

# UVM Notes

## 1. Overview

### a. What is UVM

- UVM is a standardized methodology, a set of pre-defined libraries using syntax and semantics of SystemVerilog.

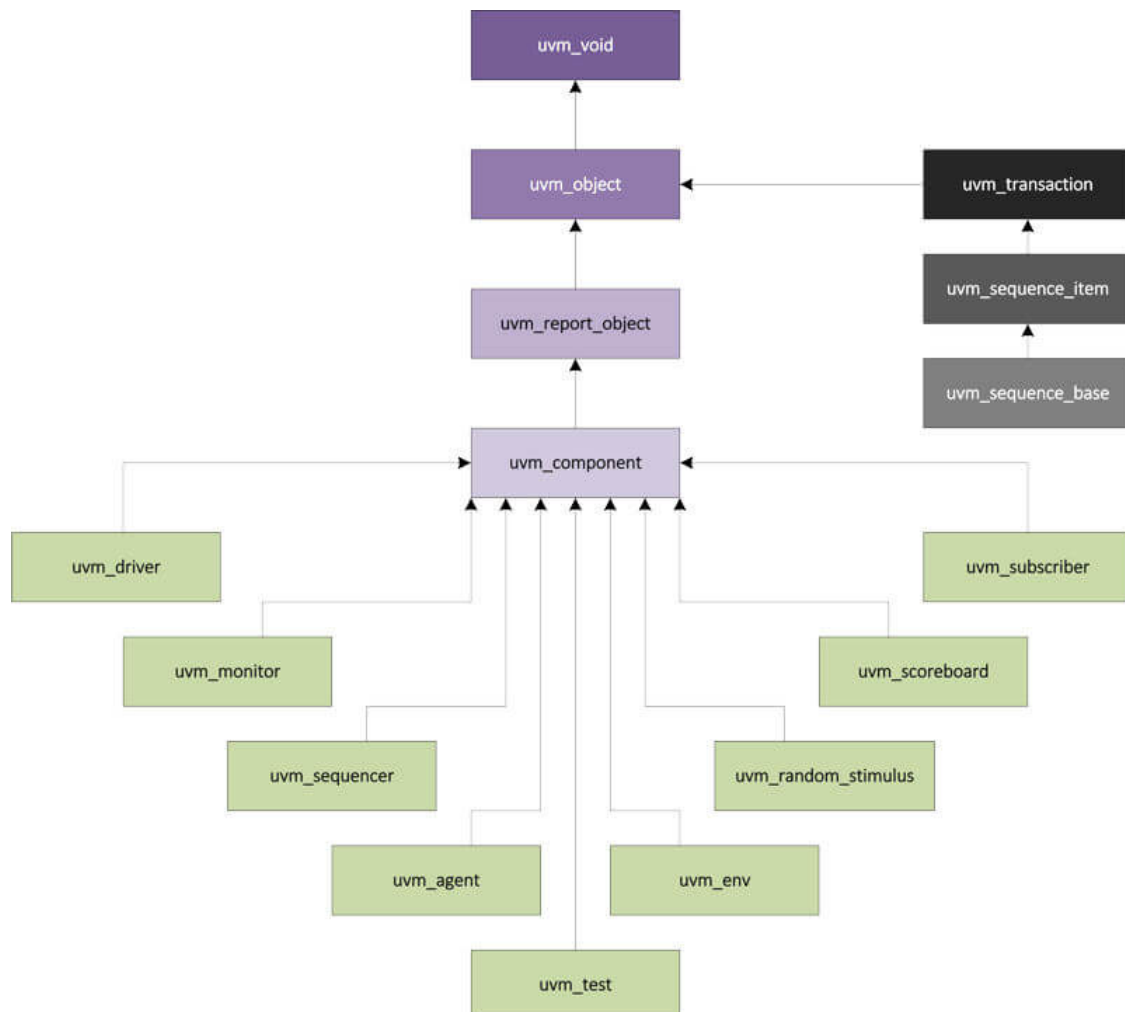
### b. Why UVM

Increases reusability

- The components(driver, sequencer, etc) are modules that can be re-used across projects.
- Stimulus are separate from the actual testbench hierarchy and can be re-used or replaced by new stimulus.
- Factory mechanisms make modifications of components easy. Create each components using factory enables them to be overridden in different tests/environments without changing the code base.

### c. UVM class hierarchy

- UVM provides a set of base classes that more complex classes can be built by inheritance.
- Two main branches. The verification components are underneath `uvm_component`, and data objects consumed and operated by components are underneath `uvm_transaction`.



- Sequence is a container for the actual stimulus to the design. Stimulus classes are inherited from `uvm_sequence`.
- Data objects that are driven to DUT are sequence items and are inherited from `uvm_sequence_item`.
- UVM utilizes TLM(transaction level modeling) which helps to send data between components in the form of transactions and class objects. It also can broadcast a packet to its listeners without creating specific channels and attach to it.
- Phases enable every component to sync with each other before proceeding to the next phases. Every component goes through the build phase when it gets instantiated, connects with each other during the connect phase, consumes simulation time during the run phase, and stops together in the final phase.

## 2. UVM Common Utilities

### a. Base Classes

- `uvm_root`
  - It is an implicit top\_level UVM component that is automatically created when simulation is run.
  - Users can access it via the global variable, `uvm_top`. Any components whose parent is set to null becomes a child of `uvm_top`.
  - `uvm_top` checks for errors during `end_of_elaboration` phase and issue `uvm_fatal` error to stop simulation.
- `uvm_report_object`
  - All messages, warnings, errors issued by components go via this interface.
  - A report has ID String, Severity, Verbosity Level, and Test Message parts. If the verbosity level is **GREATER** than the configured maximum verbosity level, it is **ignored**. For example, if maximum verbosity level is `UVM_MEDIUM`, and a info is assigned to verbosity level `UVM_HIGH`, then this message will not be seen in the output.

