

# OVM Register Package User Guide

<b>OVM REGISTER PACKAGE USER GUIDE .....</b>	<b>1</b>
CONTENTS .....	1
INTRODUCTION .....	1
USING THE REGISTER VERIFICATION ENVIRONMENT .....	2
OVERVIEW OF THE EXAMPLE .....	3
THE TOP LEVEL .....	4
<i>The top module</i> .....	4
<i>The test environment</i> .....	5
<i>The automatic test connector</i> .....	6
THE RTL AND RTL WRAPPER .....	7
DEFINING THE REGISTERS .....	8
CANONICAL EXAMPLE .....	8
<i>Defining register objects</i> .....	9
<i>Defining a register file</i> .....	11
<i>Defining a design specific register package</i> .....	14
<i>Defining a register map</i> .....	15
PUTTING IT ALL TOGETHER.....	16
APPENDIX A – THE EXAMPLE .....	18
<i>Requirements</i> .....	18
RUNNING THE EXAMPLE .....	18
<i>Easy install – parallel to an existing OVM installation</i> .....	18
<i>Separate install – point at the OVM installation</i> .....	18
<i>Separate install – point at the built-in OVM installation</i> .....	19
<i>The General compilation and simulation flow</i> .....	19
APPENDIX B – SOURCE CODE – AUTO-GENERATED CODE .....	20
APPENDIX C – SOURCE CODE – TOP LEVEL .....	24
<i>Building a register testcase</i> .....	27
<i>Building a register aware driver</i> .....	28
<i>Building a register aware monitor</i> .....	29
<i>Building a register aware scoreboard</i> .....	30

While OVM is known as the de-facto methodology for efficient verification environments, the requirements for verification productivity continues to grow.

One example of this requirement is managing and controlling registers in SOC. Not only is the register count high but the relationship between operation modes defined by registers values are very complex.

This document outlines how to use the OVM register functionality available in the beta version of the register package.

We would like to hear from you, please email any questions or feedback that you might have to [ovm\\_utils@mentor.com](mailto:ovm_utils@mentor.com)

The register verification environment can be used in many ways – generally outlined in Figure 1. You must create the DUT (the system under test), and the Bus Driver – the transactor responsible for going between bus transactions and bus pin wiggles. Additionally, you must supply the register map for the system.

This user guide discusses a simple example which demonstrates using the OVM Register Package in this way.

