modified(uvm_component cntxt, string inst_name, string field_name);</pre>	Waits (blocks) for a configuration setting to be set.

Rules

- Regular expressions are enclosed by a pair of slashes: `/regex/`.
- All of the functions in `uvm_config_db` are static so they must be called using the `::` operator.

uvm_config_db

- In the methods, *inst_name* is an explicit instance name relative to *ctxt* and may be an empty string if the *ctxt* is the instance that the configuration object applies to. *field_name* is the specific field in the scope that is being searched for.
- When setting values, both *inst_name* and *field_name* may be glob-style search strings or regular expressions. regular expressions are enclosed in a pair of slashes.

Examples

Set the value for a resource, iterations, in any hierarchical component whose name ends in "driver" to 50:

```
uvm_config_db #(int)::set(this,"*driver",
                      "iterations", 50);
```

The same, using a regular expression instead of a glob:

```
uvm_config_db #(int)::set(this,"*.driver/",
                      "iterations", 50);
```

Retrieve the resource's value. If the resource is not found, use a default value.

```
if ( !uvm_config_db #(int)::get(this,"","iterations",
                               n_iterations) )
    // In the absence of a config setting, use a default value:
    n_iterations = 20;
```

Set a sequencer's default sequence. This will cause the sequencer to start automatically.

```
uvm_config_db#(uvm_sequence_base)::set(
    null,
    "/.*m_env.m_sequencer.run_phase/", // Regular expression
    "default_sequence",
    my_sequence::type_id::get());
```

Tips

`uvm_config_db` and `uvm_resource_db` share the same underlying database. For configuration properties that are related to hierarchical position, e.g., "set all of the coverage_enable bits for all components in a specific agent", `uvm_config_db` is the correct choice. Conversely, for cases where a configuration property is being shared without regard to hierarchical context, `uvm_resource_db` should be used.

Gotchas

- When using the config database to set build-related options in an environment (like the `is_active` flag of an agent), be sure to set the values in the `build_phase` so values are inserted in time in the database.
- When the config database is accessed, it uses the full hierarchical path name. If `null` is used as the context, then it is important to recognize that the testcase's instance name is not at the top of the hierarchy. Rather, UVM inserts `uvm_test_top` in the path. Therefore, it is usually best to use wildcards when specifying the path name starting with `uvm_top` (`null`) as the context; e.g., `*.m_env*`.
- 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.

See also

Configuration; `uvm_resource_db`

Configuration

Configuration is a mechanism that UVM provides to modify the default state of components, either when they are built or when a simulation is run. It provides an alternative to factory configuration for modifying the way components are created. Configuration can act on both components and transactions.

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

Note that the configuration mechanism is implemented differently in UVM than in OVM, although the semantics are mostly unchanged.

Resources Database

There is a global resources database that is available during simulation. It is accessed using the `uvm_resource_db` and `uvm_config_db` classes.

Resources can be set and retrieved by name or by type. Components use the resource database at various simulation phases including, but not limited to, the `build_phase`.

There are no separate configuration tables for individual components or instances. Instead, resources are stored with a set of scopes to which they apply. The set of scopes is in the form of a single regular expression. When the resources database is queried, the *current scope* of the component attempting to retrieve the resource is matched against this regular expression. The way the current scope is constructed depends on which query function is used, and the values of its arguments.

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

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

Printing Configuration Information

The `print_config` method of `uvm_component` may be used to print configuration information about the component. (`print_config_settings` is deprecated.) Called without arguments, it prints all the component's configuration information. `print_config` may also recursively print configuration for the component's children (`recurse=1`).

Global Functions

The methods of `uvm_config_db` and `uvm_resource_db` are static, so they are called using the `::` operator. For example,

```
uvm_config_db #(int)::set(this, "*driver",
                         "max_iterations", 50);
```

Methods of uvm_component

These functions are members of `uvm_component` (or an extension). They have been reproduced here for convenience. (However, note that `uvm_config_db::set` and `uvm_config_db::get` work on any type, not just `int`, `string` and `uvm_object`.)

<pre>virtual function void set_config_int(string inst_name, string field_name, uvm_bitstream_t value);</pre>	Sets an integral-valued configuration item.
<pre>virtual function void set_config_string(string inst_name, string field_name, string value);</pre>	Sets a string-valued configuration item.
<pre>virtual function void set_config_object(string inst_name, string field_name, uvm_object value, bit clone = 1);</pre>	Sets a configuration item as a <code>uvm_object</code> (or <code>null</code>). By default, the object is cloned.
<pre>virtual function bit get_config_int(string field_name, inout uvm_bitstream_t value);</pre>	Gets an integral-valued configuration item. Updates member and returns 1'b1 if field name found.
<pre>virtual function bit get_config_string(string field_name, inout string value);</pre>	Gets a string-valued configuration item. Updates member and returns 1'b1 if field name found.
<pre>virtual function bit get_config_object(string field_name, inout uvm_object value, input bit clone = 1);</pre>	Gets a configuration item as a <code>uvm_object</code> (or <code>null</code>). Updates member and returns 1'b1 if field name found. By default, the object is cloned.
<pre>function void check_config_usage(bit recurse = 1)</pre>	Searches for configuration items and updates members
<pre>function void print_config(bit recurse = 0, bit audit = 0)</pre>	Prints all configuration information for this component and, if <code>recurse</code> is 1, recursively its children

Configuration

```
function void  
print_config_with_audit(  
    bit recurse = 0)
```

As print_config, with audit set to 1

Members of uvm_component

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

Examples

Automatic configuration using field automation:

```
class verif_env extends uvm_env;  
    int m_n_cycles;  
    string m_lookup;  
    instruction m_template;  
    typedef enum {IDLE, FETCH, WRITE, READ} bus_state_t;  
    bus_state_t m_bus_state;  
    ...  
    `uvm_component_utils_begin(verif_env)  
        `uvm_field_string(m_lookup, UVM_DEFAULT)  
        `uvm_field_object(m_template, UVM_DEFAULT)  
        `uvm_field_enum(bus_state_t, m_bus_state, UVM_DEFAULT)  
    `uvm_component_utils_end  
endclass: verif_env  
class test2 extends uvm_test;  
    register_instruction inst = new();  
    string str_lookup;  
    ...  
    function void build_phase(uvm_phase phase);  
        ...  
        uvm_config_db #(bus_state_t)::set(  
            null, "*env1.*", "m_bus_state", verif_env::IDLE);  
        uvm_config_db #(string)::set(  
            null, "*", "m_lookup", str_lookup);  
        uvm_config_db #(register_instruction) (  
            null, "*", "m_template", inst);  
        ...  
    endfunction : build_phase  
    ...  
endclass : test2
```

Manual configuration

```
// In a test, create an entry "count" in the global configuration settings table ...
uvm_config_db#(int)::set(this,"*","count",1000);
// ... and retrieve the value of "count"
if ( !uvm_config_db#(int)::get(null,"count",m_n_cycles) )
    m_n_cycles = 1500; // use default value
```

Tips

- Standard (Verilog) command-line plusargs may be used to modify the configuration. This provides a simple, yet flexible way of configuring a test.
- Use `uvm_resource_db::dump()` to diagnose problems.

Gotchas

- A wildcard "*" in an instance name will match any expanded path at that point in the hierarchical name, not just a single level of hierarchy.
- The instance name in a call to `get_config/::get` should not include wildcards; they are treated as normal characters when matching the scope.
- `print_config` does not give any indication about whether the configuration has "successfully" set the value of a component member.

See also

`uvm_component`; `uvm_config_db`; `uvm_factory`; Field Macros;
`uvm_resource_db`; `uvm_root`

uvm_driver

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

Declaration

```
class uvm_driver #(type REQ = uvm_sequence_item,  
                  type RSP = REQ) extends uvm_component;  
  
class uvm_push_driver #(type REQ = uvm_sequence_item,  
                  type RSP = REQ) extends uvm_component;
```

Methods

<code>function new(string name, uvm_component parent);</code>	Constructor, mirrors the superclass constructor in <code>uvm_component</code>
--	---

Members

uvm_driver

<code>uvm_seq_item_pull_port #(REQ,RSP) seq_item_port;</code>	Port for connecting the driver to the sequence item export of a sequencer
<code>uvm_analysis_port #(RSP) rsp_port;</code>	Analysis port for responses
<code>REQ req;</code>	Handle for request
<code>RSP rsp;</code>	Handle for response

uvm_push_driver

<code>uvm_blocking_put_imp #(REQ, uvm_push_driver #(REQ,RSP)) req_export;</code>	Port for connecting the driver to the blocking put port of a push_sequencer
<code>virtual task put(REQ item);</code>	Implements the push_driver's behavior
<code>uvm_analysis_port #(RSP) rsp_port;</code>	Analysis port for responses
<code>REQ req;</code>	Handle for request
<code>RSP rsp;</code>	Handle for response

uvm_seq_item_pull_port

<pre>function new(string name, uvm_component parent, int min_size = 0, int max_size = 1);</pre>	Constructor. Default minimum size of 0 makes connection to sequencer optional
<pre>task get_next_item(output REQ req_arg);</pre>	Blocks until item is returned from sequencer. There must be a subsequent call to item_done
<pre>task try_next_item(output REQ req_arg);</pre>	Attempts to fetch item. If item is available, returns immediately and there must be a subsequent call to item_done. Otherwise req_arg set to null
<pre>function void item_done(RSP rsp_arg = null);</pre>	Indicates to the sequencer that the driver has processed the item and clears the item from the sequencer fifo. Optionally also sends response
<pre>task wait_for_sequences();</pre>	Calls connected sequencer's wait_for_sequences task (by default waits #100)
<pre>function bit has_do_available();</pre>	Returns 1 if item available, otherwise 0
<pre>task get(output REQ req_arg);</pre>	Blocks until item is returned from sequencer. Calls item_done before returning.
<pre>task peek(output REQ req_arg);</pre>	Blocks until item is returned from sequencer. Does not remove item from sequencer fifo
<pre>task put(RSP rsp_arg);</pre>	Sends response back to sequencer

Example

```
class example_driver extends uvm_driver #(my_transaction);
  ...
  virtual task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(req);
      phase.raise_objection(this);

      // Code to generate physical signal activity
      // as specified by transaction data in req

      seq_item_port.item_done();
      phase.drop_objection(this);
    end
  endtask

  `uvm_component_utils_begin(example_driver)
  `uvm_component_utils_end

endclass: example_driver
Example of a uvm_push_driver
class example_push_driver
  extends uvm_push_driver #(my_transaction);
  ...
  // Called by push_sequencer through TLM export
  task put (my_transaction item);
    // code to generate physical signal activity
    // as specified by transaction data in req
  endtask : put

  // No need to implement run_phase
  ...
endclass: example_push_driver
```

Tips

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

Gotchas

Do not forget to call the sequence item pull port's `item_done` method when your code has finished consuming the transaction item. Using the `seq_item_port`'s `get` method also calls `item_done`.

See also

`uvm_agent`; Sequencer Interface and Ports; `uvm_sequencer`

End of Test

One of the trickiest things to understand when you are starting to use UVM is how to stop the simulator at the end of the test.

UVM provides an “objection” mechanism for this purpose. (Don’t be confused by the terminology: we are not talking about objects – the noun – as in object-oriented, but the verb ‘to object’; that is to express disapproval or refuse permission). Other methods of stopping simulation, which were used in OVM and UVM-EA, such as calling `global_stop_request`, are deprecated in UVM.

The reason for including this seemingly unintuitive objection mechanism in UVM is so that components can communicate their status hierarchically and in correct phase synchronization with other parts of the test environment. We don’t want one part of the test environment to stop the simulation simply because it thinks “I’ve finished, so everything else must have finished too”.

Objection

In general, the process is for a component or sequence to “object” to a specific simulation phase completing before the component’s activity in that phase has completed. It does this by “raising a phase objection” at the beginning of the activity that must be completed before the phase stops, and to dropping the objection at the end of that activity.

Once all of the raised objections for a phase have been dropped, the phase terminates.

Drain Time

When all objections are dropped, the currently running phase is ended. In practice, there are times when simulation needs to continue for a while. For example, concurrently running processes may need some additional cycles to convey the last transaction to a scoreboard.

`uvm_component`’s `phase_ready_to_end()` method is called when all objections have been dropped. This could be used to re-raise an objection to the phase ending.

Alternatively, you can set a *drain time* for a component. This delays notifying the component’s parent that all objections have been dropped. Typically a single drain time is set at the environment or test level.

Finally, you could use the phase objection’s `all_dropped()` callback.

Example

This shows how a sequence can use objections.

```
class my_sequence extends uvm_sequence#(my_transaction);  
  
  // pre_body() is called before body()  
  task pre_body();
```

```
// raise objection if started as a root sequence
if ( starting_phase != null )
    starting_phase.raise_objection(this);
endtask

// body() implements the sequence's main activity
task body();
// do interesting activity
...
endtask

// post_body() is called after body() completes
task post_body();
// drop objection if started as a root sequence
if ( starting_phase != null )
    starting_phase.drop_objection(this);
endtask
endclass
```

This would work if the sequence was started automatically. If the sequence was started manually, the sequence's `starting_phase` would be `null`. In this case, the test could set it:

```
class my_test extends uvm_test;
my_env m_env;
my_sequence seq;

...
task main_phase(uvm_phase phase);
    seq.starting_phase = phase;
    seq.start(m_env.m_sequencer);
endtask
...
endclass
```

Alternatively, the test itself could use an objection:

```
class my_test extends uvm_test;
my_env m_env;
my_sequence seq;

...
task main_phase(uvm_phase phase);
    phase.raise_objection(this);
    // start() blocks until the sequence has completed on the sequencer
    seq.start(m_env.m_sequencer);
    // Set a 'drain time' for the test
```

End of Test

```
phase.phase_done.set_drain_time(this, 100ns);
phase.drop_objection(this);
endtask
...
endclass
```

Gotchas

`global_stop_request()` is deprecated. All environments using the `global_stop_request()` mechanism must add the switch
`+UVM_USE_OVM_RUN_SEMANTIC`.

See also

`uvm_objection`; `uvm_test`; `uvm_sequence`

For further information on this and other topics, please see the UVM tutorials at
<http://www.doulos.com/knowhow/sysverilog/uvm/>.

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

Declaration

```
virtual class uvm_env extends uvm_component;
```

Methods

<code>function new(string name = "env", uvm_component parent = null);</code>	Constructor.
---	--------------

Members

Only inherited members.

Example

This is a minimal environment:

```
class verif_env extends uvm_env;  
  `uvm_component_utils(verif_env)  
  
  // Testbench methodology components  
  ...  
  
  function new(string name, uvm_component parent);  
    super.new(name,parent);  
  endfunction : new  
  
  function void build_phase(uvm_phase phase);  
    // Instantiate top-level components using "new" or  
    // the factory, as appropriate  
    ...  
  endfunction: build_phase  
  
  virtual function void connect_phase(uvm_phase phase);  
    // Connect ports-to-exports  
    ...  
  endfunction: connect_phase
```

```
virtual task run_phase(uvm_phase phase);
    // Prevent simulation from ending – must do this before the first wait.
    // Not needed here if it is in the test class
    phase.raise_objection(this);
    // Control stimulus generation
    ...
    // Allow simulation to end
    phase.drop_objection(this);
endtask: run_phase

endclass: verif_env
```

Tips

- The new, build_phase, and connect_phase methods should be overridden.
- Control simulation using the run_phase method. Call phase.raise_objection and phase.drop_objection so that simulation stops automatically.
- Instantiate one top-level environment in a (top-level) test. This may in turn instantiate other (lower-level) environments.

Gotchas

- new and build_phase should call the base class new (super.new) and build_phase (super.build_phase) methods respectively.
- Do not forget to register the environment with the factory, using `uvm_component_utils.

See also

uvm_test; Configuration

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

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

Declaration

```
class uvm_event extends uvm_object;
```

Methods

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

uvm_event

virtual function void reset (bit wakeup = 0);	Resets event and clears data. If wakeup bit is set, any process waiting for trigger resumes.
virtual function void add_callback (uvm_event_callback cb, bit append = 1);	Add callback (class with pre_trigger and post_trigger function). Adds to end of list by default
virtual function void delete_callback (uvm_event_callback cb);	Removes callback.
virtual function void cancel ();	Decrement count of waiting processes by 1.
virtual function int get_num_waiters ();	Number of waiting processes.

Example

Using an event to synchronize two tasks and send data:

```
class C extends uvm_component;  
    uvm_event e1;  
    function new (string name, uvm_component parent);  
        super.new(name,parent);  
    endfunction : new  
  
    function void build_phase(uvm_phase phase);  
        super.build_phase(phase);  
        e1 = new ("e1");  
    endfunction: build_phase  
  
    task run_phase(uvm_phase phase);  
        basic_transaction tx,rx;  
        tx = new();  
        fork  
            begin  
                tx.data = 10;  
                tx.addr = 1;  
                #10 e1.trigger(tx);  
            end  
            begin  
                e1.wait_ptrigger();  
                $cast(rx,e1.get_trigger_data());  
                rx.print();  
            end  
        end
```

```
join
endtask: run_phase

`uvm_component_utils(C)
endclass: C

Creating a callback:

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

    function void post_trigger(uvm_event e,
                               uvm_object data=null);
        basic_transaction rx;
        if (data) begin
            $cast(rx,data);
            `uvm_info("CBACK",$sformatf("Received %s",
                                         rx.convert2string()),UVM_NONE)
        end
    endfunction: post_trigger
endclass: my_e_callback
```

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

```
my_e_callback cb1;
...
cb1 = new ("cb1"); //in build_phase
...
e1.add_callback(cb1); //in run_phase
```

Tips

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

Gotchas

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

uvm_factory

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

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

Classes derived from `uvm_object` and `uvm_component` can be substituted with alternative types using the factory override methods. The substitution is made when the component or object is built. The substitution mechanism only makes sense if the substitute is an extended class of the original type. Both the original type and its replacement must have been registered with the factory, preferably using one of the utility macros ``uvm_component_utils`, ``uvm_component_param_utils`, ``uvm_object_utils` or ``uvm_object_param_utils`.

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

A singleton instance of `uvm_factory` named `factory` is instantiated within the UVM package (`uvm_pkg`). The `factory` object can therefore be accessed from SystemVerilog modules and classes as well as from within a uvm environment.

Declaration

```
class uvm_factory;
```

Methods

<pre>function uvm_object create_object_by_type(uvm_object_wrapper requested_type, string parent_inst_path = "", string name = "");</pre>	Creates and returns an object. Type is set by proxy. Name and parent specified by strings
<pre>function uvm_component create_component_by_type(uvm_object_wrapper requested_type, string parent_inst_path = "", string name, uvm_component parent);</pre>	Creates and returns a component. Type is set by proxy. Name and parent specified by strings
<pre>function uvm_object create_object_by_name(string requested_type_name, string parent_inst_path = "", string name = "");</pre>	Creates and returns an object. Type, name and parent are specified by strings

<pre>function uvm_component create_component_by_name(string requested_type_name, string parent_inst_path = "", string name, uvm_component parent);</pre>	Creates and returns a component. Type, name and parent are specified by strings
<pre>function void set_inst_override_by_type(uvm_object_wrapper original_type, uvm_object_wrapper override_type, string full_inst_path);</pre>	Register an instance override with the factory based on proxies (see below)
<pre>function void set_inst_override_by_name(string original_type_name, string override_type_name, string full_inst_path);</pre>	Register an instance override with the factory based on type names (see below)
<pre>function void set_type_override_by_type(uvm_object_wrapper original_type, uvm_object_wrapper override_type, bit replace = 1);</pre>	Register a type override with the factory based on proxies (see below)
<pre>function void set_type_override_by_name(string original_type_name, string override_type_name, bit replace = 1);</pre>	Register a type override with the factory based on type names (see below)
<pre>function uvm_object_wrapper find_override_by_type(uvm_object_wrapper requested_type, string full_inst_path);</pre>	Return the proxy to the object that would be created for the given override
<pre>function uvm_object_wrapper find_override_by_name(string requested_type_name, string full_inst_path);</pre>	Return the proxy to the object that would be created for the given override
function uvm_factory get() ;	Returns the singleton factory object.
function void register (uvm_object_wrapper obj);	Registers a proxy with the factory (called by utility macros)
<pre>function void debug_create_by_type(uvm_object_wrapper requested_type, string parent_inst_path = "", string name = "");</pre>	Prints information about the type of object that would be created with the given proxy, parent and name

uvm_factory

<pre>function void debug_create_by_name(string requested_type_name, string parent_inst_path = "", string name = "");</pre>	Prints information about the type of object that would be created with the given type name, parent and name
<pre>function void print(int all_types = 1);</pre>	<p>all_types is 0: Prints the factory overrides</p> <p>all_types is 1: Prints the factory overrides + registered types</p> <p>all_types is 2: Prints the factory overrides + registered types (including UVM types)</p>

Registration

Components and objects are generally registered with the factory using the macros ``uvm_component_utils` and ``uvm_object_utils` respectively.

Parameterized components and objects should use

``uvm_component_param_utils` and ``uvm_object_param_utils` respectively. Registration using these macros creates a specialization of the `uvm_component_registry #(T,Tname)` (for components) or `uvm_object_registry #(T,Tname)` (for objects) and adds it as a nested class to the component or object named `type_id`. If you try to create a component or object using a type that has not been registered, nothing is created, and the value `null` is returned.

Overriding instances and types

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

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

Creating Components and Objects

The `create_object_by_*` and `create_component_by_*` member functions of `uvm_factory` can be called from modules or classes using the factory instance. The type of object/component to build is requested by

passing a proxy or string argument. The instance name and path are specified by string arguments. The path argument is used when searching the configuration table for path-specific overrides. The `create_component_by_*` functions require a 4th argument: a handle to their parent component. This is not required when objects are created.

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

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

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

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

Examples

```
class verif_env extends uvm_env;
  // Register the environment with the factory
  `uvm_component_utils (verif_env)
  ...
  instruction m_template;
  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    ...
    // Use the factory to create the m_template object
    m_template = instruction::type_id::create("m_template",
                                              this);
  endfunction : build_phase
endclass: verif_env
```

uvm_factory

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

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

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

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

    ...
endclass : test1
```

Tips

- Do not create a `uvm_factory` object or derive a class from `uvm_factory`. A factory is created automatically. It is a singleton – there is only one instance named `factory`.
- Use the factory to create components and objects whenever possible. This makes the test environment more flexible and reusable.
- Prefer the type-based (proxy) functions to the string-based (type name) functions, because string based calls are more error prone and can only be detected at run-time.
- For convenience, use the `create` function from an object or component proxy – it requires less arguments, is more likely to pick up errors in class names at compile time and returns a handle of the correct type. If this is not possible, then prefer a component's own `create_component` or `create_object` method to `uvm_factory::create_component_by_name` or `uvm_factory::create_object_by_name` respectively.

- With `create_component_by_name` and `create_object_by_name` use the same name for the instance that you used for the instance (`handle`) variable.
- Both TLM1 and TLM2 ports are not registered with the factory so they must be created using a call to `new`. The only exception is the TLM2 generic payload, which is registered and can be created using the factory mechanism.

Gotchas

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

See also

Field Macros; Configuration; Sequence; `uvm_component`; `uvm_object`; Utility macros

Field Macros

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

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

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

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

Field Macros

Macro	Declares a field for this type:
<code>`uvm_field_int</code> (ARG, FLAG)	Any packed integral type.
<code>`uvm_field_real</code> (ARG, FLAG)	Real.
<code>`uvm_field_enum</code> (TYPE, ARG, FLAG)	Enum of TYPE.
<code>`uvm_field_object</code> (ARG, FLAG)	<code>uvm_object</code> .
<code>`uvm_field_event</code> (ARG, FLAG)	Event.
<code>`uvm_field_string</code> (ARG, FLAG)	String.
<code>`uvm_field_array_enum</code> (ARG, FLAG)	Dynamic array of enums.
<code>`uvm_field_array_int</code> (ARG, FLAG)	Dynamic array of packed integral type.
<code>`uvm_field_array_object</code> (ARG, FLAG)	Dynamic array of <code>uvm_object</code> .
<code>`uvm_field_array_string</code> (ARG, FLAG)	Dynamic array of string.

<code>`uvm_field_aa_int_<key_type> (</code> ARG, FLAG)	Associative array of <i>key_type</i> (an integer type: int , integer , byte , ...) with integral keys.
<code>`uvm_field_aa_int_enumkey (</code> ARG, FLAG)	Associative array of any integral type indexed by an enumerated key type.
<code>`uvm_field_aa_int_key (</code> ARG, FLAG)	Associative array of any integral type indexed by any integral key type.
<code>`uvm_field_aa_int_string (</code> ARG, FLAG)	Associative array of integral type with string keys.
<code>`uvm_field_aa_object_int (</code> ARG, FLAG)	Associative array of objects with int keys.
<code>`uvm_field_aa_object_string (</code> ARG, FLAG)	Associative array of <code>uvm_object</code> with string keys.
<code>`uvm_field_aa_string_int (</code> ARG, FLAG)	Associative array of string with int keys.
<code>`uvm_field_aa_string_string (</code> ARG, FLAG)	Associative array of string with string keys.
<code>`uvm_field_queue_enum (</code> ARG, FLAG)	Queue of enums.
<code>`uvm_field_queue_int (</code> ARG, FLAG)	Queue of packed integral type.
<code>`uvm_field_queue_object (</code> ARG, FLAG)	Queue of <code>uvm_object</code> .
<code>`uvm_field_queue_string (</code> ARG, FLAG)	Queue of string.
<code>`uvm_field_sarray_enum (</code> ARG, FLAG)	Static array of enums.
<code>`uvm_field_sarray_int (</code> ARG, FLAG)	(Fixed-size) array of packed integral type.
<code>`uvm_field_sarray_object (</code> ARG, FLAG)	(Fixed-size) array of <code>uvm_object</code> .
<code>`uvm_field_sarray_string (</code> ARG, FLAG)	(Fixed-size) array of string.

Flags

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

Field Macros

<code>UVM_ALL_ON</code>	All flags are on (default).
<code>UVM_DEFAULT</code>	Use the default settings.
<code>UVM_COPY, UVM_NOCOPY</code>	Do/Do not do a copy.
<code>UVM_COMPARE, UVM_NOCOMPARE</code>	Do/Do not do a compare.
<code>UVM_PRINT, UVM_NOPRINT</code>	Do/Do not print.
<code>UVM_PACK, UVM_NOPACK</code>	Do/Do not pack/unpack.
<code>UVM_REFERENCE</code>	Treat objects as only handles (i.e., no deep copy).
<code>UVM_PHYSICAL</code>	Treat as a physical field.
<code>UVM_ABSTRACT</code>	Treat as an abstract field.
<code>UVM_READONLY</code>	Do not allow this field to be set using <code>set_config_*</code> .
<code>UVM_BIN, UVM_DEC,</code> <code>UVM_UNSIGNED, UVM_OCT,</code> <code>UVM_HEX, UVM_STRING,</code> <code>UVM_TIME, UVM_ENUM,</code> <code>UVM_REAL, UVM_NORADIX</code>	Radix setting. The default is <code>UVM_HEX</code> .

Examples

```
class basic_transaction extends uvm_sequence_item;
  rand bit[7:0] addr, data;
  ...
  `uvm_object_utils_begin(basic_transaction)
    `uvm_field_int(addr,UVM_ALL_ON)
    `uvm_field_int(data,UVM_ALL_ON | UVM_BIN)
  `uvm_object_utils_end
endclass : basic_transaction
```

Tips

- Call field macro for every significant member of a transaction class or sequence.
- Declare as fields any data members that require configuration using `uvm_config_db`; for example, an instance of a virtual interface.
- Mark as “readonly” (`UVM_READONLY`) fields that you do not want to be affected by configuration.
- Technically, there is a difference between `UVM_ALL_ON` and `UVM_DEFAULT`, but practically, there is none. `UVM_DEFAULT` includes an additional “deep” option not included with `UVM_ALL_ON`, but the built-in

automation methods no longer use it. For now, both can be used interchangeably.

Gotchas

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

See also

Configuration

vuvm_heartbeat

Heartbeats use UVM's objection mechanism so that environments can ensure that their descendants are alive. A component that is being tracked by the heartbeat object must raise (or drop) the associated objection during the heartbeat window, which is defined by triggering a `uvm_event`. The synchronizing objection must be a `uvm_callbacks_objection` type.

The `uvm_heartbeat` object has a list of participating objects. The heartbeat can be configured so that all components (`UVM_ALL_ACTIVE`), exactly one (`UVM_ONE_ACTIVE`), or any component (`UVM_ANY_ACTIVE`) must trigger the objection in order to satisfy the heartbeat condition.

Declaration

```
class uvm_heartbeat extends uvm_object;
```

Methods

<pre>function new (string name, uvm_component cntxt, uvm_callbacks_objection objection = null);</pre>	Creates a new heartbeat instance associated with a component. The objection associated with the heartbeat is optional, but it must be set before the heartbeat monitor will activate.
<pre>function uvm_heartbeat_modes set_mode (uvm_heartbeat_modes mode = UVM_NO_HB_MODE);</pre>	Retrieve the heartbeat mode, one of <code>UVM_ALL_ACTIVE</code> , <code>UVM_ONE_ACTIVE</code> , <code>UVM_ANY_ACTIVE</code> and <code>UVM_NO_HB_MODE</code> If mode is anything other than <code>UVM_NO_HB_MODE</code> this sets the mode.
<pre>function void set_heartbeat (uvm_event e, ref uvm_component comps[\$]);</pre>	Sets up the heartbeat event and assigns a list of objects to watch. The monitoring is started as soon as this method is called, provided a trigger event has been provided previously, or <code>e</code> is not <code>null</code> here.
<pre>function void add (uvm_component comp);</pre>	Add a single component to the set of components to be monitored.

function void remove (uvm_component comp);	Remove a single component to the set of components being monitored.
function void start (uvm_event e = null);	Starts the heartbeat monitor.
function void stop ();	Stops the heartbeat monitor.

Rules

- Once heartbeat monitoring has been started with a specific event, providing a new monitor event with `set_heartbeat` or `start` results in an error.
You should first call `stop`.

Example

```
// Create an objection for heartbeat monitoring
uvm_callbacks_objection hb_objection = new("hb_objection");

// An instance of this class will be monitored
class my_driver extends uvm_driver #(trans);
    ...
    task run_phase(uvm_phase phase);
        ...
        // This must happen between every two event triggers in my_env
        // This in effect says "I'm alive"
        hb_objection.raise_objection(this);
        ...
    endtask
    ...
endclass : my_driver

class my_env extends uvm_env;
    ...
    uvm_event hb_event;
    my_driver m_driver;

    uvm_heartbeat hb;

    function new (string name, uvm_component parent);
        super.new(name,parent);
        // Create a heartbeat for myenv using hb_objection.
```

uvm_heartbeat

```
uvm_heartbeat hb = new("hb", this, hb_objection);
hb.add(m_driver);
endfunction

...
task run_phase(uvm_phase phase);
  uvm_event hb_event = new ("hb_event");
  ...
  hb.set_mode(UVM_ALL_ACTIVE);
  hb.start(hb_event);

  fork
    forever
      // At least one raise or drop must occur between successive triggers
      #10 hb_event.trigger(this);
      ...
    join_any
    hb.stop();
    hb.remove(m_driver);
  endtask
  ...
endclass : my_env
```

Gotchas

add doesn't start monitoring and remove doesn't stop it; an explicit start or stop is required.

See also

[uvm_callbacks_objection](#); [uvm_event](#); [uvm_objection](#)

HDL Backdoor Access

UVM provides a set of DPI/PLI functions, which can read, write, force, and release values anywhere inside of a design under test. Probing down into the design is referred to as *HDL backdoor access*.

While traditional Verilog hierarchical references can accomplish the same goal, hierarchical references suffer from at least two issues. First, hierarchical references are not allowed in a package, which prevents classes with hierarchical references from being added to a package. Second, they are hard-coded, set at elaboration time, and not changeable during run-time, making it difficult for the testbench to be flexible on a per test basis. UVM's backdoor access has the advantage that the references are referred to using strings, allowing them to be read from a configuration file or dynamically constructed, and dynamically accessed at run-time.

Since DPI/PLI code is used for the access, the DPI/PLI object code must be linked into simulation, which happens automatically when using a vendor's pre-compiled library. The DPI/PLI access routines can be left out of compilation by using the following plusarg:

```
+define+UVM_HDL_NO_DPI
```

Using DPI/PLI typically requires simulation access enabled to the HDL paths of interest.

Declaration

```
'define UVM_HDL_MAX_WIDTH 1024  
  
parameter int UVM_HDL_MAX_WIDTH = `UVM_HDL_MAX_WIDTH;  
  
typedef logic [UVM_HDL_MAX_WIDTH-1:0] uvm_hdl_data_t;
```

Methods

<code>function int uvm_hdl_check_path(string path);</code>	Returns 1 if HDL path exists and 0 if non-existent.
<code>function int uvm_hdl_deposit(string path, uvm_hdl_data_t value);</code>	Sets HDL path to value. Returns 1 on success, 0 on failure.
<code>function int uvm_hdl_force(string path, uvm_hdl_data_t value);</code>	Forces the HDL path to value. Returns 1 on success, 0 on failure.
<code>task uvm_hdl_force_time(string path, uvm_hdl_data_t value, time force_time = 0);</code>	Forces value on the HDL path for the specified amount of time.

<pre>function int uvm_hdl_release_and_read(string path, inout uvm_hdl_data_t value);</pre>	<p>Releases the drive by <code>uvm_hdl_force</code> and reads back the released value.</p> <p>For variables, value remains unchanged until the next procedural assignment. For wires, value immediately updates with the resolved value.</p> <p>Returns 1 on success, 0 on failure.</p>
<pre>function int uvm_hdl_release(string path);</pre>	<p>Releases the drive by a <code>uvm_hdl_force</code>.</p> <p>Returns 1 on success, 0 on failure.</p>
<pre>function int uvm_hdl_read(string path, output uvm_hdl_data_t value);</pre>	<p>Returns the value of the specified HDL path.</p> <p>Returns 1 on success, 0 on failure.</p>

Example

```
// Create an objection for heartbeat monitoring
class fullchip_sb extends uvm_scoreboard;
  ...
  // Analysis port method that compares the returned value from the
  // design's interface with the RTL's current value
  function void write(input AXItran t);
    uvm_hdl_data_t value;

    // Probe into the design
    assert (uvm_hdl_read("top_tb.dut.csr", value)) else
      `uvm_error("FULLCHIPSB",
                 "Cannot read top_tb.dut.csr");

    assert (t.data === value)
      `uvm_error("FULLCHIPSB", $sformatf("Returned value
(%h) != RTL value (%h)", t.data, value))

  endfunction

endclass : fullchip_sb
```

Tips

Use a simulator's built-in pre-compiled DPI object code to simplify linking in the DPI access functions.

Gotchas

Most simulators require read or write access enabled to access the signals in the design. This typically turns off optimizations, which may impact performance.

See also

Register Layer

uvm_in_order_*_comparator

The `uvm_in_order_*_comparator` family of components can be used to compare two streams of transactions in a UVM environment. They each provide a pair of analysis exports that act as subscribers to the transaction streams (the streams typically originate from analysis ports on UVM monitors and drivers).

The transactions may be built-in types (e.g. int, enumerations, structs) or classes: you should use `uvm_in_order_class_comparator` to compare class objects, and `uvm_in_order_builtin_comparator` to compare objects of built-in type. In each case the type is set by a parameter. Both versions are derived from a parent class `uvm_in_order_comparator`; this underlying class is not normally appropriate in user code, and is not described here.

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

Declarations

```
class uvm_in_order_class_comparator #( type T = int )
  extends uvm_in_order_comparator #(T, ...);

class uvm_in_order_builtin_comparator #( type T = int )
  extends uvm_in_order_comparator #(T, ...);
```

Methods

function new (string name, uvm_component parent) ;	Constructor.
function void flush ();	Clears (mis)matches counts.

Members

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

Example

Using uvm_in_order_class_comparator within a scoreboard component

```
class cpu_scoreboard extends uvm_scoreboard;

  uvm_analysis_export #(exec_xact) af_iss_export;
  uvm_analysis_export #(exec_xact) af_cpu_export;
  uvm_in_order_class_comparator #(exec_xact) m_comp;