### b. UVM utility & field macros

- UVM uses the concept of a factory where all objects are registered.
- Utility Macros
  - The `utils` macro is used to register an object or component with the factory.
  - Required to be used inside every user-defined class derived from `uvm_object`
  - Object Utility
    - All classes derived directly from `uvm_object` or `uvm_transaction` required to be registered using ``uvm_object_utils` macro.
    - It is mandatory for the new function to be explicitly defined for every class, and take the name of the class instance as an argument.

`uvm_sequence` is inherited from `uvm_sequence_item`, `uvm_transaction`, then `uvm_object`.

```
class fc_sequence extends uvm_sequence #(fc_transaction);
    //register fc_sequence, this user-defined class with the factory
    `uvm_object_utils(fc_sequence)
    function new(string name = "fc_sequence");
        super.new(name);
    endfunction
endclass
```

- Component Utility
  - All classes derived directly or indirectly from `uvm_component` are required to be registered with the factory using ``uvm_component_utils` macro.

- It is mandatory for the new function to be explicitly defined for every class, and takes the name of the class instance and a handle to the parent class where this object is instantiated.
- Macro Expansion
  - ``uvm_object_utils` gets expanded into its `*_begin` and `*_end` form with nothing in between.
  - `*_begin` implements other macros, such as
    - ``m_uvm_object_registry_internal(T,T)`, which implements the function `get_type()` and `get_object_type()` that returns a factory proxy object for the requested type
    - ``m_uvm_object_create_func(T)` which instantiates an object of the specified type by calling its no-args constructor
    - ``m_uvm_get_type_name_func(T)` which return the `type_name` as a string.
    - ``uvm_field_utils_begin(T)` which registers the type with UVM factory
- Creation of class object
  - Recommend all class objects are created by calling the `type_id::create()` method which is defined using the macro ``m_uvm_object_create_func(T)`. (this macro utilizes the `new()` function. When creating component object, two arguments are taken which are the name and parent).

```
fc_drv = fc_driver::type_id::create("fc_drv",this);
```

- Field Macros
  - ``uvm_field_*` macros that were used between `*_begin` and `*_end` provide automatic implementations of core methods like copy and compare.
  - ``uvm_field_*` corresponding to the data type of variables been used. Variables of type int, bit, byte should use ``uvm_field_int`, type string should use ``uvm_field_string` and so on.
  - The macro accepts at least two arguments, ARG and FLAG. ARG is the name of the variable, FLAG specifics which data method implementations will not be included(except `UVM_ALL_ON` and `UVM_DEFAULT`).
    - `UVM_ALL_ON`: all operations are turned on
    - `UVM_DEFAULT`: enables all operations, equivalent to `UVM_ALL_ON`
    - `UVM_NOCOPY`, `UVM_NOCOMPARE`, `UVM_NOPRINT`, `UVM_NOPACK`: do not copy, compare, print, pack/unpack the given variable
    - `UVM_REFERENE`: operate only on handles.
- UVM Object Print
  - After using `type_id create` to create an object, we can randomize it and print it using `obj.randomize()` and `obj.print()`.
  - `do_print()`
    - using automation macros introduces additional codes and reduces simulator performance.
    - We can use `do_*` callback. For example, we can use `do_print` inside the derived object. `do_print` is called by the print function by default.

```
virtual function void do_print(uvm_printer printer);
    super.do_print(printer);
    printer.print_string();
    //can control the radix of the given variable, such as UVM_HEX or UVM_DEC
    printer.print_field_int();
endfunction
```

- UVM Object Copy/Clone

- we can use obj2.copy(obj1) method to copy the content of obj1 into obj2
  - do\_copy()
    - A generic uvm\_object called "rhs" is received and type casted into Packet pkt. Then m\_addr is copied from the type-casted \_pkt to the variable of the current class.
- "rhs" does not contain o\_bool as its only a parent handle. We cast this rhs into child data type and access it using child handle. We then copy content of casted handle into local variables.

```
class Packet extends uvm_object;
  rand bit[15:0] m_addr;
  ...
  virtual function void do_copy(uvm_object rhs);
    Packet _pkt;
    super.do_copy(rhs);
    $cast(_pkt, rhs);
    m_addr = _pkt.m_addr;
  endfunction
  ...
endclass

class Object extends uvm_object;
  rand Packet m_pkg;
  rand bool o_bool;
  ...
  virtual function void do_copy(uvm_object rhs);
    Object _obj;
    super.do_copy(rhs);
    $cast(_obj, rhs);
    o_bool = _obj.o_bool;
    m_pkg.copy(_obj.m_pkg);
  endfunction
endclass
```

- Clone
  - Clone will return an object with the copied contents, so no need of creating the second object before copy.
- UVM Object Compare
  - we can use something like obj2.compare(obj1).
  - do\_compare

