



# System Verilog Frameworks™ Scoreboard

*User Guide*

Revision 2.0

---

## Table of Contents

1	Introduction .....	5
2	SVF Scoreboard Functionalities .....	6
2.1	Multi-Stream Posting and Checking .....	6
2.2	TLM Interfaces .....	6
2.3	Transfer Function .....	7
3	SVF Scoreboard Use Model .....	9
3.1	Installing SVF scoreboard .....	9
3.1.1	Including SVF scoreboard files .....	9
3.1.2	Importing pw_scoreboard_pkg .....	9
3.2	Scoreboarding through TLM interfaces .....	9
3.2.1	Use TLM ports, transactions derived from ovm_transaction .....	11
3.2.2	Use TLM ports, transactions derived from pw_sb_transaction .....	11
3.3	Scoreboarding through procedural interfaces .....	11
3.4	Using Transfer function .....	13
3.5	Statistic Report .....	14
3.6	Advanced Scoreboard Features .....	14
3.6.1	Posting Scoreboard Data with Timeout Events .....	14
3.6.2	Posting and Checking with support for data dropping .....	16
3.6.3	Using scoreboard events to trigger testbench activity .....	18
3.6.4	Altering scoreboard report handling via report_handler override .....	18
4	SVF Scoreboard APIs .....	20
4.1	pw_sb_transaction .....	20
4.2	sb_entry .....	20
4.3	sb_item_queue .....	21
4.3.1	Manipulating SVF scoreboard queues: late packet dropping example .....	21
4.4	pw_scoreboard .....	22
4.5	pw_predictor_checker .....	25
5	Xbus Example .....	27
5.1	Overview .....	27
5.2	Steps to configure pw_scoreboard and connect it with xbus testbench .....	27
5.3	Scoreboard Extension for Using Advanced Features with xbus_transfer .....	31
5.4	Tests to run .....	33
5.5	Running the example .....	38

## List of Figures

Figure 2-1 SystemVerilog FrameWorks™ Scoreboard Data Streams .....	6
Figure 2-2 SVF TLM Scoreboard Structure.....	7
Figure 2-3 Predictor_checker Class Used as Predictor on Transmit Side .....	7
Figure 2-4 Predictor_checker Class Used as Checker on Receive Side.....	8
Figure 3-1 Scoreboarding through TLM Interface .....	9
Figure 3-2 Scoreboarding through procedural Interface.....	12
Figure 5-1 Adding SVF scoreboard to XBus testbench .....	27

## List of Tables

Table 4-1 Class pw_sb_transaction .....	20
Table 4-2 Class sb_entry .....	20
Table 4-3 Class sb_item_queue.....	21
Table 4-4 Class pw_scoreboard.....	22
Table 4-5 Class pw_predictor_checker .....	25

## 1 Introduction

Scoreboarding is a fairly straightforward concept used in functional verification environments.

Very simply put, when an event of interest is anticipated by the verification environment, details of that event are posted to the scoreboard. Conversely, when an event of interest is actually observed, it is checked against the events already posted on the scoreboard. The mapping of posted to checked events is called the transfer function, which can range from fairly straightforward to fairly complex. For example, for every posted event, there may be multiple events that are checked and vice versa. The SVF OVM Scoreboard Component implementation is able to handle any transfer function using a combination of inheritance and callback techniques.

The SVF Scoreboard is an OVM Component that is easily incorporated into an OVM Testbench. The following features are supported and the default implementation supports most scoreboarding needs:

- Allows for both in-order and out-of-order checking
- Allows creation of complex DUT-specific transfer functions
- Timeout checking
- Has hooks for error handling
- Is extendable

## 2 SVF Scoreboard Functionalities

### 2.1 Multi-Stream Posting and Checking

Figure 2-1 below shows the basic scoreboard behavior.

The basic `pw_scoreboard` class is derived from the `ovm_scoreboard` class. The `ovm_scoreboard` is an empty built-in base component in OVM library.

Transactions are posted to the scoreboard as instances of classes derived from the `ovm_transaction` class. Similarly, transactions are checked by passing instances of classes derived from `ovm_transaction` to the scoreboard. The actual comparison takes place by calling the `ovm_compare` method of the posted object with the instance of the checked object as an argument.

Further, the scoreboard also supports the notion of an arbitrary number of streams. Objects are posted to and checked against specific streams in the scoreboard. Each stream is identified by a unique number, and objects posted to a given stream are expected to be checked in the same order as they are posted. There is no implied order between objects posted in different streams. Thus, objects can appear in any order with respect to each other if they are posted in separate streams.

In summary, in-order checking is accomplished by posting objects to the same stream. Out-of-order checking is accomplished by posting objects to different streams. Complex transfer functions are accomplished by overloading the transfer virtual function in `pw_scoreboard` with a DUT-specific transfer function.

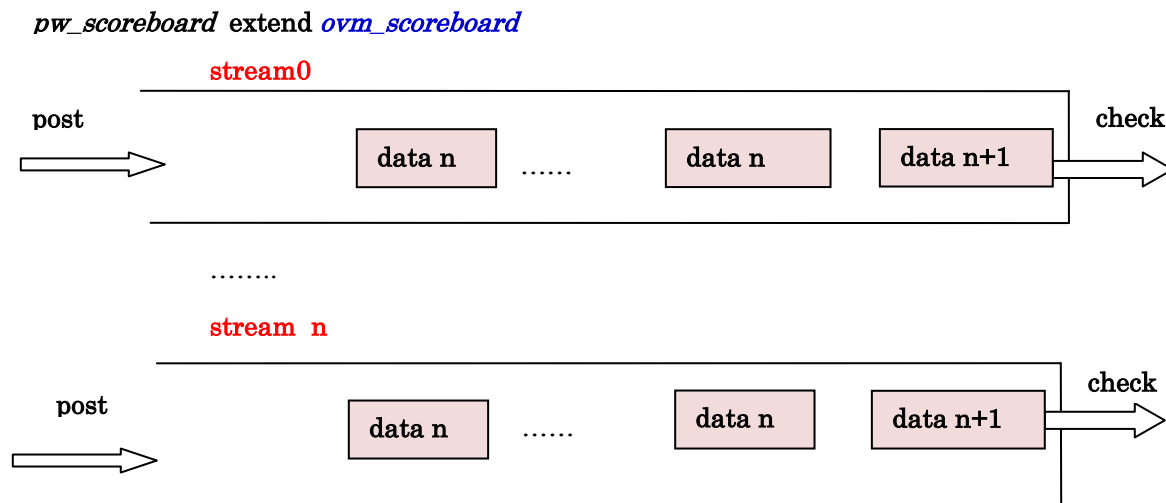


Figure 2-1 SystemVerilog FrameWorks™ Scoreboard Data Streams

### 2.2 TLM Interfaces

SVF scoreboard supports using TLM interface to communicate with other verification components. TLM interface allows users to create more reusable codes. In addition, SVF scoreboard allows data type for posting and checking to be parameterized.

Figure 2-2 illustrates the TLM interface of the SVF scoreboard.

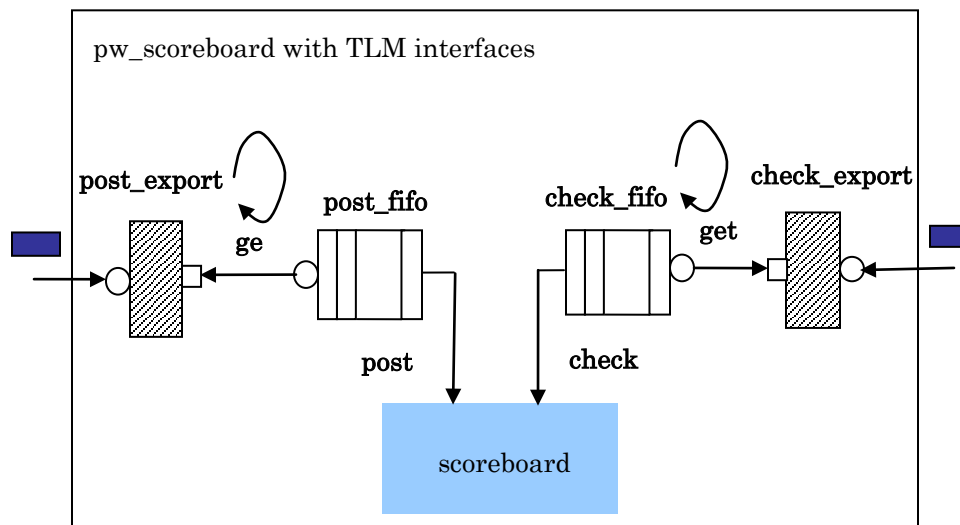


Figure 2-2 SVF TLM Scoreboard Structure

The key members and methods related to the TLM interfaces will be described in section 4.4.