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

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

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

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

  virtual function void report_phase(uvm_phase phase);
    string txt;
    $sformat(txt, "#matches = %d, #mismatches = %d",
             m_comp.m_matches, m_comp.m_mismatches);
    `uvm_info("", txt,UVM_NONE)
  endfunction: report_phase
```

uvm_in_order_*_comparator

```
`uvm_component_utils(cpu_scoreboard)
endclass: cpu_scoreboard
```

Tips

- The comparator writes a message for each match and mismatch that it finds. These messages are of class "Comparator Match" and "Comparator Mismatch" respectively. You may wish to disable these messages or redirect them to a log file as shown in the example.
- If you need your comparator also to model the transformation that a DUT applies to its data, you may find `uvm_algorithmic_comparator` more appropriate – it allows you to incorporate a reference model of the DUT's data-transformation behavior in a convenient way.
- The `uvm_in_order_class_comparator` requires two methods to be defined in a transaction object—`compare` and `convert2string`. The `compare` method can be handled by field automation. Likewise, the `convert2string` method can be easily created using the `sprint` automation method:

```
function string convert2string();
    return sprint();
endfunction
```

- Consider using `uvm_in_order_class_comparator` as a base class for a type-specific comparator that can be built by the factory, for example:

```
class exec_comp extends
    uvm_in_order_class_comparator #(exec_xact);
    `uvm_component_utils(exec_comp)
    function new(string name, uvm_component parent);
        super.new(name,parent);
    endfunction: new
endclass: exec_comp
```

Gotchas

`uvm_in_order_class_comparator` uses transaction classes' `compare` member function (which is created for you by the field automation macros). To customize the behavior of `compare`, you must override the function `do_compare`.

See also

`uvm_analysis_port`; `uvm_analysis_export`; `uvm_transaction`;
`uvm_algorithmic_comparator`

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

Declaration

```
class uvm_monitor extends uvm_component;
```

Methods

<code>function new (string name, uvm_component parent = null);</code>	Constructor, mirrors the superclass constructor in <code>uvm_component</code>
---	---

Members

<code>uvm_analysis_port #(transaction_class_type) monitor_ap;</code>	Analysis port through which monitored transactions are delivered to other parts of the testbench. Note: this field is not defined in <code>uvm_monitor</code> , but should always be provided as part of any user extensions.
--	---

Example

```
class example_monitor extends uvm_monitor;  
  uvm_analysis_port #(example_transaction) monitor_ap;  
  example_virtual_if vif;  
  
  virtual function void build_phase(uvm_phase phase);  
    super.build_phase(phase);  
  endfunction: build_phase  
  ..  
  virtual task run_phase(uvm_phase phase);  
    example_transaction tr;  
    forever begin  
      // Start with a new, clean transaction so that  
      // already-monitored transactions are unaffected  
      tr = new;  
      // code to observe physical signal activity  
      // and assemble transaction data in tr  
      monitor_ap.write(tr);
```

uvm_monitor

```
end  
endtask  
  
'uvm_component_utils (example_monitor)  
  
endclass: example_monitor
```

Tips

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

Gotchas

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

See also

[uvm_agent](#)

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

Declaration

```
virtual class uvm_object extends uvm_void;
```

Methods

<code>function new(string name = "");</code>	Constructor.
<code>virtual function void set_name(string name);</code>	Sets the name.
<code>static function int get_inst_count();</code>	Returns running total count of number of uvm_object-based objects created.
<code>static function uvm_object_wrapper get_type();</code>	Returns the type proxy for this class (overridden by utils macro).
<code>virtual function int get_inst_id();</code>	Returns unique ID for object (count value when object created).
<code>virtual function string get_name();</code>	Returns the name.
<code>virtual function string get_type_name();</code>	Returns type name. Override unless utility macros called.
<code>virtual function string get_full_name();</code>	By default calls <code>get_name()</code> . When overridden by <code>uvm_component</code> it returns the full hierarchical path name.
<code>virtual function uvm_object_wrapper get_object_type ();</code>	Returns the type proxy for this object type (overridden by utils macro).
<code>virtual function uvm_object create(string name = "");</code>	Creates a new object. Override unless utility macros called.
<code>virtual function uvm_object clone();</code>	Creates a copy of the object.

uvm_object

function bit compare (uvm_object rhs, uvm_comparer comparer = null);	Comparison against rhs.
function void copy (uvm_object rhs);	Copies rhs to this.
function void print (uvm_printer printer = null);	Prints the object.
function string sprint (uvm_printer printer = null);	Prints the object to a string.
function void record (uvm_recorder recorder = null);	Used for transaction recording.
function int pack (ref bit bitstream[], input uvm_packer packer = null);	Packs object to array of bits. Returns number of bits packed.
function int pack_bytes (ref byte unsigned bytestream[], input uvm_packer packer = null);	Packs object to array of bytes. Returns number of bytes packed.
function int pack_ints (ref int unsigned intstream[], input uvm_packer packer = null);	Packs object to array of ints. Returns number of ints packed.
function int unpack (ref bit bitstream[], input uvm_packer packer = null);	Unpacks array of bits to object.
function int unpack_bytes (ref byte unsigned bytestream[], input uvm_packer packer = null);	Unpacks array of bytes to object.
function int unpack_ints (ref int unsigned intstream[], input uvm_packer packer = null);	Unpacks array of ints to object.
virtual function bit do_compare (uvm_object rhs, uvm_comparer comparer);	Override for custom compare (called by compare).
virtual function void do_copy (uvm_object rhs);	Override for custom copying (called by copy).
virtual function void do_pack (uvm_packer packer);	Override for custom packing (called by pack).
virtual function void do_print (uvm_printer printer);	Override for custom printing (called by print).
virtual function void do_record (uvm_recorder recorder);	Override for custom reporting (called by report).

<pre>virtual function void do_unpack(uvm_packer packer);</pre>	Override for custom unpacking (called by unpack).
<pre>function void reseed();</pre>	Set seed based on object type and name if use_uvm_seeding = 1.
<pre>virtual function void set_object_local (string field_name, uvm_object value, bit clone = 1, bit recurse = 1);</pre>	Access method to set an object value used with the field automation macros.
<pre>virtual function void set_int_local (string field_name, uvm_bitstream_t value, bit recurse = 1);</pre>	Access method to set an integral value used with the field automation macros.
<pre>virtual function void set_string_local (string field_name, string value, bit recurse = 1);</pre>	Access method to set a string value used with the field automation macros.

Factory interface

The component utility macros register a component with the factory. They also define a nested proxy class named `type_id`, which provides a very convenient way to use the factory. For example, calling `my_object::type_id::create` will use the factory to create an instance of `my_object` by calling the `create` method of the nested `type_id` class. See `uvm_factory` for more details.

<pre>static function T type_id::create(string name, uvm_component parent, string context = "");</pre>	Create an instance of the object (type T) using the factory.
<pre>virtual function string type_id::get_type_name();</pre>	Returns the type name of the object as a string.
<pre>static function void type_id::set_type_override(uvm_object_wrapper override_type, bit replace = 1);</pre>	Set a factory override for all instances of this object.
<pre>static function void type_id::set_inst_override(uvm_object_wrapper override_type, string inst_path, uvm_component parent = null);</pre>	Set a factory override for a specific instance or instances of this object.

uvm_object

Members

static bit use_uvm_seeding = 1;	Enables the UVM seeding mechanism (based on type and hierarchical name).
--	--

Macros

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

```
`uvm_object_utils(TYPE)
```

or

```
`uvm_object_utils_begin(TYPE)
  `uvm_field_*(ARG, FLAG)
  ...
`uvm_object_utils_end
```

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

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

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

Tips

- Objects that need to be configured automatically at run-time using UVM configurations should use `uvm_component` as their base class instead.
- Call the utility macros in derived classes to ensure the `get_type_name` and `create` functions are automatically generated. This will also enable these classes to be used with the UVM factory.

See also

`uvm_factory`; `uvm_printer`

Objections are used to determine when it is safe to end a phase and, ultimately, the test. (See [End of Test](#).) Components and tests can 'raise an objection' to the simulator ending a particular phase or the test as a whole. They must 'drop' the objection to allow the phase or test to complete. When all the raised objections are dropped and any (optionally specified) 'drain time' has elapsed, the phase or test is free to terminate.

Each phase has a built-in objection method. (There is also a built-in end of test objection, `uvm_test_done_objection`, but this is deprecated and should not be used in new designs.)

Declaration

```
class uvm_objection extends uvm_report_object;
```

Methods

<code>function new(string name = "");</code>	Constructor.
<code>virtual function void clear(uvm_object obj = null);</code>	Clears all objections.
<code>virtual function void raise_objection(uvm_object obj = null, string description = "", int count = 1)</code>	Raises the number of objections for the source object by <code>count</code> . The object is usually the <code>this</code> handle of the caller. If <code>object</code> is not specified or null, the implicit top-level component, <code>uvm_top</code> , is chosen.
<code>virtual function void drop_objection(uvm_object obj = null, string description = "", int count = 1)</code>	Drops the number of objections for the source object by <code>count</code> .
<code>function void set_drain_time(uvm_object obj = null, time drain);</code>	Sets the <i>drain time</i> of <code>obj</code> . This is the time to wait after all objections have been dropped, default 0 ns.
<code>function void display_objections(uvm_object obj = null, bit show_header = 1);</code>	Displays objection information about the given object, or <code>uvm_root</code> if this is null. <code>show_header</code> controls whether a header is output.
<code>function void get_objectors(ref uvm_object list[\$]);</code>	Returns the list of currently objecting objects.

uvm_objection

<code>function int get_objection_count(uvm_object obj = null);</code>	Returns the current number of objections raised by the given object.
<code>function int get_objection_total(uvm_object obj = null);</code>	Returns the current number of objections raised by the given object and all descendants.
<code>function time get_drain_time(uvm_object obj = null);</code>	Returns the current drain time set for the given object.
<code>function bit trace_mode (int mode = -1);</code>	Sets or returns the trace mode. ! turns tracing on; 0 turns tracing off.
<code>task wait_for(uvm_objection_event objt_event, uvm_object obj = null)</code>	Waits for the specified event to occur in the given object.

Callback Hooks

<code>virtual function void raised(uvm_object obj, uvm_object source_obj, string description, int count)</code>	Called when an objection is raised for obj. source_obj originally raised the objection.
<code>virtual function void dropped(uvm_object obj, uvm_object source_obj, string description, int count)</code>	Called when an objection is dropped.
<code>virtual function void all_dropped(uvm_object obj, uvm_object source_obj, string description, int count)</code>	Called when all objections have been dropped and the drain time has expired.

Example

```
task run_phase(uvm_phase phase);  
    phase.raise_objection(this);  
    forever  
    begin  
        my_transaction tx;  
        #10 phase.drop_objection(this);  
        seq_item_port.get_next_item(tx);  
        phase.raise_objection(this);  
        seq_item_port.item_done();
```

```
end  
endtask: run_phase
```

Tips

- Active components, such as sequences and drivers should use objections. Passive components such as monitors don't need to use objections.
- Objections can be tricky to get working, plus there is a bit of overhead associated with hierarchical objections. Therefore, it is best to place objections only at the top levels of an environment. For example, place objections in a test case (or top sequence) and inside a scoreboard—the test case knows when the stimulus is done and the scoreboard knows when all of the responses have come back from the design.
- A top-level sequence should raise a phase objection at the beginning of an activity that must be completed before the phase stops, and drop the objection at the end of that activity. Once all of the raised objections are dropped, the phase terminates.
- Set a drain time to inject a delay between the time a component's total objection count reaches zero for the current phase and when the drop is passed to its parent. Typically, a single drain is set at the env or test level. Alternatively, you could use the `all_dropped` callback for more precise control.
- Each simulation phase includes a built-in objection, so there is no need for users to create instances of `uvm_objection`.
- Use the command-line plusarg `+UVM_OBJECTION_TRACE` to trace objection activity and find out why a simulation is not stopping when it should.
- If you want to define the callbacks, use class `uvm_callbacks_objection` instead of `uvm_objection`, and extend `uvm_objection_callback`.
- To manually force a simulation to shutdown before all objections are dropped, call the `clear` method on the phase objection object. Alternatively, the `uvm_heartbeat` mechanism or a call to `uvm_report_fatal` could be used.

See also

`uvm_callbacks_objection`; `uvm_objection_callback`; `uvm_heartbeat`; Phases

For further information on this and other topics, please see the UVM tutorials at <http://www.doulos.com/knowhow/sysverilog/uvm/>.

uvm_objection_callback

Callback class for `uvm_callbacks_objection`. The methods in this class may be defined in order to take some action when an objection is raised or dropped.

Declaration

```
class uvm_objection_callback extends uvm_callback;  
typedef uvm_callbacks #(uvm_objection,  
    uvm_objection_callback) uvm_objection_cbs_t;
```

Methods

<code>function new(string name);</code>	Constructor.
<code>virtual function void raised (uvm_objection objection, uvm_object obj, uvm_object source_obj, string description, int count);</code>	Called when an objection is raised. The arguments are passed from <code>raise_objection</code> .
<code>virtual function void dropped (</code> <code> uvm_objection objection, uvm_object obj, uvm_object source_obj, string description, int count);</code>	Called when an objection is dropped. The arguments are passed from <code>drop_objection</code> .
<code>virtual task all_dropped (</code> <code> uvm_objection objection, uvm_object obj, uvm_object source_obj, string description, int count);</code>	Called when an objection is dropped and the number of raised objections has reached zero. The arguments are passed from <code>drop_objection</code> .

These methods are called by the corresponding methods in `uvm_callbacks_objection`.

Gotcha

Be careful about the class name. This class, `uvm_objection_callback` is used to define the callback methods. `uvm_callbacks_objection` is an extended version of `uvm_objection`, which includes the callbacks.

See also

`uvm_objection`; `uvm_callbacks_objection`

When a test is started by calling `run_test`, the simulation proceeds by calling a pre-defined sequence of functions and tasks in every component. Each step in this sequence is known as a *phase*. Phases provide a synchronization mechanism between activities in multiple components.

All components implement a common set of phases. These are represented as objects of `uvm_topdown_phase` or `uvm_bottomup_phase`, except `run_ph` which is an implementation of `uvm_task_phase`. All these are extensions of `uvm_phase`. It is also possible for users to create custom phases which can be inserted into the standard UVM phase sequence.

During each phase, a corresponding callback function or task in each component in the hierarchy is invoked in either top-down or bottom-up order (depending on the phase).

The phase callback tasks and functions in a UVM component are empty (do nothing) by default – they are optionally overridden in components derived from `uvm_component` to implement the required behavior.

A phase can jump to the start of another phase by calling `jump` and specifying the destination phase. When the phases in one component jumps backwards, the phase tasks in the other components continue but wait at the next phase boundary for the phase jumping component to catch up.

Standard UVM Phases

For readability, the '`uvm_`' prefix has been omitted from the phase names here.

Phase Name (in order of execution)	Callback Type	Order	Main Activity
<code>build_phase</code>	function	top-down	Call factory to create child components
<code>connect_phase</code>	function	bottom-up	Connect ports, exports and channels
<code>end_of_elaboration_phase</code>	function	bottom-up	Check connections (hierarchy fixed)
<code>start_of_simulation_phase</code>	function	bottom-up	Prepare for simulation (e.g. open files, load memories)
<code>run_phase</code>	task	bottom-up	Run simulation until explicitly stopped or maximum time step reached

Phases

<code>extract_phase</code>	function	bottom-up	Collect results
<code>check_phase</code>	function	bottom-up	Check results
<code>report_phase</code>	function	bottom-up	Issue reports
<code>final_phase</code>	function	top-down	Close files; terminate co-simulation engines

Standard UVM Run-Time Schedule

The run-time schedule is the pre-defined phase schedule which runs concurrently with the `run_ph` global run phase. These alternative phases are intended to simplify the synchronization of actions during the `run_ph` across multiple components. By default, all `uvm_components` using the run-time schedule are synchronized with respect to the pre-defined phases in the schedule. A typical example would be to code all reset actions within the components' `reset_phase` tasks to ensure that every component has completed its reset activities before any component starts its `configure_phase` task.

(It is also possible for components to belong to different domains in which case their schedules can be unsynchronized.)

The runtime phases are all instances of `uvm_task_phase`, which is an extension of `uvm_phase`. For readability, the '`uvm_`' prefix has been omitted from the phase names here.

<i>Phase Name (in order of execution)</i>	<i>Main Activity</i>
<code>pre_reset_phase</code>	Prepare for reset to be asserted.
<code>reset_phase</code>	Perform reset.
<code>post_reset_phase</code>	Initiate idle transaction and interface training.
<code>pre_configure_phase</code>	Modify DUT configuration information.
<code>configure_phase</code>	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.
<code>post_configure_phase</code>	Wait for configuration information to propagate and take effect.
<code>pre_main_phase</code>	Wait for interface training to complete.
<code>main_phase</code>	Execute transactions, start sequences.
<code>post_main_phase</code>	Test stimulus is finished.
<code>pre_shutdown_phase</code>	Beginning of shutdown phase.

<code>shutdown_phase</code>	Wait for DUT pipelines etc. to clear
<code>post_shutdown_phase</code>	Perform final checks

User-Defined Phases

You can define a custom phase by extending an appropriate phase class: `uvm_phase`, `uvm_topdown_phase` or `uvm_bottomup_phase`; implementing `exec_task` or `exec_func`; instantiating a singleton instance for global use; and inserting the phase in a schedule by editing `uvm_phase::define_phase_schedule` to call `uvm_phase::add_phase`.

Gotchas

The `global_stop_request()` method is deprecated. Environments should use objections for phase synchronization and timing. (See [End of Test](#).) All environments using the `global_stop_request()` mechanism must add the switch `+UVM_USE_OVM_RUN_SEMANTIC`.

Examples

You will find examples of the various phases throughout this Guide

uvm_phase

`uvm_phase` is the base class for all the UVM simulation phases. It should not be extended directly. Instead, one of the predefined phases can be extended: `uvm_task_phase`, `uvm_topdown_phase`, and `uvm_bottomup_phase`. A `uvm_phase` object is passed as an argument into every UVM phase method of each component, which can then be used for a variety of purposes. Most commonly, the `uvm_phase` object is used for raising and dropping objections to prevent moving on to the next UVM phase. The `uvm_phase` object can also be used to jump between run-time phases.

Declaration

```
virtual class uvm_phase;
```

Methods

<code>function new(string name = "uvm_phase", uvm_phase_type phase_type = UVM_PHASE_SCHEDULE, uvm_phase parent = null);</code>	Constructor.
<code>function void add(uvm_phase phase, uvm_phase with_phase=null, uvm_phase after_phase=null, uvm_phase before_phase=null);</code>	Adds a phase to a schedule.
<code>virtual function void drop_objection (uvm_object obj, string description="", int count=1);</code>	Drops the phase objection.
<code>virtual function void exec_func(uvm_component comp, uvm_phase phase);</code>	Executes the phase's function-based functionality.
<code>virtual task exec_task(uvm_component comp, uvm_phase phase);</code>	Executes the phase's task- based functionality.
<code>function uvm_phase find(uvm_phase phase, bit stay_in_scope=1);</code>	Finds phase by phase object.
<code>function uvm_phase find_by_name (string name, bit stay_in_scope=1);</code>	Finds phase by name.
<code>function uvm_domain get_domain();</code>	Returns enclosing domain.
<code>function string get_domain_name();</code>	Returns the domain name.

<code>virtual function string get_full_name();</code>	Returns path from enclosing domain to this schedule.
<code>function uvm_phase get_imp();</code>	Returns the phase implementation.
<code>function uvm_phase get_jump_target();</code>	Returns the target phase of the current jump.
<code>function uvm_phase get_parent();</code>	Returns the parent schedule node.
<code>function uvm_phase_type get_phase_type();</code>	Returns phase type.
<code>function int get_run_count();</code>	Number of times the phase has been executed.
<code>function uvm_phase get_schedule(bit hier=0);</code>	Returns top schedule node.
<code>function string get_schedule_name(bit hier=0);</code>	Returns schedule name of current phase.
<code>function uvm_phase_state get_state();</code>	Returns current state of phase.
<code>function bit is(uvm_phase phase);</code>	Returns true if phase argument the same as current phase.
<code>function bit is_after(uvm_phase phase);</code>	Returns true if phase argument is after current phase.
<code>function bit is_before(uvm_phase phase);</code>	Returns true if phase argument is before current phase.
<code>function void jump(uvm_phase phase);</code>	Forces a jump to another phase.
<code>static function void jump_all(uvm_phase phase);</code>	Force all schedule phases to jump to another phase.
<code>virtual function void raise_objection (uvm_object obj, string description="", int count=1);</code>	Raises the phase objection.
<code>function void sync(uvm_domain target, uvm_phase phase=null, uvm_phase with_phase=null);</code>	Used to synchronize between domains.

uvm_phase

<pre>function void unsync(uvm_domain target, uvm_phase phase=null, uvm_phase with_phase=null);</pre>	Removes domain synchronization.
<pre>task wait_for_state(uvm_phase_state state, uvm_wait_op op=UVM_EQ);</pre>	Waits until the current state has the given state and op.

Example

```
class mytest extends uvm_test;
  ...
  virtual task run_phase(uvm_phase phase);
    uvm_objection obj;
    obj = phase.get_objection();
    obj.set_drain_time(10ns);

    phase.raise_objection();
    `uvm_do_on(seq, seqr);
    phase.drop_objection();
  endtask
endclass
```

Tips

- To create a user-defined phase, use one of the following predefined phases `uvm_task_phase`, `uvm_topdown_phase`, or `uvm_bottomup_phase`.
- The `phase` argument of any phase method can be used to get the current objection object and change the objection's drain time (see example above).

Gotchas

- Pre-UVM 2.0, jumping between phases may not work as expected.

See also

Phases

A uvm_pool is a dynamic associative array. The uvm_pool is parameterizable and works very much like a regular SystemVerilog associative array. Its advantage as a class object is that it can be dynamically created (instead of statically declared), passed by reference, and accessed globally through a global object.

Just as SystemVerilog provides methods for associative arrays, uvm_pool implements the analogous counterparts. Values are added to a pool by using add and retrieved using get. A global pool can be created for sharing items across a verification environment by using get_global_pool.

UVM provides a specialized uvm_pool for associative arrays of uvm_objects indexed by string called uvm_object_string_pool.

Declaration

```
class uvm_pool #(type KEY = int, T = uvm_void)
  extends uvm_object;
  typedef uvm_pool #(KEY,T) this_type;

class uvm_object_string_pool #(type T = uvm_object)
  extends uvm_pool #(string, T);
  typedef uvm_object_string_pool #(T) this_type;
```

Methods

uvm_pool

function new (string name = "");	Constructor.
virtual function void add (KEY key, T item);	Adds the key / item pair to the pool.
virtual function uvm_object create (string name = "");	Returns a newly created uvm_pool object.
virtual function void delete (KEY key);	Removes item from pool with the specified key.
virtual function int exists (KEY key);	Returns 1 if key exists in pool, 0 otherwise.
virtual function int first (ref KEY key);	Returns the key of the first pool entry.

uvm_pool

virtual function T get (KEY key);	Returns the pool entry at the specified key. If entry not found, an entry is created and returned for the key with the default SystemVerilog value for data type T.
static function T get_global (KEY key);	Returns the global pool entry at the specified key.
static function this_type get_global_pool() ;	Returns a reference to the global pool.
virtual function string get_type_name() ;	Returns the type name of the pool object.
virtual function int last (ref KEY key);	Returns the key of the last item in the pool. Returns 0 if pool is empty, and 1 if non-empty.
virtual function int next (ref KEY key);	Returns the key of next item in pool. Returns 0 if pool is empty, and 1 if next key is found.
virtual function int num() ;	Returns the number of keys in the pool.
virtual function int prev (ref KEY key);	Returns the key for the item before the specified key. Returns 0 if pool is empty, and 1 if previous key is found.

Also inherited are the standard methods from `uvm_object`.

uvm_object_string_pool

function new (string name = "");	Constructor.
virtual function string get_type_name() ;	Returns the type name of the pool object.
static function this_type get_global_pool() ;	Returns a reference to the global pool.
static function T get_global (string key);	Returns the global pool entry at the specified key.

virtual function T get (string key);	Returns the object for the specified key. If entry not found, an entry is created and returned for the key with the default SystemVerilog value for data type T.
virtual function void delete (string key);	Removes the object from the pool with the specified key.

Also inherited are the standard methods from `uvm_object`.

Example

```
// Create a global pool somewhere like in a package
package my_pkg;
    uvm_pool #(string,axi_cfg) m_global_pool =
        uvm_pool#(string,axi_cfg)::get_global_pool();
endpackage

// Add config objects to the global pool in the testcase
begin
    axi_cfg  axi_cfg1, axi_cfg2, axi_cfg3;

    axi_cfg1 = new(...);
    axi_cfg2 = new(...);
    axi_cfg3 = new(...);

    axi_cfg1.randomize with { data < 8'hffff; ... };
    axi_cfg2.randomize with { ... };
    axi_cfg3.randomize with { ... };

    my_pkg::m_global_pool.add("axi_cfg1", axi_cfg1);
    my_pkg::m_global_pool.add("axi_cfg2", axi_cfg2);
    my_pkg::m_global_pool.add("axi_cfg2", axi_cfg3);
end

// Retrieve the config object in a test sequence
task body();
    axi_cfg  cfg;
    cfg = my_pkg::m_global_pool.get("axi_cfg1");

    `uvm_do_with(req, { req.data == cfg.data, ... })
endtask
```

Tips

- `uvm_pool` provides a custom `print` method for displaying its contents.
- `uvm_pool` provides a custom `copy` method for copying a pool.

Gotchas

- A `uvm_pool` uses a SystemVerilog associative array to store its information. SystemVerilog limits key types to strings, classes, structs, and integral data types (e.g., `integer`, `int`, `bit`, etc.). Real values are illegal. Associative arrays do not support index values containing 4-state values of `X` or `Z`. SystemVerilog uses numerical order for numbers, alphabetical order for strings, and arbitrary ordering for all other data types.
- The `create` method is not static—it only works on an existing instance of `uvm_pool`.

See also

`uvm_queue`

Class `uvm_port_base` is the base class for all UVM ports and exports, in both the TLM-1 and TLM-2.0 subsets. It provides a set of common functions for connecting and interrogating ports and exports.

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

Declaration

```
virtual class uvm_port_base #(type IF=uvm_void) extends IF;  
  
typedef enum {  
    UVM_PORT ,  
    UVM_EXPORT ,  
    UVM_IMPLEMENTATION  
} uvm_port_type_e;  
  
typedef uvm_port_component_base uvm_port_list[string];
```

Methods

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

uvm_port_base

<code>function bit is_imp();</code>	True if imp.
<code>function int size();</code>	Number of connected interfaces for "multi-port".
<code>function void set_default_index(int index);</code>	set default interface of multi-port.
<code>function void connect(this_type provider);</code>	Connect to port/export [†] .
<code>function void debug_connected_to(int level = 0 , int max_level = -1);</code>	Print locations of interfaces connected to port. Recurse through multi-ports as necessary [†] .
<code>function void debug_provided_to(int level = 0 , int max_level = -1);</code>	Print locations of ports connected to export. Recurse through multi-ports as necessary [†] .
<code>function void get_connected_to(ref uvm_port_list list);</code>	Returns list of ports/exports connected to port.
<code>function void get_provided_to(ref uvm_port_list list);</code>	Returns list of ports connected to export.
<code>function void resolve_bindings();</code>	Resolve port connections (called automatically).
<code>function uvm_port_base #(IF) get_if(int index = 0);</code>	Returns the selected interface of multi-port.
<code>function void set_if(int i = 0);</code>	Select indexed interface of multi-port.

Definitions

<code>typedef uvm_port_base #(IF) this_type;</code>	Base type of port/export with interface IF.
---	---

See also

TLM-1 Ports, Exports and Imps

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

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

```
`uvm_field_int ( data, UVM_ALL_ON | UVM_HEX )
```

tells the UVM machinery to provide all the automation functions, including printing, and to print the data out in hexadecimal format. Automatic printing of fields can be enabled and disabled using the `UVM_PRINT` and `UVM_NOPRINT` options respectively:

```
`uvm_field_int ( data, UVM_PRINT | UVM_DEC )
```

or

```
`uvm_field_int ( data, UVM_NOPRINT )
```

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

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

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

Printer Types

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

- a tabular printer for printing in columnar format (`uvm_table_printer`)

Print

- a tree printer for printing objects in a tree format (`uvm_tree_printer`)
- a line printer that prints objects out on a single line (`uvm_line_printer`)

The default `uvm_printer` type prints objects in the raw format:

`object.members (type) (size) value`

for example:

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

The `uvm_table_printer` prints objects in the following tabular format – first a header, then the object, then its fields:

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

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

```
packet_obj: (packet_object) {
  data: {
    [0]: 'd281
    [1]: 'd428
    [2]: 'd62
    [3]: 'd892
```

```
[4]: 'd503
[5]: 'd74
}
addr: 'h95e
size: tiny
tag: good
}
```

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

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

Printer Functions

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

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

- `print_object_header()`
- `print_array_header()`
- `print_field()`
- `print_field()`
- `print_field()`
- `print_field()`
- `print_field()`
- `print_field()`
- `print_array_footer()`
- `print_string()`
- `print_string()`
- `print_string()`

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

Printer Knobs

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

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

Globals

In every UVM environment, four global printers are available:

```
uvm_default_table_printer  
uvm_default_tree_printer  
uvm_default_line_printer  
uvm_default_printer
```

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

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

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

```
initial  
  uvm_default_printer = uvm_default_tree_printer;
```

Example

See `uvm_printer` and `uvm_printer_knobs`.

Tips

- To globally change the format of all print messages, assign the `uvm_default_printer` a specific printer type or change the `uvm_default_printer.knobs`.

Gotcha

Redefining the printer functions to create a new printer type requires a bit of work and finesse. They are not as straightforward as they may appear! It is often easier to copy and modify the functions from one of the standard printers than to create them from scratch.

See also

`uvm_printer; uvm_printer_knobs`

uvm_printer

The `uvm_printer` class provides a facility for printing a `uvm_object` in various formats when the object's `print` function is called. The field automation macros specify the fields that are passed to the printer and their required format.

Alternatively, an object's virtual `do_print` function may be overridden to print its fields explicitly by calling member functions of `uvm_printer` (`do_print` is called implicitly by `print`).

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

<code>uvm_printer</code>	Raw, unformatted dump of object
<code>uvm_table_printer</code>	Prints object in tabular format
<code>uvm_tree_printer</code>	Prints multi-line tree format
<code>uvm_line_printer</code>	Prints all object information on a single line

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

Four default printers are globally instantiated in every UVM environment:

<code>uvm_default_printer</code>	Default table printer used by <code>uvm_object::print()</code> or <code>uvm_object::sprint()</code> when no printer is specified
<code>uvm_default_line_printer</code>	Line printer that can be used with <code>uvm_object::do_print()</code>
<code>uvm_default_tree_printer</code>	Tree printer that can be used with <code>uvm_object::do_print()</code>
<code>uvm_default_table_printer</code>	Table printer that can be used with <code>uvm_object::do_print()</code>

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

Declaration

```
class uvm_printer;
```

Methods

<pre>function void print_int (string name, uvm_bitstream_t value, int size, uvm_radix_enum radix =UVM_NORADIX, byte scope_separator = ".", string type_name = "");</pre>	Prints an integral field.
<pre>virtual function void print_generic(string name, string type_name, int size, string value, byte scope_separator = ".");</pre>	Prints a field with the specified name, type name, size, and value.
<pre>virtual function void print_object(string name, uvm_object value, byte scope_separator = ".");</pre>	Prints an object.
<pre>virtual function void print_string(string name, string value, byte scope_separator = ".");</pre>	Prints a string.
<pre>virtual function void print_time(string name, time value, byte scope_separator = ".");</pre>	Prints a time value.
<pre>virtual function void print_array_footer(int size = 0);</pre>	Prints footer information for arrays and marks the completion of array printing
<pre>virtual function void print_array_header(string name, int size, string arraytype = "array", byte scope_separator = ".");</pre>	Prints header information for arrays
<pre>virtual function void print_array_range(int min, int max);</pre>	Prints a range using ellipses for values
<pre>virtual protected function string adjust_name(string id, byte scope_separator = ".");</pre>	Prints a field and id name.
<pre>virtual function string emit();</pre>	Emits a string for an object.

uvm_printer

<code>virtual function string format_row(uvm_printer_row_info row);</code>	Customizable method for printing a row.
<code>virtual function string format_header();</code>	Customizable method for printing a header.

[†]Only use in derived printer classes

Members

<code>uvm_printer_knobs knobs;</code>	Knob object providing access to printer knobs
<code>string m_string</code>	Printer output is written to this string when <code>sprint</code> knob is set to 1 (only use in derived printer classes)

Example

```
class my_object extends uvm_object;
    int addr = 198;
    int data = 89291;
    string name = "This is my test string";
    `uvm_object_utils_begin( my_object )
        `uvm_field_int( addr, UVM_ALL_ON )
        `uvm_field_int( data, UVM_ALL_ON )
        `uvm_field_string( name, UVM_ALL_ON )
    `uvm_object_utils_end
    ...
endclass : my_object

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

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

        // Print using the line printer
    end
```

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

Produces the following simulation results:

```
# This is from the table printer
```

Name	Type	Size	Value
my_obj	my_object	-	@726
addr	integral	32	'hc6
data	integral	32	'h15ccb
name	string	22	This is my test str+

```
# This is from the tree printer
```

```
my_obj: (my_object)  {
    addr: 'hc6
    data: 'h15ccb
    name: This is my test string
}
```

```
# This is from the line printer
```

```
my_obj: (my_object)  { addr: 'hc6 data: 'h15ccb name: This
is my test string }
```

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

```
class my_printer extends uvm_table_printer;

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

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

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

uvm_printer

```
my_printer my_special_printer = new();  
  
module top;  
    my_object my_obj = new( "my_obj" );  
    initial begin  
        #100;  
        // Print using my_printer  
        my_obj.print( my_special_printer );  
    end  
endmodule : top
```

Produces the following simulation results:

```
Printing object my_obj at time 100:  
-----
```

Name	Type	Size	Value
my_obj	my_object	-	@726
addr	integral	32	'hc6
data	integral	32	'h15ccb
name	string	22	This is my test str+

Tips

- Set `uvm_default_printer` to `uvm_default_line_printer`, `uvm_default_tree_printer`, or `uvm_default_table_printer` to control the default format of the object printing.
- The `print_time` function is subject to the formatting set by the `$timeformat` system task.
- The `print_object` task prints an object recursively, based on the depth knob of the default printer knobs (see `uvm_printer_knobs`). By default, components are printed, but this can be disabled by setting the `uvm_component::print_enabled` bit to 0 for specific components that should not be automatically printed.

Gotchas

- The printing facility is limited to printing values up to 4096 bits.
- The UVM printing facility is separate from the reporting facility and is not affected by the severity or verbosity level settings. It is possible to create a customized printer that takes account of the reporting verbosity settings (by overriding its `print_object` function for example). A better alternative is to call `sprint` on an object and pass the string into a reporting method. For example,

```
uvm_report_info("", tr.sprint());
```

This produces the following simulation result:

```
UVM_INFO @ 100ns: uvm_test_top.m_env.m_driver []:
```

Name	Type	Size	Value
my_obj	my_object	-	@726
addr	integral	32	'hc6
data	integral	32	'h15ccb
name	string	22	This is my test str+

See also

Print; uvm_printer_knobs

uvm_printer_knobs

Printer knobs provide control over the formatting of a `uvm_printer`'s output. The `uvm_printer_knobs` class contains a set of variables that are common to all printers. The knobs class can be extended to include additional controls for derived printers. UVM defines two typedefs for compatibility with OVM: `uvm_table_printer_knobs` and `uvm_tree_printer_knobs`.