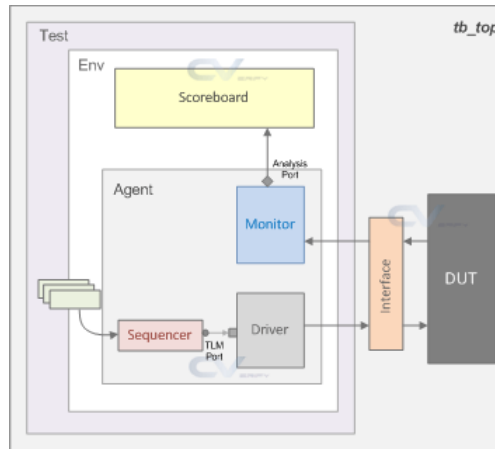
```
//inside Packet class
virtual function bit do_compare(uvm_object rhs, uvm_comparer comparer);
  bit res;
  Packet _pkt;
  $cast(_pkt, rhs);
  super.do_compare(_pkt, comparer);
  res = super.do_compare(_pkt, comparer) & m_addr == _pkt.m_addr;
  return res;
endfunction

virtual function bit do_compare(uvm_object rhs, uvm_comparer comparer);
  bit res;
  Object _obj;
  $cast(_obj, rhs);
  res = super.do_compare(_obj, comparer) & o_bool == _obj.o_bool
  & m_pkg.do_compare(_obj.m_pkg, comparer);
  return res;
endfunction
```

### 3. Testbench Structures

#### a. UVM Testbench Top

- All verification components, interfaces, and DUT are instantiated in a top level module called testbench.



- At the start of simulation, set the interface handle as a config object in UVM database(`uvm_config_db::set`). This if can be retrieved in the test using the `get()` method.
- `run_test("test_name")` accepts test name as argument, and the test\_name case will be run for simulation
- `tb_top` is a static container

#### b. UVM Test

- We can put the entire testbench into a container, `environment`, and use same environment for different test. Each testcase can manipulate `agents`, and run different `sequences` on many `sequencers` in the environment.
- As shown in the above top-level picture, we can start a virtual/normal `sequence` on a given `sequencer` in the `run_phase` of the test. Remember to raise and drop the objection
- A base test sets up all basic environment parameters and configurations that can be overridden by derivative tests. For the new test, we can define the phases we want to change, the the object will call its parent's phases that's not explicitly defined.

#### c. UVM Environment

- A UVM environment contains multiple reusable verification components and defined their default configuration as required.
- It is possible to instantiate agents and scoreboards directly in `uvm_test`, but tests become non-reusable because
  - They rely on a specific environment structure.
  - The test writer would need to know how to configure the environment
- `uvm_env` is the base class for hierarchical containers of other components that make up a complete environment
- We need to connect verification components together in the `connect_phase`

#### d. UVM Driver

- UVM driver drives transactions to a particular interface of the design. Transaction level objects are obtained from the `sequencer` and the UVM driver drivers them to the design via an interface handle.
- UVM Driver-Sequencer handshake
  - The UVM driver uses following methods to interact with the sequencer.

- **get\_next\_item**: blocks until a request item is available from the sequencer. Should be followed by **item\_done** to complete the handshake.
- **try\_next\_item**: non\_blocking method which return **null** if a request object is not available from the sequencer. Else returns a pointer to the object
- **item\_done**: non\_blocking method which completes the driver-sequencer handshake. Should be called after **get\_next\_item** or a successful **try\_next\_item** call.
- A driver-sequencer handshake allow the driver to get a series of transaction objects from the sequence and respond back to the sequence after it finishes driving the given item, so it can get the next sequence
  - **get\_next\_item + item\_done**
    - **finish\_item** call in the sequence finishes only after the driver returns **item\_done** call

```
virtual task run_phase (uvm_phase phase);
  my_data req_item;
  forever begin
    seq_item_port.get_next_item(req_item);
    @(posedge vif.clk);
    vif.en <= 1;
    seq_item_port.item_done();
  end
endtask
```

- **get + put**
  - The driver gets the next item and send back the sequence handshake in one go, before the UVM driver processes the item
  - The driver uses the **put** method to indicate that the item has been finished later.
  - So **finish\_item** call in the sequence is finished as soon as **get()** is done
- A virtual interface handle **vif** is declared and assigned later in the build phase
- Real interface object is retrieved from the database directly into a local variable using **uvm\_config\_db::get()**

#### e. UVM Sequencer

- A sequencer generates data transactions as class objects and sends it to the Driver for execution
- The **uvm\_sequencer** base class is parameterized by the **request** and **response** item types and can be handled by the sequencer. By default, response type is the same as the request type.

#### f. UVM Sequence

- UVM sequences are made up of several data items.
- Executed by assigned sequencer(s) which then send(s) data items to the driver. Sequences are **core stimuli** of any verification plan.
- We can make the **body** task virtual so child classes can override the task definition
- We can use **pre\_body()** and **post\_body()** callbacks
- We can use **`uvm\_do()** sequence macros, which we have to provide a **uvm\_sequence\_item** object or a sequence and it does the following internally:

#### g. UVM Monitor

- A UVM monitor is responsible for capturing signal activity from the design interface and translate it into transaction level data objects that can be sent to other components. It should have a virtual interface handle to the actual interface that this monitor is trying to monitor, and TLM analysis port declarations to broadcast captured data to others

- Its functionality should be limited to basic monitoring that is always required. High level functional checking should be done in a scoreboard.

#### h. UVM Agent

- An agent encapsulates a sequencer, driver, and monitor into a single entity. We can have **active** or **passive** agent, which only instantiate the monitor and is used for checking and coverage only. We can use `uvm_config_db::set` to configure a passive or active agent by using `is_active` variable.
- We can use `get_is_active()` to check whether to create sequencer and driver.