### 2.3 Transfer Function

When data is transmitted from one interface to a different kind of interface, a complex transfer function may be required to represent the relationship between one data type to another data type. Although the transfer function will be very much design specific, it can still be done in a consistent and systematic manner. By providing hooks to allow design specific transfer functions, the scoreboard can be highly reusable.

SVF scoreboard provides a **pw\_predictor\_checker** class to handle various transfer function(s) between data being transmitted and data being received. Figure 2-3 and Figure 2-4 illustrate how the **pw\_predictor\_checker** class can be used in a scoreboarding process.

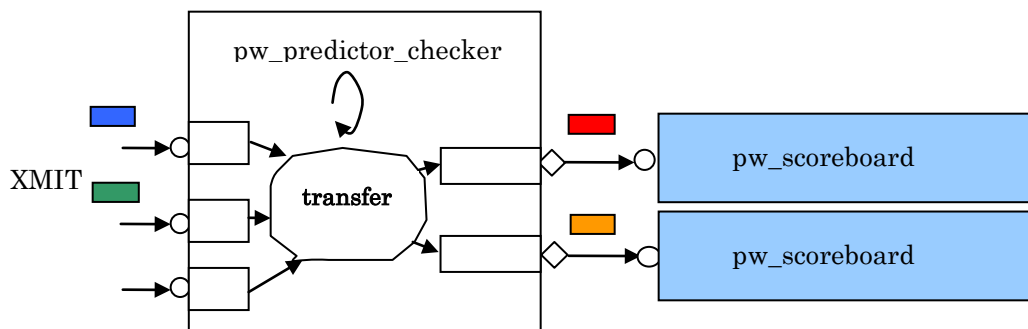
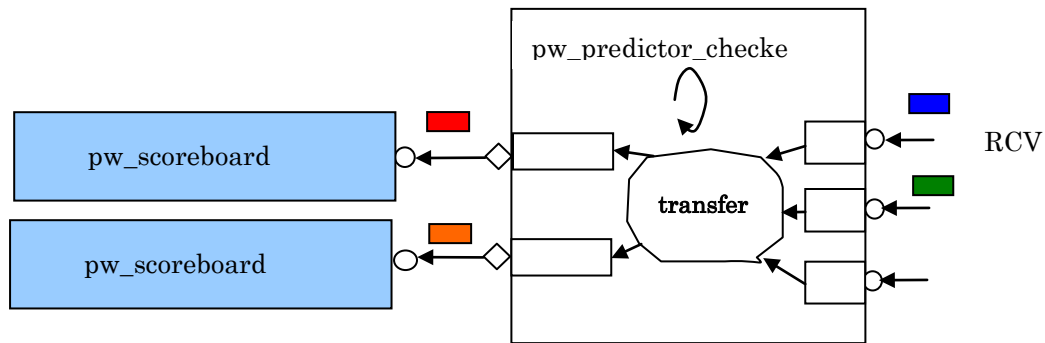


Figure 2-3 Predictor\_checker Class Used as Predictor on Transmit Side



**Figure 2-4 Predictor\_checker Class Used as Checker on Receive Side**

The members and methods of `pw_predictor_checker` class will be detailed in section 4.5.

### 3 SVF Scoreboard Use Model

#### 3.1 Installing SVF scoreboard

##### 3.1.1 Including SVF scoreboard files

The SVF scoreboard files can be included in a testbench using ``include`. For example, after downloading SVF scoreboard files to a `$SVF_SB` directory, in the top level testbench module (e.g., `top`), add the lines in red:

```
module top;
  `include "ovm.svh"
  `include "pw_scoreboard_lib.svh"
```

##### 3.1.2 Importing pw\_scoreboard\_pkg

The SVF scoreboard is also included in a SystemVerilog package, defined in the file `pw_scoreboard_pkg.sv`. Packages are a SystemVerilog construct that provide a declaration space that can be shared by other building blocks. Package declarations can be imported.

The `pw_scoreboard_pkg` can be imported to run with SVF scoreboard. In a top-level testbench module, import the SVF scoreboard packages:

```
module top;
  import ovm_pkg::*;
  import pw_scoreboard_pkg::*;
```

#### 3.2 Scoreboarding through TLM interfaces

Figure 3-1 shows an example of connecting the SVF scoreboard to a testbench via the TLM interfaces. There are two use cases when using TLM interfaces.

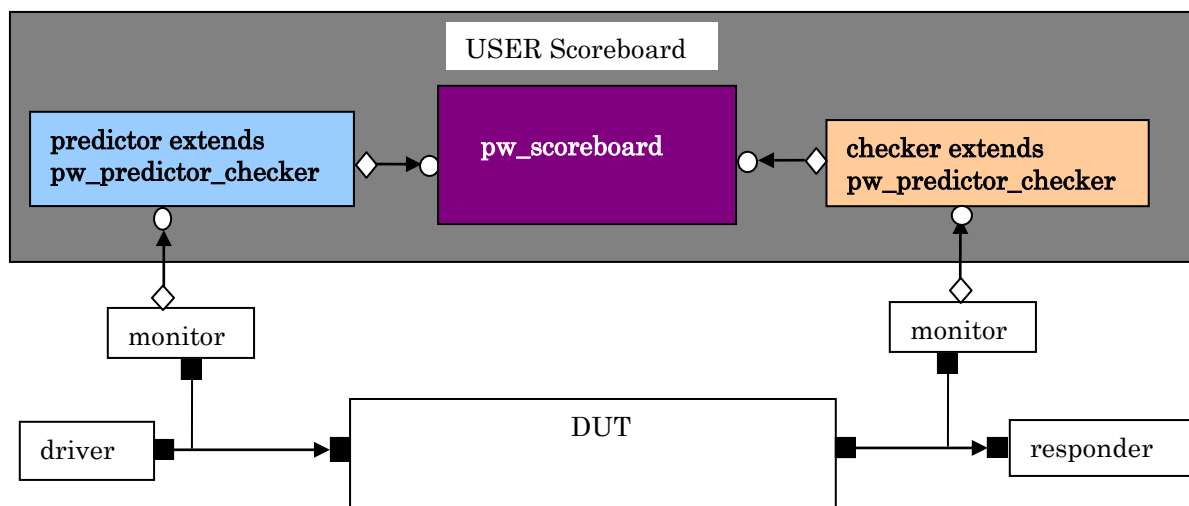


Figure 3-1 Scoreboarding through TLM Interface

In this example, two **pw\_predictor\_checker** class instances are created. The instance that is connected to the monitor at the input side (left) of the DUT works as the predictor. The predictor is responsible for converting the input data type to the expected output data type. The instance that is connected to the monitor at the output side (right) of the DUT works as the checker. The checker is responsible for comparing the observed output data with the expected data on the scoreboard. The predictor connects to the **pw\_scoreboard**'s **post\_export**, the checker connects to the **check\_export** of the scoreboard.

The following shows the example code snippets for setting up the scoreboard as shown in Figure 3-1.

**Step 1.** Declaring a scoreboard and two predictor\_checkers:

```
class xbus_demo_env extends ovm_env;
    // Instantiate a scoreboard. Both input and output data are of type xbus_transfer
    pw_tlm_scoreboard #(xbus_transfer, xbus_transfer) pw_sb;

    // Instantiate a predictor. Number of input ports is '1', number of output
    // ports is '1'. Input and output data types are both xbus_transfer
    pw_tlm_predictor_checker #(1,1,xbus_transfer,xbus_transfer) pw_predictor;

    // Instantiate a checker. Number of input ports is '1', number of output ports
    // is '1'. Input and output data types are both xbus_transfer
    pw_tlm_predictor_checker #(1,1,xbus_transfer,xbus_transfer) pw_checker;

    ...
endclass
```

**Step 2.** Constructing the scoreboard components

```
class xbus_demo_tb extends ovm_env;
    virtual function void build();
        super.build();
        ....
        pw_sb = pw_tlm_scoreboard#(xbus_transfer,xbus_transfer)::type_id::create("pw_sb",this);
        pw_predictor=pw_tlm_predictor_checker#(1,1,xbus_transfer,xbus_transfer)::type_id::create("pw_predictor",this);
        pw_checker=pw_tlm_predictor_checker#(1,1,xbus_transfer,xbus_transfer)::type_id::create("pw_checker",this);
    endfunction

    ....
endclass
```

**Step 3.** Connecting checker/predictor and scoreboard ports:

```
class xbus_demo_tb extends ovm_env;
  function void connect();
    ...
    xbus0.master[0].monitor.item_collected_port.connect(pw_predictor.inp_exports[0]);
    xbus0.slaves[0].monitor.pw_item_collected_port.connect(pw_checker.inp_exports[0]);
    pw_checker.sb_aporsts[0].connect(pw_sb.post_export);
    pw_predictor.sb_aporsts[0].connect(pw_sb.check_export);
  endfunction
  ...
endclass
```

### 3.2.1 Use TLM ports, transactions derived from ovm\_transaction

In this use case, the user is using an [ovm\\_transaction](#) derived class as the basic scoreboard data element. The user provides the functions (defined as virtual functions) `get_stream_id()`, `get_comparer()`, `get_timeout()`, and `get_canDrop()` for the transaction used. These four functions are then used by the scoreboard to define data-item specific `stream_id`, `comparer`, `timeout`, and `canDrop`. Note that the user defined [ovm\\_transaction](#) could contain multiple data types with a key member to distinguish them in systems where mixed transaction types are presented to the scoreboard.