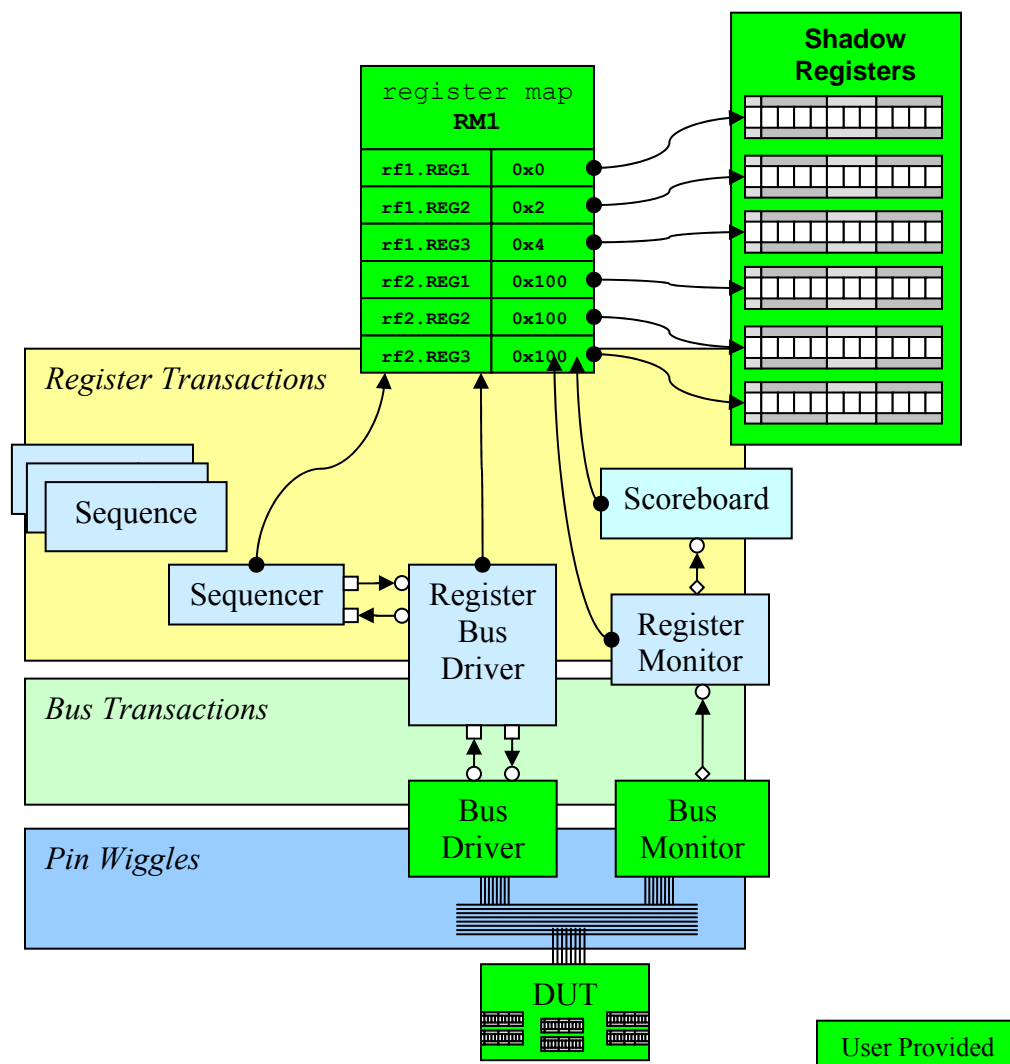


Figure 1 - General Register Verification Environment

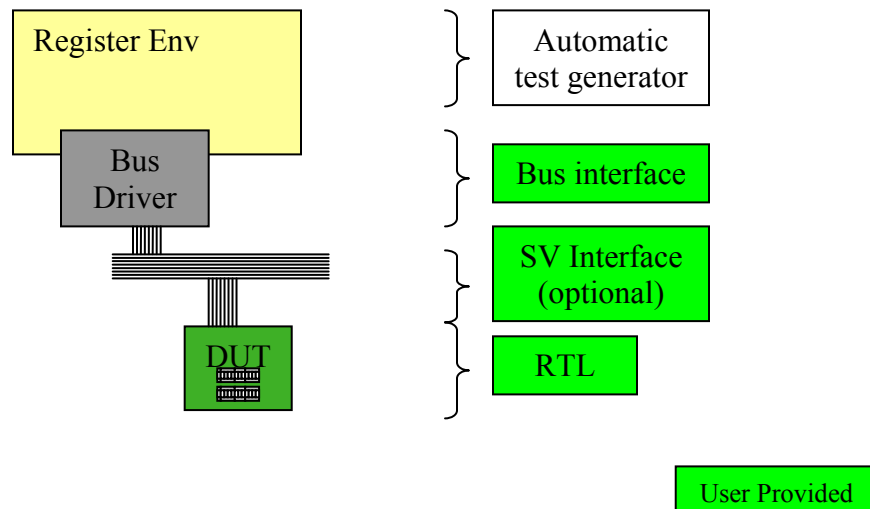
The example discussed in this document is a small, synthetic example which will demonstrate how to use the register package to do automatic test creation and verification.

The example is a small design with a collection of wires managed as a SystemVerilog interface. The interface is connected to the register test system, and tests can be automatically run.

The contents of the example are roughly what a user needs to provide in order to use the register facility. By providing these basics, automated testing can be achieved.

```
00_stopwatch/  
00_stopwatch/auto/stopwatch_register_pkg.sv  
00_stopwatch/dut/stopwatch_rtl.sv  
00_stopwatch/dut/stopwatch_rtl_wrapper.sv  
00_stopwatch/makefile  
00_stopwatch/t.sv  
00_stopwatch/vsim.do
```

The example provides the top level – t.sv, the RTL – dut/stopwatch\_rtl.sv and a SystemVerilog interface wrapper around the RTL – dut/stopwatch\_rtl\_wrapper.sv, and the actual register definitions auto/stopwatch\_register\_pkg.sv. This last file can be either generated automatically from a register description, or can be created by hand. In either case, it should describe the registers and the address map used in the design.