#### i. UVM Scoreboard

- It is a verification component that contains checkers and verifies the functionality of a design. It usually receives transaction level objects captured from the interfaces of a DUT via TLM Analysis Ports
- After receiving data objects, the scoreboard can either perform calculations and predict the expected value, or send it to a **reference model** to get expected value. The reference model is also called a **predictor that mimics the functionality of the design**. The scoreboard then compares the expected results with the actual output data from DUT
- It is not required to perform checks in the **check\_phase**. Real checkers can also check during the **run\_phase**.
- After connecting the scoreboard with other components(e.g. monitor), monitor can send data to the scoreboard via an analysis port by calling the port's **write** method.

#### j. UVM Subscriber

- Subscribers are listeners of an analysis port. They subscribe to a broadcaster and receive objects whenever an item is broadcasted via the connected analysis port.

#### k. UVM Virtual Sequencer

- It is a UVM sequencer that contain handles to other sequencers.

### 4. UVM Phases

- All testbench components are derived from `uvm_component` and goes through a pre-defined set of phases. It cannot proceed to the next phase until all components finish their execution in the current phase.
- We have **functions** that are methods that do not consume simulation time and **tasks** that consume simulation time
  - Build time phases. Functions
    - **build\_phase**, used to build testbench components and create their instances
    - **connect\_phase**, used to connect between different testbench components via TLM ports
    - **end\_of\_elaboration\_phase**, used to display UVM topology(e.g. `print_topology` displays all instantiated components in the environment to help debug) and other functions required to be done after connection
    - **start\_of\_simulation\_phase**, used to set initial run-time configuration or display topology.
  - Run time phases. Tasks
    - **run\_phase**. Actual simulation that consumes time, and runs parallel to other UVM run-time phases.
  - Clean-Up phases. Functions
    - **extract\_phase**, used to extract and compute expected data from scoreboard
    - **check\_phase**, used to perform scoreboard tasks that check for errors between expected and actual values from design
    - **report\_phase**, used to display result from checkers, or summary of other test objectives
    - **final\_phase**, used to do last minute operations before exiting the simulation

- Why doesn't Verilog tb need phases?
  - All of its components made of static containers(modules), so each module will have a set of ports/signals that it utilizes to communicate with other tb components.
  - Since a module is static, all modules will be created at the beginning of the simulation.
- Why SystemVerilog testbench require phases?
  - With OOP, entities(class objects) can be reused and deployed when required.
  - It is possible to create a new object in Xns, we it is possible to call a component that's not initialized.
  - In addition, we need synchronization between testbench components.

## 5. UVM Factory Override

- UVM Factory is a mechanism to improve flexibility and scalability of the tb by allowing the user to **substitute** an existing class object by any of its inherited child class objects.
- Factory needs to know all types of classes created within the tb via **registration**.
- Why Override? **With the help of the factory, we can override the type of underlying components or objects from the top-level component without having to edit the code.**
  - For example, if we want to replace new\_driver() with the base\_driver(), all we have to do is to override the base driver by one of the factory override methods, instead of going to code and substitute every single code that mentions base\_driver();

### Factory Override Methods

- **set\_type\_override\_by\_type()/set\_type\_override\_by\_name()**, which override all the objects of a particular type.
  - e.g. set factory to override "base\_agent" by "child\_agent" by type
    - **set\_type\_override\_by\_type(base\_agent::get\_type(), child\_agent::get\_type());**
  - e.g. set by name

```
uvm_factory factory = uvm_factory::get();
factory.set_type_override_by_name("base_agent", "child_agent");
```

- **set\_inst\_override\_by\_type()/set\_inst\_override\_by\_name()**, which override a type within a particular instance
  - When only a few instances of the given type has to be override, we can use instance override by type/name.
  - e.g. set factory to override all instances under **my\_env** of type "base\_agent" by "child\_agent".
    - **set\_inst\_override\_by\_type("my\_env.\*", base\_agent::get\_type(), child\_agent::get\_type());**
  - e.g. set by name

```
uvm_factory factory = uvm_factory::get();
factory.set_inst_override_by_name("base_agent", "child_agent", {get_full_name(), ".my_env.*"});
```

## 6. Stimulus Generation

- Sequences can do operations on sequence items, or initiate new subsequences
  - Execute using the start() method of a sequence
  - Execute sequence items via start\_item/finish\_item
  - we can also use `uvm\_do macro, which will identify if the argument is a sequence or sequenece\_item and will call start() or start\_item() accordingly.



- create the item using ``uvm_create`` if necessary.
- randomize the item or sequence.
- call the `start_item()` and `finish_item()` if its a `uvm_sequence_item` object.
- call the `start()` task if its a sequence.
- ``uvm_do``: execute this sequence on default sequencer with the item provided
- ``uvm_do_with`` : override any default constraints with inline value
- ``uvm_do_pri`` : execute based on the priority value, used when running multiple sequences simultaneously
- ``uvm_do_pri_with`` : execute based on priority and override default constraints with inline values