### 3.2.2 Use TLM ports, transactions derived from pw\_sb\_transaction

This use case also uses TLM port connections, but is simplified in that it uses the `pw_sb_transaction` (extended from [ovm\\_transaction](#)) as the scoreboard transaction. The class `pw_sb_transaction` has members `sb_stream_id`, `sb_ev_timeout`, `sb_comparer`, and `sb_canDrop` defined. In this third case, all members of [ovm\\_transaction](#) are available along with the `pw_scoreboard` specific members listed above. Definition of virtual callback functions is not required.

### 3.3 Scoreboarding through procedural interfaces

In certain cases, it may be necessary to access the scoreboard directly without going through the TLM ports/exports. The SVF scoreboard provides `post_sb_data()` and `check_sb_data()` for such purposes. These methods can be called procedurally in the testbench. Although we provide these procedural methods for flexibility, we recommend that users utilize TLM interfaces in order to promote code reuse.

Figure 3-2 shows the testbench setup.

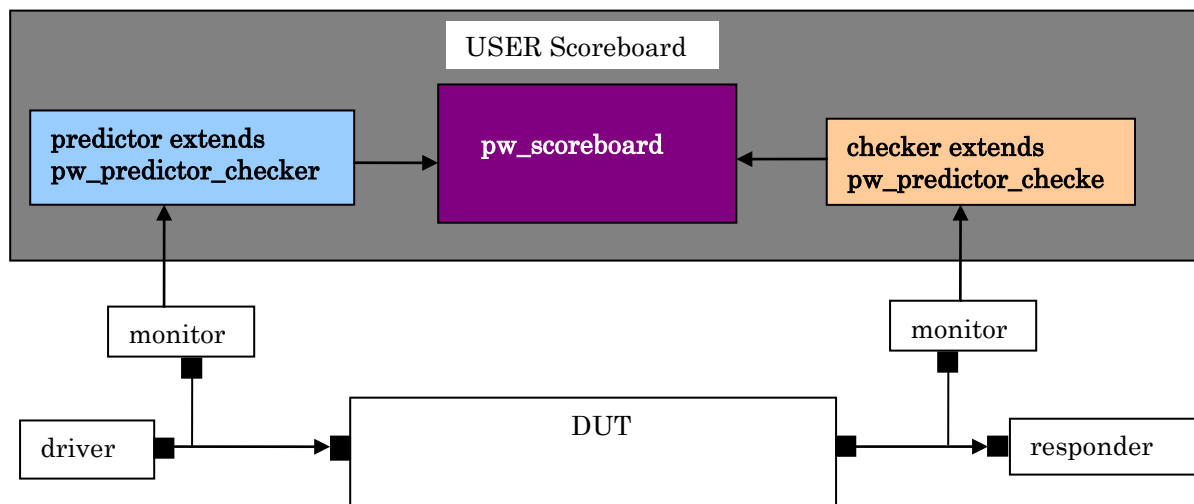


Figure 3-2 Scoreboarding through procedural Interface

The following code examples show the process of using SVF scoreboard's procedural interfaces.

**Step 1. Declaring and constructing a scoreboard with two predictor\_checkers:**

```

class xbus_demo_tb extends ovm_env;
  pw_scoreboard pw_sb;
  pw_predictor_checker #(1,1, xbus_transfer, xbus_transfer) pw_predictor;
  pw_predictor_checker #(1,1, xbus_transfer, xbus_transfer) pw_checker;
  virtual function void build();
    super.build();
    // Constructing xbus components
    // Constructing SVF scoreboard components
    pw_sb = pw_scoreboard#(xbus_transfer,xbus_transfer)::type_id::create("pw_sb",this);
    pw_predictor=pw_predictor_checker#(1,1,xbus_transfer,xbus_transfer)::type_id::create("pw_predictor",this);
    pw_checker=pw_predictor_checker#(1,1,xbus_transfer,xbus_transfer)::type_id::create("pw_checker",this);
  endfunction

  function void connect();
    // Passing SVF scoreboard component handles to the proper monitor components
    xbus0.masters[0].monitor.pw_sb = pw_sb;
    xbus0.masters[0].monitor.pw_predictor = pw_predictor;
    xbus0.slaves[0].monitor.pw_checker = pw_checker;
  endfunction

```

**Step 2. Declaring scoreboard and predictor/checker handles in the monitors**

```
class xbus_master_monitor extends ovm_monitor;
  pw_scoreboard pw_sb;
  pw_predictor_checker #(1,1,xbus_transfer,xbus_transfer) pw_predictor;
  `ovm_component_utils_begin
    `ovm_field_object(pw_sb, OVM_REFERENCE)
    `ovm_field_object(pw_predictor, OVM_REFERENCE)
  `ovm_component_utils_end
endclass

class xbus_slave_monitor extends ovm_monitor;
  pw_predictor_checker #(1,1,xbus_transfer,xbus_transfer) pw_checker;
  `ovm_component_utils_begin
    `ovm_field_object(pw_checker, OVM_REFERENCE)
  `ovm_component_utils_end
endclass
```

### Step 3. Posting directly via procedural code:

```
// In xbus_master_monitor. 'tr' is the xbus_transfer going into the predictor
// 'otr' is the xbus_transfer going into the scoreboard
xbus_transfer tr, otr;

// Convert input data type to expected output data type if needed
pw_predictor.transfer(tr, otr);

// Post to scoreboard stream 0
pw_sb.post_sb_data(otr, 0);
...
```

### Checking a transaction via procedural call:

```
// In xbus_slave_monitor. 'rcvd' is the received xbus_transfer. 'otr' is the
// xbus_transfer to be compared with the scoreboard
xbus_transfer rcvd, otr;

// If received transfer doesn't match expected data type, do next line,
// otherwise, skip the next line
pw_checker.transfer(rcvd, otr);

// Check received data (stream_id=0)
pw_sb.check_sb_data(otr, 0);
```

## 3.4 Using Transfer function

The **pw\_predictor\_checker** class provides a built-in virtual method **transfer()** to handle different input and output data type. By default, input and output data type of the **transfer()** method are the same. The following code snippet

shows the default `transfer()` method. Actual transfer functions may be quite complex and dependent upon mirrored images of DUT state and multiple modes or configurations.

```
// Transfer function that processes the arrived transaction
// Also specifies the port id at which it arrived
virtual task transfer(T_INP trans, int port_id);
    ovm_report_message("pw_predictor_checker", $psprintf("SB: transfer: %d\n", port_id));
    // Do what you need to do with trans
    // Push it to the SB port (port_id % NUM_SB) by default
    ovm_report_message("pw_predictor_checker:", $psprintf("%d port_id", port_id));
    sb_aporports[port_id % NUM_SB].write(trans);
endtask // transfer
```

### 3.5 Statistic Report

SVF scoreboard provides methods to report the statistics of the scoreboard, such as, how many data has been posted on the scoreboard, how many data has been checked, how many outstanding data are left on the scoreboard. The following example shows scoreboard reporting method is called in the `report()` phase of the simulation.

```
class xbus_demo_tb extends ovm_env;
    function void report();
        pw_sb.report_sb(1,1); // (chk_outstanding, rpt_outstanding)
    endfunction
```

### 3.6 Advanced Scoreboard Features

Besides the basic functionalities shown in sections 3.2 to 3.5, the SVF scoreboard also supports timeout, data dropping and so on. This section will show some examples of using these SVF scoreboard advanced features.

In the case of using TLM interfaces for scoreboarding, the posting and checking method calls are built into the `run()` process. Therefore users will not be able to pass arguments directly to these methods. To accomplish the advanced features described in this section, `pw_scoreboard` must be extended to provide implementations for the virtual methods `get_stream_id`, `get_timeout`, `get_comparer`, and `get_canDrop` as needed for the intended usage. Where there is no need for the functionality associated with these methods, no method implementation is required.