**Figure 2 - Simple example - block diagram**

The remainder of this document will go through each source code file, discussing relevant details to provide enough information so that the techniques can be adapted to the user design.

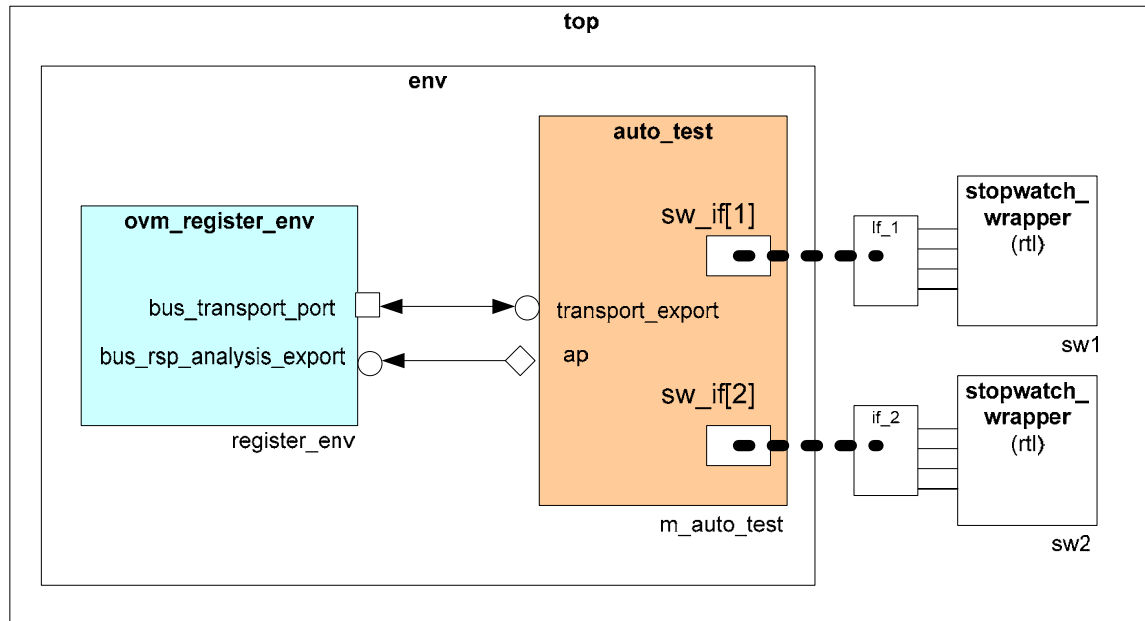


Figure 3 - Top Level Block Diagram

## The top module

The top level is the code that the verification engineer must write. It is responsible for instantiating the hardware, connecting the hardware to the register tests, and getting everything started.

In our example, the following top level module instantiates the hardware, the SystemVerilog interfaces and the test environment.

```
module top;
    bit clk = 0;

    // Our hardware
    // There are two instances of the hardware, and
    // each has its own interface.
    stopwatch_if sw_if_1(clk);
    stopwatch_if sw_if_2(clk);

    // The hardware instances.
    stopwatch_wrapper sw1(sw_if_1);
    stopwatch_wrapper sw2(sw_if_2);

    // The testbench environment
    my_env env = new("env", null);

    initial begin
        // Connect the class based testbench
        // to the hardware.
        env.assign_vif(sw_if_1, sw_if_2);
    end
endmodule
```

```

        // Run the test
        run_test();
    end

    // Hardware clock
    always
        #10 clk = ~clk;
endmodule

```

## The test environment

Once the top module is built, it starts running the test environment using “run\_test()”. This will cause the environment to have its phases called.

In this test environment we build two sub-components. We instantiate a register test environment – the thing that will cause register traffic to happen, and we instantiate a “gasket” named “auto\_test” to allow the generated register traffic to be translated into traffic on the SystemVerilog interfaces.