## 7. Driver Sequencer Handshake

- The driver contains a TLM port `uvm_seq_item_pull_port` which is connected to a `uvm_seq_item_pull_export` in the sequencer in the connect phase of a UVM agent. The driver can use TLM functions to get the next item from the sequencer when required
- We need the driver sequencer API because this helps the driver to get a series of sequence\_items from the sequencer's FIFO that contains data for the driver to drive to the DUT. The driver will send finish signal to the sequence and can request the next item.
- `seq_item_port` can be used by derived driver class to request items from the sequencer and send response back. `rsp_port` provides an alternative way of sending response back to the originating sequencer.
- `seq_item_export` is an inbuilt TLM pull implementation port in a `uvm_sequencer`, which is used to connect with the driver's pull port.
- Typically, a driver and sequencer are instantiated and connected in a `uvm_agent`
  - `drv.seq_item_port.connect(seqr.seq_item_export);`
  - This is one-to-one. Multiple drivers are not connected to a sequencer nor are multiple sequencers connected to a single driver. Once the connection is made, the driver can utilize API calls in the TLM port definitions to receive sequence items from the sequencer.
- `get_next_item()`
  - The driver is a parameterized class with the type of request and response sequence items.
  - The `uvm_driver` gets request sequence item(REQ) from the sequencer FIFO and optionally returns a response sequence item(RSP) back to the sequencer response FIFO.
    - \*The driver is allowed to send back a different sequence\_item type back to the sequencer as the response. And of course, its more common to send the same type as the request sequence item
  - A `uvm_sequence` is started on a sequencer which pushes the sequence item onto the sequencer's FIFO.
    - Create an item the connected sequencer can accept
    - Call the `start_item()` task which sends this object to the driver
    - Because the class handle passed to the driver points to the same object, we can do late randomization
    - Call the `finish_item()` method so that the sequence waits until the driver lets the sequencer know this item has finished
- Using `get()` and `put()`
  - We can let the driver use `get()` method to receive the next item and later use `put()` to give a response item back to the sequencer

- So how does a sequencer stop an item now? Because `finish_item` does not indicate that the driver has finished driving the item, the sequence has to wait until the driver explicitly tells the sequencer that the item is over. So the sequence has to wait until it gets a response back from the sequencer via `get_response()`;

## 8. Reporting Infrastructure

- Reporting Functions
  - There are four basic reporting functions with different verbosity levels
    - `uvm_report_*` ("TAG", \$sformatf("[display message]"), VERBOSITY\_LEVEL);
    - \* can be info, error, warning, fatal
    - verbosity level has six levels, `UVM_NONE(0)`, `UVM_LOW(100)`, `UVM_MEDIUM(200)`, `UVM_HIGH(300)`, `UVM_FULL(400)`, `UVM_DEBUG(500)`
    - NOTE: verbosity level is only required for `uvm_report_info`. Usage of `uvm_report_warning`, `uvm_report_error`, `uvm_report_fatal` do not require verbosity. In fact, `uvm_report_fatal` will exit the simulation.
    - We can display the filename and line number of the displayed message by using ``_FILE_`, ``_LINE_`. UVM reporting macros will automatically display the file and line information without explicitly mentioning the ``_FILE_`, ``_LINE_`.
    - Verbosity level controls whether a `uvm_report_*` statement gets displayed. Default configuration is `UVM_MEDIUM`, means every `uvm_report_*` message with a verbosity level less than `UVM_MEDIUM` will be printed. This controls the number of information will be displayed. If you want to debug, you can set verbosity level to `UVM_DEBUG`, then everything under it will be displayed.
- `uvm_printer`
  - UVM avoids the need for customized print function by incorporating its own `uvm_printer` class.
  - Every class item derived from `uvm_object` will have a printer instance within it. So a data class derived from `uvm_sequence_item` will have access to the `print()` function
  - UVM has three main printer: table printer, tree printer, line printer
    - By default, UVM assigns table printer to handle every `print()` function, hence is the `uvm_default_printer`.

```
class my_data extends uvm_sequence_item;
    bit [3:0] bit_data;
endclass

my_data data_obj;
data_obj.print(); //calls table printer by default
data_obj.print(uvm_default_line_printer); //calls line printer
```

- Calling `print()` is possible if **EITHER** of the following things are done(we can do together and `do_print()` will append to the macro)
  - Add any member that needs to be printer within ``uvm_object_utils_begin` and ``uvm_object_utils_end`
  - Define a `do_print()` function for the class

## 9. UVM Config DB

- UVM resource database
  - A resource database is a parameterized container that holds arbitrary data
  - Can put any data type into the resource database, and have another component retrieve it later at some point in simulation
  - The global resource database has both a name table and a type table into which each resource is entered

- So the same resource can be retrieved later by name and type
- Multiple resources with the same name/type are stored in a queue and hence those which were pushed earlier have more precedence over those placed later
- For example if item **red** and item **blue** in the queue have the same scope, and a **get\_by\_type()** method is called for that particular scope. Then item **red** will be returned since that sits earlier in the queue
- Resources are added to the pool by calling **set**, and they are retrieved from the pool by **get\_by\_name()** or **get\_by\_type()**
- UVM config database
  - UVM has an internal database table in which we can store values under a given name and can be retrieved later by other tb component
  - **uvm\_config\_db** class provides a convenience interface on top of the **uvm\_resource\_db** to simplify the basic interface used for uvm\_component instance
- **set()**

```
static function void set (uvm_component cntxt,
                        string inst_name,
                        string field_name,
                        T value);
```

- use this static function of the class **uvm\_config\_db** to set a variable in the configuration database
- **set()** function will set a variable of name **test\_enable** at the path **uvm\_test\_top.env.agt** with value 1
- use of **set()** will create a new or update an existing configuration setting for **field\_name** in **inst\_name** from **cntxt**. This setting is made at **cntxt** with the full scope of the set begin {**cntxt**, ".", **inst\_name**}. if **cntxt** is null, then the complete scope of getting the information will be provided by **inst\_name**.

```
//set virtual interface handle under name "fcif" available to all components under "this" phase, indicated by the *
uvm_config_db#(virtual flex_counter_if)::set(this, "", "fcif", fcif);
//if in the test phase, then it is equivalent to
uvm_config_db#(virtual flex_counter_if)::set(null, "tb_flex_counter.fc_test.*", "fcif", fcif);
```

For the **cntxt** **this**, which will be substituted by the path to the current component which in the case **tb\_flex\_counter.fc\_test**.