#### 3.6.1 Posting Scoreboard Data with Timeout Events

The following set of code snippets show the user code to set up timeout events associated with the posting of transactions to `post_sb_data()`.

##### Step 1. Define a task to fork a thread to delay and trigger an ovm\_event upon timeout

Note that this task must be defined within a class to ensure use of automatic variables. Usually this will be within the class in the testbench that is doing the post to the scoreboard. When using TLM interfaces for scoreboarding, this task may be defined within the scoreboard itself. In the testbench from which this example was drawn, that was a monitor class.

```

task delay_to_trigger_timeout(ovm_event ev_timer);
  fork
    begin
      ovm_report_info("", "In delay_to_trigger_timeout, prior to delay...");
      #3000 ; // a very simple fixed delay
      ev_timer.trigger();
      ovm_report_info("", "In delay_to_trigger_timeout, just triggered
event...");
    end
  join_none
endtask // delay_to_trigger_timeout

```

**Step 2. Declare ovm\_event handle for use in attaching an ovm\_event to a transaction as it is posted:**

```
ovm_event ev_timer ;
```

**Step 3. Create an OVM\_event and post it to the Scoreboard:**

The following code snippet shows how the monitor creates an ovm\_event via new and posts it to the scoreboard. The example uses a cloned copy of the generated transaction and uses stream\_id '0' in the call to post\_sb\_data().

```

// Clone transaction 't'
$cast(t_cpy,t.clone());
ev_timer = new ;
ovm_report_info("", "Posting to Scoreboard...");

// post the transaction with stream_id=0 and ovm_event
pw_sb.post_sb_data(t_cpy,0,ev_timer);

// call task to fork timeout thread for this transaction
delay_to_trigger_timeout(ev_timer);

```

When the timeout is set to less than the time it takes to transit the scoreboard, or when the packet is mistakenly dropped, allowing the timeout to trigger, the scoreboard will report an error similar to this:

```
# OVM_ERROR @ 1108: ovm_test_top.pwr_demo_sve0.pw_sb[0] [] Timed out on event : for
transaction:a:1 p:2 r:10 l: 10 [16_b9_74_64_fc_cc_c9_b3_b4_fc] parity : 0x1e
```

The user may also employ OVM's event pool mechanism to name timeout events and store them within an ovm\_event\_pool. An identifying string may be defined in any suitable header file:

```
`define EV_USER_TIMEOUT "EV_USER_TIMEOUT"
```

In this case, `ovm_event` and `ovm_event_pool` handles for use in attaching an `ovm_event` to a transaction as it is posted are declared:

```
ovm_event ev_timer ;
ovm_event_pool ev    p ;
```

In this more complex usage, `ovm_event_pool.get(EV_USER_TIMEOUT)` would be used instead of `new()` to instance `ev_timer`.

### 3.6.2 Posting and Checking with support for data dropping

There are three distinct facilities for allowing dropping during the `check_sb_data()` phase. They are:

- setting the `canDrop` argument in `post_sb_data()`
- overriding the virtual function `get_canDrop()` to determine dropping
- using `droppable_cnt` to set an allowable number of dropped entries

The following subsections show these mechanisms and the relationship between them.

#### 3.6.2.1 Marking posted scoreboard data with `canDrop` argument.

The SVF scoreboard method `post_sb_data()` allows marking an individual `sb_entry` to be droppable or non-droppable based upon the `canDrop` argument. The following two examples of `post_sb_data()` calls illustrate using this facility.

```
$cast(t_cpy, t.clone());    // prepare a copy of transaction t
// post t to scoreboard, with stream_id == 0 and canDrop set
pw_sb.post_sb_data(t_cpy, 0, 1); // (data, stream_id, canDrop)
```

#### 3.6.2.2 Posting Scoreboard Data Using `get_canDrop()` - Establish Drop

By inheriting from the class `pw_scoreboard`, user code can overload the `get_canDrop()` method and provide a general method to determine whether an `sb_entry` is droppable at the time it is checked. This general method could use logic to test DUT state or, perhaps, to get past a time window associated with system reset.

The following set of code snippets show the user code to overload the `get_canDrop()` method, and employ it in a testbench. Here we declare a class that inherits from `pw_scoreboard`.

```

//-----
// Class: acme_pw_scoreboard
//-----
class acme_pw_scoreboard extends pw_scoreboard;
  `ovm_component_utils_begin(acme_pw_scoreboard)
  `ovm_field_int(disable_scoreboard,OVM_ALL_ON)
  `ovm_field_int(post_cnt,OVM_ALL_ON)
  `ovm_field_int(check_cnt,OVM_ALL_ON)
  `ovm_field_int(droppable_cnt,OVM_ALL_ON)
  `ovm_component_utils_end

  extern function new(string name="acme_pw_scoreboard",
                      ovm_component parent=null);

  super.new(name, parent);
  post_cnt = 0;
  check_cnt = 0;
  ev_pool = new( {name , "_EV_POOL"} );
endfunction // new

virtual function int get_canDrop(ovm_transaction posted);
  if($time < 200ns) begin
    ovm_report_info("", "In overloaded get_canDrop, about to return 1.");
    return 1;
  end
  else begin
    ovm_report_info("", "In overloaded get_canDrop, about to return 0.");
    return 0;
  end
endfunction
endclass : acme_pw_scoreboard

```

Note that the call to **get\_canDrop()** is made in the method **check\_sb\_data()**, so this method of dropping determines dropability at check time.

### 3.6.2.3 Setting a droppable\_cnt for a Scoreboard

The pw\_scoreboard data field droppable\_cnt is used at check time to determine if a checked sb\_entry is droppable. If droppable\_cnt is >0, the sb\_entry is droppable. Every time an sb\_entry on a particular scoreboard is checked, the droppable\_cnt is decremented.

### 3.6.2.4 Relationship Between the Three Methods of Controlling Droppability.

The logical relationship between the three methods of determining if a particular sb\_entry is droppable is simple and shown in the following pseudo code:

```

if (canDrop is set in sb_entry) the entry is droppable.
else if (get_canDrop() returns 1 at check time) the entry is droppable.
else if( droppable_cnt > 0 at check time) the entry is droppable.
else the entry is not droppable.

```

### 3.6.3 Using scoreboard events to trigger testbench activity

The scoreboard events can be used to allow user testbench code to wait for these events.

In the following example, an event is fetched from the scoreboards event pool and the testbench waits for it to trigger.

```

ovm_event sbMatchEv;

// Get the Match event from the scoreboard pool
sbMatchEv = customer_demo_sve0.sb[0].ev_pool.get(^EV_SB_MATCHED);

// Wait for a check match
sbMatchEvEv.wait_pttrigger();

// Got the event.
// Do match user processing.

```

In the second example, the user code fetches a mismatch event from the event pool associated with a particular transaction. Given an ovm\_transaction, t, posted to the scoreboard via post\_sb\_data(), we pend upon a mismatch event for that particular ovm\_transaction.

```

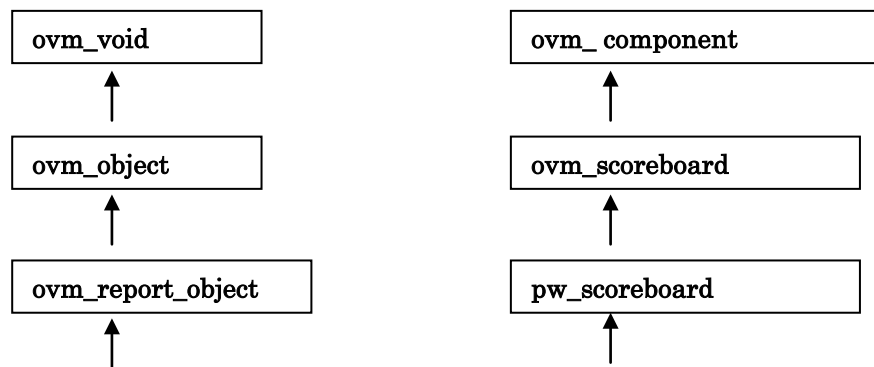
ovm_transaction t;
ovm_event ev;
ovm_event_pool evp;

// Get the event pool for the transaction
evp = t.get_event_pool();
ev = evp.get(^EV_SB_MISMATCHED);
ev.wait_pttrigger();

```

### 3.6.4 Altering scoreboard report handling via report\_handler override

The following diagram of class inheritance relationships for OVM is useful in understanding this example of overriding report handling. Note that **pw\_scoreboard** class inherits from the [ovm\\_report\\_object](#) class.