Our test environment has a function named “assign\_vi()” which takes two arguments – the two virtual interfaces that will get connected. Our environment knows it is testing a “dual” environment, and it passes those virtual interface connections to the auto\_test later in the connect() phase.

```

//
// The environment
// contains
//     1. a register environment
//     2. a translator between class-based
//         transactions and pin wiggles
//
class my_env
    extends ovm_env;

    ovm_register_env register_env;

    auto_test #(bus_request, bus_response)
        m_auto_test;

    function new(string name, ovm_component p);
        super.new(name, p);
        register_env = new("register_env", this);
        m_auto_test = new("auto_test", this);
    endfunction

    ...

    // Called from the top-level module to
    // connect to the real interface.
    function void assign_vi (
        virtual interface stopwatch_if sw_if_1,
        virtual interface stopwatch_if sw_if_2);
        m_auto_test.assign_vi(sw_if_1, sw_if_2);
    endfunction

    function void connect();
        register_env.bus_transport_port.connect(
            m_auto_test.transport_export);
        m_auto_test.ap.connect(

```

```

    register_env.bus_rsp_analysis_export);
endfunction

function void report();
    ovm_register_map register_map;
    // Some general debug routines.
    m_auto_test.print();
    register_env.m_register_map.display_address_map();
endfunction
endclass

```

## The automatic test connector

When using the automatic tests, there are many ways to connect, but the easiest is to extend the base class library element named ‘ovm\_register\_auto\_test’.

The ovm\_register\_auto\_test allows a user to implement the do\_operation() task, and do some simple connections. The do\_operation() task is expected to receive the request, process it on the “bus” and return a response. There are many other ways to interact with the automatic tests, but this transport channel version is a simple one. The other responsibility of the user of ovm\_register\_auto\_test, is to define a way that the request and response can interact with the system being tested. In our example the system we are testing is a pair of virtual interfaces. do\_operation() gets the request and checks the operation. If a READ, then the address is passed to a read() routine in the proper interface. If a WRITE, then the address and the data are passed to a write routine in the proper interface.

```

//
// Custom built code to translate between classes
// and pin wiggles
//
// bus_request --> DUT pin wiggles --> bus_response
//
class auto_test
    #(type REQ = ovm_sequence_item,
      type RSP = ovm_sequence_item)
    extends ovm_register_auto_test #(REQ, RSP);

    function new(string name, ovm_component p);
        super.new(name, p);
    endfunction

    //
    // Step 1: Define how this object connects
    //          to the DUT
    //
    virtual interface stopwatch_if sw_if[1:2];

    function void assign_vif(
        virtual interface stopwatch_if sw_if_1,
        virtual interface stopwatch_if sw_if_2);
        sw_if[1] = sw_if_1;
        sw_if[2] = sw_if_2;
    endfunction

    //
    // Step 2: Define how a request and response
    //          is handled. And how the
    //          request and response is interfaced
    //          with the DUT.

```

```

//
task do_operation(REQ req, output RSP rsp);
    int chunk;
    bit[31:0] mapped_address;

    // Create a response.
    rsp = new();
    rsp.copy_req(req);
    rsp.set_id_info(req);
    rsp.status = PASS; // Default is PASS.

    ovm_report_info("AutoTest MEMREQ", req.convert2string());

    // Do the simple address mapping here.
    // Each 1000 bytes is one chunk
    //
    // Note: Each virtual interface above manages
    //       one chunk. So each virtual interface
    //       hangs a device for an address space
    //       of 'h1000
    chunk = req.address / 'h1000;
    mapped_address = req.address % 'h1000;
    assert((chunk==1) || (chunk==2)) else
        $fatal("chunk failed");

    // Actually "execute" the transaction.
    case (req.op)
        READ: sw_if[chunk].read(mapped_address, rsp.data);
        WRITE: sw_if[chunk].write(mapped_address, req.data);
    endcase
    ovm_report_info("AutoTest MEMRSP", rsp.convert2string());
endtask
endclass

```

The RTL is simply Verilog RTL, with a pin level interface. The RTL wrapper is a SystemVerilog “interface” that wraps the pin-level RTL, and enables communication with the RTL via the interface and helper functions contained in the interface.

The contents of the RTL are not important for this document, but the system-level address map is important.

Register handing for functional verification is supported through the OVM Register package. The package contains two kinds of object categories:

- 1) Register objects - Objects used to define and capture a register instance, and
- 2) Register components - Objects used to manage the register objects.