- **get()**

```
static function void get (uvm_component cntxt,
                        string inst_name,
                        string field_name,
                        T value);
```

- use this static function to get the value of variable given in **field\_name** from the configuration database. The value will be returned only if the scope is true.

```
//get virtual interface handle under name "fcif" into local virtual interface handle at fc_test level
uvm_config_db #(virtual flex_counter_if)::get(this, "", "fcif", fcif);
```

- **exists()**

```
static function void exists (uvm_component cntxt,
                           string inst_name,
                           string field_name,
                           bit spell_chk);
```

- checks if a value for **field\_name** is available in **inst\_name**, using component **cntxt** as the starting point. If the **field\_name** does not exist at a given scope, the function will return a zero. The **spell\_chk** arg can set to 1 to turn spell checking on if it is expected that the field should exist in the database

```
if (!uvm_config_db#(virtual flex_counter_if)::exists(this, "", "fcif"))
    `uvm_error ("fcif", "cannot find fcif handle");
```

## 10. UVM TLM

- Transaction Level Modeling is a modeling style for building highly abstract models of components and systems.
- In this scheme, data is represented as transactions(class objects that contain random, protocol specific information) which flow in and out of different components via special ports called TLM interfaces.
- UVM provides a set of transaction-level communication interfaces that can be used to connect between components such that data packets can be transferred between them.
- Advantages: isolates a component from the changes in other components, and promotes reusability and flexibility because we can just swap a component with another which also have a TLM interface
- UVM TLM Blocking Put Port
  - Any component can send a transaction to another component through a TLM **put** port. The receiving component should define an implementation of the **put** port, which gives receiver the chance to define what has to be done with the incoming packet.
  - The port can be either blocking or nonblocking in nature, which will decide whether the **put** method will block execution in the sender until the receiver accepts the object.
  - a **uvm\_blocking\_put\_port** parameterized to accept a data object of type Packet. The port has to be instantiated with the **new()** method preferably in the **build\_phase** of the same component.

In the following code, of class object of type Packet is created, randomized, and sent via the **put\_port** handle by calling the **put()** method. Many such packets can be sent using a simple loop

```
uvm_blocking_put_port #(Packet) m_put_port;

virtual function void build_phase (uvm_phase phase);
    super.build_phase(phase);
    m_put_port = new ("m_port_port", this);
endfunction

//create a packet, randomize it and send it through the port
//put() is a method defined by the receiving component
//repeat N times to send N packets
virtual task run_phase (uvm_phase phase):
    phase.raise_object(this);
    repeat(N) begin
        Packet pkt = Packet::type_id::create("pkt");
        assert(pkt.randomize());
        //print the pkg
        //remember to begin and end utils of the data in the Packet class
        pkt.print(uvm_default_line_printer);
        m_put_port.put(pkt);
    end
    phase.drop_object(this);
endtask
endclass
```

- The receiver class needs to define an implementation port using `uvm_blocking_put_imp`. Since the port is blocking in nature, the `put()` implementation is a task which has to be defined by this component

```
//mention type of transaction, and type of class that implements the put()
uvm_blocking_put_imp #(Packet, component_b) m_put_imp;

virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    m_put_imp = new("m_put_imp", this);
endfunction

//implementation of the put() function. Its just a print function in this case
virtual task put(Packet pkt);
    pkg.print(uvm_default_line_printer);
endtask
```

- The connection between a port and its implementation has to be done at a higher hierarchical level, which means it can be connected during the `connect_phase` of the component that they were instantiated in, such as the `env` or the `test` class

```
//the put_port is connected to its implementation put_imp
virtual function void connect_phase(uvm_phase phase);
    compA.m_put_port.connect(compB.m_put_imp);
endfunction
```

#### • UVM TLM Nonblocking Put Port

- For `uvm_blocking_put_port`, the sender gets stalled until the receiver finishes with the `put` task
- `uvm_nonblocking_put_port`, where the sender has to use `try_put` to see if the put was successful or `can_put` method to see if the receiver is ready to accept a transfer.
- In the following example, a class object of type Packet is created, randomized, and sent via the `put_port` handle by calling the `try_put` method. The `try_put` function should return 1 if the transfer is successful and 0 if it failed.

```
uvm_nonblocking_put_port #(Packet) m_put_port;
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    m_put_port = new("m_put_port", this);
endfunction
//create a packet, randomize it, and send it through the port
//put() is a method defined by the receiving component
//repeat the step N times to send N packets
virtual task run_phase (uvm_phase phase);
    phase.raise_objection(this);
    repeat(N) begin
        bit success;
        Packet pkg = Packet::type_id::create("pkg");
        assert(pkg.randomize());
        success = m_put_port.try_put(pkg);
        if (success)
            `uvm_info();
        else
            `uvm_info();
        end
        phase.drop_objection(this);
    endtask
```

- The receiver class needs to define an implementation port using `uvm_nonblocking_put_imp`. Since the port is nonlocking in nature, `try_put()` implementation is a function which has to be defined by this component

```
uvm_nonblocking_put_imp #(Packet, component_b) m_put_imp;
virtual function void build_phase(uvm_phase phase);
```

```

    m_put_imp = new("m_put", this);
endfunction
//the defined "try_put" method accepts the packet and prints it
//It should return 1 if successful so the component A knows
//how to handle the transfer return code
virtual function bit try_put(Packet pkt);
    pkt.print(uvm_default_line_printer);
    return 1;
endfunction