This example assumes that the verification engineer wishes to coerce `ovm_report_error()` calls for a particular ID within a particular scoreboard instance to take action as if they were `ovm_report_fatal()`. This might be motivated by the desire to limit run time on failing tests in a regression suite for one class of errors.

The tactic employed is to use `set_report_severity_id_action()` method of `ovm_report_object` to override the default action for `ovm_report_error()`. Note that `ovm_report_handler` usually has a one-to-one relationship with `ovm_report_object` and `pw_scoreboard` inherits from class `ovm_report_object`.

```

pwr_demo_sve0.pi_sb[0].set_report_severity_id_action(
    OVM_ERROR, // ovm_severity
    "check_sb_data:", // id
    OVM_DISPLAY | OVM_EXIT // ovm_action
);
  
```

The code above establishes, for the particular instance `pi_sb[0]` of `pw_scoreboard`, that all `ovm_report_error()` calls with id of “check\_sb\_data:” will have their `ovm_action` set to `OVM_DISPLAY | OVM_EXIT`, which is the default for `ovm_report_fatal()`. The effect is to have errors for this particular (id, severity) finish the simulation.

It may be useful to read the sections of the OVM Reference Guide for `ovm_report_handler`, `ovm_report_object`, and global variables to understand the relationship between `pw_scoreboard` class and the classes from which it inherits.

This is just a simple example taken from the many variations on using inherited methods to modify scoreboard behavior.

## 4 SVF Scoreboard APIs

This section lists the detailed SVF scoreboard classes, their members and methods.

### 4.1 pw\_sb\_transaction

This is a transaction class extends from ovm\_transaction. The **pw\_sb\_transaction** class has built-in method to determine the stream ID.

**Table 4-1 Class pw\_sb\_transaction**

Property or Method	Description
int sb_stream_id	Which stream this item belongs to
ovm_event sb_ev_timeout	Event that indicates that data is checked after timeout period
ovm_comparar sb_comparer	Policy to use for data comparison
int sb_canDrop	Indicate this data can be dropped
function new(string name="", ovm_component initiator=null)	Class constructor

### 4.2 sb\_entry

This class is used internally by SVF scoreboard. It contains the transaction and the associated sideband information, such as, events. Normally users do not need to operate on this class directly.

**Table 4-2 Class sb\_entry**

Property or Method	Description
string name	Name of the class object
ovm_transaction data	Handle to the transaction item of this scoreboard entry
ovm_comparer comparer	Handle to the compare policy
int CanDrop	Indicate the scoreboard entry is droppable
ovm_event tmout_ev	Indicate the scoreboard entry checking timed out
function new(string name= "sb_entry", ovm_transaction data=null, ovm_comparer comparer=null, int canDrop=0, ovm_event tmout_ev=null, ovm_component initiator=null)	Class constructor
virtual function string convert2string()	Display scoreboard entry information

Property or Method	Description
function void createAndAddEvents	Create and add events to event pool
function bit comp(sb_entry rhs,ref string rpt)	Compare scoreboard data against received data

### 4.3 sb\_item\_queue

This class is used internally by SVF scoreboard. It keeps a queue of sb\_entry objects for a particular stream.

Normally users do not need to operate on this class directly.

**Table 4-3 Class sb\_item\_queue**

Property or Method	Description
string name	Name of the object
sb_entry item_queue[\$]	A queue of sb_entry objects
int streamId	Stream ID of this queue
function new(string name= "", int streamed)	Class constructor
function string get_name()	Return the name of this object

For special user defined applications where users need to manipulate the data items of data queues on a scoreboard, the following example shows how to use SVF scoreboard provided API to modify the scoreboard queues.

#### 4.3.1 Manipulating SVF scoreboard queues: late packet dropping example

Often in a verification effort, the DUT will exhibit behavior that is correct, but the exact sequence of outputs is difficult to predict at the time of stimulus generation. The following function, drop\_pkt\_from\_sb, illustrates how SVF scoreboard queues can be manipulated using `get_sb_queue()` and `set_sb_queue()` to drop specific packets. This function might be useful in a testbench where a monitor detects a planned FIFO overflow in the DUT, where packets must be dropped from the scoreboard to reflect the overflow.

```

// *****
// *** drop_pkt_from_sb ***
// *****
// Drop a pkt of data from scoreboard (usually due to FIFO overflow
// in the DUT - or other error condition)
function void fifo_fill_test::drop_pkt_from_sb(int stream_id,
        ref pw_scoreboard Sb,
        ref sb_entry drop_pkt );

    sb_entry t_q[$] ;
    // get queue of stream_id from scoreboard
    Sb.get_sb_queue(stream_id, t_q);

    // find and delete drop_pkt from temp queue
    for (int i=0; i< t_q.size(); i++) begin
        if(t_q[i].comp(drop_pkt)) begin
            t_q.delete(i) ; // delete entry matching drop_pkt
            ovm_report_message("drop_pkt_from_sb", $psprintf("Dropping pkt form sb queue $d",
                stream_id) ) ;
        end
    end
end
// set sb queue to temp queue
Sb.set_sb_queue(stream_id, t_q);

endfunction:drop_pkt_from_sb

```

#### 4.4 pw\_scoreboard

**pw\_scoreboard** class extends from `ovm_scoreboard`. This class contains TLM interfaces as well as procedural access methods to the scoreboard, i.e., `post_sb_data()` and `check_sb_data()`.

**Table 4-4 Class pw\_scoreboard**

Property or Method	Description
bit disable_scoreboard	Disable the scoreboard
bit display_post_data	When set, print the details of the data being posted
bit display_check_data	When set, print the details of the data being checked
bit display_exp_act	When set, print the details of the expected and the actual data
sb_item_queue sb_entry_queue[int]	Array of sb_entry queues
int post_cnt[int]	Number of posted data to each stream
int check_cnt[int]	Number of data checked for each stream

Property or Method	Description
int drop_cnt[int]	Number of data dropped from each stream
int timeout_cnt[int]	Number of timeouts encountered for each stream
ovm_event_pool ev_pool	Event pool of the scoreboard
int droppable_cnt	Number of droppable data
ovm_analysis_export #(T_POSTED) <b>post_export</b>	Analysis export to be connected with main data posting port. Data type is parameterized
ovm_analysis_export #(T_CHECKED) <b>check_export</b>	Analysis export to be connected with main data checking port. Data type is parameterized
function new(string name= "pw_scoreboard", ovm_component parent=null);	Class constructor
function virtual void <b>post_sb_data</b> ( ovm_transaction t, int stream_id=0, ovm_event tmout_ev = null, ovm_comparer cmp = null, int canDrop = 0 );	<p><b>t:</b> Expected value to be posted</p> <p><b>stream_id:</b> The default stream is 0, but a different queue is allocated for each unique stream.</p> <p><b>tmout_ev:</b> Timeout event associated with the scoreboard transaction.</p> <p>This tmout_ev is used by the scoreboard as follows. The argument tmout_ev is used to match an event to the posted data. The calling user code may create a separate thread used to trigger the tmout_ev. If the event is triggered before the pw_scoreboard checks the sb_data for the posted transaction, then the scoreboard issues an error. In the simplest case, the tmout_ev will be triggered by user code after a fixed timeout, which would correspond to the maximum expected interval between posting and checking. More complex use cases are possible. User code could trigger the tmout_ev based on some logical combination of events or conditions that set an expected bound on the arrival of the checked transaction.</p> <p><b>cmp:</b> Comparator associated with the scoreboard transaction.</p> <p><b>cmp</b> allows the user to set a distinct compare mechanism for each sb_data item that is posted to the scoreboard. Comparers may be set for each of several different classes of posted sb_data items. For one class of sb_data item, all fields in the item could be compared. For a second class of item, only selected fields might be compared. In the most general case, a separate comparer could be assigned to each sb_data item, on an item-by-item basis.</p>

Property or Method	Description
	<p><b>canDrop:</b> set to 1 implies that the posted packet may be a candidate for being dropped.</p> <p>For example, the DUT may be stimulated by a reset sequence while sb_data items A, B, C are in flight. The reaction of the DUT to the reset is difficult to determine exactly, but the arrival of either {A, B, C} or {A, C} is good DUT behavior. In this case, marking sb_data B as canDrop would allow the scoreboard to check either sequence of items and accept them as correct.</p>
<pre>function void check_sb_data(   ovm_transaction t,   int stream_id=0)</pre>	<p><b>t:</b> Information used to check posted data in the order they are posted in the associated stream</p> <p><b>stream_id:</b> Associated stream</p> <p>This method is used to supply the DUT response data or transaction to be checked, which usually comes from a monitor. In a testbench not using TLM transaction ports, check_sb_data() would be called procedurally from user testbench code. A null transaction in the input argument results in an error report. Compare mismatches, droppable transactions, and unexpected transactions are all handled by this method.</p>
<pre>function void cleanup_droppables();</pre>	Remove all remaining droppable transactions from scoreboard.
<pre>function bit outstanding();</pre>	Check if unchecked transactions remain on scoreboard. Returns 1 if outstanding transactions remain, 0 if not.
<pre>function void report_outstanding();</pre>	Print detailed information on outstanding scoreboard data
<pre>function void report_sb(   bit chk_outstanding=1,   bit rpt_outstanding=1)</pre>	Remove all remaining droppable data. Print number of data posted, number of data checked, number of data timed out, and number of data dropped for each stream. If enabled print detailed outstanding data information and issue an error for non-zero outstanding data
<pre>virtual function void reset_stream(   int stream_id);</pre>	Clears (empties) the queue for a particular stream_id. Resets check_cnt[stream_id]. All other counts remain unchanged.
<pre>virtual function void reset_all_streams();</pre>	Clears (empties) all the queues for each stream_id.
<pre>virtual function get_posted_streams(   ref int stream[\$]);</pre>	Get the stream_id for each queue.