This separation enables the verification environment and ultimately the test writer, to define testcases based on register views instead of a bus level view.

In order to build a verification environment where the abstraction is at the register level we need to follow these simply steps to build the register information:

- 1) Describe each registers types
  - a. Define register data fields,
  - b. Define field access,
  - c. Define field relationships
- 2) Define register objects
  - a. Specialize the parameterized (templated) register base class with a bit vector or structure representing the bit fields defined in 1a) above.
  - b. Define functional coverage for the register
  - c. Define random constraints for the register.
- 3) Define register file
  - a. Map the registers to specific addresses, usually in a “register file”.
  - b. Map register files together into a larger “register map” describing one or more address spaces.
  - c. Define random constraints for the relationships between registers.

All code snippets used in this document are based on the stopwatch example included with the register package. The example uses a typical OVM structure with scenarios to build the register aware transactions and shows the different aspects outlined in this document, from the register definitions, to the register based stimuli and analysis. See the appendix section for documentation of the stopwatch example.



## Defining register objects

In this example we have two kinds of registers: a register where the data is basically a bit vector and a register where the data is partitioned into fields.

In order to define a register, we must first define the type of the data contained in the register – in this example we have two types – `bit32_t` and `stopwatch_csr_t`. Once we define these basic types, we can use them along with the `ovm_register` parameterized class.

```
typedef bit[31:0] bit32_t;

typedef struct packed {
    bit [3:0] stride;
    bit updown;
    bit upper_limit_reached;
    bit lower_limit_reached;
} stopwatch_csr_t;
```

These types define the types that we need for the register specialization.

To define a register with the data values of type bit vector we would do:

```
class stopwatch_lower_limit extends ovm_register #(bit32_t);
```

To define a register with the data values of fields we would do:

```
class stopwatch_csr extends ovm_register #(stopwatch_csr_t);
```

This approach enables us to access the register data values as whole bit vector or as field indexed by the struct. This is becomes valuable in many cases, e.g. for bus level access and coverage.

If we need to capture functional coverage for a register field, all we need is to add an embedded coverage group in our register type. In addition, to enable sampling of the functional coverage we define the `sample()` method for the register type:

```
class stopwatch_csr extends ovm_register #(stopwatch_csr_t);
...
    covergroup c;
        stride: coverpoint data.stride;
        updown: coverpoint data.updown;
        upper_limit_reached: coverpoint data.upper_limit_reached;
        lower_limit_reached: coverpoint data.lower_limit_reached;
    endgroup

    function new(...);
        c = new();
    ...
endfunction

function void sample();
    c.sample();
endfunction
```

Often we need to define the access rights or permissions of a register i.e. which field are readable and writable. This is most easily done using the built-in masks in the `ovm_register` class.

Consider the case where a register of type `stopwatch_csr_t`, it is not allowed to write to the bit's `upper_limit_reached` and `lower_limit_reached`.

Recall, the type is defined as :

```
typedef struct packed {
    bit [3:0] stride;
    bit updown;
    bit upper_limit_reached;
    bit lower_limit_reached;
} stopwatch_csr_t;
```

We would need to define the WMASK as follows:

```
class stopwatch_csr extends ovm_register #(stopwatch_csr_t);

    function new(string name, ovm_component p);
        super.new(name, p);

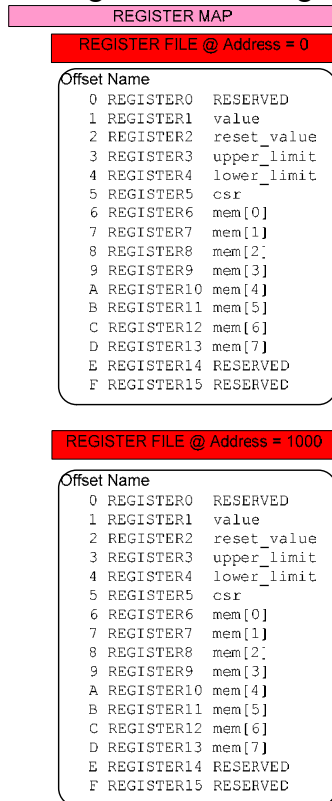
        // All bits are writable
        // except the upper and lower limits reached.
        WMASK = 'b1111_1_0_0;
    endfunction
```

The bit mask is applied when the register data is assessed using the write and read `ovm_register` methods. The '0' bits in the WRITE mask mean that those bits are not writable. The `ovm_register` package comes with built-in masks and behavior, but those can be easily changed or extended as needed by the register designer.