Declaration

```
class uvm_printer_knobs;  
  
typedef uvm_printer_knobs uvm_table_printer_knobs;  
typedef uvm_printer_knobs uvm_tree_printer_knobs;
```

Methods

<code>function string get_radix_str(uvm radix enum radix);</code>	Returns <i>radix</i> in a printable form
--	--

Members

From `uvm_printer_knobs`:

<code>int begin_elements = 5;</code>	Number of elements at the head of a list that should be printed.
<code>string bin_radix = "'b";</code>	String prepended to any integral type when <code>UVM_BIN</code> used for a radix.
<code>string dec_radix = "'d";</code>	String prepended to any integral type when <code>UVM_DEC</code> used for a radix.
<code>uvm radix enum default_radix = UVM_HEX;</code>	Default radix to use for integral values when <code>UVM_NORADIX</code> is specified.
<code>int depth = -1;</code>	Indicates how deep to recurse when printing objects, where depth of -1 prints everything.
<code>int end_elements = 5;</code>	Number of elements at the end of a list that should be printed.
<code>bit footer = 1;</code>	Specifies if the footer should be printed.
<code>bit full_name = 0;</code>	Specifies if leaf name or full name is printed.

<code>bit header = 1;</code>	Specifies if the header should be printed.
<code>string hex_radix = "'h";</code>	String prepended to any integral type when UVM_HEX used for a radix.
<code>bit identifier = 1;</code>	Specifies if an identifier should be printed.
<code>int indent = 2;</code>	Number of spaces for level indentation.
<code>integer mcd = UVM_STDOUT;</code>	File descriptor or multi-channel descriptor where print output is directed.
<code>string oct_radix = "'o";</code>	String prepended to any integral type when UVM_OCT used for a radix.
<code>string prefix = "";</code>	Specifies string prepended to each line.
<code>bit reference = 1;</code>	Specifies if a unique reference ID for a uvm_object should be printed.
<code>string separator = "{}";</code>	(For tree printers only) Determines the opening and closing separators for nested objects.
<code>bit show_radix = 1;</code>	Specifies if the radix should be printed for integral types.
<code>bit show_root = 0;</code>	Indicates whether or not the initial object that is printed (when current depth is 0) prints the full path name. By default, only the leaf name is printed.
<code>bit size = 1;</code>	Specifies if the size of the field should be printed.
<code>bit type_name = 1;</code>	Specifies if the type name of a field should be printed.
<code>string unsigned_radix = "'d";</code>	Default radix to use for integral values when UVM_UNSIGNED is specified.

Example

Using default printer:

uvm_printer_knobs

```
uvm_default_printer.knobs.indent = 5; // Indent by 5 spaces  
uvm_default_printer.knobs.type_name = 0; // No type values
```

Output:

Name	Type	Size	Value
my_obj	-		@730
payload	6		-
[0]	32		'h95e
[1]	32		'h2668
[2]	32		'h206a
...			
[5]	32		'he7
crc	32		'h3
kind	32		RX
msg	7		send_tx

Using tree printer:

```
uvm_default_printer = uvm_default_tree_printer;  
uvm_default_printer.knobs.hex_radix = "0x"; // Change radix  
uvm_default_printer.knobs.separator = "@@";
```

Output:

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

Turning off identifiers (not very useful in practice):

```
uvm_default_printer = uvm_default_line_printer;  
uvm_default_printer.knobs.identifier = 0;
```

Output:

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

Tips

- To turn off all printing to STDOUT, set the `mcd` field to 0.
`uvm_default_printer.knobs.mcd = 0;`
- This will stop all standard printing messages issued for transactions and sequences.

Gotchas

- The results of the `reference` knob is simulator-dependent.
- When negative numbers are printed, the radix is not printed.
- If the maximum width of a column is reached, then nothing else is printed until a new line is printed.

See also

Print

uvm_queue

A `uvm_queue` is a dynamic queue. The `uvm_queue` is parameterizable and works very much like a regular SystemVerilog queue. Its advantage as a class object is that it can be dynamically created (instead of statically declared), passed by reference, and accessed globally through a global object.

Just as SystemVerilog provides methods for queues, `uvm_queue` implements the analogous counterparts. Values are added to a queue by using `insert` and retrieved using `get`. A global queue can be created for sharing items across a verification environment by using `get_global_queue`.

Declaration

```
class uvm_queue #(type T = int) extends uvm_object;  
typedef uvm_queue #(T) this_type;
```

Methods

<code>function new(string name = "");</code>	Constructor.
<code>virtual function uvm_object create(string name = "");</code>	Returns a newly created <code>uvm_queue</code> object.
<code>virtual function void delete(int index = -1);</code>	Removes item from queue at the specified index. If no index is specified, the entire queue is deleted.
<code>virtual function T get(int index);</code>	Returns the pool entry at the specified key. If entry not found, an entry is created and returned for the key with the default SystemVerilog value for data type T.
<code>static function T get_global(int index);</code>	Returns the global queue entry at the specified key.
<code>static function this_type get_global_queue();</code>	Returns a reference to the global queue.
<code>virtual function string get_type_name();</code>	Returns "uvm_queue".
<code>virtual function void insert(int index, T item);</code>	Inserts item into queue at the specified index.

virtual function T pop_back() ;	Returns the last element in the queue. Returns null if queue is empty.
virtual function T pop_front() ;	Returns the first element in the queue. Returns null if queue is empty.
virtual function void push_back (T item);	Inserts item at the back of the queue.
virtual function void push_front (T item);	Inserts item at the front of the queue.
virtual function int size() ;	Returns the number of items in the queue.

Also inherited are the standard methods from `uvm_object`.

Example

```
'uvm_analysis_imp_decl(_pci)
'uvm_analysis_imp_decl(_usb)

class top_sb extends uvm_scoreboard;
  uvm_analysis_imp_usb #(usb_trans,top_sb) usb_ap;
  uvm_analysis_imp_pci #(pci_trans,top_sb) pci_ap;

  uvm_queue #(usb_trans) expected_queue;

  function void build_phase(uvm_phase phase);
    super.build_phase();
    usb_ap = new("usb_ap",this);
    pci_ap = new("pci_ap",this);

    // Create the queue
    expected_queue = new();
  endfunction : build_phase

  function void write_pci(pci_trans t);
    // Save expected result in queue
    expected_queue.push_back( compute_result(t) );
  endfunction : write_pci

  function void write_usb(usb_trans t);
```

```
// Compare the transaction with expected value from the queue
assert ( t.compare( expected_queue.pop_front() ) ) else
    `uvm_error("TOP_SB", "Actual != Expected!")
endfunction : write_usb
endclass
```

Tips

- **uvm_queue** provides a custom `copy` for copying a **uvm_queue**.
- **uvm_queue** provides a custom `convert2string` method for displaying the contents of a **uvm_queue**.
- **uvm_queue** uses an unbounded queue.

Gotchas

- A **uvm_queue** uses a SystemVerilog queue to store its information. Four-state index values of X and Z are automatically converted to 0. Index values greater or less than the queue size issue a warning and immediately return, having no effect.
- The `create` method is not static—it only works on an existing instance of **uvm_queue**.

uvm_random_stimulus

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

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

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

Declaration

```
class uvm_random_stimulus  
#(type T=uvm_transaction) extends uvm_component;
```

Methods

<code>function new(string name , uvm_component parent);</code>	Constructor
<code>virtual task generate_stimulus(T t = null, input int max_count = 0);</code>	Starts stimulus of length <code>max_count</code> (0 = infinite). Optional transaction template <code>t</code>
<code>virtual function void stop_stimulus_generation();</code>	Ends <code>generate_stimulus</code> task

Members

<code>uvm_blocking_put_port #(T) blocking_put_port;</code>	Port for generated stimulus
--	--------------------------------

Example

Using `uvm_random_stimulus` in an environment

```
class verif_env extends uvm_env;  
  uvm_random_stimulus #(basic_transaction) m_stimulus;  
  dut_driver m_driver;
```

uvm_random_stimulus

```
uvm_tlm_fifo #(basic_transaction) m_fifo;
int test_length;
...

virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    get_config_int("run_test_length",test_length);
    m_stimulus = new ("m_stimulus",this);
    m_fifo = new("m_fifo",this);
    m_driver
        = dut_driver::type_id::create("m_driver",this);
endfunction: build_phase

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

virtual task run_phase(uvm_phase phase);
    m_stimulus.generate_stimulus(null,test_length);
endtask: run_phase

`uvm_component_utils(verif_env)
endclass: verif_env
```

Tips

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

Gotchas

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

See also

Sequence; uvm_transaction

Register and Memory Sequences

The UVM library includes a number of pre-defined register and memory test sequences. Once a register model has been instantiated in an environment and integrated with the DUT, it is possible to execute any of the predefined register tests sequences to verify the proper operation of the registers and memories in the DUT.

You can combine them in a higher-level virtual sequence to better verify your design. Test sequences are not applied to any block, register, or memory with the NO_REG_TESTS attribute defined.

Some of the predefined test sequences require back-door access be available for registers or memories.

Declaration

```
class sequence_name  
  extends uvm_reg_sequence #(uvm_sequence #(uvm_reg_item));  
where sequence_name is the name of one of the pre-defined test sequences  
described in the table below.
```

Pre-Defined Test Sequences

uvm_reg_hw_reset_seq	Reads all the registers in a block and checks their values are the specified reset value.
uvm_reg_single_bit_bash_seq	Sequentially writes 1s and 0s in each bit of the register, checking it is appropriately set or cleared, based on the field access policy specified for the field containing the target bit.
uvm_reg_bit_bash_seq	Executes uvm_reg_single_bit_bash_seq for all registers in a block and sub-blocks.
uvm_reg_single_access_seq	For each address map in which the register is accessible, writes the register then confirms the value was written using the back-door. Subsequently writes a value via the backdoor and checks the corresponding value can be read through the address map.

Register and Memory Sequences

uvm_reg_access_seq	Executes uvm_reg_single_access_seq for all registers in a block and sub-blocks
uvm_mem_single_walk_seq	Writes a walking pattern into the memory then checks it can be read back with the expected value.
uvm_mem_walk_seq	Executes uvm_mem_single_walk_seq for all memories in a block and sub-blocks.
uvm_mem_single_access_seq	For each address map in which the memory is accessible, writes the memory locations for each memory then confirms the value was written using the back-door. Subsequently writes a value via the back-door and checks the corresponding value can be read through the address map.
uvm_mem_access_seq	Executes uvm_mem_single_access_seq for all memories in a block and sub-blocks.
uvm_reg_shared_access_seq	For each address map in which the register is accessible, writes the register via one map then confirms the value was written by reading it from all other address maps.
uvm_mem_shared_access_seq	For each address map in which the memory is accessible, writes each memory location via one map then confirms the value was written by reading it from all other address maps
uvm_reg_mem_shared_access_seq	Executes uvm_reg_shared_access_seq for all registers in a block and sub-blocks and uvm_mem_shared_access_seq for all memories in a block and sub-blocks.

uvm_reg_mem_built_in_seq	Execute all the selected predefined block-level sequences. By default, all pre-defined block-level sequences are selected
uvm_reg_mem_hdl_paths_seq	Verify the HDL path(s) specified for registers and memories are valid.

Attributes

The following bit-type resources in the resource database cause the corresponding tests to be skipped (see Examples):

Resource	Tests skipped
NO_REG_HW_RESET_TEST	uvm_reg_hw_reset_seq
NO_REG_TESTS	All
NO_REG_BIT_BASH_TEST	uvm_reg_single_bit_bash_seq, uvm_reg_bit_bash_seq
NO_REG_ACCESS_TEST	uvm_reg_single_access_seq, uvm_reg_access_seq
NO_MEM_WALK_TEST	uvm_mem_single_walk_seq, uvm_mem_walk_seq
NO_MEM_TESTS	All _mem and reg_mem tests
NO_MEM_ACCESS_TEST	uvm_mem_single_access_seq, uvm_mem_access_seq
NO_REG_SHARED_ACCESS_TEST	uvm_reg_shared_access_seq
NO_MEM_SHARED_ACCESS_TEST	uvm_mem_shared_access_seq

Rules

- These tests require the back-door be defined for the register:
`uvm_reg_single_access_seq` and `uvm_mem_single_access_seq`.
- These tests requires the register be mapped in multiple address maps:
`uvm_reg_shared_access_seq`, `uvm_mem_shared_access_seq` and
`uvm_reg_mem_shared_access_seq`

Example

If a bit-type resource named “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.

Register and Memory Sequences

```
uvm_resource_db#(bit)::set(  
    {"REG:::", regmodel.blk.get_full_name(), ".*"},  
    "NO_REG_TESTS", 1, this);
```

Using the hardware reset test

```
class smoke_test extends uvm_test;  
    my_env m_env;  
    ...  
    task main_phase(uvm_phase phase);  
        uvm_reg_hw_reset_test test_seq = new();  
        phase.raise_objection(this, "Running hw_reset test");  
        test_seq.model = m_env.regmodel;  
        test_seq.start(null);  
        phase.drop_objection(this, "End of hw_reset test");  
    endtask : main_phase  
endclass : smoke_test
```

Tips

Start with the simplest test—the hardware reset test—to debug the register model, the environment, the physical transactors, and the DUT to a level where it can be taken through more complicated tests.

Gotchas

- The DUT should be idle and not modify any register during these tests:
`uvm_reg_single_bit_bash_seq`, `uvm_reg_bit_bash_seq`,
`uvm_reg_single_access_seq`, `uvm_reg_access_seq`,
`uvm_reg_mem_access_seq`, `uvm_reg_shared_access_seq`, and
`uvm_reg_mem_shared_access_seq`.
- The DUT should be idle and not modify any memory during these tests:
`uvm_reg_mem_access_seq`, `uvm_mem_shared_access_seq`,
`uvm_reg_mem_shared_access_seq`, `uvm_mem_single_access_seq` and
`uvm_mem_access_seq`.

See also

[Register Layer; uvm_sequence](#)

Register Generators

UVM provides the class library for creating register models, but it is typically easier and more maintainable to represent a design's registers in a high-level description format and automatically generate the register model. UVM does not provide any specific register generators, but two popular generators are mentioned here for quick reference.

RALGEN

A popular tool for automatically generating register models is *ralgen*, which is provided as part of Synopsys' VCS simulator installation. For *ralgen* to generate the UVM register definitions, simply supply the additional **-uvm** command line switch:

```
% ralgen -l sv -t top -uvm ral_file.ralf
```

Ralgen uses a simple register definition syntax called *RALF* that is based on the Tcl scripting language. Standard Tcl commands may be used and multiple RALF files can be included using Tcl's *source* command. The following provides a summary of the RALF syntax.

Common Tcl Syntax

# Comment	Comment.
set <i>name</i> <i>value</i>	Sets variable <i>name</i> to <i>value</i> .
source <i>filename</i>	Reads in the specified Tcl file.
for {set i 0} {\$i < n} {incr i} ... }	For loop.
if (\$var) { ... }	If statement.

RALF Syntax

field – defines a register field

```
field name {  
    properties  
}
```

Properties

<i>bits</i> <i>n</i> ;	Number of bits.
-------------------------------	-----------------

Register Generators

access rw ro wo w1 ru wlc rc a1 a0 other user0 user1 user2 user3 dc;	Writeable (rw), read-only (ro), write-only (wo), write-once (w1), read-only with design update (ru), write 1 to clear (wlc), clear on read (rc), auto-set by design (a1), auto-cleared by design (a0), other (other), user-defined (user0–3), don't care (dc).
reset hard_reset value;	Hard reset value.
soft_reset value;	Soft reset value.
constraint name { expression }	Sets a constraint on field values. Use value to refer to a field value. Must be valid SystemVerilog syntax.
enum { name[=val], ... }	Symbolic name for field values
cover + - b f	Turns on/off field coverage. + - Include / exclude: b = bits from register-bit coverage f = fields value is a coverage goal
coverpoint { bins name [[[n]]] = { n [n:n], ... } default }	Defines bins for field value coverage. Must be valid SystemVerilog syntax.

register – defines a register

```
register name {  
    properties  
}
```

Properties

attributes { name value, ... }	User defined attribute.
bytes n;	Number of bytes in the register

constraint name [{ expression }]	Sets a constraint on field values. Use <i>fieldname.value</i> to refer to a field value. Must be valid SystemVerilog syntax.
cover + - a b f	Turns on/off register coverage. + - Include / exclude: a = address from address map b = bits from register-bit coverage f = fields from field value coverage
cross item1, item2, [...] [{ label name }]	Cross-coverage point. Label required if used in another cross-coverage point.
doc { text }	User specified documentation.
field name[=rename][[n]] [(hdl_path)] [@bit_offset[+incr]]; OR field name [[n]] [(hdl_path)] [@bit_offset[+incr]] { field properties }	Defines a specific register field. Use <i>name</i> of unused or reserved to skip bits. An index [n] indicates a field array. hdl_path represents hierarchical path to register for backdoor access. bit_offset direction relative to <i>left_to_right</i> value.
left_to_right;	Concatenate fields from left to right (the default is reverse).
noise ro rw no;	Defines register access during normal operations of device. ro = read at any time rw = read/write at any time no = not accessible during normal operations

Register Generators

shared [(hdl_path)];	Register is physically shared wherever it is instantiated. hdl_path represents hierarchical path to register for backdoor access.
-----------------------------	--

regfile – register file to define a group of registers

```
regfile name {  
    properties  
}
```

Properties

constraint name [{ expression }]	Sets a constraint on field values. Must be valid SystemVerilog syntax.
cover + - a b f	Turns on/off register file coverage. + - Include / exclude registers from: a = address map coverage b = register bits coverage f = field value coverage
doc { text }	User specified documentation.
register name[=rename] [[n]] [(hdl_path)] [@offset] [read write]; OR register name[[n]] [(hdl_path)] [@offset] { field properties }	Instances of a previously defined register. <i>name</i> must be unique. An index [n] indicates a register array. hdl_path represents hierarchical path to register for backdoor access. Register arrays must contain the %d placeholder for the index. offset may be specified as @none. <i>Cannot contain the shared property.</i>

shared [(hdl_path)];	Register file is physically shared by all domains in the block that instantiates it. hdl_path represents hierarchical path to register file for backdoor access.
-----------------------------	---

Memory – defines a memory

```
memory name {
    properties
}
```

Properties

access rw ro;	RAM (rw) or ROM (ro).
attributes { name value, ... }	User defined attribute.
bits n;	Number of bits in each memory location. <i>Required property.</i>
cover + - a	Turns on/off memory address coverage.
doc { text }	User specified documentation.
initial x 0 1 addr literal[++ -];	Initial memory contents: x = unknowns (default) 0 = zeros 1 = ones addr = physical address value literal = constant value (++ incrementing or – decrementing)

Register Generators

noise ro rw unused no;	Defines memory access during normal operations of device. ro = read at any time rw = read/write at any time unused = read/write to only unused memory locations no = not accessible during normal operations
shared [(hdl_path)];	Memory is physically shared wherever it is instantiated. <i>hdl_path</i> represents hierarchical path to memory for backdoor access.
size m[k M G];	Specifies memory size where k=kilobytes, M=megabytes, and G=gigabytes.

Virtual register – register across memory locations by combining virtual fields

```
virtual register name {  
    properties  
}
```

Properties

bytes n;	Number of bytes in the register.
field name[=rename] [@bit_offset]; OR field name [@bit_offset] { bits n; [doc { text }] }	Virtual field of the specified bits at bit offset. <i>At least one field property is required.</i>
left_to_right ;	Concatenate fields from left to right (the default is reverse).

Block – define a set of registers and memories

```
block name {  
    properties  
}
```

OR

```
block name {
    domain name {
        properties
    }
    domain name {
        properties
    }
    [doc {
        text
    }]
}
```

Properties

attributes { name value, ... }	User defined attribute.
bytes n;	Number of concurrent byte accesses through physical interface. <i>Required property.</i>
constraint name [{ expression }]	Sets a constraint on field values. Must be valid SystemVerilog syntax. <i>Cannot constraint memory or virtual register contents.</i>
cover + - a b f	Turns on/off register and memory coverage in this block (in this domain). + - Include / exclude from: a = address map coverage b = register bits coverage f = field value coverage
doc { text }	User specified documentation.
Endian little big fifo_ls fifo_ms;	Register endianess. Default little endian.

Register Generators

<pre>memory name[=rename] [(hdl_path)] [@offset] [read write]; OR memory name [(hdl_path)] [@offset] { properties }</pre>	Previously defined memory. hdl_path represents hierarchical path to register for backdoor access. Register arrays must contain the %d placeholder for the index. <i>Cannot contain the shared property.</i>
<pre>register name[=rename] [[i]] [(hdl_path)] [@offset] [+incr] [read write]; OR register name[[n]] [(hdl_path)] [@offset] [+incr] { properties }</pre>	Instances of a previously defined register. name must be unique. An index [n] indicates a register array. hdl_path same as above. offset may be specified as @none. <i>Cannot contain the shared property.</i>
<pre>regfile name[=rename] [[n]] [(hdl_path)] [@offset] [+incr]; OR regfile name[[n]] [(hdl_path)] [@offset] [+incr] { properties }</pre>	Previously defined register file. +incr represents the increment amount. Other options same as above.
<pre>virtual register name[=rename[n]] mem@offset [+incr]; OR virtual register name[[n]] mem@offset [+incr] { properties }</pre>	Previously defined virtual register. Options same as above.

system – define a set of blocks or subsystems

```
system name {
    properties
}
```

OR

```

system name {
    domain name {
        properties
    }
    domain name {
        properties
    }
    [doc {
        text
    }]
}

```

Properties

attributes { name value, ... }	User defined attribute.
block name[[.domain]=rename][[n]] [(hdl_path)] @offset [+incr]; OR block name[[n]] [(hdl_path)] @offset [+incr] { properties }	Previously defined register file. name must be unique. An index [n] indicates a block array. hdl_path represents hierarchical path to register for backdoor access. Register arrays must contain the %d placeholder for the index. +incr represents the increment amount.
bytes n;	Number of concurrent byte accesses through physical interface. Required property.
constraint name [{ expression }]	Sets a constraint on field values. Must be valid SystemVerilog syntax. Cannot constraint memory or virtual register contents.

Register Generators

cover + - a b f	Turns on/off register and memory coverage in this block (in this domain). + - Include / exclude from: a = address map coverage b = register bits coverage f = field value coverage
doc { text }	User specified documentation.
Endian little big fifo_ls fifo_ms;	Register endianess. Default little endian.
system <i>name</i> [<i>.domain</i>]= <i>rename</i> [[<i>n</i>]] [(<i>hdl_path</i>) @ <i>offset</i> [+ <i>incr</i>] ; OR system <i>name</i> [<i>n</i>] [<i>hdl_path</i>] @ <i>offset</i> [+ <i>incr</i>] { <i>properties</i> }	Previously defined virtual register. Options same as above.

RALF Example

```
register ctrl {  
    left_to_right;  
    field soft_RST;  
    field endianess;  
    field mode {  
        bits 3;  
        reset 0x0;  
  
        constraint my_lmt {  
            value < 3'h7;  
        }  
    }  
    field RDY;  
    ...  
}  
memory memsys {  
    size 256;  
    bits 32;  
}  
block sub_system {
```

```
bytes 4;  
Endian little;  
  
register ctrl @'h0;  
...  
memory memsys @'h1000;  
}
```

UVM Register and Memory Package (UVM_RGM)

The UVM Register and Memory Package is a UVM contribution by Cadence Design Systems that can be downloaded from the UVM World website. It is a separate register implementation built on top of UVM, but not on top of the UVM Register Layer.

UVM_RGM uses a SPIRIT IP-XACT XML file to capture a device's register description, which can easily be edited using any XML enabled viewer (like Eclipse). Once the registers are described, UVM_RGM uses a Java-based converter to generate the corresponding UVM register implementation:

```
% $JAVA_HOME/bin/java -jar  
$UVM_RGM_HOME/builder/ipxact/uvmrgm_ipxact2sv_parser.jar -i  
INPUT_FILE
```

The following is a brief description of the UVM_RGM register syntax:

Field

```
<spirit:field>
```

<code><spirit:access></code>	Register access (e.g., read-write)
<code><spirit:bitOffset></code>	Bit offset of the field.
<code><spirit:bitWidth></code>	Width of the field.
<code><spirit:name></code>	Name of the field.

Register Generators

```
<spirit:vendorExtensions>
```

<vendorExtensions:access_policy>	<p>RW1C – bitwise write of 1 clears bit</p> <p>RW1S – bitwise write of 1 sets bit</p> <p>RC – clear-on-read</p> <p>RS – set-on-read</p> <p>RWOnce – Write once, read first write value</p> <p>WOnce – Write once</p> <p>RW0C – bitwise write of 0 clears bits</p> <p>RW0S – bitwise write of 0 sets bits</p> <p>RWC – Writes clear register</p> <p>RWS – Writes set register</p> <p>RW1T – Write 1 to toggle</p> <p>RW0T – Write 0 to toggle</p> <p>R0W – WO, reads return 0</p> <p>R1W – WO, reads return 1</p> <p>RSVD – reserve field</p> <p>USER – call user hook</p> <p>ACP_NONE – no access policy</p>
<vendorExtensions:collect_coverage>	<p>cov_update – coverage enabled on update()</p> <p>cov_compare_and_update – coverage enabled on compare_and_update()</p> <p>cov_all – coverage enabled</p> <p>cov_none – coverage disabled</p>
<vendorExtensions:constraint>	Valid SystemVerilog constraint.

<code><vendorExtensions:external_event></code>	Optional. Default is none.
<code><vendorExtensions:hd़l_path></code>	Path for backdoor register/field access.
<code><vendorExtensions:is_rsv></code>	TRUE = field is reserved.
<code><vendorExtensions:type></code>	Enumerated type.

Register

```
<spirit:register>
```

<code><spirit:access></code>	Register access (e.g., read-write)
<code><spirit:addressOffset></code>	Address of register.
<code><spirit:name></code>	Name of the register.
<code><spirit:size></code>	Number of bits in register.

```
<spirit:vendorExtensions>
```

<code><vendorExtensions:alternate_properties></code>	Same register having alternative name, offset, and access.
<code><vendorExtensions:compare_mask></code>	Compare mask for register.
<code><vendorExtensions:constraint></code>	Valid SystemVerilog constraint. Optional.
<code><vendorExtensions:external_event></code>	Optional. Default is none.
<code><vendorExtensions:hd़l_path></code>	Path for backdoor access of register.
<code><vendorExtensions:indirect_addr_reg></code>	Address register for an indirect register.
<code><vendorExtensions:indirect_areg_fld></code>	Field of indirect register that holds the address.
<code><vendorExtensions:indirect_offset></code>	Offset of an indirectly accessed register.

Register Generators

<vendorExtensions:is_fifo_with_depth>	FIFO register with given depth.
<vendorExtensions:randomize_mask>	Mask specifying fields not to randomize.
<vendorExtensions:reg_tag>	String tag.
<vendorExtensions:size>	Size of the register.
<vendorExtensions:soft_reset_mask>	Soft reset mask. Default is 0.
<vendorExtensions:soft_reset_value>	Soft reset value of register. Default is 0.
<vendorExtensions:type>	Type of the register.
<vendorExtensions:update_mask>	The update mask used when updating.

Address Block (Register File)

```
<spirit:addressBlock>
```

<spirit:baseAddress>	Base address of block.
<spirit:name>	Name of the block.
<spirit:range>	Number of bytes in block.
<spirit:width>	Width of registers.

```
<spirit:vendorExtensions>
```

<vendorExtensions:alternate_properties>	Same register file having alternative name, offset, and access.
<vendorExtensions:constraint>	Valid SystemVerilog constraint.
<vendorExtensions:external_event>	Optional. Default is none.
<vendorExtensions:hdl_path>	HDL path for register file.

<code><vendorExtensions:logical_addresssing></code>	Changes address mode from the default of MEMORY_MAPPED to LOGICAL.
<code><vendorExtensions:type></code>	Type of the register file.

Memory Map

```
<spirit:memoryMap>
```

<code><spirit:name></code>	Name of the memory.
----------------------------------	---------------------

Component

<code><spirit:library></code>	Name of library.
<code><spirit:name></code>	Name of component.
<code><spirit:vendor></code>	Name of vendor.
<code><spirit:version></code>	Version number.

```
<spirit:vendorExtensions>
```

<code><vendorExtensions:constraint></code>	Valid SystemVerilog constraint.
<code><vendorExtensions:external_event></code>	Optional. Default is none.
<code><vendorExtensions:hdl_path></code>	HDL path for register file.
<code><vendorExtensions:size></code>	Size of the address map.
<code><vendorExtensions:type></code>	Type of the register file.

Example

```
<spirit:component ... >
<spirit:memoryMaps>
  <spirit:memoryMap>
    <spirit:name>DUT</spirit:name>
    <spirit:addressBlock>
```

Register Generators

```
<spirit:name>Regs</spirit:name>
<spirit:baseAddress>0x0</spirit:baseAddress>
<spirit:range>0x2c</spirit:range>
<spirit:width>32</spirit:width>
<spirit:register>
    <spirit:name>rxtx0</spirit:name>
    <spirit:description>
        Data receive/transmit register 0
    </spirit:description>
<spirit:addressOffset>0x0
</spirit:addressOffset>
<spirit:size>32</spirit:size>
<spirit:access>read-write</spirit:access>
    <spirit:reset>
<spirit:value>0x0</spirit:value>
<spirit:mask>0xffffffff</spirit:mask>
</spirit:reset>
<spirit:vendorExtensions>
<vendorExtensions:type> rxtx_register
</vendorExtensions:type>
</spirit:vendorExtensions>
    </spirit:register>

    <!-- Other register definitions here ... -->
</spirit:addressBlock>
</spirit:memoryMap>
</spirit:memoryMaps>
</spirit:component>
```

See also

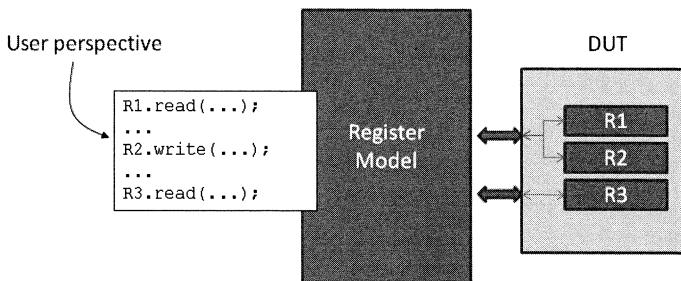
Register Layer

For further information on this and other topics, please see the UVM tutorials at
<http://www.doulos.com/knowhow/sysverilog/uvm/>.

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 DUT.

The UVM Register Layer classes are used to create a high-level, object-oriented model for memory-mapped registers and memories in a design under test (DUT).



This abstraction mechanism allows the migration of verification environments and tests from block to system levels without any modifications. It also can move uniquely named fields between physical registers without requiring modifications in the verification environment or tests.

Register Model

A register model is typically composed of a hierarchy of blocks that map to the design hierarchy. Blocks can contain registers, register files and memories, as well as other blocks.

Register Model Generator

Most of the UVM register layer classes 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 thus are able to provide an up-to-date, correct-by-construction register model. Possible formats are XML (IP-XACT), RALF and CSV. Whilst model generators are outside the scope of the UVM library, two generators are described in **Register Generators**.

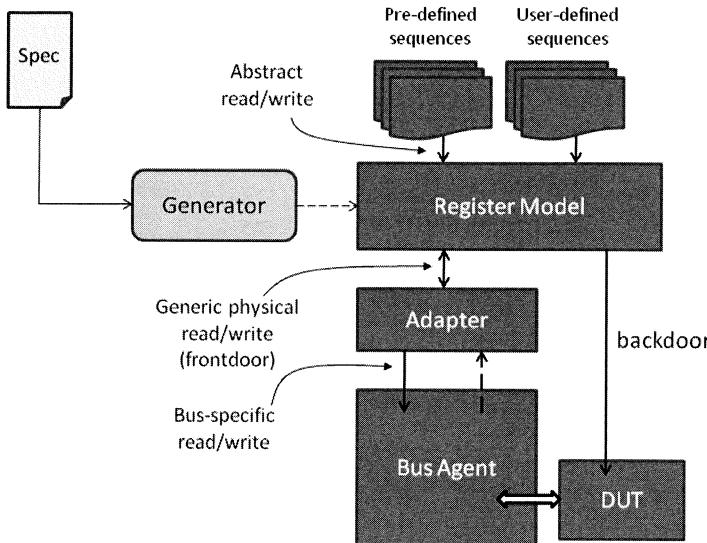
Register Access

The register layer classes support front-door and back-door access to provide redundant paths to the register and memory implementation and verify the correctness of the decoding and access paths as well as increased performance

Register Layer

after the physical access paths have been verified. Designs with multiple physical interfaces, as well as registers, register files, and memories shared across multiple interfaces, are also supported.

Register Test Sequence Library



UVM also provides a register test sequence library containing predefined testcases you can use to verify the correct operation of registers and memories in a DUT.

Register Layer Classes

For each element in a register model—field, register, register file, memory or block—there is a class instance that abstracts the read and write operations on that element.

A *block* generally corresponds to a design component with its own host processor interface(s), address decoding, and memory-mapped registers and memories. If a memory is physically implemented externally to the block, but accessed through the block as part of the block's address space, then the memory is considered as part of the block register model.

All data values are modeled as *fields*. Fields represent a contiguous set of bits. Fields are wholly contained in a *register*. A register may span multiple addresses.

The smallest register model that can be used is a *block*. A block may contain one register and no memories, or thousands of registers and gigabytes of

memory. Repeated structures may be modelled as register arrays, register file arrays, or block arrays

Register Model Classes

<code>uvm_reg_block</code>	Block abstraction base class.
<code>uvm_reg</code>	Register abstraction base class.
<code>uvm_reg_indirect_data</code>	Indirect data access abstraction class. Class includes an address (index) register.
<code>uvm_reg_field</code>	Field abstraction base class.
<code>uvm_vreg</code>	Virtual register abstraction base class. All virtual register accesses eventually turn into memory accesses.
<code>uvm_vreg_field</code>	Virtual field abstraction class.
<code>uvm_reg_file</code>	Register file abstraction base class.
<code>uvm_mem</code>	Memory abstraction base class.
<code>uvm_reg_fifo</code>	Models a write/read FIFO. backdoor access is not supported.
<code>uvm_reg_map</code>	Address map – collection of registers and maps accessible via a specific physical interface.
<code>uvm_mem_mam</code>	Memory allocation manager.

Users (or register model generators) do not use most of these classes directly; instead, they create classes that extend them. For full details, please refer to the *UVM Class Reference Manual*.

Enums

`uvm_status_e` – Return status for register operations

<code>UVM_IS_OK</code>	Operation completed successfully.
<code>UVM_NOT_OK</code>	Operation completed with error.
<code>UVM_HAS_X</code>	Operation completed successfully but had unknown bits.

`uvm_path_e` – Path used for register operation

<code>UVM_FRONTDOOR</code>	Use the front door.
<code>UVM_BACKDOOR</code>	Use the back door.

Register Layer

UVM_PREDICT	Operation derived from observations by a bus monitor via the <code>uvm_reg_predictor</code> class.
UVM_DEFAULT_PATH	Operation specified by the context.

`uvm_check_e` – Read-only or read-and-check

UVM_NO_CHECK	Read only.
UVM_CHECK	Read and check.

`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.

`uvm_elem_kind_e` – Type of element being read or written

UVM_REG	Register.
UVM_FIELD	Field.
UVM_MEM	Memory location.

`uvm_access_e` – Type of operation being performed

UVM_READ	Read operation.
UVM_WRITE	Write operation.
UVM_BURST_READ	Burst read operation.
UVM_BURST_WRITE	Burst write operation.

`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.

`uvm_predict_e` – How the mirror is to be updated

<code>UVM_PREDICT_DIRECT</code>	Predicted value is as-is.
<code>UVM_PREDICT_READ</code>	Predict based on the specified value having been read.
<code>UVM_PREDICT_WRITE</code>	Predict based on the specified value having been written.

`uvm_coverage_model_e` – Coverage models available or desired. Multiple models may be specified by bitwise OR'ing individual model identifiers

<code>UVM_NO_COVERAGE</code>	None.
<code>UVM_CVR_REG_BITS</code>	Individual register bits.
<code>UVM_CVR_ADDR_MAP</code>	Individual register and memory addresses.
<code>UVM_CVR_FIELD_VALS</code>	Field values.
<code>UVM_CVR_ALL</code>	All coverage models.