Property or Method	Description
function void <b>get_sb_queue</b> ( int stream_id, ref sb_entry out_q[\$]);	If a queue exists for stream_id, return a copy of the queue with its queue items in out_q[\$], else report an OVM warning.
function ovm_transaction <b>get_sb_data</b> ( int stream_id, int item_idx);	If a queue exists for stream_id, and it has an entry for index item_idx, return the queue entry indexed by item_idx.
function void <b>export_connections</b> ()	OVM built-in method to make exports and implementations of subcomponents visible externally.
task <b>run</b> ()	OVM built-in method
virtual function int <b>get_stream_id</b> (ovm_transaction t)	Tells the scoreboard how to obtain the stream id of a given transaction. Called by <b>run()</b> for all transactions not derived from <b>pw_sb_transaction</b> . Default implementation returns 0.
virtual function ovm_comparer <b>get_comparer</b> (ovm_transaction t)	Tells the scoreboard how to associate a comparison policy for the given transaction. Called by <b>run()</b> for all transactions not derived from <b>pw_sb_transaction</b> . Default implementation returns null object.
virtual function int <b>get_timeout</b> (ovm_transaction posted)	Tells the scoreboard how many units to wait for a match (or mismatch) event to take place before a timeout is generated for the given transaction. Called by <b>run()</b> for all transactions not derived from <b>pw_sb_transaction</b> . Default implementation returns null object.
virtual function int <b>get_canDrop</b> ( ovm_transaction posted)	Tells the scoreboard how to determine whether a given transaction can be dropped. Called by <b>run()</b> for all transactions not derived from <b>pw_sb_transaction</b> . Default implementation returns 0.

#### 4.5 pw\_predictor\_checker

The **pw\_predictor\_checker** class allows a user-defined transformation of data to take place in a testbench component that is distinct from and feeds the **pw\_scoreboard**. The essential model for most testbenches is a DUT which is some variation of a multiple-input/ multiple-output device with associated transformation(s) of the input streams to the output streams. Transformations may be simple or complex.

**Table 4-5 Class pw\_predictor\_checker**

Property or Method	Description
ovm_analysis_export #(T_INP) <b>inp_exports</b> [NUM_INP]	Analysis exports to be connected with main port(post or check). Data type is parameterized
ovm_analysis_port #(T_SB) <b>sb_apor</b> [NUM_OUTP]	Analysis ports to connect with scoreboard export. Data type is parameterized

---

Property or Method	Description
function <b>new</b> ( string name, ovm_component parent=null);	Create a new instance of the predictor_checker class.
virtual task <b>transfer</b> ( T_INP trans, int port_id T_SB)	Transfer method processes the arrived transaction and specified the port id at which it arrived. DUT-specific transfer transformations are done here. Default action would be to push the transaction to the SB port (port_id % NUM_SB).
function void <b>export_connections</b> ()	Internal implementation. Hooks the main ports with corresponding post/check exports and internal fifos.
task <b>run</b> ()	Main process of fetching data from xmit/rcv side, conducting transfer function and sending to scoreboard

## 5 Xbus Example

### 5.1 Overview

XBus OVC from OVM distribution is used as an example to demonstrate the usage of `pw_scoreboard`. For detail about XBus OVC, please refer to OVM user guide.

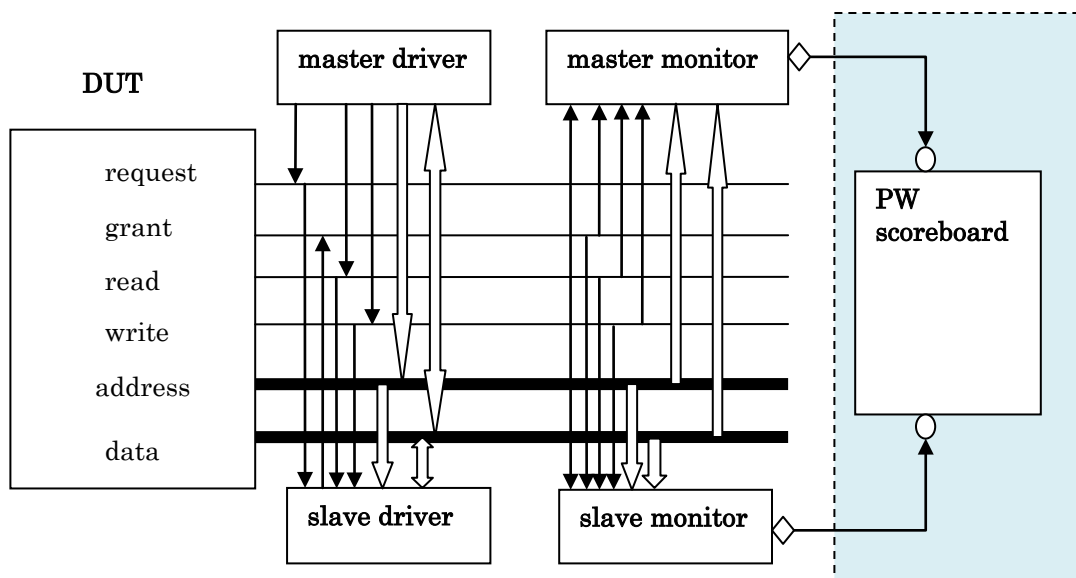
Since XBus example does not use SystemVerilog packages, we use ``include` to bring in the SVF scoreboard classes to the XBus testbench as shown in the Step 1 code snippet.

The `pw_scoreboard` has two analysis port: `post_export` and `check_export`. `Post_export` is used to post transaction to scoreboard. `Check_export` is used to check the expected transaction against the posted transactions in the scoreboard.

The Xbus master monitor has an analysis port that can provide transactions that post to scoreboard. The Xbus slave monitor has an analysis port that can provide transactions to check against the posted transactions.

The predictor takes transactions from Xbus master monitor and sends the transactions to `pw_scoreboard` via `post_export` port. The Checker takes the transactions from Xbus slave monitor and sends the transactions to `pw_scoreboard` via `check_export`. Those transactions should match as both Xbus master monitor and slave monitor are on the same bus.

Figure 5-1 illustrates how the SVF scoreboard is added to the XBus testbench.



**Figure 5-1 Adding SVF scoreboard to XBus testbench**

Note that in the example provided, a DUT specific extension of `pw_scoreboard`, defined in class `xbus_pw_scoreboard`, is used. This is not inherently necessary, but is provided to demonstrate usage of advanced features of `pw_scoreboard` with a user defined transaction class which is not derived from `pw_sb_transaction`. It should be understood that, subsequent to Step 1 below, references to `xbus_pw_scoreboard` could be replaced by `pw_scoreboard` were such DUT-specific usage is not required.