## Defining a register file

With the register objects defined, we can create a link from a register name, register type and the address for a register instance; this is done using the `ovm_register_file` class.

The register file and register map we are building are shown in the figure below:



Our goal is to build a look-up mechanism that given either an address or a name can find the register at that address or with that name.

For example if we like to have one instance of a `stopwatch_csr` register and multiple instances of a `stopwatch_memory` register we declare a register file as follows:

```
class stopwatch_register_file extends ovm_register_file;
...
    stopwatch_csr      stopwatch_csr_reg;
    stopwatch_memory    stopwatch_memory_reg[8];
...
endclass
```

As the `ovm_register_file` is-a `ovm_component` the constructor is identical to any other `ovm_component`, simply call the parent constructor with the provide name and handle to the parent:

```
class stopwatch_register_file extends ovm_register_file;
...
    function new(string name, ovm_component parent);
        super.new(name, parent);
    endfunction
endclass
```

```

    endfunction
endclass

```

The `stopwatch_register_file` now contains handles to the registers which must be constructed. In the build phase of our register file we need to construct all the register instances and add the registers to the register file.

We define the register name, the parent handle i.e. the register file, and an optional default value, for reset, for this register instance:

```

class stopwatch_register_file extends ovm_register_file;
...
function void build();
    stopwatch_csr_reg = new("CSR", this, 'b1);
    foreach (stopwatch_memory_reg[i])
        stopwatch_memory_reg[i] = new($sprintf("MEM[%0d]", i), this);
    ...
endfunction

```

This will create an instance of the `stopwatch_csr_reg` type with the instance name CSR and define the reset value to be all ones.

By using a foreach loop we create a set of register instances of the `stopwatch_memory_reg` type; the instances are named MEM[0], MEM[1] etc. Since we didn't define any default reset value, the `ovm_register` default result value is used, i.e. all bit are set to zero when method `reset()` are called.

In order to insert a register instance into the register file we use the `add_register()` method.

The arguments for `add_register()` are a string for the register identification, the address where this register is mapped to, and the handle for the register instance that shall be added to the register file:

```

class stopwatch_register_file extends ovm_register_file;
...
function void build();
    // -----
    // Construct the registers
    // -----
    ...
    // -----
    // Add the registers to the register file
    // -----
    add_register("CSR", 5, stopwatch_csr_reg);
    add_register("MEM[0]", 6, stopwatch_memory_reg[0]);
    add_register("MEM[1]", 7, stopwatch_memory_reg[1]);
    ...
endfunction

```

Here we added the instance of the `stopwatch_csr_reg` to address 5 and we defined the string CSR as the register identification. The address and the register identification string become the keys to find the register later in the verification environment, for example in drivers and monitors.

In the same way, we explicitly map all instances of the `stopwatch_memory_reg`, to the addresses starting from 6.

Note that the loop structure used for construction of the `stopwatch_memory_reg` array could also be the used to reduce the amount of coding for `add_register()` calls.

## Defining a design specific register package

The register data types, the register types and the register file are typically defined in a self contained SystemVerilog package. In fact, this collection of information is usually auto-generated, since it is tedious to write and error prone. Furthermore, the address maps and bit structure frequently changes during the early stages of verification – so automated tools to generated this code increases productivity.

Packaging the “register definitions” this way also helps create a clean separation from the register declaration and the usage of the register in the OVM verification environment.

A typical structure of a design specific register package looks as follows:

```
package <dut_name>_register_pkg;

    typedef bit[31:0] <dut_name>_<register data kind>_t;
    typedef struct packed {...}
        <dut_name>_<a other register data kind>_t;
    ...

    class <register kind> extends ovm_register #( <register data type> );
    ...
endclass

    class <a other register kind> extends
        ovm_register #( <other register data type> )
    ...
endclass

    class <dut_name>_register_file extends ovm_register_file;
        // Declare the register members

        function void build();
            // Construct the registers
            // Add the registers to the register file
        endfunction
    endclass

endpackage
```

A number of tools exist so you can automatically generate the register declarations from input format like IP-XACT and SystemRDL.