`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.

<code>UVM_DO_REG_HW_RESET</code>	Run <code>uvm_reg_hw_reset_seq</code> .
<code>UVM_DO_REG_BIT_BASH</code>	Run <code>uvm_reg_bit_bash_seq</code> .
<code>UVM_DO_REG_ACCESS</code>	Run <code>uvm_reg_access_seq</code> .
<code>UVM_DO_MEM_ACCESS</code>	Run <code>uvm_mem_access_seq</code> .
<code>UVM_DO_SHARED_ACCESS</code>	Run <code>uvm_reg_mem_shared_access_seq</code> .
<code>UVM_DO_MEM_WALK</code>	Run <code>uvm_mem_walk_seq</code> .
<code>UVM_DO_ALL_REG_MEM_TESTS</code>	Run all of the above.

Mirror

The register model maintains a mirror of what it thinks the current value of the registers is inside the DUT. The mirrored value is not guaranteed to be correct because the only information the register model has is the read and write accesses to those registers. If the DUT internally modifies the content of any field or register through its normal operations (for example, by setting a status bit or incrementing an accounting counter), the mirrored value becomes outdated.

The register model takes every opportunity to update its mirrored value.

Register Layer

Memories are not mirrored.

Access API

Registers and memories are normally accessed using the `read()` and `write()` methods. When using the front-door (`path=UVM_FRONTDOOR`), one or more physical transactions are executed on the DUT to read or write the register. The mirrored value is then updated to reflect the expected value in the DUT register after the observed transactions.

When using the back-door (`path=UVM_BACKDOOR`), peek or poke operations are executed on the DUT to read or write the register via the back-door mechanism, bypassing the physical interface.

Using the `peek()` and `poke()` methods, reads or writes are made directly to the register respectively, which bypasses the physical interface. The mirrored value is then updated to reflect the actual sampled or deposited value in the register after the observed transactions.

Using the `get()` and `set()` methods, reads or writes are made directly to the desired mirrored value respectively, without accessing the DUT. The desired value can subsequently be uploaded into the DUT using the `update()` method.

Using the `randomize()` method causes copies of the randomized value in the `uvm_reg_field::value` property into the desired value of the mirror by the `post_randomize()` method. The desired value can subsequently be uploaded into the DUT using the `update()` method.

Using the `update()` method invokes the `write()` method if the desired value (previously modified using `set()` or `randomize()`) is different from the mirrored value. The mirrored value is then updated to reflect the expected value in the register after the executed transactions.

Using the `mirror()` method invokes the `read()` method to update the mirrored value based on the readback value. `mirror()` can also compare the readback value with the current mirrored value before updating it.

Field Access

When accessing a field (modelled using `uvm_reg_field`), it may be possible to access only that field, but sometimes the entire register that contains the field must be accessed.

If a frontdoor access (read or write) 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 supports byte-enabling, then only the field is accessed. Otherwise, the entire register containing the field is accessed. For a write, the mirrored values of the other fields in the same register are used in a best-effort not to modify their values. For a read, other fields in the same register are updated.

For a backdoor write, a peek-modify-poke process is used in a best-effort not to modify the value of the other fields in the register. For a backdoor read, the entire register is peeked, and the other fields are updated in the mirror.

Field Access Modes

These are the field access modes. The effect of a read operation is applied after the current value of the field is sampled.

"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.
"WOC"	W: 1/0 no effect on/clears matching bit, R: no effect.
"WOS"	W: 1/0 no effect on/sets matching bit, R: no effect.
"WOT"	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.
"W01"	W: first one after HARD reset is as-is, other W have no effects, R: error.

Register Layer

Indirect Registers

The class `uvm_reg_indirect_data` models a register that is used to access a register array, indexed by a second (address) register. A read or write to an indirect register results in a corresponding read or write to the register that is indexed by the second register.

Virtual Registers and Fields

A virtual register represents a set of fields that are logically implemented in consecutive memory locations. It is modelled using the class `uvm_vreg`. Similarly, a virtual field represents a set of adjacent bits that are logically implemented in consecutive memory locations. It is modelled using `uvm_vreg_field`.

All virtual register and field accesses eventually turn into memory accesses.

Field Aliases

If a field name is unique across all registers' fields within a block, it may also be accessed independently of their register location using an alias declared in the block. This enables models to be independent of the names of the registers in which the fields actually reside. If this behaviour is required, the fields must be configured appropriately using `uvm_reg_field::configure()`.

Callbacks

The Register Layer also provides a number of callback hooks, and some pre-defined callback routines.

These are the various callback classes:

<code>uvm_reg_cbs</code>	Facade class for callback methods (except virtual registers and fields).
<code>uvm_vreg_cbs</code>	Facade class for callback methods for virtual registers.
<code>uvm_vreg_field_cbs</code>	Facade class for callback methods for virtual fields.
<code>uvm_reg_cb,</code> <code>uvm_reg_cb_iter</code>	Convenience callback type and iterator for registers.
<code>uvm_reg_bd_cb,</code> <code>uvm_reg_bd_cb_iter</code>	Convenience callback type and iterator for backdoor access.
<code>uvm_mem_cb,</code> <code>uvm_mem_cb_iter</code>	Convenience callback type and iterator for memories.
<code>uvm_reg_field_cb,</code> <code>uvm_reg_field_cb_iter</code>	Convenience callback type and iterator for fields.
<code>uvm_vreg_cb,</code> <code>uvm_vreg_cb_iter</code>	Convenience callback type and iterator for virtual registers.

<code>uvm_vreg_field_cb,</code> <code>uvm_vreg_field_cb_iter</code>	Convenience callback type and iterator for virtual fields.
<code>uvm_reg_read_only_cbs</code>	Pre-defined callback for readonly registers – issues an error if a write is attempted.
<code>uvm_reg_write_only_cbs</code>	Pre-defined callback for writeonly registers – issues an error if a read is attempted.

These are the callback methods:

<code>pre_write</code>	Called before a write.
<code>post_write</code>	Called after a backdoor write.
<code>pre_read</code>	Called before a read.
<code>post_read</code>	Called after a read.
<code>post_predict[†]</code>	Called by <code>uvm_reg_field::predict</code> after a successful UVM_PREDICT_READ or UVM_PREDICT_WRITE.
<code>encode[†]</code>	Data encoder.
<code>decode[†]</code>	Data decoder.

[†]The post_predict, encode and decode callback methods are not included in `uvm_vreg_cbs` or `uvm_vreg_field_cbs`

Transactions, Sequences and Adapters

The Register Layer provides generic bus transactions and adapters, analogous to sequence items and drivers in UVM agents and test environments. UVM also provides a number of built-in test sequences, which can be used with a register model. See **Register and Memory Sequences** for more information.

<code>uvm_reg_item</code>	Defines an abstract register transaction item.
<code>uvm_reg_backdoor</code>	Facade class for register and memory backdoor access.
<code>uvm_reg_bus_op</code>	Struct that defines a generic bus transaction for register and memory accesses.
<code>uvm_reg_adapter</code>	This class defines an interface for converting between <code>uvm_reg_bus_op</code> and a specific bus transaction.

Register Layer

uvm_reg_tlm_adapter	For converting between uvm_reg_bus_op and uvm_tlm_gp (Generic Payload) items.
uvm_reg_frontdoor	Facade class for register and memory frontdoor access.
uvm_reg_sequence	This class provides base functionality for both user defined RegModel test sequences and "register translation sequences."
uvm_reg_predictor	Updates the register model mirror based on observed bus transactions.

Gotcha

A register model, which is usually derived from `uvm_reg_block`, should have a `build()` method. This is a user-defined method; it is not a base class method and without it there will be no method to call. (There is no inherited `build_phase()` method because `uvm_reg_block` is derived from `uvm_object`, not `uvm_component`.)

Example

The register model will usually be generated by a memory model generator.

```
class my_model extends uvm_reg_block;  
  ...  
endclass : my_model
```

The register-to-bus adapter provides functions to convert between abstract register transactions and physical transactions. An adapter needs to have the `reg2bus` and `bus2reg` functions overloaded by the user and then installed with the register map.

```
class reg_to_bus_adapter extends uvm_reg_adapter;  
  `uvm_object_utils(reg_to_bus_adapter)  
  function uvm_sequence_item reg2bus(uvm_reg_bus_op rw);  
    bus_trans tx = bus_trans::type_id::create("bus_item");  
    tx.addr = rw.addr[7:0];  
    ...  
    return tx;  
  endfunction : reg2bus  
  function void bus2reg(uvm_sequence_item bus_item,  
    uvm_reg_bus_op rw);  
    ...  
  endfunction : bus2reg
```

```
endclass : reg_to_bus_adapter

Add the memory model to a test environment:

class my_env extends uvm_env;
    my_model regmodel;
    bus_agent agt;

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        agt = bus_agent::type_id::create(...);
        if (regmodel == null) begin
            regmodel = my_model::type_id::create(...);
            regmodel.build();
            // Lock the register model to prevent further structural changes
            // and build the address map(s)
            regmodel.lock_model();
        end
    endfunction : build_phase

    function void connect_phase(uvm_phase phase);
        if (regmodel.get_parent() == null) begin
            // Create an instance of the register-to-bus adapter
            reg_to_bus_adapter adapter
                = reg_to_bus_adapter::type_id::create(...);
            // Set the sequencer for the register model's default map to be
            // the sequencer in the bus agent and connect it to the adapter
            regmodel.default_map.set_sequencer(
                agt.m_sequencer, adapter);
            // Turn on implicit prediction
            regmodel.set_auto_predict(1);
        end
        ...
    endfunction : connect_phase

endclass : my_env

Create the test:

class my_test extends uvm_test;
    my_env env;
    virtual function void run_phase(uvm_phase phase);
        // Create and start a user-defined or built-in sequence
        my_reg_sequence seq =
            my_reg_sequence::type_id::create("seq",this);
        seq.start(env.agt.m_sequencer);
    endfunction : run_phase
```

Register Layer

```
endclass : my_test
```

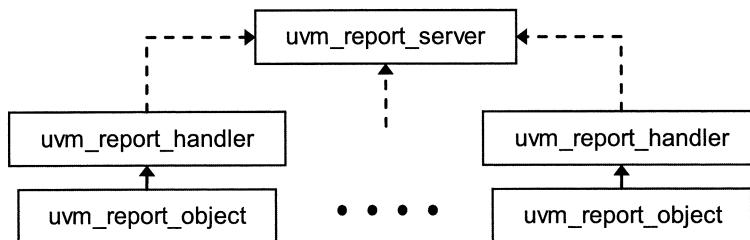
See also

Register and Memory Sequences; Register Generators

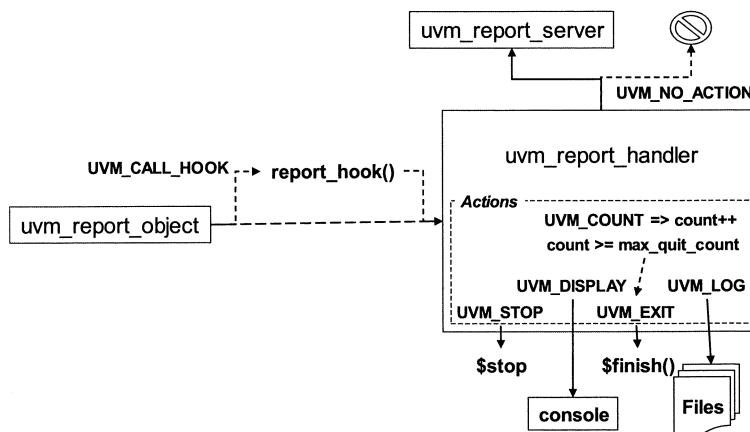
For further information on this and other topics, please see the UVM tutorials at
<http://www.doulos.com/knowhow/sysverilog/uvm/>.

UVM offers a powerful reporting facility, providing the mechanisms to display messages in a uniform format to different destinations, filtering of messages, and assigning actions to trigger when specific messages are issued. All reporting messages are issued through a global `uvm_report_server`, but the report server is not intended to be directly accessed. Rather, a `uvm_report_object` is provided as the user interface into the reporting machinery.

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



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



Report Messages

There are four basic functions used for issuing messages: `uvm_report_info`, `uvm_report_warning`, `uvm_report_error` and `uvm_report_fatal`.

UVM provides a set of four corresponding global macro wrappers, which should be used in preference to calling the functions.

```
`uvm_info(string id, string message,  
         uvm_verbosity verbosity_level)  
'uvm_warning(string id, string message)  
'uvm_error(string id, string message)  
'uvm_fatal(string id, string message)
```

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

UVM_INFO	Informational message.
UVM_WARNING	Warning message.
UVM_ERROR	Error message.
UVM_FATAL	Fatal message.

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

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

The verbosity level represents an arbitrary value used for filtering messages. If the reporting macros are used, a verbosity level is required for `'uvm_info` and is not permitted for the other macros. It is commonly set to UVM_NONE, meaning that the message will be written, irrespective of the global verbosity level. For the other macros, the severity level is fixed as UVM_NONE so that these messages cannot mistakenly be turned off using the global verbosity. (It is still possible turn messages off using `actions` – see below).

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

UVM_NONE	0
----------	---

UVM_LOW	100
UVM_MEDIUM	200
UVM_HIGH	300
UVM_FULL	400
UVM_DEBUG	500

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

Report Actions

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

UVM_NO_ACTION	no action.
UVM_DISPLAY	send report to STDOUT.
UVM_LOG	send report to one or more files.
UVM_COUNT	increment report counter.
UVM_EXIT	calls the <code>pre_abort()</code> callback and <code>\$finish</code> to terminate simulation immediately.
UVM_CALL_HOOK	call the <code>report_hook()</code> methods.
UVM_STOP	calls <code>\$stop</code> .

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

UVM_INFO	UVM_DISPLAY
UVM_WARNING	UVM_DISPLAY
UVM_ERROR	UVM_DISPLAY UVM_COUNT
UVM_FATAL	UVM_DISPLAY UVM_EXIT

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

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

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

Report

```
set_report_severity_action( UVM_WARNING, UVM_NO_ACTION );
```

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

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

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

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

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

Report Hooks

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

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

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

Report State

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

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

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

```
#  
# --- UVM Report Summary ---  
#  
# ** Report counts by severity  
# UVM_INFO : 76  
# UVM_WARNING : 0  
# UVM_ERROR : 16  
# UVM_FATAL : 0  
# ** Report counts by id  
# [ENV ] 1  
# [RNTST ] 1  
...  
...
```

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

Globals

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

Because `uvm_top` is globally accessible, it acts as a global reporter. (OVM's `_global_reporter` const is provided for backward compatibility: in UVM, it is the same as `uvm_top`.)

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

+UVM_SEVERITY=value	sets the global severity level threshold, where <i>value</i> equals one of the following: INFO WARNING ERROR FATAL
+UVM_VERBOSITY=value	sets the global verbosity level threshold, where <i>value</i> equals one of the following: <i>any integer value</i> NONE UVM_NONE LOW UVM_LOW MEDIUM UVM_MEDIUM HIGH UVM_HIGH FULL UVM_FULL DEBUG UVM_DEBUG

Example

See `uvm_report_object`.

Tips

- Use the ``uvm_(info|warning|error|fatal)` macros for full control of messages rather than `$display` and `$fdisplay`. The system tasks do not provide configure and filtering capabilities in your environment.
- To pass in multiple variables and format them into a single string that can be passed to ``uvm_(info|warning|error|fatal)` macros, use a system function like `$sformatf`. (The UVM reporting functions do not accept variable length arguments nor a format string specifier as `$display`)

does)::

```
`uvm_info( get_name(),
    $sformatf("Received transaction. Data = %d",
    data), UVM_NONE)
```

Gotchas

- Don't forget the verbosity argument for `uvm_info. You can't change the verbosity when using `uvm_warning, `uvm_error or `uvm_fatal.
- Macros represent text substitutions, not statements, so they should not be terminated with semi-colons.
- The command line options +UVM_SEVERITY and +UVM_VERBOSITY cause the format of the leading information printed before the message string to be changed.
- Not all information printed out by UVM is controlled through the reporting mechanism. UVM also provides a printing facility for traversing object hierarchies and printing their internal contents. To control printing of data objects (such as sequence items), see Printing.

See also

[uvm_report_object](#)

uvm_report_catcher

The `uvm_report_catcher` is used to intercept messages issued by the `uvm_report_server`. Caught messages may be modified before being passed to the report server; they can be issued immediately, or they can be quietly dropped.

Multiple report catchers can be registered with a report object, i.e., a component. Catchers can be registered as default catchers which catch all reports on all `uvm_report_object` reporters, or catchers can be attached to specific 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) message is passed on to other registered catchers. Lastly, the report can be immediately issued from the catcher by calling `issue`. The catchers are processed in the order in which they are registered.

The catcher maintains a count of all reports with FATAL,ERROR or WARNING severity and a count of which of these had their severity lowered. These statistics are reported in the summary of the `uvm_report_server`.

Declaration

```
virtual class uvm_report_catcher extends uvm_callback;
```

Methods

function new(string name = "uvm_report_catcher");	Constructor.
function uvm_report_object get_client() ;	Returns the <code>uvm_report_object</code> that generated the message
function string get_context() ;	Returns the hierarchical path to the component issuing the message.
function uvm_severity get_severity() ;	Returns the message's severity.
function int get_verbosity() ;	Returns the message's verbosity.
function string get_id() ;	Returns the message's id.
function string get_message() ;	Returns the message's text.
function uvm_action get_action() ;	Returns the message's action.

function string get_fname() ;	Returns the name of the source file that generated the message.
function int get_line() ;	Returns the line number in the source file that generated the message.
protected function void set_severity(uvm_severity severity);	Change the message's severity.
protected function void set_verbosity(int verbosity) ;	Change the message's verbosity.
protected function void set_id(string id);	Change the message's id.
protected function void set_message(string message) ;	Change the message's text.
protected function void set_action(uvm_action action) ;	Change the message's action.
static function uvm_report_catcher get_report_catcher(string name) ;	Returns the first report catcher called name.
static function void print_catcher(UVM_FILE file = 0) ;	Prints information about all the registered report catchers.
pure virtual function action_e catch() ;	callback – called for each registered report catcher.
protected function void uvm_report_fatal(string id, string message, int verbosity, string fname = "", int line = 0);	Issue the message, bypassing any report catchers.
protected function void uvm_report_error(string id, string message, int verbosity, string fname = "", int line = 0);	Issue the message, bypassing any report catchers.

uvm_report_catcher

<pre>protected function void uvm_report_warning(string id, string message, int verbosity, string fname = "", int line = 0);</pre>	Issue the message, bypassing any report catchers.
<pre>protected function void uvm_report_info(string id, string message, int verbosity, string fname = "", int line = 0);</pre>	Issue the message, bypassing any report catchers.
<pre>protected function void issue();</pre>	Immediately issues the message which is currently being processed.

Rules

The `catch` method is a pure virtual function so it must be implemented.

Tips

- The `uvm_callbacks #(uvm_report_object, uvm_report_catcher)` class is aliased to `uvm_report_cb` to make it easier to use.
- Catchers are `uvm_report_cb` objects so all facilities in the `uvm_callback` and `uvm_callbacks#(T,CB)` classes are available for registering catchers and controlling catcher state.
- 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.

Example

```
// Create a new report catcher class
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

// Instantiate the new report catcher
my_error_demoter demoter = new;

initial begin
    // Catch messages for all reporters
    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 using the component name
    uvm_report_cb::add_by_name(".*.driver", demoter);

    `uvm_error("MY_ID",
              "This message will be demoted to UVM_INFO")
    // Disable demotion
    demoter.callback_mode(0);
    `uvm_error("MY_ID",
              "This message will NOT be demoted")
end
```

See also

Report; uvm_callbacks; uvm_report_object

uvm_report_object

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

- tagging each message with a specific id
- assigning 4 levels of severity (INFO, WARNING, ERROR, and FATAL)
- controlling the verbosity of a reported message.

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

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

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

Since every `uvm_component` is derived from `uvm_report_object`, the report member functions are also members of every component. Typically, the `uvm_report_info`, `uvm_report_warning`, `uvm_report_error`, and `uvm_report_fatal` methods are called to issue messages of a specified severity level. The `uvm_component` class extends several of the `uvm_report_object` functions to operate recursively on a component and all its subcomponents. These functions have the additional `_hier` suffix added to their name (see `uvm_component`).

Objects not derived from `uvm_report_object` can also use `uvm_top`, which acts as a global reporter, or by calling the macros ``uvm_info`, ``uvm_warning`, ``uvm_error` and ``uvm_fatal`.

Report objects provide a mechanism to increment a message count in the report server (by setting a message's action to `UVM_COUNT`) and to specify a maximum permitted count. If this maximum count is exceeded, the report server will call the `die` function. A `UVM_EXIT` action will also invoke `die`. `die` calls the `report_summarize` function and simulation is terminated immediately with a call to `$finish`.

Declaration

```
virtual class uvm_report_object extends uvm_object;
```

Methods

<code>function new(string name = "");</code>	Constructor
<code>function void uvm_report_fatal(string id, string message, int verbosity_level = UVM_NONE, string filename = "", int line = 0);</code>	Produces reports of severity UVM_FATAL (see Report).
<code>function void uvm_report_error(string id, string message, int verbosity_level = UVM_LOW, string filename = "", int line = 0);</code>	Produces reports of severity UVM_ERROR (see Report).
<code>function void uvm_report_warning(string id, string message, int verbosity_level = UVM_MEDIUM, string filename = "", int line = 0);</code>	Produces reports of severity UVM_WARNING (see Report).
<code>function void uvm_report_info(string id, string message, int verbosity_level = UVM_MEDIUM, string filename = "", int line = 0);</code>	Produces reports of severity UVM_INFO (see Report).
<code>virtual function void die();</code>	Called by report server max quit count reached or UVM_EXIT action (fatal error).
<code>function void dump_report_state();</code>	Dumps the report handler's internal state.
<code>function int get_report_action(uvm_severity severity, string id);</code>	Returns the action associated with the specified severity and id.
<code>function int get_report_file_handle(uvm_severity severity, string id);</code>	Returns the file descriptor associated with the specified severity and id.
<code>function uvm_report_handler get_report_handler();</code>	Returns a reference to the report handler.

uvm_report_object

<pre>function uvm_report_server get_report_server();</pre>	Returns the report server associated with this report.
<pre>function int get_report_verbosity_level(uvm_severity severity = UVM_INFO, string id = "");</pre>	Returns the verbosity level for current object. The severity and id arguments are used to check if the verbosity has changed for the severity/id combination.
<pre>virtual function void report_header(FILE f = 0);</pre>	Prints copyright and version numbers to command line if <code>f=0</code> or to a file if <code>f</code> is a file descriptor.
<pre>virtual function bit report_hook(string id, string message, int verbosity, string filename, int line);</pre> <pre>virtual function bit report_info_hook(string id, string message, int verbosity, string filename, int line);</pre> <pre>virtual function bit report_warning_hook(string id, string message, int verbosity, string filename, int line);</pre> <pre>virtual function bit report_error_hook(string id, string message, int verbosity, string filename, int line);</pre> <pre>virtual function bit report_fatal_hook(string id, string message, int verbosity, string filename, int line);</pre>	User-definable functions allowing additional actions to be performed when reports are issued if the report action is <code>UVM_CALL_HOOK</code> . Return value of 1 (default) allows reporting to proceed; otherwise, reporting is not processed. The <code>report_*_hook</code> functions are only called for messages with a corresponding severity.

<pre>virtual function void report_summarize(FILE f = 0);</pre>	Prints report server statistical information to command line if <code>f=0</code> or to a file if <code>f</code> is a file descriptor.
<pre>function void reset_report_handler();</pre>	Resets this object's report handler to default values.
<pre>function void set_report_handler(uvm_report_handler handler);</pre>	Sets the report handler.
<pre>function void set_report_max_quit_count(int max_count);</pre>	Sets maximum number of UVM_COUNT actions before die method is called; default value of 0 sets no maximum upper limit.
<pre>function void set_report_default_file(UVM_FILE file); function void set_report_severity_file(uvm_severity severity, UVM_FILE file); function void set_report_id_file(string id, UVM_FILE file); function void set_report_severity_id_file(uvm_severity severity, string id, UVM_FILE file);</pre>	Sets the output file for the UVM_LOG action. Specifying both severity and id takes precedence over id only which takes precedence over severity only which takes precedence over the default.
<pre>function void set_report_id_verbosity(string id, int verbosity); function void set_report_severity_id_verbosity(uvm_severity severity, string id, int verbosity);</pre>	Sets the report handler to associate the respective id/severity to a specific verbosity level.
<pre>function void set_report_severity_action(uvm_severity severity, uvm_action action); function void set_report_id_action(string id,</pre>	Sets the report handler to perform a specific action for all reports matching the specified severity, id, or both, respectively, where action equals UVM_NO_ACTION

uvm_report_object

<pre>uvm_action action); function void set_report_severity_id_action(uvm_severity severity, string id, uvm_action action);</pre>	or UVM_DISPLAY UVM_LOG UVM_COUNT UVM_EXIT UVM_CALL_HOOK
<pre>function void set_report_severity_override(uvm_severity cur_severity, uvm_severity new_severity); function void set_report_severity_id_override(uvm_severity cur_severity, string id, uvm_severity new_severity);</pre>	Upgrades or downgrades the severity level to the newly specified level for the specific id.
<pre>function void set_report_verbosity_level(int verbosity_level);</pre>	Sets the maximum verbosity threshold (reports with a lower level are not processed).
<pre>function int uvm_report_enabled(int verbosity, uvm_severity severity = UVM_INFO, string id = "");</pre>	Returns 1 if component's verbosity is greater than the specified verbosity and the component's action for the specified severity and id are not UVM_NO_ACTION.

Members

<code>protected uvm_report_handler m_rh;</code>	Handle to a report handler.
--	--------------------------------

Example

```
class my_test extends uvm_test;  
...  
// Turn off messages tagged with the id = "debug" for the my_env  
// component only  
my_env.set_report_id_action ("debug", UVM_NO_ACTION);  
  
// Turn all UVM_INFO messages off -  
// (Use set_report_*_hier version [from uvm_component]  
// to recursively traverse the hierarchy and set the action)  
set_report_severity_action_hier(UVM_INFO, UVM_NO_ACTION);
```

```

// Turn all messages back on
set_report_severity_action_hier( UVM_INFO,
                                 UVM_DISPLAY | UVM_LOG );
set_report_severity_action_hier( UVM_WARNING,
                                 UVM_DISPLAY | UVM_LOG );
set_report_severity_action_hier( UVM_ERROR,
                                 UVM_DISPLAY | UVM_COUNT | UVM_LOG );
set_report_severity_action_hier( UVM_FATAL,
                                 UVM_DISPLAY | UVM_EXIT | UVM_LOG );

// Setup the global reporting for messages that use uvm_top as a
// global report handler (like the sequences machinery)
uvm_top.set_report_verbosity_level( UVM_ERROR );
uvm_top.dump_report_state(); // Print out state

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

// Example with user-definable report hooks
class my_env extends uvm_env;
    bit under_reset = 0; // Indicates device under reset
    UVM_FILE f;
    ...
    // Override the report_hook function
    function bit report_hook(input id, string message,
                             int verbosity, string
                             filename, int line);
        // Turn off all reporting during the boot-up,
        // initialization, and reset period
        if (!under_reset && ( $time > 100ns ))
            return 1;
        else
            return 0; // Either under_reset or time less than
                      // 100ns so do not issue report messages
    endfunction : report_hook

    function void start_of_simulation_phase(uvm_phase phase);
        // Duplicate report messages to a file
        f = $fopen( "sim.log", "w" );
        set_report_default_file_hier( f );
    endfunction
endclass

```

```
// Setup the environment to not print INFO, WARNING,  
// and ERROR message during reset or initialization by  
// adding the UVM_CALL_HOOK reporting action  
set_report_severity_action_hier( UVM_INFO,  
                                 UVM_DISPLAY | UVM_CALL_HOOK );  
set_report_severity_action_hier( UVM_WARNING,  
                                 UVM_DISPLAY | UVM_CALL_HOOK );  
set_report_severity_action_hier( UVM_ERROR,  
                                 UVM_DISPLAY | UVM_COUNT | UVM_CALL_HOOK );  
endfunction : start_of_simulation_phase  
endclass : my_env
```

Tips

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

```
set_report_id_verbosity_hier  
set_report_severity_id_verbosity_hier  
set_report_severity_action_hier  
set_report_id_action_hier  
set_report_severity_id_action_hier  
set_report_severity_file_hier  
set_report_default_file_hier  
set_report_id_file_hier  
set_report_severity_id_file_hier  
set_report_verbosity_level_hier  
pre_abort
```

See `uvm_component` for function details.

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

Gotchas

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

See also

[Report](#)

uvm_resource_db

The parameterized class, `uvm_resource_db` provides a convenience interface for UVM's resources facility. A `resource` is a parameterized container (`uvm_resource`) that contains arbitrary data. The type of the data can be any built-in SystemVerilog or user-defined type, including scalar objects, class handles, queues, lists and virtual interfaces.

Resources are stored in a resource database (`uvm_resource_pool`) so that each resource can be retrieved by name or by type. The database has both a name table and a type table and each resource is entered into both. The database is implementing as a singleton and is globally accessible.

Resources can be used to configure components, supply data to sequences, or enable sharing of information across disparate parts of a testbench.

Each resource has a set of scopes over which it is visible. This could be an arbitrary string to provide a namespace utility, and it is also used to enable the resource to be visible in only certain parts of the testbench. The set of scopes is represented as a regular expression. When a resource is looked up, the current scope of the entity doing the looking up is supplied to the lookup function. If the current scope is in the set of scopes over which a resource is visible then the resource can be returned in the lookup.

The class `uvm_config_db` provides a convenience layer on top of `uvm_resource_db` for when the hierarchical context is important. Both these classes share the same underlying database, so it is possible to access the same resources through both classes.

Multiple resources that have the same name are stored in a queue. Each resource is pushed into a queue with the first one at the front of the queue and each subsequent one behind it. The same happens for multiple resources that have the same type. The resource queues are searched front to back, so those placed earlier in the queue have precedence over those placed later.

The precedence of resources with the same name or same type can be altered. One way is to set the precedence member of the resource container to any arbitrary value, or to move a resource to either the front or back of the queue using the `set_priority` function. The search algorithm will return the resource with the highest precedence. In the case where there are multiple resources that match the search criteria and have the same (highest) precedence, the earliest one located in the queue will be one returned.

Declaration

```
class uvm_resource_db #(type T = uvm_object);
```

Methods

In the following methods, the `type` is provided by the class parameter, `T`, and the `scope` is the scope in which the resource is visible.

<pre>static function rsrc_t get_by_type(string scope);</pre>	Get a resource by type.
<pre>static function rsrc_t get_by_name(string scope, string name, bit rpterr = 1);</pre>	Get a resource by name. If rpterr is 1 a warning is issued if there is no matching resource.
<pre>static function rsrc_t set_default(string scope, string name);</pre>	Create a new item in the resources database with the default value for its type.
<pre>static function void set(input string scope, input string name, T val, input uvm_object accessor = null);</pre>	Create a new item in the resources database with the value <i>val</i> . <i>accessor</i> is used for auditing.
<pre>static function void set_anonymous(input string scope, T val, input uvm_object accessor = null);</pre>	Create a new anonymous item in the resources database with the value <i>val</i> . <i>accessor</i> is used for auditing.
<pre>static function bit read_by_name(input string scope, input string name, inout T val, input uvm_object accessor = null);</pre>	Retrieve a resource by <i>name</i> and <i>scope</i> . Returns 1 if successful.
<pre>static function bit read_by_type(input string scope, inout T val, input uvm_object accessor = null);</pre>	Retrieve a resource by <i>type</i> and <i>scope</i> . Returns 1 if successful.
<pre>static function bit write_by_name(input string scope, input string name, T val, input uvm_object accessor = null);</pre>	Write a value into the resources database, creating a new resource if one isn't located using <i>name</i> and <i>scope</i> .
<pre>static function bit write_by_type(input string scope, input T val, input uvm_object accessor = null);</pre>	Write a value into the resources database, creating a new resource if one isn't located using the <i>type</i> and <i>scope</i> .

uvm_resource_db

<pre>static function void dump();</pre>	(For debugging purposes) Dumps all the resources in the resource pool.
---	--

Rules

- Regular expressions are enclosed by a pair of slashes: `/regex/`. (Without the slashes the scope is a *glob*: * , + and ? are the only wildcards. Globs are converted to regular expressions before being stored in the database.)
- All of the functions in `uvm_resource_db#(T)` are static, so they must be called using the `::` operator (See examples)

Examples

Create a resource called "max_count" of type `int` in a scope called "shared_resources" and give it the value 100:

```
uvm_resource_db #(int)::set("shared_resources",
                           "max_count", 100, null);
```

Retrieve the value of the "max_count" resource by name:

```
max_count = uvm_resource_db #(int)::get_by_name(
  "shared_resources", "max_count");
```

and by type (assuming no other ints in "shared_resources"):

```
max_count = uvm_resource_db #(int)::get_by_type(
  "shared_resources");
```

This shows how a virtual interface can be configured to point to an actual interface using the resource database.

```
module top;

  import uvm_pkg::*;
  import my_pkg::*;

  // Instance of the interface that connects the DUT and TB
  dut_if dut_if1();

  // Instance the DUT
  dut    dut1( .i_f(dut_if1) );

  initial
  begin
    // Set virtual interface of driver to point to actual interface
    uvm_resource_db #(virtual dut_if)::set("DUT",
```

```
        "m_dut_if",
        dut_if1);

    run_test();
end

endmodule: top

class my_driver extends uvm_driver #(my_transaction);
...
virtual interface dut_if m_dut_if;

function void connect_phase(uvm_phase phase);
    if ( ! uvm_resource_db #(virtual dut_if)::read_by_name(
            "DUT",
            "m_dut_if",
            m_dut_if) )
        `uvm_fatal("NOVIF", {"No virtual interface set for",
                            get_full_name(),".m_dut_if"})
endfunction : connect_phase
...
endclass : my_driver
```

Tips

uvm_config_db and uvm_resource_db share the same underlying database. For configuration properties that are related to hierarchical position, e.g., “set all of the coverage_enable bits for all components in a specific agent”, uvm_config_db is the correct choice. Likewise, for cases where a configuration property is being shared without regard to hierarchical context, uvm_resource_db should be used.

Gotchas

- Care must be taken with write_by_name and write_by_type, because the scope is matched to a resource which may be a regular expression, and consequently, may target other scopes beyond the scope argument.
- The scope provided to the get_* and read_* members is *not* a glob or regular expression.

See also

Configuration; uvm_config_db

uvm_root

The top level object in a uvm testbench. Every UVM testbench contains a single instance of `uvm_root` named `uvm_top`. Users should not attempt to create any other instances of `uvm_root`. Any component that does not have a parent specified when it is created has its parent set automatically to `uvm_top`. This allows components created in multiple modules to share a common parent.

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

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

Declaration

```
virtual class uvm_root extends uvm_component;
```

Methods

<code>virtual task run_test(string test_name = "");</code>	Runs all simulation phases for all components in the environment.
<code>virtual function string get_type_name();</code>	Returns "uvm_root".
<code>function void stop_request();</code>	Deprecated. Stops execution of the current phase.
<code>function uvm_component find(string comp_match);</code>	Returns handle to component with name matching pattern.

<pre>function void find_all(string comp_match, ref uvm_component comps[\$], input uvm_component comp = null);</pre>	Returns queue of handles to components with matching names. <code>comp</code> specifies component to start search.
<pre>function void print_topology(uvm_printer printer = null);</pre>	Print the verification environment's component topology.
<pre>function void set_timeout(time timeout, bit overridable = 1);</pre>	Sets the timeout for the entire simulation.

Members

<pre>bit finish_on_completion = 1;</pre>	When <code>set</code> , <code>run_test</code> calls <code>\$finish</code> on completion of the report phase.
<pre>bit enable_print_topology = 0;</pre>	When set the testbench hierarchy is printed when <code>end_of_elaboration</code> completes.
<pre>time phase_timeout = `UVM_DEFAULT_TIMEOUT;</pre>	Sets the maximum simulated-time for task-based phases (e.g. <code>run_phase</code>). Exits with error if reached.
<pre>uvm_component top_levels[\$];</pre>	List of all of the top level components, including the <code>uvm_test_top</code> component that is created by <code>run_test</code> .