### 5.2 Steps to configure `pw_scoreboard` and connect it with `xbus` testbench.

#### Step 1: `xbus_tb_top.sv`

Include `pw_scoreboard.sv` within module `xbus_tb_top`. Also, include the extended scoreboard, `xbus_pw_scoreboard.sv`

```
`define XBUS_ADDR_WIDTH 16

`include "dut_dummy.v"
`include "xbus_if.sv"

module xbus_tb_top;

    `include "ovm.svh"
    `include "pw_scoreboard_lib.svh"
    `include "xbus_pw_scoreboard.sv"
    `include "xbus.svh"
    `include "test_lib.sv"
```

### Step 2: `xbus_demo_tb.sv`

- a. Add `xbus_pw_scoreboard`, `pw_predictor` and `pw_checker` to `xbus_demo_tb` class.
- b. In build function, instantiate the `xbus_pw_scoreboard`, `pw_predictor` and `pw_checker` by calling the correspondent create method.
- c. In connect function, connect the master monitor with `pw_predictor`'s input port, connect the slave monitor with `pw_checker`'s input port. Also connect the `pw_predictor` sb port to `xbus_pw_scoreboard`'s post port and connect the `pw_checker` sb port to `xbus_pw_scoreboard`'s check port.
- d. Finally in report stage, do scoreboard clean up and print out a scoreboard report.

```

class xbus_demo_tb extends ovm_env;
.....
// xbus environment
xbus_env xbus0;

// PW scoreboard to check transaction between masters and slaves
xbus_pw_scoreboard #(xbus_transfer, xbus_transfer) pw_tlm_sb;
pw_predictor_checker #(1,1,xbus_transfer,xbus_transfer) pw_predictor;
pw_predictor_checker #(1,1,xbus_transfer,xbus_transfer) pw_checker;
.....
// build
virtual function void build();
super.build();
.....
pw_sb = xbus_pw_scoreboard #(xbus_transfer,
                             xbus_transfer)::type_id::create("pw_sb", this);

pw_predictor=pw_predictor_checker#(1,1,xbus_transfer,
                                   xbus_transfer)::type_id::create("pw_predictor", this);

pw_checker =pw_predictor_checker#(1,1,xbus_transfer,
                                   xbus_transfer)::type_id::create("pw_checker",this);
endfunction : build

function void connect();
.....
xbus0.masters[0].monitor.item_collected_port.connect(pw_predictor.inp_exports[0]);
xbus0.slaves[0].monitor.pw_item_collected_port.connect(pw_checker.inp_exports[0]);
pw_predictor.sb_aporsts[0].connect(pw_sb.post_export);
pw_checker.sb_aporsts[0].connect(pw_sb.check_export);
.....
endfunction : connect

function void report();
sb_cleanup();
endfunction : report

function void sb_cleanup();
pw_tlm_sb.report_sb(1,1); // (chk_outstanding,rpt_outstanding)
endfunction // void

endclass : xbus_demo_tb

```

### Step 3: Modifications to the xbus OVC

Usually the above two steps are sufficient to connect to the pw\_scoreboard. But due to the fact that master monitor and slave monitor are seeing the same transaction on the same bus at the same time in the xbus example, a delay in slave monitor is deliberately added to make sure that transaction arrived at check port is always later than the corresponding transaction arrived at check port in scoreboard. The delay is made configurable for use in demonstrating timeout functionality.

```
class xbus_slave_monitor extends ovm_monitor;
.....
// This bit is used to enable transaction lost to send to pw_scoreboard.
bit pw_item_lost_enable = 0;

// This controls how many cycles to delay transactions before sending to
// pw_scoreboard for check
int pw_check_send_delay = 1;

// This is the analysis port same as item_collected_port, except that the
// writing to the port is delayed a little bit, to make sure the slave
// post to pw_scoreboard a little bit later than master.
ovm_analysis_port#(xbus_transfer) pw_item_collected_port;

// The following property holds the transaction information currently
// begin captured (by the collect_address_phase and data_phase methods).
protected xbus_transfer trans_collected;
protected xbus_transfer trans_delayed;

// event to make sure the posting of transaction from slave to scoreboard is always
// later then transaction from mater.
protected event data_phase_done;

// new - constructor
function new(string name, ovm_component parent=null);
.....
pw_item_collected_port = new("pw_item_collected_port", this);
.....
endfunction : new

// run phase
virtual task run();
.....
    forever begin
        @data_phase_done;
        repeat(pw_check_send_delay)
            @(posedge xsi.sig_clock); // add one or more cycles delay

        if(pw_item_lost_enable) begin
            //only 50% chance to pass the data to port.
            if({$random} % 2 ==0)
                pw_item_collected_port.write(trans_delayed);
            end else
                pw_item_collected_port.write(trans_delayed);
        end
    end
end
.....
endtask : run
```

```

// collect_transactions
virtual protected task collect_transactions();
  forever begin
    ....
    item_collected_port.write(trans_collected);
    $cast(trans_delayed, trans_collected.clone());
    -> data_phase_done;
  end
  ...
endtask : collect_transactions

```

### 5.3 Scoreboard Extension for Using Advanced Features with xbus\_transfer

To provide a basic demonstration of the advanced features of **pw\_scoreboard** in conjunction with the `xbus_transfer` transaction class, a simple extension of **pw\_scoreboard** is declared to provide the following:

- Configurable inclusion of timeout support with a simple, timeout from time posted mechanism
- Configurable multi-stream management of transactions, with separate streams for read and write transactions
- Configurable declaration of transactions as droppable

The necessary code is shown following.

```

class xbus_pw_scoreboard
#(type T_POSTED=ovm_transaction,
  type T_CHECKED=ovm_transaction)
  extends pw_scoreboard #(T_POSTED, T_CHECKED);

  // This bit controls inclusion of timeout events for foreign transactions
  bit pw_enable_timeout = 0;

  // This field controls how many ticks to wait before declaring a timeout
  // Value is measured from time of posting.
  int pw_timeout_value = 10;

  // This bit determines whether transactions are lumped into a single stream,
  // or assigned to streams based on transaction type. This applies strictly
  // to xbus_transfer transaction types.
  bit pw_enable_stream_by_type = 0;

  // This bit determines whether to tolerate drops or not
  bit pw_enable_drop_without_error = 0;

  `ovm_component_param_utils_begin(xbus_pw_scoreboard #(T_POSTED,T_CHECKED))
  `ovm_field_int(pw_timeout_value,OVM_ALL_ON)
  `ovm_field_int(pw_enable_timeout,OVM_ALL_ON)
  `ovm_field_int(pw_enable_stream_by_type,OVM_ALL_ON)
  `ovm_field_int(pw_enable_drop_without_error,OVM_ALL_ON)
  `ovm_component_utils_end

```

```
// Create an instance
function new(string name="xbus_pw_scoreboard", ovm_component parent=null);
    super.new(name, parent);
endfunction: new

// Post the expected transaction t to the stream tagged stream_id
// If a tmout_ev is provided, fork off a delay
virtual task post_sb_data(ovm_transaction t, int stream_id=0, ovm_event tmout_ev = null,
ovm_comparer cmp = null, int canDrop = 0);
    super.post_sb_data(t, stream_id, tmout_ev, cmp, canDrop);
    if (tmout_ev != null) begin
        fork
            begin
                #pw_timeout_value;
                tmout_ev.trigger();
            end
        join_none
    end
endtask: post_sb_data

// Tells the scoreboard how to obtain the stream id of a given foreign transaction.
//
virtual function int get_stream_id(ovm_transaction t);
    xbus_transfer chk_t;
    get_stream_id = 0;

    if ((pw_enable_stream_by_type == 1) && $cast(chk_t, t))
        get_stream_id = chk_t.read_write;

endfunction

// Tells the scoreboard to process timeouts for a given foreign transaction.
//
virtual function ovm_event get_timeout(ovm_transaction posted);
    ovm_event xbus_to_ev;
    get_timeout = null;
    if (pw_enable_timeout) begin
        xbus_to_ev = new("XBUS_PW_TO");
        get_timeout = xbus_to_ev;
    end
endfunction

// Tells the scoreboard whether a given foreign transaction is acceptable to drop.
//
virtual function int get_canDrop(ovm_transaction posted);
    get_canDrop = pw_enable_drop_without_error;
endfunction

endclass : xbus_pw_scoreboard
```

## 5.4 Tests to run

### Test 1: test\_read\_modify\_write

Where three transactions were sent from master and the three transactions are posted and matched in pw scoreboard.

```
// Read Modify Write Read Test
class test_read_modify_write extends xbus_demo_base_test;

    `ovm_component_utils(test_read_modify_write)

    function new(string name = "test_read_modify_write", ovm_component parent=null);
        super.new(name,parent);
    endfunction : new

    virtual function void build();
        // Set the default sequence for the master and slave
        set_config_string("xbus_demo_tb0.xbus0.masters[0].sequencer",
            "default_sequence", "read_modify_write_seq");
        set_config_string("xbus_demo_tb0.xbus0.slaves[0].sequencer",
            "default_sequence", "slave_memory_seq");
        // Create the tb
        super.build();
    endfunction : build

endclass : test_read_modify_write
```

#### **Result:**

```
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: -----
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: Posted to Stream 0->0: 3
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: Checked in Stream 0->0: 3
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: -----
```