```

- Instead of directly trying to put a packet, the sender can first query to see if the receiver is ready or not with **can\_put** function and then send the packet

```

//loop until can_put returns a 1. Its not even attempted to send a transaction
//using put, until the sender knows for sure that the receiver is ready to accept it
do begin
    success = m_put_port.can_put();
end while (!success); //repeat as long as success is 0, we keep checking
//if success, we break out of the loop
m_put_port.try_put(pkg);

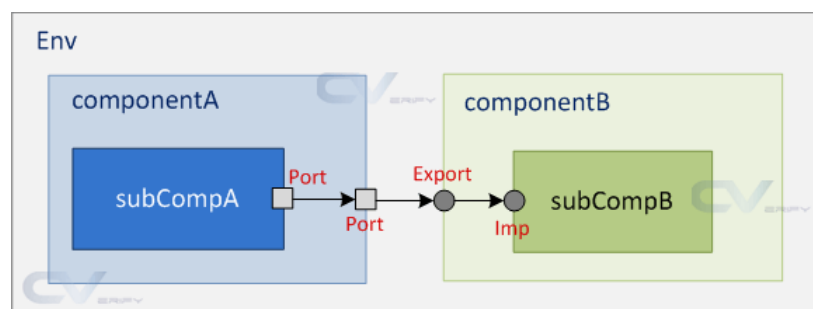
```

- UVM TLM Port to Export to Imp

- UVM TLM ports and exports are also used to send transaction objects across different levels of testbench hierarchy
- Ports shall be used to initiate and forward packets to the top layer of the hierarchy.
- Exports shall be used to accept and forward packets from the top layer to destination
- Implementation ports shall be used to define the **put** method at the target.subCom
- subCompA is trying to send transactions to another subcomponent subCompB. We should allow subCompA to send data to CompA, which forward them to the top layer of the hierarchy. ComponentB shall then accept the transaction and forward it to subCompB.

The connection flows in one direction, from left to right in this image.

We connect subCompA **TO** componentA, componentB **TO** implementation of subCompB. in the test env, we connect componentA **TO** componentB.



```

//this is a class data object that can be sent from one component to another
class Packet extends uvm_object;
    rand bit[7:0] addr;
    rand bit[7:0] data;
    //remember to turn on to utilize UVM macros
    `uvm_object_utils_begin(Packet)
        `uvm_field_int(addr, UVM_ALL_ON)
        `uvm_field_int(data, UVM_ALL_ON)
    `uvm_object_utils_end
    function new(string name = "Packet");
        super.new(name);
    endfunction
endclass

```

```

        endfunction
    endclass

    //this is the subcomponentA which is embedded inside the componentA and env
    class subCompA extends uvm_component;
        `uvm_component_utils(subCompA)
        ...
        uvm_blocking_put_port #(Packet) m_put_port;
        virtual function void build_phase(uvm_phase phase);
            super.build_phase(phase);
            m_put_port = new("m_put_port", this);
        endfunction
        virtual task run_phase(uvm_phase phase);
            phase.raise_objection(this);
            repeat(N) begin
                Packet pkt = Packet::type_id::create("pkt");
                assert(pkt.randomize());
                m_put_port.put(pkg);
            end
            phase.drop_objection(this);
        endtask
    endclass

    //this is the componentA which communicates with the top env and subCompA
    class componentA extends uvm_component;
        `uvm_component_utils(componentA);
        ...
        subCompA m_subcomp_A; //instantiate the subcomponent
        uvm_blocking_put_port #(Packet) m_put_port;
        //create the putport via new and subcomponent via type_id::create in build_phase
        //connection with subCompA
        virtual function void connect_phase(uvm_phase phase);
            super.connect_phase(phase);
            m_subcomp_A.m_put_port.connect(this.m_put_port);
        endfunction
    endclass

    //this is the componentB whilch communicates with the top env and subCompB
    class componentB extends uvm_component;
        ...
        subCompB m_subcomp_B;
        uvm_blocking_put_export #(Packet) m_put_export;
        //create the object and component
        virtual function void connect_phase(uvm_phase phase);
            m_put_export.connect(m_subcomp_b.m_put_imp);
        endfunction
    endclass

    class subCompB extends uvm_component;
        ...
        //mention type of transaction, and type of class that implements the put()
        uvm_blocking_put_imp #(Packet, subCompB) m_put_imp;
        //new() the port
        //implementation of the put() port
        virtual task put(Packet pkt);
            pkt.print(uvm_default_line_printer);
        endtask
    endclass

    //overall test env
    class my_test extends uvm_test;
        ...
        componentA compA;
        componentB compB;
        ...
        //type_id::create in the build_phase
        //connect componentA with componentB
        virtual function void connect_phase(uvm_phase phase);
            compA.m_put_port.connect(compB.m_put_export);
        endfunction
    endclass

```

- UVM TLM Blocking/Nonblocking Get Port

- Any component can request to receive a transaction from another component through a TLM **get** port. The **sending component** should define an implementation of the get port. The implementation let the sender to define what needs to be sent to the requestor. This is opposite of a put port
- The port can be either blocking or nonblocking, which decides whether the get method will block execution in the receiver until the sender sends the object
- To implementation. A class object of type Packet is created, randomized, and sent via the implementation of **get()** method.

```
class componentA extends uvm_component;
...
//create an export to send data to componentB
uvm_blocking_get_imp #(Packet, componentA) m_get_imp;
Packet pkt;
//create the port with new in the build_phase
//create a get task which will output a new packet
virtual task get(output Packet pkt);
    //create a new packet
    pkt = new();
    assert(pkt.randomize());
endtask
endclass
```

A receiver class needs to define a **get** port using **uvm\_blocking\_get\_port** to receive the packet.

```
class componentB extends uvm_component;
...
//create a get_port to request for data from componentA
uvm_blocking_get_port #(Packet) m_get_port;
...
//new() the port in the build_phase
virtual task run_phase(uvm_phase phase);
    Packet pkt;
    phase.raise_objection(this);
    repeat(N) begin
        m_get_port.get(pkt);
    end
    phase.drop_objection(this);
endtask
endclass
```

The connection between a port and its implementation has to be done at a higher level

```
virtual function void connect_phase(uvm_phase phase);
    //connect get port of B to the implementation port of A
    compB.m_get_port.connect(compA.m_get_imp);
endfunction
```

The blocking nature will stall the receiver from resuming until the get task returns, componentB in this case.