Global defines and methods

<pre>`define UVM_DEFAULT_TIMEOUT 9200s</pre>	Default timeout for simulation.
<pre>task run_test(string test_name = "");</pre>	Calls <code>uvm_top.run_test</code> .
<pre>function void global_stop_request();</pre>	Deprecated. Convenience function for <code>uvm_test_done.stop_request</code> .
<pre>function void set_global_timeout(time timeout, bit overridable = 1);</pre>	Deprecated. Calls <code>uvm_top.set_timeout</code> .
<pre>function void set_global_stop_timeout(time timeout);</pre>	Deprecated. Sets <code>uvm_test_done.stop_timeout</code> .

Gotchas

`global_stop_request()` is deprecated. All environments using the `global_stop_request()` mechanism must add the switch
`+UVM_USE_OVM_RUN_SEMANTIC`.

Example

Configuring and running a test:

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

Searching for components:

```
class test1 extends uvm_test;
    `uvm_component_utils(test1)
    verif_env env1;
    uvm_component c;
    uvm_component cq[$];
    function new (string name, uvm_component parent);
        super.new(name,parent);
    endfunction : new
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        env1 = verif_env::type_id::create("env1",this);
    endfunction : build_phase

    function void end_of_elaboration_phase(uvm_phase phase);
        c = uvm_top.find("env1.m_driver");
        uvm_top.find_all("*",cq,c);
        foreach(cq[i])
            `uvm_info("FIND_ALL",
                $sformatf("Found %s of type %s",
                    cq[i].get_full_name(),cq[i].get_type_name()),
                UVM_NONE)
    endfunction: end_of_elaboration_phase
endclass: test1
```

Tips

- Use objections to control end of simulation. (`uvm_top.stop_request` and `uvm_top.phase_timeout` are deprecated.)
- For flexibility, use `+UVM_TESTNAME` command-line “plusarg” to set the test name, rather than passing it as an argument to `run_test`. If it is passed as an argument to `run_test`, the command line option takes precedence. The `run_test` argument can serve as a default test case if nothing is specified on the command line.

Gotchas

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

See also

End of Test; `uvm_test`

uvm_scoreboard

Class `uvm_scoreboard` is derived from `uvm_component`. User-defined scoreboards should be built using classes derived from `uvm_scoreboard`.

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

Declaration

```
class uvm_scoreboard extends uvm_component;
```

Methods

<code>function new(string name, uvm_component parent);</code>	Constructor, mirrors the superclass constructor in <code>uvm_component</code>
---	---

Members

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

Tips

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

Gotchas

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

See also

`uvm_algorithmic_comparator`; `uvm_in_order_*_comparator`

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

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

UVM sequences are derived from the `uvm_sequence` class. Each sequence is associated with a sequencer (derived from the `uvm_sequencer` class). The sequencer is used to execute the sequence and place the generated sequence items on the sequencer's built-in sequence item export (`seq_item_export`). Generated sequence items are *pulled* from the sequencer by a driver (see `uvm_driver`). (Alternatively, `uvm_push_driver` and `uvm_push_sequencer` could be used to implement a *push* semantic). Each driver has a built-in sequencer port (called `seq_item_port`) that can be connected to a sequencer's sequence item export. The driver pulls sequence items from the sequencer by calling `get()` or `get_next_item()` and sends an acknowledgement back to the sequencer by calling `put()` or `item_done()`, respectively.

Sequences and sequencers are registered with the UVM factory in the usual way by calling the UVM macros. If the sequence requires access to the derived type-specific functionality of its associated sequencer, use the ``uvm_declare_p_sequencer` macro to declare the desired sequencer pointer.

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

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

Steps 2 and 3 are performed by invoking a sequence's `start_item` method. Steps 5-8 are performed by invoking `finish_item`. Therefore, these 9 steps can be reduced to the following:

1. Call the factory to create the sequence item.
2. Call `start_item`.

Sequence

3. Randomize the sequence item.
4. Call `finish_item`.

Alternatively, the following UVM macros perform most or all of these steps automatically:

```
`uvm_do
`uvm_do_pri
`uvm_do_with
`uvm_do_pri_with
`uvm_create
`uvm_send
`uvm_send_pri
`uvm_rand_send
`uvm_rand_send_pri
`uvm_rand_send_with
`uvm_rand_send_pri_with
`uvm_create_on
`uvm_do_on
`uvm_do_on_with
`uvm_do_on_pri
`uvm_do_on_pri_with
`uvm_create_seq
`uvm_do_seq
`uvm_do_seq_with
```

Variables must be declared for any `uvm_sequence` or `uvm_sequence_item` used by the UVM macros with the exception of `req` and `rsp`, which are pre-defined members of a sequence. Class members of the `uvm_sequence` or `uvm_sequence_item` can be declared `rand` so when the sequence is generated, the field values may be constrained using the ``uvm_do_with`, ``uvm_do_pri_with`, ``uvm_rand_send_with`, ``uvm_rand_send_pri_with`, ``uvm_do_on_with` or ``uvm_do_on_pri_with` macros. The ``uvm_*_pri` macros allow sequence items and sequences to be assigned a priority that is used when multiple sequence items are waiting to be pulled by a driver.

A uvm sequence has the following basic structure:

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

    virtual task pre_do(); endtask      // Optional

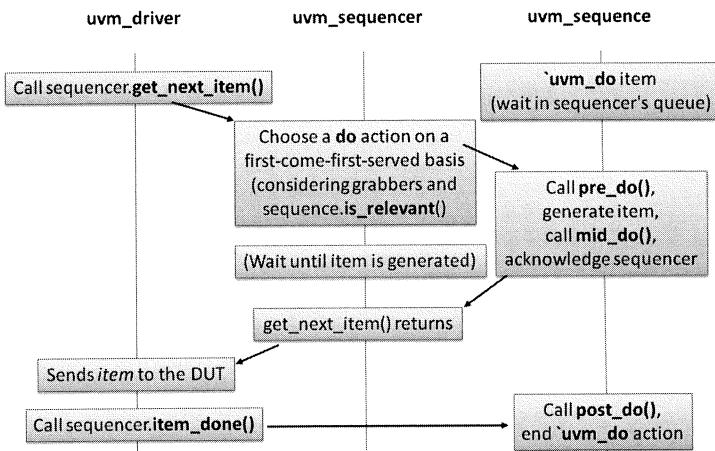
    virtual task body();
        // UVM sequence macro such as `uvm_do or `uvm_do_with
        `uvm_do ( trans )
    endtask : body
```

```

virtual task post_do; endtask      // Optional
endclass : my_sequence

```

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



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

Example

Declaring a sequence item:

```

class my_seq_item extends uvm_sequence_item;
    rand int data;
    rand bit [4:0] addr;
    ...
    `uvm_object_utils_begin ( my_seq_item )
        `uvm_field_int ( data, UVM_ALL_ON + UVM_HEX )
        `uvm_field_int ( addr, UVM_ALL_ON + UVM_HEX )
    ...
    `uvm_object_utils_end
endclass : my_seq_item

```

Sequence

Specialize a sequencer:

```
typedef uvm_sequencer #(my_seq_item) my_sequencer;
```

Declaring a sequence:

```
class my_sequence extends uvm_sequence #(my_seq_item);
  `uvm_object_utils ( my_sequence )
  my_seq_item      m_seq_item;

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

  virtual task body();
    // Create a sequence item using a macro
    `uvm_do ( m_seq_item )

    // Manually create another sequence item
    m_seq_item = my_seq_item:::type_id::create();
    m_seq_item.start_item();
    m_seq_item.randomize with { ... };
    m_seq_item.finish_item();
  endtask : body
endclass : my_sequence
```

Tips

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

Gotchas

- The `uvm_sequencer` implements only PULL mode, meaning that the `uvm_driver` controls or pulls the `uvm_sequence_items` from the `uvm_sequencer`. For the sequencer to control the interaction (i.e., PUSH mode), use `uvm_push_sequencer` and a corresponding `uvm_push_driver`.

- No `uvm_virtual_sequence` exists. You must use `uvm_sequence` for both virtual and non-virtual sequences.
- Take care not to confuse `uvm_sequence` with `uvm_sequencer`. They differ by one letter only, but have quite different functionality.

See also

`uvm_sequence_item`; `uvm_sequence`; `uvm_sequencer`; Virtual Sequences; Sequence Action Macros

uvm_sequence

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

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

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

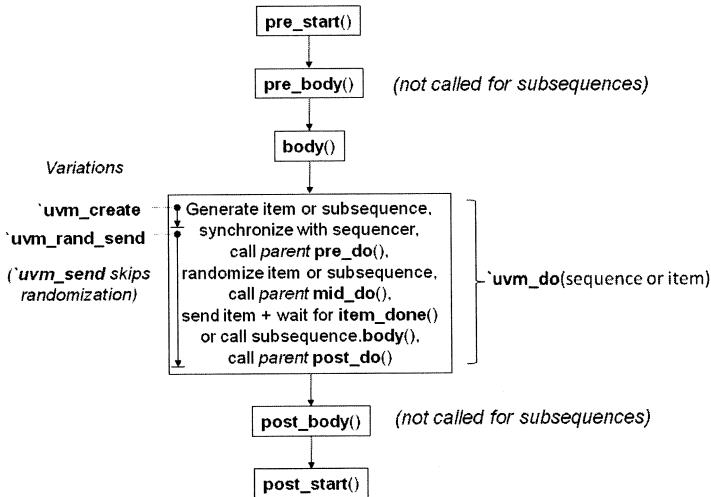
It is also possible to create sequence items, send them to a sequencer, and wait for a response by explicitly calling the member tasks and functions of `uvm_sequence`. A sequence has a response queue that buffers the responses from a driver and permits multiple sequence items to be sent before the responses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

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

Additional `pre_do`, `mid_do`, and `post_do` virtual tasks can be defined for the sequence. These tasks provide additional control over the sequence's behavior.

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

The following diagram illustrates the flow of execution for a UVM sequence:



Declaration

```

class uvm_sequence_base extends uvm_sequence_item;
  typedef enum {CREATED, PRE_START, PRE_BODY, BODY,
                POST_BODY, POST_START, ENDED, STOPPED,
                FINISHED} uvm_sequence_state;

  virtual class uvm_sequence #(type REQ = uvm_sequence_item,
                                type RSP = REQ)
    extends uvm_sequence_base;
  
```

Methods

uvm_sequence_base

<pre> function new(string name = "uvm_sequence"); </pre>	Constructor.
<pre> function uvm_sequence_state_enum get_sequence_state(); </pre>	Returns the state of the current sequence.
<pre> task wait_for_sequence_state(uvm_sequence_state_enum state); </pre>	Waits until the sequence is in the given state.

uvm_sequence

<pre>virtual task start(uvm_sequencer_base sequencer, uvm_sequence_base parent_sequence = null, int this_priority = -1, bit call_pre_post = 1);</pre>	Starts execution of the sequence on the specified sequencer. If <code>call_pre_post = 1</code> , <code>pre_body</code> and <code>post_body</code> tasks are called.
<code>virtual task pre_start();</code>	User-definable callback.
<code>virtual task pre_body();</code>	User-definable callback.
<code>virtual task pre_do(bit is_item);</code>	User-defined task called at the start of a <code>do</code> action performed by a sequence.
<code>virtual function void mid_do(uvm_sequence_item this_item);</code>	User-defined function called during a <code>do</code> action just after <code>this_item</code> is randomized and before either the item is sent to the consumer or subsequence body executed.
<code>virtual task body();</code>	Main method of the sequence.
<code>virtual function void post_do(uvm_sequence_item this_item);</code>	User-defined function called after either the consumer indicates the item is done or a subsequence <code>body()</code> completes.
<code>virtual task post_body();</code>	User-defined task called on root sequences immediately after the <code>body()</code> is executed.
<code>virtual task post_start();</code>	User-definable callback.
<code>virtual function bit is_item();</code>	Overrides <code>is_item</code> from <code>uvm_sequence_item</code> base class. Returns 0.
<code>function void set_priority(int value);</code>	Changes the priority of a sequence (default = 100).
<code>function int get_priority();</code>	Returns the current priority of a sequence.
<code>function bit is_blocked();</code>	Returns 1 if sequence is blocked; otherwise, 0.

<pre>virtual function bit is_relevant();</pre>	User-defined function that defines when sequence items are relevant for a <i>do</i> action; i.e., a <i>do</i> action on a sequence item is only selected if function returns 1'b1 (default implementation is 1'b1).
<pre>virtual task wait_for_relevant();</pre>	User-defined task that defines the trigger condition when the sequencer should re-evaluate if the sequence item is relevant (only called when <i>is_relevant()</i> returns 1'b0).
<pre>task lock(uvm_sequencer_base sequencer = null);</pre>	Gets a lock on the given sequencer, or blocks until it can (see Gotchas)..
<pre>function bit has_lock();</pre>	Returns 1 if sequence has a lock; otherwise, 0.
<pre>task grab(uvm_sequencer_base sequencer = null);</pre>	Gets a lock on the given sequencer, or blocks until it can (see Gotchas).
<pre>function void unlock(uvm_sequencer_base sequencer = null);</pre>	Unlocks the given sequencer.
<pre>function void ungrab(uvm_sequencer_base sequencer = null);</pre>	Unlocks the given sequencer.
<pre>virtual task wait_for_grant(int item_priority = -1, bit lock_request = 0);</pre>	Blocks until sequencer granted. Must be followed by call to <i>send_request</i> in same time step.
<pre>virtual task start_item(uvm_sequence_item item, int set_priority = -1, uvm_sequencer_base sequencer = null);</pre>	<i>start_item</i> and <i>finish_item</i> together will initiate operation of a sequence item.
<pre>virtual task finish_item(uvm_sequence_item item, int set_priority = -1);</pre>	See <i>start_item</i> .

uvm_sequence

<pre>virtual function void send_request(uvm_sequence_item request, bit rerandomize = 0);</pre>	Sends item to sequencer. Must follow call to wait_for_grant. Item is randomized if rerandomize = 1.
<pre>virtual task wait_for_item_done(int transaction_id = -1);</pre>	Blocks until driver calls item_done or put. If specified, will wait until response with matching ID is seen.
<pre>function void kill();</pre>	Kills the sequence.
<pre>virtual function void do_kill();</pre>	User hook, called whenever a sequence is terminated by using either sequence.kill() or sequencer.stop_sequences().
<pre>function void use_response_handler(bit enable);</pre>	Changes the behavior to use the response handler if enable = 1.
<pre>function bit get_use_response_handler();</pre>	Returns 1 if behavior set to use response handler.
<pre>virtual function void response_handler(uvm_sequence_item response);</pre>	The response handler – called automatically with response item if enabled.
<pre>function void set_response_queue_ error_report_disabled(bit value);</pre>	1 = disables errors from being reported if the response queue overflows.
<pre>function bit get_response_queue_ error_report_disabled();</pre>	Returns 0 if response queue overflow results in errors, 1 if disabled.
<pre>function void set_response_queue_depth(int value);</pre>	-1 indicates arbitrary depth. Default is 8.
<pre>function int get_response_queue_depth();</pre>	Returns response queue depth.
<pre>virtual function void clear_response_queue();</pre>	Empties the response queue.

uvm_sequence

<pre>function new(string name = "uvm_sequence");</pre>	Constructor.
--	--------------

<pre>function void send_request(uvm_sequence_item request, bit rerandomize = 0);</pre>	Sends sequence item to driver. Randomize item if rerandomize = 0.
<pre>function REQ get_current_item();</pre>	Returns sequence item currently executing (on sequencer).
<pre>task get_response(output RSP response, input int transaction_id = -1)</pre>	Retrieves response with matching ID (or next response if ID = -1) and removes it from queue. Blocks until response available.

Members

uvm_sequence_base

<pre>bit do_not_randomize;</pre>	Turns off randomization when using the `uvm_do and `uvm_rand_send macros.
<pre>uvm_phase starting_phase;</pre>	If non-null, specifies the phase in which this sequence was started. Automatically set when this sequence is started as the default sequence.

uvm_sequence

<pre>REQ req;</pre>	Request sequence item.
<pre>RSP rsp;</pre>	Response sequence item.
<pre>SEQUENCER p_sequencer†;</pre>	Handle to sequencer executing this sequence.

[†]Added by calling the `uvm_declare_p_sequencer macro. See [Sequence Action Macros](#).

Example

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

uvm_sequence

```
// Register sequence item with the factory and add the
// field automation macros
`uvm_object_utils_begin( my_seq_item )
  `uvm_field_int( data, UVM_ALL_ON )
  `uvm_field_int( addr, UVM_ALL_ON )
  `uvm_field_enum( dir_t, dir, UVM_ALL_ON )
`uvm_object_utils_end

endclass : my_seq_item

// Create a sequence that uses the sequence item
class my_seq extends uvm_sequence #(my_seq_item);

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

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

  // Define the sequence functionality in the body() using macros
  virtual task body();
    `uvm_info ( get_name(),
      "Starting the sequence ...", UVM_NONE )

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

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

endclass : my_seq
```

Here is an equivalent example that uses `start_item` and `finish_item` to generate a sequence item, and `start` to invoke a subsequence instead of using macros.

```
class my_seq extends uvm_sequence #(my_seq_item);
  `uvm_declare_p_sequencer(uvm_sequencer #(my_seq_item))
  ...
  virtual task body();
```

```

// Generate a sequence item
req = my_seq_item::type_id::create("req");
start_item(req);
randomize(req) with { addr > 10'hfff; dir == read; };
finish_item(req);

// Invoke a nested subsequence
subseq = my_other_seq::type_id::create("subseq");
subseq.start(p_sequencer);
...
endtask : body

```

Tips

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

```

`uvm_info(get_name(), "Now executing sequence",
          UVM_NONE )

`uvm_error(get_name(), $sformatf(
    "Write to an invalid address! Address = %s", addr)
)

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

```

// Modify the normal sequence to send error items
class error_seq extends normal_seq;
  // intf_seq_item seq_item; // defined in normal_seq
  ...
  factory.set_type_override_by_name(
    "intf_seq_item", "intf_error_seq_item" );

  `uvm_do ( seq_item )
  ...
endclass : error_seq

// Create a random sequence using randcase and factory type overrides

```

uvm_sequence

```
class rand_seq extends uvm_sequence;
    intf_seq_item seq_item;
    ...
    randcase
        // Send an error item 25% of the time
        1 : factory.set_type_override_by_name(
            "intf_seq_item", "error_seq_item" );
        // Send a complex item 25% of the time
        1 : factory.set_type_override_by_name(
            "intf_seq_item", "complex_seq_item" );
        // Send a normal item 50% of the time
        2 : factory.set_type_override_by_name(
            "intf_seq_item", "intf_seq_item" );
    endcase
    // Now send the randomly selected item
    `uvm_do ( seq_item )
    ...
endclass : rand_seq
```

Gotchas

- Take care to use a `uvm_sequence_item` or `uvm_sequence` instead of a `uvm_transaction` with the `do` sequence action macros.
- No `uvm_virtual_sequence` exists so use `uvm_sequence` for both virtual and non-virtual sequences.
- In UVM 1.1 `start_item/finish_item` cannot be used with sequences, only `sequence_items`. This is not backward compatible with UVM 1.0.
- A `uvm_sequence_item` must be created before calling `start_item`. A `uvm_sequence_item` must have its `randomize` method called explicitly (if random fields are required) before calling `finish_item`.
- A `uvm_sequence` object must be created before calling its `start` method.
- The `grab` and `lock` methods both attempt to lock the given sequencer immediately. Either will block if the sequencer is already locked, in which case `grab` goes to the front of the queue and `lock` goes to the back of the queue. When the sequencer is unlocked, the pending `grab` or `lock` nearest the front of the queue will be selected next, passing over any regular sequence items on the queue, which can only get selected once there are no more pending lock requests.

See also

[Sequence](#); [uvm_sequence_item](#); [uvm_sequencer](#); [Sequence Action Macros](#)

Sequence Action Macros

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

1. *Create* – allocate sequence item or sequence.
2. *Synchronize with sequencer* – if a sequence item, wait until the sequencer is ready.
3. *pre_do* – execute the user defined `pre_do` task of the executing sequence with an argument of 1 for a sequence item and 0 for a sequence.
4. *Randomize* – randomize the sequence item or sequence.
5. *mid_do* – execute the user defined `mid_do` task of the executing sequence with the specified sequence item or sequence as an argument.
6. *Post-synchronization or body execution* – for a sequence item, indicate to the sequencer that the item is ready to send to the consumer and wait for it to be consumed; for a sequence, execute the `body` task.
7. *post_do* – execute the user defined `post_do` task of the executing sequence with the specified item or sequence as an argument.

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

Macros

Macros used on regular sequences or sequence items:

<code>`uvm_create(item_or_sequence)</code>	Performs <i>ONLY</i> the create sequence action on an item or sequence using the factory.
<code>`uvm_declare_p_sequencer(sequencer)</code>	Declare a variable <code>p_sequencer</code> , which points to the sequencer.
<code>`uvm_do(item_or_sequence)</code>	Performs all sequence actions on an item or sequence. Sets priority to -1.
<code>`uvm_do_pri(item_or_sequence,priority)</code>	Same as <code>`uvm_do</code> but assigns a priority.
<code>`uvm_do_pri_with(item_or_sequence,priority,{constraint-block})</code>	Same as <code>`uvm_do_with</code> but assigns a priority.

Sequence Action Macros

<code>`uvm_do_with(item_or_sequence, {constraint-block})</code>	Performs all sequence actions on an item or sequence using the specified constraints for randomization. Sets priority to -1.
<code>`uvm_rand_send(item_or_sequence)</code>	Similar to `uvm_do but skips the create stage. Sets priority to -1.
<code>`uvm_rand_send_pri(item_or_sequence,priority)</code>	Same as `uvm_rand_send but assigns a priority.
<code>`uvm_rand_send_pri_with(item_or_sequence, priority, {constraint-block})</code>	Same as `uvm_rand_send_with but assigns a priority.
<code>`uvm_rand_send_with(item_or_sequence, {constraint-block})</code>	Similar to `uvm_do_with but skips the create stage. Sets priority to -1.
<code>`uvm_send(item_or_sequence)</code>	Similar to `uvm_do but skips the create and randomization stages. Sets priority to -1.
<code>`uvm_send_pri(item_or_sequence,priority)</code>	Same as `uvm_send but assigns a priority.

Macros used only with virtual sequences:

<code>`uvm_do_on(item_or_sequence, sequencer)</code>	Performs all sequence actions on a sequence started on the specified sequencer. Its parent member is set to this sequence. Sets priority to -1.
<code>`uvm_do_on_with(item_or_sequence, sequencer, {constraints-block})</code>	Same as `uvm_do_on but uses the specified constraints for randomization.
<code>`uvm_do_on_pri(item_or_sequence, sequencer, priority)</code>	Same as `uvm_do_on but assigns a priority.
<code>`uvm_do_on_pri_with(item_or_sequence, sequencer, priority, {constraints-block})</code>	Same as `uvm_do_on but assigns a priority and adds constraints.
<code>`uvm_create_on(item_or_sequence, sequencer)</code>	Creates sequence, assigns parent sequence, and associated sequencer.

Example

```
// Examples of `uvm_do and `uvm_do_with
class example_sequence extends
uvm_sequence #(example_sequence_item);
...
task body();
    // Send the sequence item to the driver
    `uvm_do( req )

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

// Examples of `uvm_create and `uvm_rand_send
class fixed_size_sequence extends
uvm_sequence #(example_sequence_item);
...
task body();
    // Allocate the sequence item
    `uvm_create ( req )

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

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

// Example of `uvm_do_on in a virtual sequence
class virtual_sequence extends uvm_sequence;
    `uvm_object_utils(virtual_sequence)
    `uvm_declare_p_sequencer(my_sequencer)
    write_sequence    wr_seq;
    read_sequence    rd_seq;
    ...
task body();
    fork // Launch the virtual sequences in parallel
        `uvm_do_on_with ( wr_seq,
            p_sequencer.seqr_a,
```

Sequence Action Macros

```
{ parity == 1; addr > 48; })  
  
`uvm_do_on_with ( rd_seq,  
                  p_sequencer.seqr_b,  
                  { width == 32; type == LONG; })  
join  
endtask : body  
endclass : virtual_sequence
```

Tips

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

Gotchas

- Do not use a semicolon after a macro.
- Take care to use a semicolon after the last constraint in a macro's constraint block.
- If you override a pre/mid/post_do callback in the situation where a sequence starts another sequence, it is the callback of the parent sequence that gets called, not the callback of the child sequence being started.

See also

[Sequence](#); [uvm_sequence](#); [Virtual Sequences](#); [uvm_sequencer](#)

uvm_sequence_item

Class `uvm_sequence_item` is derived from `uvm_transaction` and is used to represent the most basic transaction item within a sequence.

`uvm_sequence_item` should be used for user-defined transactions instead of `uvm_transaction`. When sequences are executed, they generate one or more sequence items, which the sequencer passes through its consumer interface to the driver's producer interface. The driver calls the `get_next_item` and `item_done` functions provided by its `uvm_seq_item_port` to pull the sequence items from the sequencer.

Note that `uvm_sequence` and `uvm_sequence_library` ultimately derive from `uvm_sequence_item`. In other words, a sequence is a sequence item and so is a sequence library; therefore, `uvm_sequence` and `uvm_sequence_library` inherit the `uvm_sequence_item`'s methods.

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

Declaration

```
class uvm_sequence_item extends uvm_transaction;
```

Methods

<code>function new(string name = "uvm_sequence_item");</code>	Constructor.
<code>function void set_item_context(uvm_sequence_base parent_seq, uvm_sequencer_base sequencer = null);</code>	Sets the sequence and sequencer context for a sequence item.
<code>function void set_use_sequence_info(bit value);</code>	Sets sequence info bit. Controls whether sequence information is printed, copied or recorded.
<code>function bit get_use_sequence_info();</code>	Returns current value of sequence info bit.
<code>function void set_id_info(uvm_sequence_item item);</code>	Copies sequence ID and transaction ID between sequence items. Should be called by drivers to ensure response matches request.

uvm_sequence_item

<pre>function void set_sequencer(uvm_sequencer_base sequencer);</pre>	Sets the sequencer for the sequence item (<i>not needed when using sequence action macros</i>).
<pre>function uvm_sequencer_base get_sequencer();</pre>	Returns the sequencer for the sequence item.
<pre>function void set_parent_sequence(uvm_sequence_base parent);</pre>	Set the item's parent sequence.
<pre>function uvm_sequence_base get_parent_sequence();</pre>	Get the item's parent sequence.
<pre>virtual function bit is_item();</pre>	Returns 1. Used to distinguish sequences from sequence_items.
<pre>function void set_depth(integer value);</pre>	Sets depth of sequence item (for example, 1 for a root sequence item, 2 for a child sequence item, etc.).
<pre>function integer get_depth();</pre>	Returns depth of a sequence item from its parent.
<pre>function string get_full_name();</pre>	Get the hierarchical sequence name (sequencer and parent sequences).
<pre>function string get_root_sequence_name();</pre>	Returns top-most sequence name.
<pre>function uvm_sequence_base get_root_sequence();</pre>	Returns reference to top-most sequence.
<pre>function string get_sequence_path();</pre>	Get the sequence name (including its parent sequences).

Members

<pre>bit print_sequence_info = 0;</pre>	If 1, sequence specific information is printed by item's print() function (set to 1 automatically by sequence macros).
---	--

Example

```
typedef enum { RX, TX } kind_t;
class my_seq_item extends uvm_sequence_item;
    rand bit [4:0] addr;
    rand bit [31:0] data;
    rand kind_t kind;
    // Constructor
    function new ( string name = "my_seq_item" );
        super.new ( name );
    endfunction : new

    // Register with UVM factory and use field automation
    `uvm_object_utils_begin( my_seq_item )
        `uvm_field_int ( addr, UVM_ALL_ON + UVM_DEC)
        `uvm_field_int ( data, UVM_ALL_ON + UVM_DEC)
        `uvm_field_enu ( kind_t, kind, UVM_ALL_ON )
    `uvm_object_utils_end
endclass : my_seq_item
```

Tips

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

Gotchas

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

See also

`uvm_transaction`; `uvm_sequence`; `uvm_sequencer`; Sequencer Interface and Ports; Sequence Action Macros

uvm_sequence_library

The `uvm_sequence_library` is a sequence (`uvm_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, depending on the `selection_mode` chosen.

The convenience class `uvm_sequence_library_cfg` is provided to make configuration of sequence libraries more straightforward.

The macro ``uvm_add_to_seq_lib` simplifies the task of adding sequences to sequence libraries.

Declaration

```
typedef enum {
    UVM_SEQ_LIB RAND,
    UVM_SEQ_LIB RANDC,
    UVM_SEQ_LIB ITEM,
    UVM_SEQ_LIB USER
} uvm_sequence_lib_mode;

class uvm_sequence_library #(type REQ=uvm_sequence_item,
                           RSP=REQ)
    extends uvm_sequence #(REQ,RSP);

class uvm_sequence_library_cfg extends uvm_object;
```

Methods

uvm_sequence_library

<code>virtual function int unsigned select_sequence(int unsigned max);</code>	In UVM_SEQ_LIB_USER mode, returns index of the next sequence to execute.
<code>static function void add_typewide_sequence(uvm_object_wrapper seq_type);</code>	Registers a sequence type with the library – affects all instances of the class.
<code>static function void add_typewide_sequences(uvm_object_wrapper seq_types[\$]);</code>	Registers the sequence types with the library – affects all instances of the class.
<code>function void add_sequence(uvm_object_wrapper seq_type);</code>	Registers the sequence type with an instance of the class.
<code>virtual function void add_sequences(uvm_object_wrapper seq_types[\$]);</code>	Registers the sequence types with an instance of the class.

<pre>virtual function void remove_sequence(uvm_object_wrapper seq_type);</pre>	Remove the sequence from the library.
<pre>virtual function void get_sequences(ref uvm_object_wrapper seq_types[\$]);</pre>	Get list of sequences in the library.
<pre>function void init_sequence_library();</pre>	Must be called by subclass constructor.

uvm_sequence_library_cfg

<pre>function new(string name = "", uvm_sequence_lib_mode mode = UVM_SEQ_LIB_RAND, int unsigned min = 1, int unsigned max = 10);</pre>	Constructor.
---	--------------

Members

<code>uvm_sequence_lib_mode selection_mode;</code>	The mode used to select sequences for execution.
<code>int unsigned min_random_count=10;</code>	The minimum number of items to execute.
<code>int unsigned max_random_count=10;</code>	The maximum number of items to execute.
<code>rand int unsigned sequence_count;</code>	The number of items to execute.
<code>protected int unsigned sequences_executed;</code>	Number of sequences executed (not including current sequence).
<code>rand int unsigned select_rand;</code>	In UVM_SEQ_LIB_RAND mode, the index of the next sequence to execute.
<code>randc bit [15:0] select_randc;</code>	In UVM_SEQ_LIB_RANDC mode, the index of the next sequence to execute.
<code>protected uvm_object_wrapper sequences[\$];</code>	All registered sequence types.

uvm_sequence_library

Macros

<code>`uvm_sequence_library_utils(TYPE)</code>	All sequence libraries must use this macro.
<code>`uvm_add_to_seq_lib (TYPE, LIBTYPE)</code>	Adds the sequence TYPE to the sequence library LIBTYPE.

Selection Modes

<code>UVM_SEQ_LIB RAND</code>	Random sequence selection.
<code>UVM_SEQ_LIB RANDC</code>	Random cyclic sequence selection.
<code>UVM_SEQ_LIB ITEM</code>	Emit only items, no sequence execution.
<code>UVM_SEQ_LIB USER</code>	Apply a user-defined random-selection algorithm – causes <code>select_sequence</code> to be called.

Rules

- When extending `uvm_sequence_library`, the constructor must call `init_sequence_library()`.

When extending `uvm_sequence_library`, the macro ``uvm_sequence_library_utils` must be used.

Example

```
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
```

Individual sequence types may then be added to the sequence library, by type, using the ``uvm_add_to_seq_lib` macro:

```
class my_seq extends uvm_sequence #(my_item);
  `uvm_object_utils(my_seq);
  `uvm_add_to_seq_lib (my_seq, my_seq_lib)
  ...
endclass
```

Here is a test that uses the sequence library:

```
class my_test extends uvm_test;
  ...
  function void build_phase(uvm_phase phase);
    uvm_sequence_library_cfg cfg;
    super.build_phase(phase);

    // Create an instance of the configuration convenience class
    cfg = new("seqlib_cfg", UVM_SEQ_LIB_RANDOM, 1000, 2000);
    uvm_config_db #(uvm_sequence_library_cfg)::set(
      null,
      "*.env.agent.sequencer.main_phase",
      "default_sequence.config",
      cfg);

    // Configure the sequence library and cause it to be run on a sequencer -
    // it is a sequence and is run in exactly the same way, either by setting
    // the sequencer's default_sequence or starting it manually.
    uvm_config_db #(uvm_sequencer_base)::set(
      this,"*.my_sequencer.main_phase",
      "default_sequence",
      my_seq_lib::type_id::get());
  
```

See also

[Sequence](#); [uvm_sequence](#); [uvm_sequence_item](#); [uvm_sequencer](#)

uvm_sequencer

The sequencer controls the flow of `uvm_sequence_item` transactions generated by one or more `uvm_sequence` sequences. A sequencer is connected to a driver using a TLM export. A sequencer randomizes sequence items and sends them to the driver.

A *virtual sequencer* includes references to subsequencers, which are configured by a higher-level component, typically the testbench. These subsequencers may themselves be virtual sequencers or (driver) sequencers. A virtual sequencer is not connected directly to a driver.

`uvm_sequencer` is used for both sequencers and virtual sequencers.

A sequencer may be configured with a `default_sequence`. If so, the sequencer will start the sequence automatically. By default, a sequencer will not generate sequences; either a `default_sequence` must be configured, or sequences must be started manually on the sequencer.

A sequencer can be configured to generate different sorts of transactions, using the factory.

`uvm_sequencer` implements a pull semantic: the driver pulls transactions (`sequence_items`) from the sequencer. `uvm_push_sequencer` uses a push semantic: the sequencer pushes transactions to the (push) driver.