## Defining a register map

Previously we saw how to define a register file. A register file is really just a “small register” map – usually just modeling one device. When we add more devices, we need to combine register files (or register maps) to describe the complete system.

As register files are “local” to the DUT they are typically instantiated together with the DUT / bus specific components like pin level drivers and monitor i.e. an agent in OVM terminology,

```
class stopwatch_agent extends ovm_threaded_component;
  virtual interface stopwatch_if sw_if;
  stopwatch_driver driver;
  stopwatch_monitor monitor;
  stopwatch_register_file register_file;
  ...
endclass
```

With the register files in place, we can start to build the register map, i.e. define the memory map for our system.

The register map is common for the whole verification environment, so we instantiate it in the top level environment. Once constructed we need to add the register files with the address offset to match our system.

Consider the following code, where we add DUT[1]’s register file (in the agent) to the register map, with address offset 0h, and the DUT[2]’s register file are added with address offset 1000h:

```
class register_env extends ovm_env;

  stopwatch_agent sw_agent[1:8];
  ovm_register_map register_map;
  ...

  function void build();
    ...
    register_map.add_register_file(sw_agent[1].register_file, 0);
    register_map.add_register_file(sw_agent[2].register_file, 'h1000);
    ...
  endfunction
endclass
```

With the register map in place, we can start to use all the elements to build a register aware verification environment.

For example if we want to check what is in the current register memory map, we can use the register map's API for checking:

```
class register_env extends ovm_env;
  task run();
    ...
    register_map.display_address_map();
    ...
  endtask
  ...
endclass
```

The call to `display_address_map()` could produce a simulation transcript similar to this:

```
# OVM_INFO @ 10: env.register_map [regmap]
                                display_address_map_by_address()
# Name  Address Value
# CSR    00000005 env.sw_agent[1].register_file.CSR = '{padding: 0,
stride: 0, updown: 0, upper_limit_reached: 0, lower_limit_reached: 0}
# MEM[0] 00000006 env.sw_agent[1].register_file.MEM[0] = 0
# MEM[1] 00000007 env.sw_agent[1].register_file.MEM[1] = 0
# MEM[2] 00000008 env.sw_agent[1].register_file.MEM[2] = 0
# MEM[3] 00000009 env.sw_agent[1].register_file.MEM[3] = 0
# MEM[4] 0000000a env.sw_agent[1].register_file.MEM[4] = 0
# MEM[5] 0000000b env.sw_agent[1].register_file.MEM[5] = 0
# MEM[6] 0000000c env.sw_agent[1].register_file.MEM[6] = 0
# MEM[7] 0000000d env.sw_agent[1].register_file.MEM[7] = 0
# CSR    00001005 env.sw_agent[2].register_file.CSR = '{padding: 0,
stride: 0, updown: 0, upper_limit_reached: 0, lower_limit_reached: 0}
# MEM[0] 00001006 env.sw_agent[2].register_file.MEM[0] = 0
# MEM[1] 00001007 env.sw_agent[2].register_file.MEM[1] = 0
# MEM[2] 00001008 env.sw_agent[2].register_file.MEM[2] = 0
# MEM[3] 00001009 env.sw_agent[2].register_file.MEM[3] = 0
# MEM[4] 0000100a env.sw_agent[2].register_file.MEM[4] = 0
# MEM[5] 0000100b env.sw_agent[2].register_file.MEM[5] = 0
# MEM[6] 0000100c env.sw_agent[2].register_file.MEM[6] = 0
# MEM[7] 0000100d env.sw_agent[2].register_file.MEM[7] = 0
```

Notice how the packed struct previously defined to specialize the CSR register, is getting automatically expanded as fields through the SystemVerilog `'%p'` construct. This is a convenient way to view the bit vectors – built-in to SystemVerilog and the `ovm_register` package.