- For the nonblocking get, the sender use **try\_get** to to see if the get was successful or **can\_get** method to see if the sender is ready to start a transfer
  - Use **can\_get** as a condition. If **can\_get** returns true, we **try\_get** the packet.
- UVM TLM FIFO(**uvm\_tlm\_fifo**)
  - Assume data rate of the sender is much faster than the rate the receiver can get the packets, a FIFO element is required to store packets. A TLM FIFO is placed in between testbench components that transfer data objects at different rates



- We can connect components via a **TLM FIFO** at a higher level. FIFO's **put\_export** is connected to first component's **put** port and the **get\_export** is connected to the receiver's **get** port.

```

uvm_tlm_fifo #(Packet) m_tlo_fifo;
//in the build_phase, create a FIFO with depth 2
virtual function void build_phase (uvm_phase phase);
    super.build_phase(phase);
    ...
    m_tlm_fifo = new ("uvm_tlm_fifo", this, 2);
endfunction
virtual function void connect_phase (uvm_phase phase);
    compA.m_put_port.connect(m_tlm_fifo.put_export);
    compB.m_get_port.connect(m_tlm_fifo.get_export);
endfunction
virtual task run_phase(uvm_phase phase);
    forever begin
        #10;
        if(m_tlm_fifo.is_full())
            `uvm_info();
    end
endtask

```



- **TLM Analysis Port**
  - The put/get communication typically require a corresponding export to study the implementation. The idea of an **analysis** port is that a component like monitor should be able to generate a stream of transactions regardless of whether there is a target actually connected to it
  - The **uvm\_analysis\_port** is a specialized TLM based class whose interface consists of a single function **write()** and can be embedded within any component. When the component calls **analysis\_port.write()**, it cycles through the list and calls the **write()** method of each connected export. If nothing is connected to it, then it simply does not do anything.
  - The idea of that an analysis port may be connected to 0,1, or many analysis exports and allows a component to call **write()** method without depending on the number of connected exports.
  - Example of sending information to multiple subscribers

Declare, instantiate, and pass the packet via analysis port to other components

```

uvm_analysis_port #(simple_packet) ap;
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    ap = new ("analysis_port", this);
endfunction
//in the run_phase, after randomizing the packet, we call the write method
virtual task run_phase(uvm_phase phase);
    ...
    ap.write(pkt);
    ...
endtask

```

The subscriber should implement the **write()** method. Note: if the subscriber is derived from **uvm\_subscriber**, then it already has an analysis port implementation object predefined by the name **analysis\_export**, hence we don't have to declare another implementation port.

Then we implement the **write()** function

```

class ... #(type T = ...) extends uvm_subscriber #(T)
...
virtual function void write(T t); //the class object name has to be "t"
...
endfunction
endclass

```

The connection should be implemented in the connect\_phase

```

virtual function void connect_phase(uvm_phase phase);
  compA.ap.connect(sub1.analysis_export);
  compA.ap.connect(sub2.analysis_export);
  compA.ap.connect(sub3.analysis_export);
endfunction

```

- TLM Sockets

- TLM 2.0 introduced socket which enables asynchronous bi-directional data transfer between the initiator and target component. A socket is derived from the same base class as ports and export - [uvm\\_port\\_base](#). Components that initiate transactions have initiator sockets and are called [initiators](#), while components that receive transactions have target sockets and are called [targets](#). They can only be connected to each other.
- In the initiator component, the timing annotation argument used in [b\\_transport\(\)](#) method allows the timing points to be offset from simulation times at which the task is called and returned

```

//declare a blocking transport socket(using initiator socket class)
uvm_tlm_b_initiator_socket #(packet) initSocket;
uvm_tlm_time delay;
packet pkt;

//in the build phase, create instances of the socket
virtual function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  initSocket = new("initSocket", this);
  delay = new();
endfunction

//in the run phase, we use the socket to send data
virtual task run_phase (uvm_phase phase);
  repeat (N) begin
    ...
    initSocket.b_transport(pkt,delay);
  end
endtask

```

- In the target socket

```

uvm_tlm_b_target_socket #(target, packet) targetSocket;
//create the instance of the target socket in the build_phase
targetSocket = new ("targetSocket", this);
//provide the implementation method of b_transport in the target class
task b_transport(packet pkt, uvm_tlm_time delay);
  ...
endtask

```

- We can connect them in the higher level hierarchy. We can do it where both initiator and target components are instantiated

```

//create the initiator and target via type_id::create in the build_phase
virtual function void connect_phase(uvm_phase phase);
  init.initSocket.connect(target.targetSocket);
endfunction

```

## 11. UVM Register Layer

- 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 verification(DUV). The register layer defined classes which can be extended to abstract read and write operations to the DUV.
- Most digital design blocks have software controllable registers that can be accessed via a peripheral bus. These registers allow the hardware to behave in certain ways when programmed with certain values. Each register may