Declaration

```
class uvm_sequencer_base extends uvm_component;
class uvm_sequencer_param_base
  #(type REQ = uvm_sequence_item, type RSP = REQ)
  extends uvm_sequencer_base;

class uvm_sequencer #(type REQ = uvm_sequence_item,
                     RSP = REQ)
  extends uvm_sequencer_param_base #(REQ, RSP);
class uvm_push_sequencer
  #(type REQ = uvm_sequence_item, RSP = REQ)
  extends uvm_sequencer_param_base #(REQ, RSP);

typedef enum {SEQ_TYPE_REQ, SEQ_TYPE_LOCK, SEQ_TYPE_GRAB}
  seq_req_t;
typedef enum {SEQ_ARB_FIFO, SEQ_ARB_WEIGHTED,
             SEQ_ARB_RANDOM, SEQ_ARB_STRICT_FIFO,
             SEQ_ARB_STRICT_RANDOM, SEQ_ARB_USER}
  SEQ_ARB_TYPE;
```

Methods

uvm_sequencer_base

<code>function new(string name, uvm_component parent);</code>	Constructor.
<code>function bit is_child(uvm_sequence_base parent, uvm_sequence_base child);</code>	Returns 1 if child is child of parent.
<code>virtual function integer user_priority_arbitration(integer avail_sequences[\$]);</code>	Called if arbitration mode = SEQ_ARB_USER to arbitrate between sequences. Default behavior is SEQ_ARB_FIFO.
<code>virtual task execute_item(uvm_sequence_item item);</code>	Cause sequencer to execute item.
<code>virtual function void start_phase_sequence(uvm_phase phase);</code>	Start the default sequence for this phase, if any.
<code>virtual task wait_for_grant(uvm_sequence_base sequence_ptr, int item_priority = -1, bit lock_request = 0);</code>	Issues request for sequence and waits until granted.
<code>virtual task wait_for_item_done(uvm_sequence_base sequence_ptr, int transaction_id);</code>	Waits until driver calls item_done or put.
<code>function bit is_blocked(uvm_sequence_base sequence_ptr);</code>	Returns 1 if sequence is blocked.
<code>function bit has_lock(uvm_sequence_base sequence_ptr);</code>	Returns 1 if sequence has a lock on this sequencer, 0 otherwise.
<code>virtual task lock(uvm_sequence_base sequence_ptr);</code>	The given sequence locks the calling sequencer, or blocks until it can (see Gotchas).
<code>virtual task grab(uvm_sequence_base sequence_ptr);</code>	The given sequence locks the calling sequencer, or blocks until it can (see Gotchas).
<code>virtual function void unlock(uvm_sequence_base sequence_ptr);</code>	Unlocks the sequencer.
<code>virtual function void ungrab(uvm_sequence_base sequence_ptr);</code>	Ungrabs the sequencer.
<code>virtual function void stop_sequences();</code>	Kills all sequences running on this sequencer.

uvm_sequencer

<pre>virtual function bit is_grabbed();</pre>	Returns 1 if a sequence is currently grabbing exclusive access, 0 if no sequence is grabbing.
<pre>virtual function uvm_sequence_base current_grabber();</pre>	Returns a reference to the current grabbing sequence, <i>null</i> if no grabbing sequence.
<pre>virtual function bit has_do_available();</pre>	Returns 1 if the sequencer has an item available for immediate processing, 0 if no items are available.
<pre>function void set_arbitration(SEQ_ARB_TYPE val);</pre>	Change the arbitration mode. Default mode is SEQ_ARB_FIFO.
<pre>function SEQ_ARB_TYPE get_arbitration();</pre>	Returns the arbitration mode.
<pre>virtual task wait_for_sequences();</pre>	User overridable task used to introduce delta delay cycles, allowing processes placing items in the consumer interface to complete before the producer interface retrieves the items.

uvm_sequencer_param_base

<pre>virtual function void send_request(uvm_sequence_base sequence_ptr, uvm_sequence_item t, bit rerandomize = 0);</pre>	Called after wait_for_grant, sends transaction to the driver.
<pre>function REQ get_current_item();</pre>	Returns the sequence item being executed.
<pre>function int get_num_reqs_sent();</pre>	Returns number of requests sent by sequencer.
<pre>function int get_num_rspns_received();</pre>	Returns number of responses received by sequencer.
<pre>function void set_num_last_reqs(int unsigned max);</pre>	Sets size of last request buffer (default=1,max=1024)
<pre>function int unsigned get_num_last_reqs();</pre>	Gets size of last requests buffer.

<code>function REQ last_req(int unsigned n = 0);</code>	Gets the last request from the buffer (or position within buffer).
<code>function void set_num_last_rsp(</code> int unsigned max);	Sets size of last response buffer (default=1,max=1024)
<code>function int unsigned get_num_last_rsp();</code>	Gets size of last response buffer.
<code>function RSP last_rsp(int unsigned n = 0);</code>	Gets the last response from the buffer (or position within buffer).

uvm_sequencer

<code>virtual function void stop_sequences();</code>	Kill all sequences and child sequences currently operating on the sequencer.
--	--

uvm_push_sequencer

<code>task run_phase(uvm_phase phase);</code>	Sends sequence items on its req_port.
---	---------------------------------------

Members**uvm_sequencer_param_base**

<code>uvm_analysis_export #(RSP) rsp_export;</code>	Analysis export that may be used to send responses to sequencer.
---	--

uvm_sequencer

<code>uvm_seq_item_pull_imp #(REQ, RSP, this_type) seq_item_export;</code>	Sequence item export (connects to driver).
--	--

uvm_push_sequencer

<code>uvm_blocking_put_port #(REQ) req_port;</code>	Sequence item export (connects to push-driver).
---	---

Examples

Two examples follow, each using a slightly different coding idiom. The first is a straightforward environment with a single sequencer. The second shows communication between two sequencers.

uvm_sequencer

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

// Register sequence item with the factory and add the
// field automation macros
`uvm_object_utils_begin( my_seq_item )
  `uvm_field_int( data, UVM_ALL_ON )
  `uvm_field_int( addr, UVM_ALL_ON )
  `uvm_field_enum( dir_t, dir, UVM_ALL_ON )
`uvm_object_utils_end
endclass : my_seq_item

// Define a sequence – generates between 1 and 20 transactions
class my_sequence extends uvm_sequence #(my_seq_item);
  rand int num_items = 5;
  constraint c_num_items { num_items > 0; num_items <= 20; }
  task body;
    if ( starting_phase != null )
      starting_phase.raise_objection(this);
    repeat (num_items)
      begin
        my_seq_item tx;
        `uvm_do(tx)
      end
    if ( starting_phase != null )
      starting_phase.drop_objection(this);
  endtask : body
endclass

// Create a typedef for the sequencer
typedef uvm_sequencer #(my_seq_item) my_sequencer;

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

  function void connect_phase(uvm_phase phase);
    // Hook up the sequencer to the driver
  
```

```
m_drv.seq_item_port.connect(m_seqr.seq_item_export);
endfunction : connect_phase
...
endclass : my_env

// Create a test
class my_test extends uvm_test;
    my_env m_env;

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

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        // Use the factory to create the environment
        m_env = my_env::type_id::create("m_env", this);
        // Set the sequencer's default sequence
        uvm_config_db #(uvm_sequence_base)::set(
            null,
            "*.m_seqr.main_phase",
            "default_sequence",
            my_sequence::type_id::get());
    endfunction : build_phase

endclass : my_test
The following example shows communication between two sequencers using a
sequencer interface.

class my_transaction extends uvm_sequence_item; ...

class my_program extends uvm_sequence_item;
    `uvm_object_utils(my_program)

    // my_program contains a dynamic array of my_transactions
    my_transaction tx_h[];

    ...
    // When a my_program object is randomized, a random number (n) of
    // randomized my_transactions are created and stored in the dynamic array
    function void pre_randomize;
        int n = $urandom_range(3, 6);
        tx_h = new[n];
    endfunction
endclass
```

uvm_sequencer

```
for (int i = 0; i < n; i++)
begin
    tx_h[i] = new;
    assert( tx_h[i].randomize() );
end
endfunction
endclass : my_program

class my_program_seq extends uvm_sequence #(my_program);
`uvm_object_utils(my_program_seq)

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

// Create one my_program item, which contains n my_transaction items
task body;
    my_program tx;
    tx = my_program:::type_id::create("tx");
    start_item(tx);
    assert( tx.randomize() );
    finish_item(tx);
endtask: body

endclass: my_program_seq

// The sequencer for my_transactions contains a uvm_seq_item_pull_port;
// the my_program sequencer will pull my_program items through this
class my_tx_sqr extends uvm_sequencer #(my_transaction);
`uvm_component_utils(my_tx_sqr)

uvm_seq_item_pull_port #(my_program) seq_item_port;

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

function void build_phase(uvm_phase phase);
    seq_item_port = new("seq_item_port", this);
endfunction: build_phase
endclass: my_tx_sqr

class my_tx_seq extends uvm_sequence #(my_transaction);
`uvm_object_utils(my_tx_seq)
```

```
// Includes the p_sequencer member – points to this sequence's sequencer
`uvm_declare_p_sequencer(my_tx_sqr)

my_program my_program_h;

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

task body;
fork
    forever
begin
    int n;
    // Get a my_program item from the my_program sequencer using
    // the interface that connects it to the my_transaction sequencer
    p_sequencer.seq_item_port.get(my_program_h);
    // Find out how many my_transaction items there are
    n = my_program_h.tx_h.size();
    // Create the same number here, copy across and generate
    for (int i = 0; i < n; i++)
begin
    my_transaction tx;
    tx = my_transaction::type_id::create("tx");
    start_item(tx);
    tx.cmd  = my_program_h.tx_h[i].cmd;
    tx.addr = my_program_h.tx_h[i].addr;
    tx.data = my_program_h.tx_h[i].data;
    finish_item(tx);
end
end
join_none
endtask: body

endclass: my_tx_seq
```

uvm_sequencer

```
class my_env extends uvm_env;
  `uvm_component_utils(my_env)
  uvm_sequencer #(my_program) my_program_sqr_h;
  my_tx_sqr          my_tx_sqr_h;

  ...

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // Create the sequencers (and other components)
    my_program_sqr_h
      = uvm_sequencer #(my_program)::type_id::create(
        "my_program_sqr_h", this);
    my_tx_sqr_h     = my_tx_sqr     ::type_id::create(
        "my_tx_sqr_h"     , this);
    ...
  endfunction: build_phase

  function void connect_phase(uvm_phase phase);
    // Connect the sequencers' port and export
    my_tx_sqr_h.seq_item_port.connect(
      my_program_sqr_h.seq_item_export );
    // Other component connections
    ...
  endfunction: connect_phase
endclass: my_env

class my_test extends uvm_test;
  `uvm_component_utils(my_test)

  my_env my_env_h;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    my_env_h = my_env::type_id::create("my_env_h", this);
  endfunction: build_phase

  task run_phase(uvm_phase phase);
    my_program_seq program_seq;
    my_tx_seq      tx_seq;
```

```
// Create and start a my_tx_seq
tx_seq = my_tx_seq::type_id::create("tx_seq");
tx_seq.start( my_env_h.my_tx_sqr_h );

// Create and start a my_program_seq
program_seq = my_program_seq::type_id::create(
    "program_seq");
assert( program_seq.randomize() );
program_seq.start( my_env_h.my_program_sqr_h );
phase.phase_done.set_drain_time( this, 30ns );
endtask: run_phase

endclass: my_test
```

Gotchas

- By default, a sequencer will not execute any sequences. You must either use a `default_sequence` (see examples above in Tips section) or start a sequence manually.
- The `grab` and `lock` methods both attempt to lock the given sequencer immediately. Either will block if the sequencer is already locked, in which case `grab` goes to the front of the queue and `lock` goes to the back of the queue. When the sequencer is unlocked, the pending `grab` or `lock` nearest the front of the queue will be selected next, passing over any regular sequence items on the queue, which can only get selected once there are no more pending lock requests.

See also

[Sequence; Virtual Sequences; uvm_sequence; uvm_sequence_item;](#)
[uvm_driver; Sequence Action Macros; Sequencer Interface and Ports](#)

Sequencer Interface and Ports

In UVM, the passing of transactions between sequencers and drivers happens through a sequencer interface export/port pair. A sequencer producing items or sequences contains a sequence item export, and a driver consuming items or sequences contains a sequence item port.

The UVM library provides an interface class and an associated export and port to handle the communication between sequencers and drivers. The interface is defined by class `uvm_sqr_if_base`.

An instance of class `uvm_seq_item_pull_port` named `uvm_seq_item_port` is a member of `uvm_driver`. It enables the driver to call interface methods such as `get_next_item` and `item_done`, which pull sequence items from the sequencer's item queue.

An instance of class `uvm_seq_item_pull_imp` named `uvm_seq_item_export` is a member of `uvm_sequencer`. It provides the implementation of the sequencer interface methods which manage the queuing of sequence items and sequencer synchronization.

Declarations

```
virtual class uvm_sqr_if_base #(type T1=uvm_object, T2=T1);  
  
class uvm_seq_item_pull_port #(type REQ=int, type RSP=REQ)  
  extends uvm_port_base #(uvm_sqr_if_base #(REQ, RSP));  
  
class uvm_seq_item_pull_export #(type REQ=int,  
                           type RSP=REQ)  
  extends uvm_port_base #(uvm_sqr_if_base #(REQ, RSP));  
  
class uvm_seq_item_pull_imp #(type REQ=int,  
                           type RSP=REQ, type IMP=int)  
  extends uvm_port_base #(uvm_sqr_if_base #(REQ, RSP));
```

Parameters

<code>type T1=uvm_object</code>	Base type for retrieved item
<code>type T2=T1</code>	Base type for response item
<code>type REQ=int</code>	Base type for requested item
<code>type RSP=REQ</code>	Base type for response item
<code>type IMP=int</code>	Base type for parent component

Methods**uvm_sqr_if_base**

<code>virtual task get_next_item(output T1 t);</code>	Blocks until an item is returned from the sequencer.
<code>virtual task try_next_item(output T1 t);</code>	Returns immediately an item if available, otherwise, <i>null</i> .
<code>virtual function void item_done(input T2 t = null);</code>	Indicates to the sequencer that the driver is done processing the item.
<code>virtual task wait_for_sequences();</code>	Calls the <code>wait_for_sequences</code> task of the connected sequencer (see <code>uvm_sequencer</code>).
<code>virtual function bit has_do_available();</code>	Returns 1 if item is available, 0 if no item available.
<code>virtual function void put_response(input T2 t);</code>	Puts a response into the sequencer queue.
<code>virtual task get(output T1 t);</code>	Blocks until item is returned from sequencer. Call <code>item_done</code> before returning.
<code>virtual task peek(output T1 t);</code>	Blocks until item is returned from sequencer. Does not remove item from sequencer fifo.
<code>virtual task put(input T2 t);</code>	Sends response back to sequencer.

uvm_seq_item_pull_port

<code>function new (string name, uvm_component parent, int min_size=0, int max_size=1);</code>	Constructor.
--	--------------

Plus implementation of `sqr_if_base` methods

Sequencer Interface and Ports

uvm_seq_item_pull_export

<pre>function new (string name, uvm_component parent, int min_size=1, int max_size=1);</pre>	Constructor.
--	--------------

Plus implementation of sqr_if_base methods

uvm_seq_item_pull_imp

<pre>function new (string name, IMP parent);</pre>	Constructor.
--	--------------

Plus implementation of sqr_if_base methods

Example

This example demonstrates the use of `uvm_seq_item_pull_port` and `uvm_seq_item_pull_imp` between a driver and sequencer.

```
class my_driver extends uvm_driver #(my_trans);
  ...
  task run_phase(uvm_phase phase);
    forever begin
      // Pull a sequence item from the interface. We could use get
      // instead of get_next_item, in which case we don't need to call
      // to call item_done.
      seq_item_port.get_next_item( req );
      ...
      // Apply transaction to DUT interface
      ...
      // Set the response id and indicate that item is done
      rsp = new("rsp");
      rsp.set_id_info(req);
      seq_item_port.item_done( rsp );
    end
  endtask : run_phase
endclass : my_driver

// Connect the sequencer and driver together
class my_env extend uvm_env;
  ...
  function void connect_phase(uvm_phase phase);
    my_drv.seq_item_port.connect(
      my_seqr.seq_item_export );
  endfunction
endclass : my_env
```

```
endfunction : connect_phase  
endclass : my_env
```

Tips

A driver can call `get_next_item` multiple times before indicating `item_done` to the sequencer (in other words, there is no one-to-one correspondence of function calls).

See also

`uvm_driver`; `uvm_sequencer`; `Virtual Sequences`; `uvm_sequence`

uvm_subscriber

`uvm_subscriber` is a convenient base class for a user-defined *subscriber* (an analysis component that will receive transaction data from another component's analysis port).

A subscriber that is derived from `uvm_subscriber` must override its inherited `write` method, which will be called automatically whenever a transaction data item is written to a connected analysis port. The `analysis_export` member of a subscriber instance should be connected to the analysis port that produces the data (typically on a monitor or other verification component).

Declaration

```
virtual class uvm_subscriber #(type T = int)
  extends uvm_component;
```

Methods

<code>function new(string name, uvm_component parent);</code>	Constructor.
<code>pure virtual function void write(transaction_type t);</code>	Override this method to implement your subscriber's behavior when it receives a transaction data item.

Members

<code>uvm_analysis_imp #(transaction_type, this_type) analysis_export;</code>	Implementation of an analysis export, ready for connection to an analysis port.
---	---

Example

```
// Define a specialized subscriber class. This class does nothing except to
// report all transactions it receives.
class example_subscriber extends
    uvm_subscriber #(example_transaction);
int transaction_count;
function new(string name, uvm_component parent);
    super.new(name, parent);
endfunction
function void write (example_transaction t);
    `uvm_info("WRITE", $sformatf(
        "Received transaction number %0d:\n%s",
        transaction_count++, t.sprint() ), UVM_NONE )
endfunction
endclass

// In the enclosing environment class:
class subscriber_test_env extends uvm_env;
    example_subscriber m_subscriber;
    example_monitor     m_monitor; // see article uvm_monitor
    ...
    function void build_phase(uvm_phase phase);
        ...
        m_monitor      = example_monitor::type_id::create(
            "m_monitor",this);
        m_subscriber   = example_subscriber::type_id::create(
            "m_subscriber",this);
        ...
    endfunction
    function void connect_phase(uvm_phase phase);
        // Connect monitor's analysis port (requires)
        // to subscriber's export (provides)
        m_monitor.monitor_ap.connect (
            m_subscriber.analysis_export );
    endfunction
    ...
endclass
```

Tips

- Use `uvm_subscriber` as the base for any custom class that needs to monitor a single stream of transactions. Typical uses include coverage collection, protocol checking, or data logging to a file. It is not appropriate

uvm_subscriber

for end-to-end checkers since these typically need to monitor transaction streams from more than one analysis port (see *Gotchas* below). For such applications, one of the built-in comparator components such as

`uvm_in_order_class_comparator` or
`uvm_algorithmic_comparator` may be appropriate.

Gotchas

- Unless one of the built-in comparator components meets your need, a component that needs to subscribe to more than one analysis port must be created as a specialized extension of `uvm_component` with a separate `uvm_analysis_export` for connection to each analysis port that will provide data. Classes derived from `uvm_subscriber` are applicable only for monitoring the output from a single analysis port (but note that a custom comparator could contain multiple subscriber members that implement the `write` function for each of its `analysis_exports`).
- The argument for the overridden `write` function MUST be named "t".

See also

`uvm_in_order_*_comparator`; `uvm_analysis_export`

A class derived from `uvm_test` should be used to represent each test case. A test will create and configure the environment(s) required to verify particular features of the device under test (DUT). There will typically be multiple test classes associated with a testbench: a single test object is created from one of these at the start of each simulation run. This approach separates the testbench from individual test cases and improves its reusability. `uvm_test` is itself derived from `uvm_component` so a test may include a `run_phase` task. A test class is sometimes defined, but never explicitly instantiated, in the top-level testbench module (a test class cannot be defined in a package if it needs to include hierarchical references).

The `uvm_top.run_test` task or the global `run_test` task is called from an initial block in the top-level testbench module to instantiate a test (using the factory) and then run it (`run_test` is a wrapper that calls `uvm_top.run_test`).

The test's `build_phase` method creates the top-level environment(s).

The simulator's command-line plusarg `+UVM_TESTNAME=testname` specifies the name of the test to run (a name is associated with a test by registering the test class with the factory). If this plusarg is not used, then the `test_name` argument of `run_test` may be used to specify the test name instead. If no test name is given, no test or environment will be created and a fatal message will be issued.

Configuration and/or factory overrides may be used within the test to customize a reusable test environment (or any of its components) without having to modify the environment code.

Declaration

```
virtual class uvm_test extends uvm_component;
```

Methods

<code>function new (string name, uvm_component parent);</code>	Constructor.
---	--------------

Also, *inherited methods like* `build_phase`, `configure_phase`, `connect_phase`, and `run_phase`.

Example

This example shows a test being defined and run from a top-level module.

```
module top;

class test1 extends uvm_test;
  ...
  function void build_phase(uvm_phase phase);
    // Create environment
  endfunction: build_phase

  function void connect_phase(uvm_phase phase);
    // Connect test environment's virtual interface to the DUT's interface
    m_env.m_mon.m_bus_if = tf.cpu_if.mon;
  endfunction: connect_phase

  task run_phase(uvm_phase phase);
    // Call methods in environment to control test (optional)
  endtask: run_phase

  // Register test with factory
  `uvm_component_utils(test1)
endclass: test1

initial begin
  run_test("test1"); // Use test1 by default
                     // Can override using +UVM_TESTNAME
end

// Contains DUT, DUT interface, clock/reset, ancillary modules etc.
test_harness tf ();
```

```
endmodule
```

Tips

- Write a new test class for each test case in your verification plan.
- Keep the test classes simple: functionality that is required for every test should be part of the testbench and not repeated in every test class.
- It is a good idea to start with a “default” test that provides random stimulus before spending time developing more directed tests.

- Use the `connect_phase` method to connect the virtual interface in the driver to the DUT's interface.
- The name of the test instance is "uvm_test_top".

Gotchas

- Do not forget to register the test with the factory, using ``uvm_component_utils`.
- Do not call the `set_inst_override` member function (inherited from `uvm_component`) for a top-level test.

See also

`uvm_env`; Configuration; `uvm_factory`

TLM Implementation Declaration Macros

TLM imp ports require the target method to be defined or implemented in the class a port is instantiated. For example, a class that uses a `uvm_put_imp` port must implement the corresponding `put` method. A problem arises, however, if more than one imp port is used which require the same method to be implemented. To work around this, UVM defines a set of macros that can uniquify the name of the imp method with a custom suffix. For example,

```
`uvm_put_imp_decl(_usb)
```

creates a custom TLM put port called `uvm_put_imp_usb`, which requires a method called `put_usb`. By using these customized TLM ports, multiple implementations can exist in the same component, and these customized ports can still be connected to the default TLM port and export types.

Macros

These macros create TLM ports of the same name (without the `_decl`) with the *SUFFIX* appended to the end of the name.

```
`uvm_blocking_put_imp_decl(SUFFIX)
`uvm_nonblocking_put_imp_decl(SUFFIX)
`uvm_put_imp_decl(SUFFIX)
`uvm_blocking_get_imp_decl(SUFFIX)
`uvm_nonblocking_get_imp_decl(SUFFIX)
`uvm_get_imp_decl(SUFFIX)
`uvm_blocking_peek_imp_decl(SUFFIX)
`uvm_blocking_get_peek_imp_decl(SUFFIX)
`uvm_nonblocking_get_peek_imp_decl(SUFFIX)
`uvm_get_peek_imp_decl(SUFFIX)
`uvm_blocking_master_imp_decl(SUFFIX)
`uvm_nonblocking_master_imp_decl(SUFFIX)
`uvm_master_imp_decl(SUFFIX)
`uvm_blocking_slave_imp_decl(SUFFIX)
`uvm_nonblocking_slave_imp_decl(SUFFIX)
`uvm_slave_imp_decl(SUFFIX)
`uvm_blocking_transport_imp_decl(SUFFIX)
`uvm_nonblocking_transport_imp_decl(SUFFIX)
`uvm_transport_imp_decl(SUFFIX)
`uvm_analysis_imp_decl(SUFFIX)
```

Example

Declaring two analysis ports inside of a scoreboard:

```
'uvm_analysis_imp_decl(_ahb)
'uvm_analysis_imp_decl(_pci)

class fullchip_sb extends uvm_scoreboard;

  uvm_analysis_imp_ahb #(ahb_trans, fullchip_sb) ahb_ap;
  uvm_analysis_imp_pci #(pci_trans, fullchip_sb) pci_ap;

  ...

  function void write_ahb(ahb_trans t) {
    // AHB implementation
    expected_queue.push_back( compute_result(t) );
  }

  function void write_pci(pci_trans t) {
    // PCI implementation
    assert ( t.compare( expected_queue.pop_front() ) ) else
      `uvm_error("FULLCHIP_SB", "Actual != Expected!")
  }

endclass
```

Tips

- Include a separator like an underscore before the *SUFFIX* name to make the new TLM port names more readable.
- Place these macros in a package to avoid namespace collisions and to make it easier to share the new, customized TLM ports.

Gotchas

- Be careful to not call the same macro with the same suffix in the same context or namespace collisions will occur.

See also

[TLM-1 Ports, Exports and Imps](#)

TLM-1 Interfaces

UVM provides a set of interface methods (not to be confused with SystemVerilog interfaces) that provide standard transaction-level communication methods for ports and exports (they are based on the SystemC TLM-1 library). These methods exist in “blocking” and “non-blocking” forms. Blocking methods may wait before returning and are always tasks. Non-blocking methods are not allowed to wait and are implemented as functions. The interface methods are virtual tasks and functions that are not intended to be called directly by applications. (They will issue an error message if they are called.)

Three types of operation are supported. The semantics are as follows:

- *Putting* a transaction into a channel, e.g. a request message from an initiator. This adds the transaction to the current channel but does not overwrite the existing contents.
- *Getting* a transaction from a channel, e.g. a response message from a target. This removes the transaction from the channel.
- *Peeking* at a transaction in the channel without removing it.

Unidirectional “blocking interfaces” contain a single task. Unidirectional “non-blocking interfaces” contain one function to access a transaction and typically one or more other functions to test the state of the channel conveying the transaction. For convenience, “combined interfaces” are provided which include all of the methods from related blocking and non-blocking interfaces.

Bidirectional interfaces combine a pair of related unidirectional interfaces so that request and response transactions can be sent between an initiator and target using a single bidirectional channel.

TLM-1 ports and exports are associated with each of the TLM-1 interfaces. They provide a mechanism to decouple the initiator and target of a transaction, providing encapsulation and improving the reusability.

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

Declarations

```
virtual class uvm_tlm_if_base #(type T1 = int,  
                           type T2 = int);
```

Methods

virtual task put (input T t);	Blocks until t can be accepted
virtual task get (output T t);	Blocks until t can be fetched
virtual task peek (output T t);	Blocks until t is available

virtual function bit try_put (input T t);	Returns 1 if successful, otherwise 0
virtual function bit can_put ();	Returns 1 if transaction would be accepted, otherwise 0
virtual function bit try_get (output T t);	Returns 1 if t successful, otherwise 0
virtual function bit can_get ();	Returns 1 if transaction is available, otherwise 0
virtual function bit try_peek (output T t);	Returns 1 if successful, otherwise 0
virtual function bit can_peek ();	Returns 1 if transaction is available, otherwise 0
virtual task transport (input T1 request, output T2 response);	Blocks until request accepted and response is returned
virtual function bit nb_transport (input T1 request, output T2 response);	Returns 1 if request accepted and response returned immediately, otherwise 0
virtual function void write (input T t);	Broadcasts transaction t to listeners.

Example

Declaring port to connect to `uvm_tlm_req_rsp_channel`:

```
uvm_master_port #(my_req,my_rsp) m_master;
```

Calling `uvm_tlm_req_rsp_channel` methods via port:

```
m_master.put(req1);
m_master.get(req1);
```

Use of `master_imp` and `slave_imp` in `uvm_tlm_req_rsp_channel` to
implement exports:

```
class uvm_tlm_req_rsp_channel
#( type REQ = int , type RSP = REQ )
extends uvm_component;

typedef uvm_tlm_req_rsp_channel #( REQ , RSP ) this_type;

protected uvm_tlm_fifo #( REQ ) m_request_fifo;
```

TLM-1 Interfaces

```
protected uvm_tlm_fifo #( RSP ) m_response_fifo;  
...  
uvm_master_imp  
#( REQ , RSP , this_type , uvm_tlm_fifo #( REQ ) ,  
  uvm_tlm_fifo #( RSP ) ) master_export;  
  
uvm_slave_imp  
#( REQ , RSP , this_type , uvm_tlm_fifo #( REQ ) ,  
  uvm_tlm_fifo #( RSP ) ) slave_export;  
...
```

Exports instantiated in function called from `uvm_tlm_req_rsp_channel` constructor:

```
master_export = new( "master_export" , this ,  
                     m_request_fifo, m_response_fifo );  
  
slave_export = new( "slave_export" , this ,  
                     m_request_fifo , m_response_fifo );
```

Tips

- It is often easier to use the combined exports when creating a channel since these can be connected to blocking, non-blocking, or combined ports.

Gotchas

- Remember that the export that provides the actual implementation of the interface methods should use `uvm_*_imp` rather than `uvm_*export`.
- A `uvm_*_imp` instance requires a type parameter to set the type of the class that will define its interface methods (this is often its parent class). This object should also be passed as an argument to its constructor.

See also

`uvm_tlm_fifo`; TLM-1 Ports, Exports and Imps

TLM-1 Ports, Exports and Imps

UVM *ports*, *exports* and *imps* are classes that are associated with one of the UVM TLM-1 interfaces.

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

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

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

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

Port and export binding is achieved by calling the port's/export's `connect` function. The `connect` function takes a single argument: the port or export that provides the required interface:

`p_or_e_requires.connect(p_or_e_provides)`. The order in which `connect` functions are called does not matter – bindings are resolved at the end of the `connect` phase.

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

The class name of a TLM port, export, or imp is based on the type of the TLM interface it implements; for example, the `uvm_blocking_put_port` can call the `put` task defined in `uvm_tlm_if_base`. Every TLM interface has a corresponding port, export, and imp. The complete set is listed below.

Blocking unidirectional interfaces

`uvm_blocking_put_port`
`uvm_blocking_get_port`
`uvm_blocking_peek_port`
`uvm_blocking_get_peek_port`

TLM-1 Ports, Exports and Imps

```
uvm_blocking_put_export  
uvm_blocking_get_export  
uvm_blocking_peek_export  
uvm_blocking_get_peek_export
```

```
uvm_blocking_put_imp  
uvm_blocking_get_imp  
uvm_blocking_peek_imp  
uvm_blocking_get_peek_imp
```

Non-blocking unidirectional interfaces

```
uvm_nonblocking_put_port  
uvm_nonblocking_get_port  
uvm_nonblocking_peek_port  
uvm_nonblocking_get_peek_port
```

```
uvm_nonblocking_put_export  
uvm_nonblocking_get_export  
uvm_nonblocking_peek_export  
uvm_nonblocking_get_peek_export
```

```
uvm_nonblocking_put_imp  
uvm_nonblocking_get_imp  
uvm_nonblocking_peek_imp  
uvm_nonblocking_get_peek_imp
```

Combined Interfaces

```
uvm_put_port  
uvm_get_port  
uvm_peek_port  
uvm_get_peek_port
```

```
uvm_put_export  
uvm_get_export  
uvm_peek_export  
uvm_get_peek_export
```

```
uvm_put_imp  
uvm_get_imp  
uvm_peek_imp  
uvm_get_peek_imp
```

Bidirectional Interfaces

```
uvm_blocking_master_port  
uvm_nonblocking_master_port  
uvm_master_port  
  
uvm_blocking_master_export  
uvm_nonblocking_master_export  
uvm_master_export  
  
uvm_blocking_master_imp  
uvm_nonblocking_master_imp  
uvm_master_imp  
  
uvm_blocking_slave_port  
uvm_nonblocking_slave_port  
uvm_slave_port  
  
uvm_blocking_slave_export  
uvm_nonblocking_slave_export  
uvm_slave_export  
  
uvm_blocking_slave_imp  
uvm_nonblocking_slave_imp  
uvm_slave_imp  
  
uvm_blocking_transport_port  
uvm_nonblocking_transport_port  
uvm_transport_port  
  
uvm_blocking_transport_export  
uvm_nonblocking_transport_export  
uvm_transport_export  
  
uvm_blocking_transport_imp  
uvm_nonblocking_transport_imp  
uvm_transport_imp
```

Sequencer interface

```
uvm_seq_item_pull_port  
uvm_seq_item_pull_export  
uvm_seq_item_pull_imp
```

TLM-1 Ports, Exports and Imps

Declarations

```
class uvm_put_port #( type T = int )
extends uvm_port_base #( uvm_tlm_if_base #(T,T) ) ;

class uvm_blocking_put_export #( type T = int )
extends uvm_port_base #( uvm_tlm_if_base #(T,T) ) ;

class uvm_master_port #( type REQ = int , type RSP = REQ )
extends uvm_port_base #( uvm_tlm_if_base #(REQ, RSP) ) ;

class uvm_put_imp #( type T = int , type IMP = int )
extends uvm_port_base #( uvm_tlm_if_base #(T,T) ) ;

class uvm_master_imp
#( type REQ = int , type RSP = REQ , type IMP = int ,
    type REQ_IMP = IMP , type RSP_IMP = IMP )
extends uvm_port_base #( uvm_tlm_if_base #(REQ, RSP) ) ;

class uvm_seq_item_pull_port #(type REQ=int, type RSP=REQ)
extends uvm_port_base #(sqr_if_base #(REQ, RSP)) ;
```

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

Common methods for TLM ports and exports

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

plus methods inherited from `uvm_port_base`

Common methods for unidirectional imp

function new (string name, IMP imp);	Constructor. <code>imp</code> is handle to object that implements interface methods.
--	--

Plus methods inherited from `uvm_port_base`

Common methods for bidirectional imps

```
function new(string name,
    IMP imp,
    REQ_IMP req_imp = null,
    RSP_IMP rsp_imp = null);
```

Constructor. `imp` is handle to object that implements interface methods

Plus methods inherited from uvm_port_base

Example

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

```
class compA extends uvm_component;
  uvm_blocking_put_port #(int) p0;
  function new(string name, uvm_component parent);
    super.new(name,parent);
  endfunction: new
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    p0 = new("p0",this,1,UVM_UNBOUNDED_CONNECTIONS);
  endfunction: build_phase
  task run_phase(uvm_phase phase);
    p0.debug_connected_to();
    for (int i=1; i<= p0.size(); i++) begin
      p0.put(i);
      p0.set_if(i);
    end
  endtask: run_phase
  `uvm_component_utils(compA)
endclass: compA
```

A component that provides the implementation of a single interface

```
class compB extends uvm_component;
  uvm_blocking_put_imp #(int,compB) put_export;
  function new(string name, uvm_component parent);
    super.new(name,parent);
  endfunction: new
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    put_export = new("put_export",this);
  endfunction: build_phase
  task put(int val); // interface method
    `uvm_info("compB",$sformatf("Received %0d",val),
              UVM_NONE)
  endtask: put
```

TLM-1 Ports, Exports and Imps

```
`uvm_component_utils(compB)
endclass: compB

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

class compC extends uvm_component;
    compA A;
    compB B00,B01;
    uvm_blocking_put_port #(int) put_port;
    function new(string name, uvm_component parent);
        super.new(name,parent);
    endfunction: new
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        put_port = new("put_port",this,
            1,UVM_UNBOUNDED_CONNECTIONS);
        A = compA::type_id::create("A", this);
        B00 = compB::type_id::create("B00",this);
        B01 = compB::type_id::create("B01",this);
    endfunction: build_phase
    function void connect_phase(uvm_phase phase);
        // The order here does not matter
        A.p0.connect(B00.put_export);
        A.p0.connect(B01.put_export);
        A.p0.connect(put_port);
    endfunction: connect_phase
`uvm_component_utils(compC)
endclass: compC
```

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

```
class sve extends uvm_env;
    compC C;
    compB ZB1,B2,B03;
    ...
    function void connect_phase(uvm_phase phase);
        C.put_port.connect(ZB1.put_export);
        C.put_port.connect(B03.put_export);
        C.put_port.connect(B2.put_export);
    endfunction: connect_phase
`uvm_component_utils(sve)
endclass: sve
```

Simulation output (note order of outputs):

```
UVM_INFO @ 0: svel.C.A.p0 [Connections Debug] has 5
interfaces from 3 places
```

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

Tips

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

Gotchas

- Remember that the export that provides the actual implementation of the interface methods should use `uvm_*_imp` rather than `uvm_*_export`.
- A `uvm_*_imp` instance requires a type parameter that gives the type of the class that defines its interface methods (this is often its parent class). This object should also be passed as an argument to its constructor.
- The order that interfaces are stored in a multi-port depends on their hierarchical names, not the order in which `connect` is called.
- The interface elements in a multi-port are accessed using index 1 to `size()`. Index 0 and index 1 return the same interface!

See also

`uvm_port_base`; `uvm_tlm_fifo`; TLM-1 Interfaces

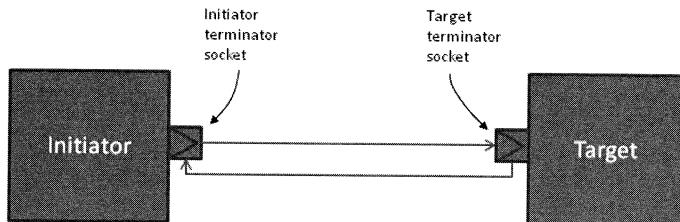
TLM-2.0

UVM's implementation of TLM-2.0 is loosely based on the original SystemC TLM-2.0 implementation.

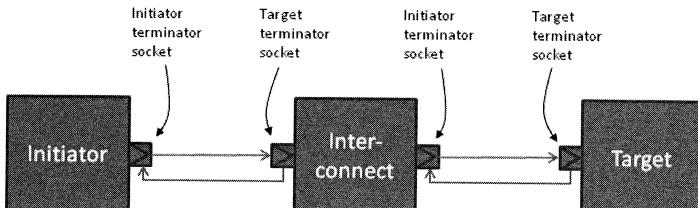
In TLM-1, the primary means of making a connection between two processes is through ports and exports, whereas in TLM-2.0, this is done through *sockets*. A socket is like a port or export; in fact, it is derived from the same base class as ports and exports, namely `uvm_port_base`. However, unlike a 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.

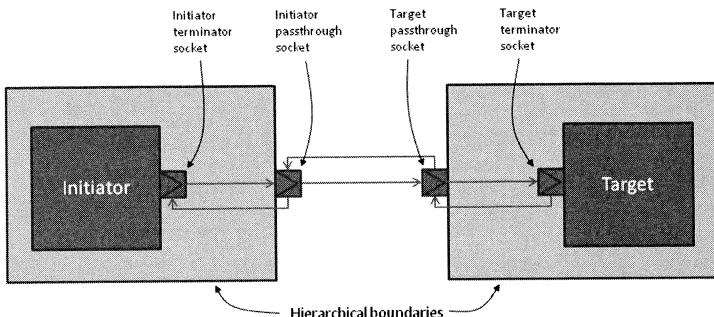
Initiator sockets can only connect to target sockets; target sockets can only connect to initiator sockets.



Interconnect components have both target and initiator sockets.



Sockets may be *terminators* or *passthroughs*. Passthrough sockets are used to enable connections across hierarchical boundaries. There can be an arbitrary number of passthrough sockets in the path between the initiator and target.



Furthermore, any particular socket implements either *blocking* interfaces or *non-blocking* interfaces. Thus there are eight different kinds of socket (see [TLM-2.0 Sockets](#)).

Blocking and non-blocking transport

The SystemVerilog implementation of TLM-2.0 provides only the basic transport interfaces, which are defined in the `uvm_tlm_if` class.

Blocking transport (`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.

For blocking interfaces, the `b_transport` task transports a transaction from the initiator to the target in a blocking fashion. The call to `b_transport` by the initiator marks the first timing point in the execution of the transaction. The return from `b_transport` by the target marks the final timing point in the execution of the transaction.

For non-blocking interfaces, `nb_transport_fw` transports a transaction in the forward direction, that is from the initiator to the target (the forward path).

`nb_transport_bw` does the reverse, it transports a transaction from the target back to the initiator (the backward path). An initiator and target will use the forward and backward paths to update each other on the progress of the transaction execution. Typically, `nb_transport_fw` is called by the initiator whenever the protocol state machine in the initiator changes state, and `nb_transport_bw` is called by the target whenever the protocol state machine in the target changes state.

Use Models

Since sockets are derived from `uvm_port_base`, they are created and connected in the same way as TLM-1 port and exports. You can create them in the build phase and connect them in the connect phase by calling `connect_phase()`.

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. Non-blocking initiator sockets must be bound to an implementation of `nb_transport_bw` and non-blocking target sockets must be bound to an implementation of `nb_transport_fw`. Typically, the task or function is implemented in the component where the socket is instantiated and the component type and instance are provided to complete the binding.

Generic Payload

TLM-2.0 defines a base object, called the generic payload, for moving data between components. In SystemVerilog, `uvm_tlm_generic_payload` is the default transaction type, but other transaction types could be used. Users may extend `uvm_tlm_generic_payload`, for example, to add constraints.

Generic Payload Extension Mechanism

Extensions are used to attach additional application-specific or bus-specific information to the generic bus transaction described in the generic payload.

An extension is an instance of a user-defined container class based on the `uvm_tlm_extension` class. The set of extensions for any particular generic payload object are stored in that generic payload object instance. To add an extension to a generic payload object, allocate an instance of the extension container class and attach it to the generic payload object using the `set_extension()` method:

```
gp_Xs_ext Xs = new();  
gp.set_extension(Xs);
```

Various methods exist for managing extensions.

Example

This example shows how to use a TLM-2.0 socket interface between two components.