### Test 2: pw\_test\_read\_modify\_write

Same as the first test except that configuration `pw_item_lost_enable` is set in slave monitor to cause transactions to be missing. As a result, the pw scoreboard will report outstanding transactions still in scoreboard at end of the test. To change the test, just update the Makefile and set the `TESTNAME` to be `pw_test_read_modify_write`.

```

// Read Modify Write Read Test with pw_item_lost_enable set.
class pw_test_read_modify_write extends xbus_demo_base_test;

.....
virtual function void build();
.....
    // Set the slave monitor to randomly drop transactions before
    // sending to scoreboard.
    set_config_int("xbus_demo_tb0.xbus0.slaves[0].monitor",
                  "pw_item_lost_enable", 1);

    // Set scoreboard configs
    set_config_int("xbus_demo_tb0.pw_sb", "disable_scoreboard", 0);
    set_config_int("xbus_demo_tb0.pw_sb", "display_post_data", 1);
    set_config_int("xbus_demo_tb0.pw_sb", "display_check_data", 1);
    set_config_int("xbus_demo_tb0.pw_sb", "display_exp_act", 0);

    // Create the tb
    super.build();
endfunction : build

endclass : pw_test_read_modify_write

```

**Result:**

```

[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: -----
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: Posted to Stream 0->0: 3
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: Checked in Stream 0->0: 2
*** OVM: DUT error at time 2000
Checked at line 0 in <unknown>
In ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard
Outstanding elements seen
*** Error: A DUT error has occurred

[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: 1 outstanding elements
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: -----
-----
Name                               Type                               Size                               Value
-----
xbus_transfer_inst                  xbus_transfer                      -                                @6017
addr                                integral                            16                               'ha881
read_write                           xbus_read_write_en+                32                               READ
size                                  integral                            32                               'h1
data                                  da(integral)                        1                                -
  [0]                                 integral                            8                                'hff
wait_state                           da(integral)                        0                                -
error_pos                             integral                            32                               'h0
transmit_delay                       integral                            32                               'h0
master                               string                               10                               masters[0]
slave                                 string                               0                                -
begin_time                           time                                 64                               310
end_time                              time                                 64                               350
-----

[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: -----

```

**Test 3: pw\_test\_r8\_w8\_r4\_w4**

This test sends a sequence of four transactions, interleaving two reads and two writes. The test increases the delay in the xbus\_slave\_monitor from one cycle to two, and defines a timeout value equivalent to one cycle.

```

class pw_test_r8_w8_r4_w4 extends test_r8_w8_r4_w4;

  `ovm_component_utils(pw_test_r8_w8_r4_w4)

  function new(string name = "pw_test_r8_w8_r4_w4", ovm_component parent=null);
    super.new(name,parent);
  endfunction : new

  virtual function void build();
    // bump the delay cycle count in xbus monitor
    set_config_int("xbus_demo_tb0.xbus0.slaves[0].monitor", "pw_check_send_delay", 2);
    // set timeout value to equivalent of one cycle
    set_config_int("xbus_demo_tb0.xbus_pw_scoreboard", "pw_timeout_value", 10);
    // Create the tb
    super.build();
  endfunction : build

endclass : pw_test_r8_w8_r4_w4

```

Result:

```

[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: -----
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: Posted to Stream 0->0: 4
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: Checked in Stream 0->0: 4
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: -----

```

**Test 4:** pw\_test\_r8\_w8\_r4\_w4\_to

This test extends the previous test to enable timeouts. Since the timeout value is specified at a value less than the two cycle xbus\_slave\_monitor's delay, all transactions should be expected to fail.

```

// Large word read/write test (w/ timeouts enabled)
class pw_test_r8_w8_r4_w4_to extends pw_test_r8_w8_r4_w4;

  `ovm_component_utils(pw_test_r8_w8_r4_w4_to)

  function new(string name = "pw_test_r8_w8_r4_w4_to", ovm_component parent=null);
    super.new(name,parent);
  endfunction : new

  virtual function void build();
    // enable timeout checks
    set_config_int("xbus_demo_tb0.xbus_pw_scoreboard", "pw_enable_timeout", 1);
    // Create the tb
    super.build();
  endfunction : build

endclass : pw_test_r8_w8_r4_w4_to

```

Result:

```

[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: -----
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: Posted to Stream 0->0: 4
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: Checked in Stream 0->0: 4
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: Timeout in Stream 0->0: 4
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: -----

```

**Test 5:** pw\_test\_r8\_w8\_r4\_w4\_ms

This test extends the basic pw\_test\_r8\_w8\_r4\_w4 test to demonstrate multi-stream management of the scoreboard. Read and write transactions are posted and checked in separate streams.

```
// Large word read/write test (w/ multi-stream scoreboarding enabled)
class pw_test_r8_w8_r4_w4_ms extends pw_test_r8_w8_r4_w4;

  `ovm_component_utils(pw_test_r8_w8_r4_w4_ms)

  function new(string name = "pw_test_r8_w8_r4_w4_ms", ovm_component parent=null);
    super.new(name,parent);
  endfunction : new

  virtual function void build();
    // enable timeout checks
    set_config_int("xbus_demo_tb0.xbus_pw_scoreboard", "pw_enable_stream_by_type", 1);
    // Create the tb
    super.build();
  endfunction : build

endclass : pw_test_r8_w8_r4_w4_ms
```

Result:

```
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: -----
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: Posted to Stream 0->1: 2
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: Checked in Stream 0->1: 2
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: Posted to Stream 0->2: 2
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: Checked in Stream 0->2: 2
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: -----
```

**Test 6:** pw\_test\_r8\_w8\_r4\_w4\_d

This test extends the basic pw\_test\_r8\_w8\_r4\_w4 test to enable dropping of transactions by the xbus\_slave\_monitor. It should be expected to fail do to dropped transactions.

```
// Large word read/write test (w/ drops)
class pw_test_r8_w8_r4_w4_d extends pw_test_r8_w8_r4_w4;

    `ovm_component_utils(pw_test_r8_w8_r4_w4_d)

function new(string name = "pw_test_r8_w8_r4_w4_d", ovm_component parent=null);
    super.new(name,parent);
endfunction : new

virtual function void build();
    // Set the slave monitor to randomly drop transactions before sending to scoreboard.
    set_config_int("xbus_demo_tb0.xbus0.slaves[0].monitor", "pw_item_lost_enable", 1);
    // Create the tb
    super.build();
endfunction : build

endclass : pw_test_r8_w8_r4_w4_d
```

Result:

```
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: -----
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: Posted to Stream 0: 4
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: Checked in Stream 0: 2
*** OVM: DUT error at time 2000
Checked at line 0 in <unknown>
In ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard
Outstanding elements seen
*** Error: A DUT error has occurred

[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: 2 outstanding elements
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: -----
-----
Name                               Type                               Size                               Value
-----
xbus_transfer_inst                 xbus_transfer                      -                                @6033
  addr                             integral                            16                               'h4008
  read_write                       xbus_read_write_en+               32                               WRITE
  size                             integral                            32                               'h4
  data                             da(integral)                       4                                -
    [0]                             integral                            8                                'hc7
    [1]                             integral                            8                                'h65
    [2]                             integral                            8                                'hfd
    [3]                             integral                            8                                'hf4
  wait_state                       da(integral)                       0                                -
  error_pos                        integral                            32                               'h0
  transmit_delay                   integral                            32                               'h0
  master                           string                               10                               masters[0]
  slave                            string                               0
  begin_time                       time                                64                               990
  end_time                         time                                64                               1120
-----
```

. . . .

**Test 7:** pw\_test\_r8\_w8\_r4\_w4\_cd

This test extends the preceding test to declare all posted transactions as droppable. The test should now pass.

```
// Large word read/write test (w/ drops legalized)
class pw_test_r8_w8_r4_w4_cd extends pw_test_r8_w8_r4_w4_d;

    `ovm_component_utils(pw_test_r8_w8_r4_w4_cd)

    function new(string name = "pw_test_r8_w8_r4_w4_cd", ovm_component parent=null);
        super.new(name,parent);
    endfunction : new

    virtual function void build();
        // enable timeout checks
        set_config_int("xbus_demo_tb0.xbus_pw_scoreboard", "pw_enable_drop_without_error", 1);

        // Create the tb
        super.build();
    endfunction : build

endclass : pw_test_r8_w8_r4_w4_cd
```

**Result:**

```
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: -----
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: Posted to Stream 0->0: 4
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: Checked in Stream 0->0: 2
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: Dropped in Stream 0->0: 2
[2000] hier=ovm_test_top.xbus_demo_tb0.xbus_pw_scoreboard: -----
```

## 5.5 Running the example

See instructions in PACKAGE\_README.txt file.