```
// Initiator component  
class initiator extends uvm_component;  
  
  // Blocking initiator socket  
  uvm_tlm_b_initiator_socket#(my_seq_item) tlm2_sock;  
  
  `uvm_component_utils(initiator)  
  
  function new(string name, uvm_component parent);  
    super.new(name, parent);  
    // Create the socket
```

```

    tlm2_sock = new("tlm2_sock", this);
endfunction : new

// During simulation, use the socket to pass transactions to the target
virtual task run_phase(uvm_phase phase);
    my_seq_item rw;
    uvm_tlm_time delay = new;

    phase.raise_objection(this);

    rw = my_seq_item::type_id::create("rw",
                                      get_full_name());
    ...
    // Pass a transaction through the socket
    tlm2_sock.b_transport(rw, delay);
    ...

    phase.drop_objection(this);
endtask : run_phase

endclass : initiator

// Target component
class target extends uvm_component;
    uvm_tlm_b_target_socket #(target, my_seq_item) tlm2_sock;
    `uvm_component_utils(target)

    function new(string name = "target",
                uvm_component parent = null);
        super.new(name, parent);
        tlm2_sock = new("tlm2_sock", this);
    endfunction : new

    // Implementation of the blocking transport method. the initiator calls this
    // through the socket
    task b_transport(my_seq_item rw, uvm_tlm_time delay);
        // Process the transaction
        // ...
    endtask : b_transport

endclass : target

```

```
// Environment that instantiates the initiator and target
class tb_env extends uvm_component;

  `uvm_component_utils(tb_env)

  initiator m_initiator;
  target    m_target;

  // Construct the environment
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction : new

  // Build the initiator and target components
  function void build_phase(uvm_phase phase);
    m_initiator = initiator::type_id::create("m_initiator",
                                              this);
    m_target   = target::type_id::create("m_target", this);
  endfunction : build_phase

  // Connect the target and the initiator sockets
  function void connect_phase(uvm_phase phase);
    m_initiator.tlm2_sock.connect(m_target.tlm2_sock);
  endfunction : connect_phase

endclass : tb_env
```

See also

[TLM-2.0 Sockets; TLM-2.0 Ports, Exports and Imps](#)

TLM-2.0 Ports, Exports and Imps

In TLM-2.0 interfaces, ports, exports and imps are used as the basis for sockets (See [TLM-2.0 Sockets](#)).

The SystemC TLM-2.0 standard recommends the use of sockets for maximal interoperability, convenience, and a consistent coding style. The use of ports and exports directly with the TLM-2.0 core interfaces is not recommended.

uvm_tlm_b_transport_port	Blocking transport port.
uvm_tlm_nb_transport_fw_port	Non-blocking forward transport port.
uvm_tlm_nb_transport_bw_port	Non-blocking backward transport port.
uvm_tlm_b_transport_export	Blocking transport export.
uvm_tlm_nb_transport_fw_export	Non-blocking forward transport export.
uvm_tlm_nb_transport_bw_export	Non-blocking backward transport export.
uvm_tlm_b_transport_imp	Blocking transport implementation.
uvm_tlm_nb_transport_fw_imp	Non-blocking forward transport implementation.
uvm_tlm_nb_transport_bw_imp	Non-blocking backward transport implementation.

Declaration

```
class uvm_tlm_b_transport_port
  #(type T = uvm_tlm_generic_payload)
  extends uvm_port_base #(uvm_tlm_if #(T));

class uvm_tlm_nb_transport_fw_port
  #(type T = uvm_tlm_generic_payload,
    type P = uvm_tlm_phase_e)
  extends uvm_port_base #(uvm_tlm_if #(T,P));

class uvm_tlm_nb_transport_bw_port
  #(type T = uvm_tlm_generic_payload,
    type P = uvm_tlm_phase_e)
  extends uvm_port_base #(uvm_tlm_if #(T,P));

class uvm_tlm_b_transport_export
  #(type T = uvm_tlm_generic_payload)
  extends uvm_port_base #(uvm_tlm_if #(T));

class uvm_tlm_nb_transport_fw_export
  #(type T = uvm_tlm_generic_payload,
```

TLM-2.0 Ports, Exports and Imps

```
type P = uvm_tlm_phase_e)
extends uvm_port_base #(uvm_tlm_if #(T,P));
class uvm_tlm_nb_transport_bw_export
#(type T = uvm_tlm_generic_payload,
  type P = uvm_tlm_phase_e)
extends uvm_port_base #(uvm_tlm_if #(T,P));
class uvm_tlm_b_transport_imp
#(type T = uvm_tlm_generic_payload,
  type IMP = int)
extends uvm_port_base #(uvm_tlm_if #(T));
class uvm_tlm_nb_transport_fw_imp
#(type T = uvm_tlm_generic_payload,
  type P = uvm_tlm_phase_e,
  type IMP = int)
extends uvm_port_base #(uvm_tlm_if #(T,P));
class uvm_tlm_nb_transport_bw_imp
#(type T = uvm_tlm_generic_payload,
  type P = uvm_tlm_phase_e,
  type IMP = int)
extends uvm_port_base #(uvm_tlm_if #(T,P));
```

Example

This example shows a TLM-2.0 socket, which uses the port and imp classes.

```
class master extends uvm_component;
  // Non-blocking initiator socket – is a forward port and has a backward imp
  uvm_tlm_nb_initiator_socket
    #(trans, uvm_tlm_phase_e, this_t) initiator_socket;
  function void build_phase(uvm_phase phase);
    // Construct the socket, hence the port and imp
    initiator_socket = new("initiator_socket", this, this);
  endfunction
  // Required implementation
  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
```

See also

[TLM-2.0 Sockets](#)

TLM-2.0 Sockets

Sockets provide interconnections in TLM-2.0 models. Each `uvm_tlm_*_socket` class is derived from a corresponding `uvm_tlm_*_socket_base` class. The base class contains most of the implementation of the class, the derived class contains the connection semantics.

Sockets come in several flavors: each socket is either an initiator or a target, a passthrough or a terminator. Further, any particular socket implements either the blocking interfaces or the non-blocking interfaces. Terminator sockets are used on initiators and targets as well as interconnect components (See [TLM-2.0](#)).

Passthrough sockets are used to enable connections to cross hierarchical boundaries.

There are eight socket types: the cross of blocking and non-blocking, passthrough and termination, target and initiator.

Sockets are specified based on what they are (IS-A, that is an inheritance relationship) and what they contain (HAS-A).

Declaration

```
class uvm_tlm_b_initiator_socket
  #(type T = uvm_tlm_generic_payload)
  extends uvm_tlm_b_initiator_socket_base #(T);

class uvm_tlm_b_target_socket
  #(type IMP = int,
    type T = uvm_tlm_generic_payload)
  extends uvm_tlm_b_target_socket_base #(T);

class uvm_tlm_nb_initiator_socket
  #(type IMP = int,
    type T = uvm_tlm_generic_payload,
    type P = uvm_tlm_phase_e)
  extends uvm_tlm_nb_initiator_socket_base #(T,P);

class uvm_tlm_nb_target_socket
  #(type IMP = int,
    type T = uvm_tlm_generic_payload,
    type P = uvm_tlm_phase_e)
  extends uvm_tlm_nb_target_socket_base #(T,P);

class uvm_tlm_b_passthrough_initiator_socket
  #(type T = uvm_tlm_generic_payload)
  extends uvm_tlm_b_passthrough_initiator_socket_base #(T);
```

```

class uvm_tlm_b_passthrough_target_socket
  #(type T = uvm_tlm_generic_payload)
  extends uvm_tlm_b_passthrough_target_socket_base #(T);

class uvm_tlm_nb_passthrough_initiator_socket
  #(type T = uvm_tlm_generic_payload,
    type P = uvm_tlm_phase_e)
  extends
    uvm_tlm_nb_passthrough_initiator_socket_base #(T,P);

class uvm_tlm_nb_passthrough_target_socket
  #(type T = uvm_tlm_generic_payload,
    type P = uvm_tlm_phase_e)
  extends uvm_tlm_nb_passthrough_target_socket_base #(T,P);

```

Methods

<pre> function new(string name, uvm_component parent); </pre>	Constructor, except for <u>vdm_tlm_b_target_socket,</u> <u>vdm_tlm_nb_initiator_socket</u> and <u>vdm_tlm_nb_target_socket</u>
<pre> function new(string name, uvm_component parent, IMP imp = null); </pre>	Constructor for <u>vdm_tlm_b_target_socket</u> <u>vdm_tlm_nb_initiator_socket</u> <u>vdm_tlm_nb_target_socket</u> <i>imp</i> is a reference to the class implementing the transport method. If not specified, it is assumed to be the same as parent.
<pre> function void connect(this_type provider); </pre>	Connect this socket to the specified socket.

Rules

- 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.
- There can be an arbitrary number of passthrough sockets in the path between an 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()`. Non-blocking initiator sockets must be bound to an implementation of `nb_transport_bw()` and non-

TLM-2.0 Sockets

blocking target sockets must be bound to an implementation of `nb_transport_fw()`.

Example

See **TLM-2.0**, **TLM-2.0 Ports, Exports and Imps** and **uvm_tlm_if** for examples.

Tips

Create sockets in the build phase by calling `new()` and connect them in the connect phase by calling `connect_phase()`.

See also

TLM-2.0; TLM-2.0 Ports, Exports and Imps; uvm_tlm_if

uvm_tlm_analysis_fifo

Class `uvm_tlm_analysis_fifo` is provided for use as part of an analysis component that expects to receive its data from an analysis port. It is derived from `uvm_tlm_fifo` and adds a `write` member function. Only the methods directly relevant to analysis FIFOs are described here; for full details, consult the article on [uvm_tlm_fifo](#).

The “put” end of an analysis FIFO is intended to be written-to by an analysis port. Consequently, an analysis FIFO is invariably set up to have unbounded depth so that it can never block on `write`. The FIFO’s `try_put` method is re-packaged as a `write` function, which is then exposed through an `analysis_export` that can in its turn be connected to an `analysis_port` on a producer component. The `get_export` (or similar) at the other end of the FIFO is typically connected to an analysis component, such as a scoreboard, that may need to wait for other data before it is able to process the FIFO’s output.

The type of data carried by the analysis FIFO is set by a type parameter.

Declaration

```
class uvm_tlm_analysis_fifo #(type T = int)
  extends uvm_tlm_fifo #(T);
```

Methods

<code>function new(string name, uvm_component parent = null);</code>	Constructor.
<code>function void write(input T t);</code>	Puts transaction on to the FIFO; cannot block (FIFO is unbounded).
<code>function void flush();</code>	Clears FIFO contents.
<code>task get(output T t);</code>	Blocks if the FIFO is empty.
<code>function bit try_get(output T t);</code>	Returns 0 if FIFO empty.
<code>function bit try_peek(output T t);</code>	Returns 0 if FIFO empty.
<code>function bit try_put(input T t);</code>	Returns 0 if FIFO full.
<code>function int used();</code>	Number of items in FIFO.
<code>function bit is_empty();</code>	Returns 1 if FIFO empty.

Members

<code>uvm_analysis_imp #(T, uvm_tlm_analysis_fifo #(T)) analysis_export;</code>	For connection to an analysis port on another component.
<code>blocking_put_export[†]; nonblocking_put_export[†]; put_export[†];</code>	Exports blocking/non-blocking/combined put interface.
<code>blocking_get_export[†]; nonblocking_get_export[†]; get_export[†];</code>	Exports blocking/non-blocking/combined get interface.
<code>blocking_peek_export[†]; nonblocking_peek_export[†]; peek_export[†];</code>	Exports blocking/non-blocking/combined peek interface.
<code>blocking_get_peek_export[†]; nonblocking_get_peek_export[†]; get_peek_export[†];</code>	Exports blocking/non-blocking/combined get_peek interface.
<code>uvm_analysis_port #(T) put_ap; uvm_analysis_port #(T) get_ap;</code>	Analysis ports.

[†]Export type omitted for clarity. `*_export` is an implementation of the corresponding `uvm_tlm_*_if` interface (see the article on [TLM-1 Interfaces](#))

Example

Declaring and connecting an analysis FIFO so that the monitor can write to it, and the analyzer can get from it:

```
class example_env extends uvm_env;
  uvm_tlm_analysis_fifo #(example_transaction) an_fifo;
  example_monitor m_monitor;      // has analysis port
  example_analyzer m_analyzer;   // has "get" port
  ...
  virtual function void build_phase(uvm_phase phase);
    m_monitor     = ...; // Create
    m_analyzer   = ...; // Create
    an_fifo       = new("an_fifo", this);
  endfunction
  virtual function void connect_phase(uvm_phase phase);
    m_monitor.monitor_ap.connect(an_fifo.analysis_export);
    m_analyzer.get_port.connect(an_fifo.get_export);
  endfunction
  ...
endclass
```

Tips

It is not necessary to use a `uvm_tlm_analysis_fifo` to pass data to an analysis component that does not consume time, such as a coverage checker or logger. In such a situation, it is preferable to derive the analysis component from `uvm_subscriber` so that it can provide its own `write` method directly. Use analysis FIFOs to buffer analysis input to a component such as a scoreboard that may not be able to service each analysis transaction immediately because it is waiting for a corresponding transaction on some other channel.

The `size()` and `is_full()` methods are inherited from `uvm_tlm_fifo`, and will both return 0.

Gotchas

If the type parameter of a `uvm_tlm_analysis_fifo` is a class, the FIFO stores object handles. If an object is updated after it has been put into a FIFO but before it is retrieved, `peek` and `get` will return the updated object. If the FIFO is required to transport an object with its state preserved, the object should first be “cloned” and the clone written to the FIFO instead.

Do not use the `put` or `try_put` methods inherited from `uvm_tlm_fifo` to write data to an analysis FIFO (they are not members of `uvm_analysis_port`).

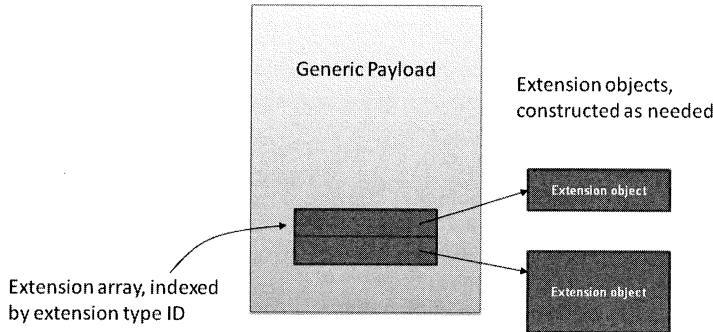
See also

`uvm_tlm_fifo`; TLM-1 Interfaces; `uvm_subscriber`; `uvm_analysis_port`;
`uvm_analysis_export`

uvm_tlm_extension

Extension class for the Generic Payload. The class is parameterized with an arbitrary type, which represents the type of the extension.

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.



Declaration

```
virtual class uvm_tlm_extension_base  
  extends uvm_object;  
class uvm_tlm_extension #(type T = int)  
  extends uvm_tlm_extension_base;
```

Methods

function new (string name = "");	Constructor.
virtual function uvm_tlm_extension_base get_type_handle ();	An interface to polymorphically retrieve a handle that uniquely identifies the type of the sub- class.
virtual function string get_type_handle_name ();	An interface to polymorphically retrieve the name that uniquely identifies the type of the sub-class.
static function this_type_ID();	Return the unique ID of this TLM extension type.

Rules

An instance of the generic payload can contain one extension object of each type; it cannot contain two instances of the same extension type.

Example

```
// Extension class
class my_extension
    extends uvm_tlm_extension#(my_extension);
    `uvm_object_utils(my_extension)
    int when;
endclass

// Add an extension to a payload
uvm_tlm_gp gp = new;
my_extension ext = new;
ext.when= $time;
gp.set_extension( ext );

// Retrieve the extension value
my_extension ext;
if ( $cast( ext, gp.get_extension( my_extension::ID() ) )
    // Extension present
    ...
else
    `uvm_error( ... )
```

Tips

- The extension type can be identified using the `ID()` method. This can be used to retrieve an extension from a `uvm_tlm_generic_payload` instance, using the `uvm_tlm_generic_payload::get_extension()` method.
- When extending class `uvm_tlm_extension` to create a user-defined extension class, always specialize `uvm_tlm_extension` with the user-defined type being defined, as shown in the example above

See also

[uvm_tlm_generic_payload](#)

uvm_tlm_fifo

Class `uvm_tlm_fifo` is provided for use as a standard channel. Data is written and read in first-in first-out order. The FIFO depth may be set by a constructor argument – the default depth is 1.

The `uvm_tlm_fifo` channel has both blocking and non-blocking `put` and `get` methods. A `put` operation adds one data item to the front of the FIFO. A `get` operation retrieves and removes the last data item from the FIFO. Blocking and non-blocking peek methods are also provided that retrieve the last data item without removing it from the FIFO. Exports are provided to access the channel methods. Analysis ports enable the data associated with each put or get operation to be monitored.

The type of data carried is set by a type parameter.

Declaration

```
class uvm_tlm_fifo #(type T = int)
  extends uvm_tlm_fifo_base #(T);
```

Methods

<code>function new(string name, uvm_component parent = null, int size_ = 1);</code>	Constructor.
<code>function bit can_get();</code>	Returns 1 if there is data available for reading from the FIFO. Returns 0 if the FIFO is empty.
<code>function bit can_peek();</code>	Returns 1 if there is data available in the FIFO for peeking at . Returns 0 if the FIFO is empty.
<code>function bit can_put();</code>	Returns 1 if there is space available in the FIFO to accept new data. Returns 0 if the FIFO is full.
<code>function void flush();</code>	Clears FIFO contents.
<code>task get(output T t);</code>	Blocks if the FIFO is empty. Once there is data available in the FIFO, removes the next item and assigns t.to it

task peek (output T t);	Blocks if the FIFO is empty. Once there is data available, reads the next item without removing it from the FIFO, and assigns t to it
task put (input T t);	Blocks if the FIFO is full. Once space is available, writes the data t to the FIFO.
function bit try_get (output T t);	If there is data available in the FIFO, removes the next item from the FIFO, assigns t to it and returns 1. Returns 0 immediately if the FIFO is empty.
function bit try_peek (output T t);	Returns 0 if the FIFO is full. If there is data available in the FIFO, reads the next item of data without removing it, assigns t to it and returns 1. Returns 0 immediately if the FIFO is empty.
function bit try_put (input T t);	If there is space available in the FIFO, writes data t to it and returns 1. Returns 0 immediately if the FIFO is full.
function int size ();	Returns the FIFO depth.
function int used ();	Returns the number of items in the FIFO.
function bit is_empty ();	Returns 1 if the FIFO is empty.
function bit is_full ();	Returns 1 if the FIFO is full.

Members

blocking_put_export [†] nonblocking_put_export [†] put_export [†]	Exports blocking/non-blocking/combined put interface.
---	---

<code>blocking_get_export[†]</code> <code>nonblocking_get_export[†]</code> <code>get_export[†]</code>	Exports blocking/non-blocking/combined get interface.
<code>blocking_peek_export[†]</code> <code>nonblocking_peek_export[†]</code> <code>peek_export[†]</code>	Exports blocking/non-blocking/combined peek interface.
<code>blocking_get_peek_export[†]</code> <code>nonblocking_get_peek_export[†]</code> <code>get_peek_export[†]</code>	Exports blocking/non-blocking/combined get_peek interface.
<code>uvm_analysis_port #(T) put_ap</code> <code>uvm_analysis_port #(T) get_ap</code>	Analysis ports.

[†]Export type omitted for clarity. `*_export` is an implementation of the corresponding `uvm_tlm_*_if` interface (see the article on [TLM-1 Interfaces](#)).

Example

Declaring port to connect to `uvm_tlm_fifo`:

```
uvm_get_port #(basic_transaction) m_trans_in;
```

Calling `uvm_tlm_fifo` method via port:

```
m_trans_in.get(tx);
```

Using `uvm_tlm_fifo` between `uvm_random_stimulus` and a driver:

```
class verif_env extends uvm_env;
  uvm_random_stimulus #(basic_transaction) m_stimulus;
  dut_driver m_driver;
  uvm_tlm_fifo #(basic_transaction) m_fifo;
  ...
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    m_stimulus = ... // Create
    m_fifo = ... // Create
    m_driver =
      dut_driver::type_id::create("m_driver",this);
  endfunction: build_phase

  virtual function void connect_phase(uvm_phase phase);
```

```
m_stimulus.blocking_put_port.connect(m_fifo.put_export);
m_driver.m_trans_in.connect(m_fifo.get_export);
endfunction: connect_phase
...
endclass: verif_env
```

Tips

- The `uvm_tlm_fifo` uses a SystemVerilog mailbox class as its internal buffer so an infinite depth FIFO can be created by setting the size constructor argument to 0.
- In general you should make TLM connections between components by connecting a port on one component directly to an export on another component rather than connecting the two components through an intermediate FIFO. Use a FIFO only where you need to buffer transactions.

Gotchas

If the type parameter of a `uvm_tlm_fifo` is a class, the FIFO stores object handles. If an object is updated after it has been put into a FIFO but before it is retrieved, peek and get will return the updated object. If the FIFO is required to transport an object with its state preserved, the object should first be copied and the copy put into the FIFO instead of the original.

See also

[uvm_tlm_analysis_fifo; TLM-1 Interfaces](#)

uvm_tlm_generic_payload

The SystemC TLM-2.0 standard defines a class, the generic payload, that represents a generic bus transaction with command, address, data, and other attributes. A generic payload transaction can represent either a read or a write operation over a memory-mapped bus. The UVM equivalent, `uvm_tlm_generic_payload`, is the default transaction type for TLM interfaces in UVM. Although other transaction types can be used, the TLM-2.0 standard strongly recommends either using the generic payload off-the-shelf or extending the generic payload using the built-in extension mechanism (see `uvm_tlm_extension`).

Unlike the native SystemC TLM-2.0 standard, UVM explicitly permits the `uvm_tlm_generic_payload` class to be extended in order to add constraints.

Declaration

Globals

```
typedef enum
{
    UVM_TLM_READ_COMMAND,
    UVM_TLM_WRITE_COMMAND,
    UVM_TLM_IGNORE_COMMAND
} uvm_tlm_command_e;

typedef enum
{
    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_response_status_e;
```

Classes

```
class uvm_tlm_generic_payload extends uvm_sequence_item;
typedef uvm_tlm_generic_payload uvm_tlm_gp;
```

Methods

<code>function new(string name = "");</code>	Constructor.
<code>virtual function uvm_tlm_command_e get_command();</code>	Accessor.

virtual function void set_command (uvm_tlm_command_e command);	Accessor.
virtual function bit is_read ();	Accessor.
virtual function void set_read ();	Accessor.
virtual function bit is_write ();	Accessor.
virtual function void set_write ();	Accessor.
virtual function void set_address (bit [63:0] addr);	Accessor.
virtual function bit [63:0] get_address ();	Accessor.
virtual function void get_data (output byte unsigned p []);	Accessor.
virtual function void set_data (ref byte unsigned p []);	Accessor.
virtual function int unsigned get_data_length ();	Accessor.
virtual function void set_data_length (int unsigned length);	Accessor.
virtual function int unsigned get_streaming_width ();	Accessor.
virtual function void set_streaming_width (int unsigned width);	Accessor.
virtual function void get_byte_enable (output byte unsigned p[]);	Accessor.
virtual function void set_byte_enable (ref byte unsigned p[]);	Accessor.
virtual function int unsigned get_byte_enable_length ();	Accessor.
virtual function void set_byte_enable_length (int unsigned length);	Accessor.
virtual function void set_dmi_allowed (bit dmi);	Accessor.
virtual function bit is_dmi_allowed ();	Accessor.

uvm_tlm_generic_payload

virtual function uvm_tlm_response_status_e get_response_status() ;	Accessor.
virtual function void set_response_status(uvm_tlm_response_status_e status);	Accessor.
virtual function bit is_response_ok() ;	Accessor.
virtual function bit is_response_error() ;	Accessor.
virtual function string get_response_string() ;	Accessor.
function uvm_tlm_extension_base set_extension(uvm_tlm_extension_base ext);	Accessor.
function int get_num_extensions() ;	Accessor.
function uvm_tlm_extension_base get_extension(uvm_tlm_extension_base ext_handle);	Accessor.
function void clear_extension(uvm_tlm_extension_base ext_handle);	Accessor.
function void clear_extensions() ;	Accessor.

Rules

The members are protected and should only be accessed using the accessor methods listed above.

Example

```
// Create a generic payload transaction using the factory:  
uvm_tlm_gptx = uvm_tlm_gp::type_id::create("tx");  
  
// Randomize the transaction  
assert( randomized_tx.randomize() with {  
    m_address >= 0 && m_address < 256;  
    m_length == 4 || m_length == 8;  
    m_data.size == m_length;  
    m_byte_enable_length <= m_length;
```

```
(m_byte_enable_length % 4) == 0;
m_byte_enable.size == m_byte_enable_length;
m_streaming_width == m_length;
m_response_status == UVM_TLM_INCOMPLETE_RESPONSE;
} )
else
`uvm_error("Initiator", "tx.randomize() failed")

// Create a copy of the transaction
tx.copy(randomized_tx);
// Use the blocking transport, which is implemented in the target
initiator_socket.b_transport(tx, delay);
// Was the bus transaction successful?
assert( tx.is_response_ok() );
```

Tips

The primary purpose of having the TLM-2.0 interfaces and the generic payload in UVM is for communication with SystemC models that run alongside the SystemVerilog verification environment.

To fully understand the intended use of the generic payload and other features of TLM-2.0, refer to the IEEE 1666 SystemC standard.

See also

TLM-2.0; TLM-2.0 Sockets; uvm_tlm_extension

uvm_tlm_if

The UVM TLM-2.0 subset provides the following two transport interfaces:
blocking (`b_transport`) completes the entire transaction within a single method call and *non-blocking* (`nb_transport`) describes the progress of a transaction using multiple `nb_transport()` method calls going back and forth between initiator and target.

`uvm_tlm_if` is the base class type to define the transport functions, `nb_transport_fw`, `nb_transport_bw` and `b_transport`.

For blocking transport, a call to `b_transport()` indicates start-of-life of a transaction, and return from `b_transport()` end-of-life of the transaction.

For non-blocking transport, significant timing points during the lifetime of a transaction (for example, the start-of response-phase) are indicated by calling `nb_transport()` in either the 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. 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 it (for simulation speed).

Declaration

Globals

```
typedef enum
{
    UNINITIALIZED_PHASE,
    BEGIN_REQ,
    END_REQ,
    BEGIN_RESP,
    END_RESP
} uvm_tlm_phase_e;
```

```
typedef enum
```

```
{  
    UVM_TLM_ACCEPTED,  
    UVM_TLM_UPDATED,  
    UVM_TLM_COMPLETED  
} uvm_tlm_sync_e;
```

Macros

```
`define UVM_TLM_TASK_ERROR \  
    "TLM-2 interface task not implemented"  
  
'define UVM_TLM_FUNCTION_ERROR \  
    "TLM-2 interface function not implemented"
```

Class

```
class uvm_tlm_if #(type T = uvm_tlm_generic_payload,  
                    type P = uvm_tlm_phase_e );
```

Methods

virtual function uvm_tlm_sync_e nb_transport_fw (T t, ref P p, input uvm_tlm_time delay);	Non-blocking forward transport.
virtual function uvm_tlm_sync_e nb_transport_bw (T t, ref P p, input uvm_tlm_time delay);	Non-blocking backward transport.
virtual task b_transport (T t, uvm_tlm_time delay);	Blocking transport.

Rules

- Each of the interface methods takes a handle to the transaction to be transported and a reference argument for the delay. In addition, the non-blocking interfaces take a reference argument for the phase.
- Any component that has either a blocking termination socket, a non-blocking initiator socket, or a non-blocking termination socket must provide implementations of the relevant functions. This includes initiator and target components as well as interconnect components that have these kinds of sockets. (Components with passthrough sockets do not need to provide implementations of any sort).

Example

Example of a target component with a blocking transport socket. There is an example of a non-blocking transport in **TLM-2.0 Ports, Exports and Imps**.

```
class Target extends uvm_component;
  `uvm_component_utils(Target)

  local byte unsigned m_data[4];

  // Declaration of target socket
  uvm_tlm_b_target_socket #(Target, my_transaction)
    t_socket;

  function new ...
    super.new("target", this);

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    t_socket = new( "t_socket", this, this );
  endfunction : build_phase

  // Called by initiator through the socket
  task b_transport(my_transaction t, uvm_tlm_time delay);
    if ( t.get_address() == 64'HC000)
      if ( t.get_command() == UVM_TLM_READ_COMMAND )
        t.set_data(m_data);
      else
        t.get_data(m_data);
    #(delay.get_realtime(1ns));
  endtask : b_transport

endclass : Target
```

Tips

In general, any component might modify a transaction object during its lifetime (subject to the rules of the protocol).

See also

[TLM-2.0; TLM-2.0 Ports, Exports and Imps; TLM-2.0 Sockets](#)

`uvm_tlm_time` is used to represent time values in a canonical form that can bridge initiators and targets located in different timescales and time precisions.

Integers are not sufficient, on their own, to represent time without any ambiguity: you need to know the scale of that integer value. The scale is information conveyed outside of that integer. In SystemVerilog, it is based on the timescale that was active when the code was compiled. SystemVerilog properly scales time literals, but not integer values that happen to be used to store times. That is because it does not know the difference between an integer that carries an integer value and an integer that carries a time value. The 'time' variables are simply 64-bit integers, they are not scaled back and forth to the underlying precision.

Declaration

```
class uvm_tlm_time;
```

Methods

<code>function new(string name = "uvm_tlm_time", real res = 0);</code>	Constructor.
<code>static function void set_time_resolution(real res);</code>	Set default resolution. By default, the default resolution is 1.0e-12 (ps)
<code>function string get_name();</code>	Return name.
<code>function void set_abstime(real t, real secs);</code>	Set time to <code>t</code> in units of secs.
<code>function real get_abstime(real secs);</code>	Return time, scaled to secs.
<code>function void reset();</code>	Reset time to 0.
<code>function real get_realtime(time scaled, real secs = 1.0e-9);</code>	Return time using current timescale. Set <code>scaled</code> to 1ns (see Tips & Examples).
<code>function void incr(real t, time scaled, real secs = 1.0e-9);</code>	Increment by <code>t</code> in the current timescale. Set <code>scaled</code> to 1ns (see Tips & Examples).
<code>function void decr(real t, time scaled, real secs);</code>	Decrement by <code>t</code> in the current timescale. Set <code>scaled</code> to 1ns (see Tips & Examples).

uvm_tlm_time

Rules

Resolution, although a `real` number, must be a power of 10; for example, `1.0e-9` for nanoseconds.

Tips

For `get_realtime`, `incr`, and `decr`, the scaled argument should always be `1ns` and the `secs` argument left at its default value of `1e-9`. (The exception would be if the `timeunit` is greater than `1ns`). These arguments are used so that `uvm_tlm_time` can work out what the current timescale value is.

Example

```
'timescale 100ps/1ps
...
// Create a new uvm_tlm_time value
uvm_tlm_time delay = new("delay"); // Value will be 0ns
delay.set_abstime(10.5,1.0e-9);      // Set to 10.5ns
delay.incr(25,1ns);                // Increment by 25 x 100ps (timescale) = 2.5ns
delay.incr(25ns,1ns);              // Increment by 25ns

// Display value in ns and ps
`uvm_info("", $sformatf("delay is %.3f ns, %.3 ps",
                      delay.get_abstime(1.0e-9),
                      delay.get_abstime(1.0e-12)), UVM_NONE)
// Display value using the current timescale
`uvm_info("", $sformatf("delay is %.3f x100ps",
                      delay.get_realtime(1ns)), UVM_NONE)
```

Tips

When co-simulating with SystemC, it is recommended that default canonical time resolution be set to the SystemC time resolution.

Gotchas

- The constructor does not set the time value: use `set_abstime` or `incr`.
- Remember that time values provided as actual arguments to `uvm_tlm_time`'s methods use the timescale of the unit in which they are compiled.

See also

TLM-2.0

uvm_transaction

`uvm_transaction` is a virtual class that is used as the base class for transactions in a UVM environment. It is derived from `uvm_object`. It adds functions for managing transactions and hooks to support transaction recording.

`uvm_transaction` is the base class for `uvm_sequence_item`; you should derive all user-defined transaction classes from `uvm_sequence_item`, not `uvm_transaction`. The intended use of the methods provided by `uvm_transaction` is for a `uvm_driver` to call the `accept_tr`, `begin_tr`, and `end_tr` methods of `uvm_component`.

Transactions are often used as the stimulus in a UVM testbench. The name of the component that initiates a transaction may be recorded as a member field. Knowing where each transaction in a complex testbench originates from can be a useful debugging aid. The initiator name can be set as a constructor argument or by calling the `set_initiator` function. A `get_initiator` function is provided to retrieve the initiator name.

The start and end time of a transaction may be recorded and stored within the transaction using the functions `begin_tr` and `end_tr` respectively. By default, the time recorded is the current simulation time: if a different time is required, this can be specified as a function argument. Many transaction-level models support the concept of a transaction being accepted by a target some time after it has been sent. An `accept_tr` function is provided to indicate when this occurs. The start, end and acceptance of a transaction are notified by events named "begin_event", "end_event" and "accept" respectively. These events are contained within a pool of events managed by each transaction. The event pool can be accessed by calling the `get_event_pool` function. Callback functions are provided that correspond to the "begin_event", "end_event", and "accept" events. They are virtual functions that by default do nothing but may be overridden in a derived class.

A transaction recording interface is provided but is not implemented in UVM itself. This is intended to be implemented by UVM tools, where required.

Declaration

```
virtual class uvm_transaction extends uvm_object;
```

Methods

<code>function new(string name = "", uvm_component initiator = null);</code>	Constructor.
<code>function void set_initiator(uvm_component initiator);</code>	Sets initiator (component that creates transaction).
<code>function uvm_component get_initiator();</code>	Get initiator.

uvm_transaction

function void accept_tr (time accept_time = 0);	Accept transaction (triggers "accept" event).
function integer begin_tr (time begin_time = 0);	Indicate start of transaction (triggers "begin" event).
function integer begin_child_tr (time begin_time = 0, integer parent_handle = 0);	Indicate start of child transaction (triggers "begin" event).
function void end_tr (time end_time = 0, bit free_ _handle = 1);	Indicate end of transaction (triggers "end" event).
function integer get_tr_handle ();	Returns transaction handle.
function void enable_recording (string stream, uvm_recorder recorder = null);	Enable (start) recording to specified stream.
function void disable_recording ();	Disable (stop) recording.
function bit is_recording_enabled ();	Check if recording enabled.
function bit is_active ();	Returns 1 if transaction started but not yet ended, otherwise 0.
virtual protected function void do_accept_tr ();	User-defined callback function.
virtual protected function void do_begin_tr ();	User-defined callback function.
virtual protected function void do_end_tr ();	User-defined callback function.
function uvm_event_pool get_event_pool ();	Returns the local event pool.
function time get_begin_time ();	Return transaction begin time.
function time get_end_time ();	Return transaction end time.
function time get_accept_time ();	Returns time that transaction was accepted.
function void set_transaction_id (integer id);	Sets the transaction id, which is otherwise -1.
function integer get_transaction_id ();	Returns the transaction id.

Example

```
class basic_transaction extends uvm_sequence_item;
    rand bit[7:0] m_var1, m_var2;
    static int tx_count = 0;

    function new (string name = "",
                  uvm_component initiator=null);
        super.new(name,initiator);
        tx_count++;
    endfunction: new

    virtual protected function int do_begin_tr();
        `uvm_info("TRX",$sformatf(
            "Transaction %0d beginning",tx_count),
            UVM_NONE)
    endfunction: do_begin_tr

    virtual protected function void do_end_tr();
        `uvm_info("TRX",$sformatf(
            "Transaction %0d ended",tx_count),
            UVM_NONE)
    endfunction: do_end_tr

    `uvm_object_utils_begin(basic_transaction)
        `uvm_field_int(m_var1,UVM_ALL_ON + UVM_DEC)
        `uvm_field_int(m_var2,UVM_ALL_ON + UVM_DEC)
    `uvm_object_utils_end
endclass : basic_transaction
```

Generating constrained random transactions:

```
virtual task generate_stimulus(
    basic_transaction t = null, input int max_count = 30 );
    basic_transaction temp;
    uvm_event_pool tx_epool;
    uvm_event tx_end;
    if( t == null ) t = new("trans",this);
    for( int i = 0; (max_count == 0 || i < max_count-1);i++ )
    begin
        assert( t.randomize() );
        $cast( temp , t.clone() );
        `uvm_info("stimulus generation",
            temp.convert2string(), UVM_NONE )
```

uvm_transaction

```
tx_epool = temp.get_event_pool();
blocking_put_port.put( temp );
tx_end = tx_epool.get("end");
tx_end.wait_trigger();
end
endtask: generate_stimulus
```

Tips

- The functions to accept, begin and end transactions are optional – they are only really useful when transaction recording is active.
- It is best to implement the beginning and ending of transactions in a driver. To do this, call `uvm_component::begin_tr` and `uvm_component_end_tr`, which will in turn call the transaction recording methods of the transaction object.
- The transaction handle returned by `begin_tr` may be used to create child transactions, by passing the handle to `begin_child_tr`

Rules

Do not derive classes directly from `uvm_transaction`: use `uvm_sequence_item` as the base for user-defined transaction classes.

Gotchas

Each transaction object maintains its own event pool. If an initiator needs to wait for a transaction to be accepted/ended before continuing, it needs to save a copy of the transaction handle to access the associated events.

See also

`uvm_object`; `uvm_sequence_item`; `uvm_random_stimulus`; `uvm_component`

Utility Macros

UVM defines a set of utility macros for objects and components. These register a class with the UVM factory and define functions required by the factory. Two of these functions are visible to users:

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

Also, the following typedefs are used for factory automation:

```
typedef uvm_object_registry #(T) type_id;  
typedef uvm_component_registry #(T,S) type_id;
```

These macros also provide a wrapper around the *field automation macros*.

Macros

Macros used only with objects:

<code>`uvm_object_utils (T)</code>	Registers simple object T with factory and defines factory methods.
<code>`uvm_object_utils_begin (T)</code>	Registers simple object T with factory and defines factory methods. May be followed by list of field automation macros .
<code>`uvm_object_utils_end</code>	Terminates list of field automation macros.
<code>`uvm_object_param_utils (T)</code>	Registers parameterized object T with factory and defines factory methods.
<code>`uvm_object_param_utils_begin (T)</code>	Registers parameterized object T with factory and defines factory methods. May be followed by list of field automation macros.

Macros used only with components:

<code>`uvm_component_utils (T)</code>	Registers simple component T with factory and defines factory methods.
---------------------------------------	--

Utility Macros

`uvm_component_utils_begin(T)	Registers simple component T with factory and defines factory methods. May be followed by list of field automation macros.
`uvm_component_utils_end	Terminates list of field automation macros.
`uvm_component_param_utils(T)	Registers parameterized component T with factory and defines factory methods.
`uvm_component_param_utils_begin(T)	Registers parameterized component T with factory and defines factory methods. May be followed by list of field automation macros.

Macros that do not register objects with the factory and do not define factory methods (used to call field automation macros in abstract base classes or where the default factory methods are not suitable):

`uvm_field_utils_begin(T)	May be followed by list of field automation macros.
`uvm_field_utils_end	Terminates list of field automation macros.

Example

Registering a transaction with the factory and calling the field automation macros on its fields:

```
class basic_transaction extends uvm_sequence_item;
  rand bit[7:0] addr, data;
  ...
  `uvm_object_utils_begin(basic_transaction)
    `uvm_field_int(addr,UVM_ALL_ON)
    `uvm_field_int(data,UVM_ALL_ON)
  `uvm_object_utils_end
endclass : basic_transaction
```

Registering a driver component with the factory and calling the field automation macro for its virtual interface wrapped in a UVM object:

```
class dut_driver extends uvm_driver #(basic_transaction);
```

```
uvm_get_port #(basic_transaction) m_trans_in;
vif_wrapper vif;
...
`uvm_component_utils_begin(dut_driver)
`uvm_field_object(vif,UVM_ALL_ON)
`uvm_component_utils_end
endclass: dut_driver
```

Tips

- Call the `uvm_object_utils macro(s) at the end of the class definition. This will ensure any members that are referenced by the field automation macros will have been declared.
- Call the `uvm_component_utils macro(s) at the beginning of the class definition. This will ensure the `type_id` proxy class, `create` method, and `get_type_name` methods are defined in the proper order for use in other class methods.

Gotchas

- Do not use a semicolon after a macro or a compiler error may result.
- Make sure you call `uvm_object_utils[_begin/_end] with objects and `uvm_component[_begin/_end] with components.
- Classes that call `uvm_field_utils_* cannot be built by the factory unless they have `create` and `get_type_name` member functions.
- Parameterized classes must use the `uvm_object_param_utils* or `uvm_component_param_utils* versions. See **uvm_component** and **uvm_object**, for more details and examples.
- There are no macros called `uvm_object_param_utils_end or `uvm_component_param_utils_end. Use `uvm_object_utils_end or `uvm_component_utils_end.

See also

[uvm_object](#); [uvm_component](#); [uvm_factory](#); [Field Macros](#)

Virtual Sequences

A *virtual sequence* is a sequence whose purpose is to manage the behavior of sequences on other sequencers. Like ordinary sequences, a virtual sequence runs on a sequencer. However, the sequencer for a virtual sequence is not linked to a driver, and the virtual sequence does not itself have data items. A sequencer used in this way is commonly called a *virtual sequencer*. A virtual sequencer acts only upon sequences (not sequence items) and is used to coordinate the execution of sequences on one or more other sequencers. A (non-virtual) sequencer can only be associated with a single driver, which typically only controls one device interface. A virtual sequencer provides a mechanism to control the interaction of multiple device interfaces by managing the sequencers that generate their transactions.

For each sequencer that it controls, a virtual sequencer must have a variable holding a reference to that sequencer. It is the user's responsibility to populate these variables with references to the appropriate subsequencer instances; this should be done as part of the `connect_phase` method of the enclosing component or environment. Note that a virtual sequencer does *not* use TLM interfaces to control its subsequencers.

Note that no `uvm_virtual_sequence` or `uvm_virtual_sequencer` classes exist; rather, both sequencers and virtual sequencers use sequences derived from the `uvm_sequence` class.

Creating a new virtual sequence is similar to creating any ordinary sequence. However, virtual sequences invoke an alternative set of sequence action macros: ``uvm_do_on` and ``uvm_do_on_with`. These macros require an additional argument to specify the sequencer instance that should execute the sequence.

Declarations

```
class uvm_sequencer #(type REQ = uvm_sequence_item,
    RSP = REQ) extends uvm_sequencer_param_base #(REQ, RSP);
virtual class uvm_sequence #(type REQ = uvm_sequence_item,
    type RSP = REQ) extends uvm_sequence_base;
```

Virtual sequences and sequencers use the same base classes as ordinary sequences and sequencers.

Example

```
// Create a virtual sequencer
class my_virtual_sequencer extends uvm_sequencer;
    // Variables to reference the sequencers we will control
    uvm_sequencer #(seq_item) m_write_sqr;
    uvm_sequencer #(seq_item) m_read_sqr;

    // Register this sequencer with the factory
```

```

`uvm_component_utils ( my_virtual_sequencer )

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

// Create a virtual sequence
class read_write_seq extends uvm_sequence;
    my_read_seq    read_seq; // Sequences on sequencers
    my_write_seq   write_seq;

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

    // Define the virtual sequence functionality.
    // Note the use of `uvm_do_on() instead of `uvm_do() !!
    virtual task body();
        // Write to a bunch of register locations
        for ( int i = 0; i < 32; i += 4 ) begin
            `uvm_do_on_with ( write_seq,
                p_sequencer.m_write_sqr, { addr == i; })
        end
        // Now read the results on another interface
        `uvm_do_on ( read_seq, p_sequencer.m_read_sqr )
        ...
    endtask : body
endclass : read_write_seq

...
// Connect the sequencer to the driver
class my_env extends uvm_env;
    ...
    my_virtual_sequencer      m_vseqr;
    uvm_sequencer #(seq_item)  m_seqr_w;
    uvm_sequencer #(seq_item)  m_seqr_r;

    // Build the components
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        m_vseqr = my_virtual_sequencer::type_id::create(

```

Virtual Sequences

```
"m_vseqr", this);
m_seqr_w = uvm_sequencer#(seq_item)::type_id::create(
    "m_seqr_w", this);
m_seqr_r = uvm_sequencer#(seq_item)::type_id::create(
    "m_seqr_r", this);
endfunction : build_phase

// Connect up the sequencers to the virtual sequencer
function void connect_phase(uvm_phase phase);
    m_vseqr.m_write_sqr = m_seqr_w;
    m_vseqr.m_read_sqr = m_seqr_r;
endfunction : connect_phase
...
endclass : my_env
```

Tips

- The configuration mechanism could be used to connect the sequencers to the variables in the virtual sequencer.
- Multiple virtual sequencers can be connected together in the same way that a sequencer connects to a virtual sequencer. A higher-level virtual sequencer contains references to lower-level virtual sequencers, just as a regular virtual sequencer contains references to ordinary sequencers. In this way, multiple layers of stimulus and control can be configured using a hierarchy of virtual sequencers.
- A virtual sequence can also be run on a null sequencer.

Gotchas

- Do not use the macros `uvm_do, `uvm_do_with, `uvm_rand_send, etc. with virtual sequences. Instead, use the `uvm_do_on and `uvm_do_on_with.
- Use uvm_sequence and uvm_sequencer for virtual sequences. No uvm_virtual_sequence class exists.
- Virtual sequence action macros such as `uvm_do_on automatically create and randomize a sequence object. If instead you call seq.start manually, the argument this_item must first be allocated (using new or sequence_type::type_id::create) and randomized.
- By default, a sequencer will not execute any sequences.

See also

Sequence; uvm_sequence; uvm_sequencer; Sequence Action Macros

Index

Before A

\$finish	184, 191
_global_reporter	178
`finish_item	202
`start_item	202
`uvm_*_decl	246
`UVM_ABSTRACT	88
`uvm_add_to_seq_lib	222, 224
`UVM_ALL_ON	88
`UVM_BIN	88
`UVM_COMPARE	88
`uvm_component_param_utils	58, 292
`uvm_component_utils	57, 291
`UVM_COPY	88
`uvm_create	202, 206, 215
`uvm_create_on	202, 216
`uvm_create_seq	202
`UVM_DEC	88
`uvm_declare_p_sequencer	201, 211, 215
`UVM_DEFAULT	88
`uvm_do	202, 206, 213, 215, 221
`uvm_do_on	202, 216, 294, 296
`uvm_do_on_pri	202, 216
`uvm_do_on_pri_with	216
`uvm_do_on_with	216, 294, 296
`uvm_do_pri	215
`uvm_do_pri_with	202, 215
`uvm_do_seq	202
`uvm_do_seq_with	202
`uvm_do_with	202, 206, 213, 216, 221
`UVM_ENUM	88
`uvm_error	174, 184
`uvm_fatal	174, 184
`uvm_field_*	86
`uvm_field_utils	58, 292
`UVM_HEX	88
`uvm_info	174, 184
`UVM_NORADIX	88
`uvm_object_param_utils	291
`uvm_object_utils	291
`UVM_OCT	88
`UVM_PACK	88
`UVM_PHYSICAL	88
`uvm_pri	202

`UVM_PRINT	88
`uvm_rand_send	202, 206, 216
`uvm_rand_send_pri	202, 216
`uvm_rand_send_pri_with	202, 216
`uvm_rand_send_with	202, 216
`UVM_READONLY	88
`UVM_REAL	88
`UVM_REFERENCE	88
`uvm_send	202, 206, 216
`uvm_send_pri	202, 216
`uvm_sequence_library_utils	222, 224
`UVM_STRING	88
`UVM_TIME	88
`UVM_UNSIGNED	88
`uvm_warning	174, 184
+UVM_CONFIG_DB_TRACE	40
+UVM_DUMP_CMDLINE_ARGS	40
+UVM_MAX_QUIT_COUNT	40
+UVM_OBJECTION_TRACE	40
+UVM_PHASE_TRACE	40
+UVM_RESOURCE_DB_TRACE	40
+uvm_set_action	40
+uvm_set_config_int	40
+uvm_set_config_string	40
+uvm_set_inst_override	40
+uvm_set_severity	40
+uvm_set_type_override	41
+uvm_set_verbose	41
+UVM_SEVERITY	178
+UVM_TESTNAME	41, 243
+UVM_TIMEOUT	41
+UVM_VERBOSITY	41, 178

A

accept_tr()	56
add()	
uvm_callbacks	36
uvm_heartbeat	90
uvm_phase	112
uvm_pool	115
add_by_name()	36
add_callback()	
uvm_event	78
add_sequence()	222
add_sequences()	222
add_typewide_sequence()	222
add_typewide_sequences()	222

after_export	96
Agent	14
all_dropped()	
uvm_callbacks_objection	38
uvm_component	53
uvm_objection	106
uvm_objection_callback	108
analysis	21, 24, 240

B

b_transport()	282
backdoor access	44
barrier	26
before_export	96
begin_child_tr()	56
begin_tr()	56
bit_exists()	61
body()	206, 208
build_phase()	48, 64, 109

C

callback_mode()	29
Callbacks	32
Register layer	168
can_get()	
TLM-1	249
uvm_tlm_fifo	274
can_peek()	
TLM-1	249
uvm_tlm_fifo	274
can_put()	
TLM-1	249
uvm_tlm_fifo	274
cancel()	78
catch()	181
check_config_usage()	65
check_phase()	48, 110
clear()	
uvm_objection	105
clone()	
uvm_object	101
compare()	
uvm_object	102
Compilation Directives	43
Configuration	64
Field Macros	86
configure_phase()	49, 110

connect()	
TLM-2.0 Sockets	267
uvm_port_base	120
connect_phase()	48, 109
copy()	
uvm_object	102
create()	
type_id	53, 59, 83, 103
uvm_object	101, 103
uvm_pool	115
create_component()	54
create_component_by_name()	81
create_component_by_type()	80
create_object()	54
create_object_by_name()	80
create_object_by_type()	80

D

debug_connected_to()	120
debug_create_by_name()	82
debug_create_by_type()	81
debug_provided_to()	120
default_sequence	235
delete()	
uvm_callbacks	36
uvm_pool	115
uvm_queue	136
delete_by_name()	36
delete_callback()	
uvm_event	78
die()	191
display_objections()	105
do_accept_tr()	57
do_begin_tr()	57
do_compare()	
uvm_object	102
do_copy()	
uvm_object	102
do_end_tr()	57
do_flush ()	51
do_global_phase()	75
do_pack()	
uvm_object	102
do_print()	121, 126, 130
uvm_object	102
do_report()	
uvm_object	102
do_resolve_bindings()	51

do_unpack()	
uvm_object	103
drop_objection()	
uvm_objection	105
uvm_phase	112
dropped()	
uvm_callbacks_objection	38
uvm_component	53
uvm_objection	106
uvm_objection_callback	108
dump()	
uvm_resource_db	194
dump_report_state()	177

E

End of Test	72
end_of_elaboration_phase()	48, 109
end_tr()	57
exec_func()	112
exec_task	112
exists()	
uvm_pool	115
Export	21
TLM-1	251
TLM-2.0	263
extract_phase()	48, 110

F

Factory	80, 243
create	83
create_component	83
create_object	83
field access modes	167
Field Macros	86
final_phase()	48, 110
find()	196
uvm_phase	112
find_all()	196
find_by_name()	
uvm_phase	112
find_override_by_name()	
uvm_factory	81
find_override_by_type()	
uvm_factory	81
finish_item()	209
first()	
uvm_pool	115

flush()

uvm_component	51
uvm_in_order_*_comparator	96
uvm_tlm_analysis_fifo	269
uvm_tlm_fifo	274

G

generate_stimulus()	139
get()	
TLM-1	248
uvm_config_db	61
uvm_factory	81
uvm_pool	116
uvm_queue	136
uvm_seq_item_pull_port	69
uvm_tlm_analysis_fifo	269
uvm_tlm_fifo	274
get_arbitration()	228
get_arg_matches()	39
get_arg_value()	39
get_arg_values()	40
get_args()	39
get_by_name()	193
get_by_type()	193
get_child()	47
get_children()	47
get_config_int()	65
get_config_object()	65
get_config_string()	65
get_connected_to()	120
get_depth()	47
get_domain()	50
uvm_phase	112
get_domain_name()	
uvm_phase	112
get_drain_time()	106
get_first_child()	47
get_full_name()	
uvm_component	47
uvm_object	101
uvm_phase	113
get_global()	
uvm_pool	116
uvm_queue	136
get_global_pool()	116
get_global_queue()	
uvm_queue	136
get_if()	120

get_imp()	113
get_inst()	39
get_inst_count()	101
get_inst_id()	101
get_jump_target()	113
get_name()	
uvm_component	47
uvm_object	101
uvm_sequence	213
get_next_child()	47
get_next_item()	219, 236, 239
get_num_children()	47
get_num_waiters()	26, 78
get_object_type()	
uvm_object	101
get_objection_count()	106
get_objection_total()	106
get_objectors()	105
get_parent()	
uvm_component	47
uvm_phase	113
get_phase_type()	113
get_plusargs()	39
get_provided_to()	120
get_run_count()	113
get_schedule()	50, 113
get_schedule_name()	113
get_sequence_path()	221
get_sequencer()	220
get_sequences()	223
get_state()	113
get_threshold()	26
get_tool_name()	40
get_trigger_data()	77
get_trigger_time()	77
get_type()	
uvm_component	54
uvm_object	101
get_type_name()	
uvm_component	48
uvm_object	101, 103
uvm_pool	116
get_uvm_args()	39
global_stop_request()	72, 111, 197
grab()	209, 227

H

has_child()	48
-------------	----

has_do_available()	69
HDL Backdoor Access	93

I**Imp**

TLM-1	251
TLM-2.0	263
init_sequence_library()	223
insert()	
uvm_queue	136
IP-XACT	155
is()	113
is_after()	113
is_before()	113
is_empty()	
uvm_tlm_analysis_fifo	269
uvm_tlm_fifo	275
is_enabled()	29
is_export()	119
is_imp()	120
is_off()	77
is_on()	77
is_port()	119
is_relevant()	209
is_unbounded()	119
item_done()	69, 219, 236, 239

J

jump()	113
jump_all()	113

L

last()	116
lock()	209, 227

M

m_matches	96
m_mismatches	96
main_phase()	49, 110
max_quit_count	176
max_size()	119
mid_do()	201, 208
min_size()	119

N

nb_transport()	
TLM-1	249
nb_transport_bw()	282, 283
nb_transport_fw()	282, 283
next()	116
next_item()	69
null sequencer	296
num()	116

O

Objection	
example	72
Override (factory)	80

P

pack()	
uvm_object	102
pair_ap	96
peek()	
TLM-1	248
uvm_seq_item_pull_port	69
uvm_tlm_fifo	275
phase_ended()	49
phase_ready_to_end()	50
phase_started()	49
Phases	109
Callbacks	48
pop_back()	137
pop_front()	137
Port	24
TLM-1	251
TLM-2.0	263
post_body()	206, 208
post_configure_phase()	49, 110
post_do()	201, 208
post_main_phase()	49, 110
post_reset_phase()	49, 110
post_shutdown_phase()	49, 111
post_start()	206, 208
pre_abort()	56
pre_body()	206, 208
pre_configure_phase()	49, 110
pre_do()	201, 208
pre_main_phase()	49, 110
pre_reset_phase()	49, 110
pre_shutdown_phase()	49, 110

pre_start()	206, 208
prev ()	116
Print	121
print()	
uvm_component	48
uvm_factory	82
uvm_object	102
print_config()	65
print_config_matches()	66
print_config_with_audit()	66
print_enabled	57
print_override_info()	54
push_back()	137
push_front()	137
put()	
TLM-1	248
uvm_seq_item_pull_port	69
uvm_tlm_fifo	275

R

raise_objection()	
uvm_objection	105
uvm_phase	113
raised()	
uvm_callbacks_objection	38
uvm_component	52
uvm_objection	106
uvm_objection_callback	108
RALF	145
block	150
field	145
memory	149
regfile	148
register	146
system	152
virtual register	150
ralgen	145
read_by_name()	193
read_by_type()	193
Record	56, 287
record()	
uvm_object	102
record_error_tr()	57
record_event_tr()	57
Register and Memory Package	155
Register and Memory Sequences	141
Register Generators	145
Register Layer	161

Callbacks	168
register()	81
Regular expression	64
remove()	
uvm_heartbeat	91
remove_sequence()	223
Report	173
report verbosity level	174
report_hook()	177, 184
report_phase()	48, 110
report_summarize()	177, 184, 191
req	
uvm_driver	68
reseed()	103
reset()	
uvm_event	78
reset_phase()	49, 110
resolve_bindings()	50, 120
resume()	50, 51
RGM	155
addressBlock	158
field	155
memoryMap	159
register	157
rsp	
uvm_driver	68
rsp_port	
uvm_driver	68
run_phase()	48, 109
run_test()	75, 196, 243

S

select_sequence()	222
seq_item_port	
Sequence	201
uvm_driver	68
Sequence	201
Sequence Action Macros	215
Sequencer Interface and Ports	236
set()	
uvm_config_db	61
uvm_resource_db	193
set_arbitration()	228
set_auto_reset()	26
set_config()	204
set_config_int()	65
set_config_object()	65
set_config_string()	65

set_default()	193
set_default_index()	120
set_domain()	50
set_drain_time()	105
set_heartbeat()	90
set_if()	120
set_inst_override()	
uvm_component	54
uvm_object	103
set_inst_override_by_name()	
uvm_factory	81
set_inst_override_by_type()	
Sequence	204
uvm_component	54
uvm_factory	81
set_mode()	90
set_name()	101
set_phase_imp()	50
set_report_default_file()	176, 187, 191
set_report_default_file_hier()	55, 190
set_report_id_action()	175, 187, 191
set_report_id_action_hier()	55, 190
set_report_id_file()	176, 187
set_report_id_file_hier()	55, 190
set_report_id_verbose()	187, 191
set_report_id_verbose_hier()	55, 190
set_report_max_quit_count()	176, 187, 190
set_report_severity_action()	175, 187, 191
set_report_severity_action_hier()	55, 190
set_report_severity_file()	176, 187
set_report_severity_file_hier()	55, 190
set_report_severity_id_action()	175, 187, 188, 191
set_report_severity_id_action_hier()	55, 190
set_report_severity_id_file()	176, 187
set_report_severity_id_file_hier()	55, 190
set_report_severity_id_override()	188
set_report_severity_id_verbose()	187, 191
set_report_severity_id_verbose_hier()	55, 190
set_report_severity_override()	188
set_report_verbose_level()	174, 188
set_report_verbose_level_hier()	56, 190
set_timeout()	197
set_type_override()	
uvm_component	54
uvm_object	103
set_type_override_by_name()	
uvm_factory	81
set_type_override_by_type()	
Sequence	204

uvm_component	54
uvm_factory	81
shutdown_phase()	49, 111
size()	
uvm_port_base	120
uvm_queue	137
uvm_tlm_fifo	275
SPIRIT IP-XACT	155
sprint ()	
uvm_object	102
start()	
uvm_heartbeat	91
uvm_sequence	208
start_item()	209
start_of_simulation_phase()	48, 109
starting_phase	211
state()	114
STDERR	190
STDIN	190
STDOUT	190
stop()	
uvm_heartbeat	91
stop_stimulus_generation()	139
suspend()	50, 51
sync()	113

T

TLM Implementation Declaration Macros	246
TLM-1 Interfaces	248
TLM-1 Ports, Exports and Imps	251
TLM-2.0 Ports, Exports and Imps	263
TLM-2.0 Sockets	266
TLM-2.0	258
trace_mode()	
uvm_objection	106
transport()	
TLM-1	249
TLM-2.0	283
trigger()	77
try_get()	
TLM-1	249
uvm_tlm_analysis_fifo	269
uvm_tlm_fifo	275
try_peek()	
TLM-1	249
uvm_tlm_analysis_fifo	269
uvm_tlm_fifo	275
try_put()	

TLM-1	249
uvm_tlm_analysis_fifo	269
uvm_tlm_fifo	275
type_id::create()	53, 59, 83, 103
type_id::get_type_name()	53
type_id::set_inst_override()	54
type_id::set_type_override()	53

U

ungrab()	209, 227
unlock()	227
unpack()	
uvm_object	102
unsync()	114
used()	
uvm_tlm_analysis_fifo	269
uvm_tlm_fifo	275
Utility Macros	291
uvm_access_e	164
UVM_ACTIVE	14
uvm_active_passive_enum	14
uvm_agent	14
uvm_algorithmic_comparator	17
uvm_analysis_export	21
uvm_analysis_port	24, 240
uvm_barrier	26
uvm_bottomup_phase	112
UVM_CALL_HOOK	175, 186, 187
uvm_callback	29
example	32
uvm_callback_iter	30
uvm_callbacks	36
uvm_callbacks_objection	38
UVM_CB_TRACE_ON	43
uvm_check_e	164
UVM_CMDLINE_NO_DPI	43
uvm_cmdline_proc	39
uvm_cmdline_processor	39
uvm_component	47
uvm_component_lookup()	48
uvm_component_registry	80
uvm_config_db	61
UVM_COUNT	175, 187, 190
uvm_coverage_e	165
uvm_default_line_printer	124, 126
uvm_default_printer	124, 126
uvm_default_table_printer	124, 126
uvm_default_tree_printer	124, 126

UVM_DISABLE_AUTO_ITEM_RECORDING	43
UVM_DISPLAY	175, 184, 187
uvm_driver	68 , 201
uvm_elem_kind_e	164
UVM_EMPTY_MACROS	43
UVM_ENABLE_FIELD_CHECKS	43
uvm_endianness_e	164
uvm_env	75
UVM_ERROR	174, 185
uvm_event	77 , 90
uvm_event_callback	77
uvm_event_pool	77
UVM_EXIT	175, 184, 185, 187
uvm_factory	80
UVM_FATAL	174, 185
UVM_FILE	56
uvm_hdl_check_path()	93
uvm_hdl_deposit()	93
uvm_hdl_force()	93
uvm_hdl_force_time()	93
UVM_HDL_MAX_WIDTH	44, 93
UVM_HDL_NO_DPI	44
uvm_hdl_read()	94
uvm_hdl_release()	94
uvm_hdl_release_and_read()	94
uvm_heartbeat	90
uvm_hier_e	164
uvm_in_order_*_comparator	96
uvm_in_order_builtin_comparator	96
uvm_in_order_class_comparator	96
uvm_in_order_comparator	96
UVM_INFO	174, 185
uvm_is_match()	64
uvm_line_printer	122, 126
UVM_LINE_WIDTH	44
UVM_LOG	175, 184, 187, 191
UVM_MAX_STREAMBITS	44
uvm_mem	163
uvm_mem_access_seq	142
uvm_mem_shared_access_seq	142
uvm_mem_single_access_seq	142
uvm_mem_single_walk_seq	142
uvm_mem_walk_seq	142
uvm_monitor	99
UVM_NO_ACTION	175, 184, 187
UVM_NO_DEPRECATED	44
UVM_NO_DPI	44
UVM_NOPRINT	121
UVM_NUM_LINES	44

uvm_object	101
uvm_object_registry	80
uvm_object_string_pool	115
uvm_objection	105
uvm_objection_callback	108
uvm_objection_cbs_t	108
UVM_PACKER_MAX_BYTES	44, 45
UVM_PASSIVE	14
uvm_path_e	163
uvm_phase	112
uvm_pool	115
uvm_port_base	119
uvm_predict_e	165
UVM_PRINT	121
uvm_printer	121, 126
uvm_printer_knobs	121, 123, 132
uvm_push_driver	68
uvm_push_sequencer	226
uvm_queue	136
uvm_random_stimulus	139
uvm_reg	163
uvm_reg_access_seq	142
UVM_REG_ADDR_WIDTH	45
uvm_reg_bit_bash_seq	141
uvm_reg_block	163
UVM_REG_BYTENABLE_WIDTH	45
UVM_REG_CVR_WIDTH	45
UVM_REG_DATA_WIDTH	45
uvm_reg_field	163
uvm_reg_file	163
uvm_reg_hw_reset_seq	141
uvm_reg_map	163
uvm_reg_mem_built_in_seq	143
uvm_reg_mem_hdl_paths_seq	143
uvm_reg_mem_shared_access_seq	142
uvm_reg_mem_tests_e	165
UVM_REG_NO_INDIVIDUAL_FIELD_ACCESS	45
uvm_reg_sequence	141
uvm_reg_shared_access_seq	142
uvm_reg_single_access_seq	141
uvm_reg_single_bit_bash_seq	141
uvm_report_catcher	180
UVM_REPORT_DISABLE_FILE	45
UVM_REPORT_DISABLE_FILE_LINE	46
UVM_REPORT_DISABLE_LINE	46
uvm_report_enabled()	188
uvm_report_error()	174, 185
uvm_report_fatal()	174, 185
uvm_report_handler	173

uvm_report_info()	174, 185
uvm_report_object	173, 184
uvm_report_server	173
uvm_report_warning()	174, 185
uvm_resource_db	192
read_by_name()	193
set()	193
UVM_RGM (Register and Memory Package)	155
uvm_root	196
uvm_scoreboard	200
uvm_seq_item_export	236
uvm_seq_item_port	219, 236
uvm_seq_item_pull_export	236
uvm_seq_item_pull_port	236
uvm_sequence	201, 206 , 219
uvm_sequence_base	207
uvm_sequence_item	201, 219
uvm_sequence_library	219, 222
uvm_sequence_library_cfg	222
uvm_sequencer	201, 226
uvm_sequencer_base	226
uvm_sequencer_param_base	226
uvm_status_e	163
UVM_STOP	175
uvm_subscriber	21, 240
uvm_table_printer	122, 126
uvm_table_printer_knobs	132
uvm_task_phase	112
uvm_test	243
uvm_test_done_objection	105
uvm_tlm_analysis_fifo	269
uvm_tlm_command_e	278
uvm_tlm_extension	272 , 278
uvm_tlm_extension_base	272
uvm_tlm_fifo	274
uvm_tlm_generic_payload	278
uvm_tlm_if	282
uvm_tlm_phase_e	282
uvm_tlm_response_status_e	278
uvm_tlm_sync_e	283
uvm_tlm_time	285
uvm_top	196
uvm_topdown_phase	112
uvm_transaction	221, 287
uvm_tree_printer	122, 126
uvm_tree_printer_knobs	132
UVM_USE_CALLBACKS_OBJECTION_FOR_TEST_DONE	46
UVM_WARNING	174, 185, 190

V

Verbosity level	174
Virtual Sequencer	294
Virtual Sequences	203, 294

W

wait_for()	
uvm_barrier	26
uvm_objection	106
wait_for_relevant()	209
wait_for_sequences()	69
wait_modified()	61
wait_off()	77
wait_on()	77
wait_ptrigger()	77
wait_ptrigger_data()	77
wait_trigger()	77
wait_trigger_data()	77
Wildcard (* and ?)	64
write()	
TLM-1	249
uvm_subscriber	240
uvm_tlm_analysis_fifo	269
write_by_name()	193
write_by_type()	193

Free UVM Tutorials

To assist new users in understanding and applying UVM, Doulos has created a number of tutorials, which are available on our website. Please visit www.doulos.com/knowhow

The Golden Reference Guide (GRG) series

- SystemC
- SystemVerilog
- OVM
- VMM
- PSL
- VHDL
- Verilog

About Doulos

Doulos is the global leader for the development and delivery of world class training solutions for SoC, FPGA and ASIC design and verification. Established in 1990 and fully independent, Doulos sets the industry standard for high quality training in SystemC™, SystemVerilog, PSL, Verilog®, VHDL, Embedded Systems, ARM®, Xilinx®, Altera®, Perl & Tcl/Tk.

Doulos know-how is delivered worldwide through regularly scheduled classes in major locations in the U.S and Europe, and through in-house training at customer locations. To find out more about the Doulos training portfolio please visit our website www.doulos.com