Sometimes it's advantageous to see the registers sorted by name, instead of by address as above; this could be done using the `get_register_array()` method in the following code structure:

```
class register_env extends ovm_env;
  task run();
    ...
    int i;
    ovm_register_base register_array[];
    register_array = register_map.get_register_array();
    foreach (register_array[i]) ;
      ovm_report_message("register_env",
        register_array[i].convert2string());
    ...
  endtask
  ...
```

This could produce a simulation transcript like this:

```
# env [register_env] env.sw_agent[1].register_file.CSR = '{padding: 0,
stride: 0, updown: 0, upper_limit_reached: 0, lower_limit_reached: 0}
# env [register_env] env.sw_agent[1].register_file.MEM[0] = 0
# env [register_env] env.sw_agent[1].register_file.MEM[1] = 0
# env [register_env] env.sw_agent[1].register_file.MEM[2] = 0
# env [register_env] env.sw_agent[1].register_file.MEM[3] = 0
# env [register_env] env.sw_agent[1].register_file.MEM[4] = 0
# env [register_env] env.sw_agent[1].register_file.MEM[5] = 0
# env [register_env] env.sw_agent[1].register_file.MEM[6] = 0
# env [register_env] env.sw_agent[1].register_file.MEM[7] = 0
...
# env [register_env] env.sw_agent[2].register_file.CSR = '{padding: 0,
stride: 0, updown: 0, upper_limit_reached: 0, lower_limit_reached: 0}
# env [register_env] env.sw_agent[2].register_file.LOWER_LIMIT = 0
# env [register_env] env.sw_agent[2].register_file.MEM[0] = 0
...
```

The Stopwatch example included with the register package can be seen as a generic example on how to use the different aspects outlined in this document.

## Requirements

The stopwatch example requires OVM version 2.0 available from [www.ovmworld.org](http://www.ovmworld.org). To execute the example we recommend Questa 6.4a or newer. The code has been testing using Questa 6.4 and 6.4a on Windows and Linux.

It is possible to use OVM 1.1 with the new register code, but you cannot use OVM 1.1 with the automatic tests – they rely on the sequence implementation which changed between OVM 1.1 and OVM 2.0. To use OVM 1.1, you must define the variable ‘USING\_OVM\_1’ on your vlog command line when you compile/include the ovm\_register code.

```
vlog +define+USING_OVM_1 \
+incdir+$(OVM_REGISTER_HOME)/src+$(OVM_HOME)/src \
$(OVM_REGISTER_HOME)/src/ovm_register_pkg.sv
```

The easiest way to run the example code is to install it in parallel with the OVM. The other way to run the example is to install the OVM Register code somewhere, and simply point at the OVM installation. You can use the environment variables OVM\_HOME and OVM\_REGISTER\_HOME to customize which versions are used in the example.

## Easy install – parallel to an existing OVM installation

```
cd $OVM_HOME
<Now in the place which contains 'src' and 'examples' from the OVM>
cd ..
tar xvfz ovm_register-XXX.tar.gz

cd ovm_register-XXX/examples/registers/00_stopwatch
make
```

## Separate install – point at the OVM installation

```
cd <Your_OVM_Register_Install_Area>
tar xvfz ovm_register-XXX.tar.gz

cd ovm_register-XXX/examples/registers/00_stopwatch
```

```
setenv OVM_HOME <Your_OVM_Install>
make
```

## Separate install – point at the built-in OVM installation

```
cd <Your_OVM_Register_Install_Area>
tar xvfz ovm_register-XXX.tar.gz

cd ovm_register-XXX/examples/registers/00_stopwatch
setenv OVM_HOME <Your_Questa_Install_Area>/verilog_src/ovm
```

**<NOTE: You must also change/edit the makefile or compile scripts. The makefile and compile scripts assume that OVM\_HOME/src exists. The Standard Questa install has removed the 'src' subdirectory. If you want to use the pre-compiled Questa source code, then you need to remove the 'src' references from the OVM\_HOME specifications in the makefile and compile scripts. This problem will be fixed in future Questa releases.>**

```
make
```

## The General compilation and simulation flow

The compilation consists of

1. Creating the Questa work directory  
`vl ib work`
2. Compiling the OVM source, if you are NOT using the built-in OVM.  
`vl og +i ncdi r+$(OVM_HOME)/src $(OVM_HOME)/src/ovm_pkg. sv`
3. Compiling the OVM Register package.  
`vl og \
+i ncdi r+$(OVM_REGISTER_HOME)/src+$(OVM_HOME)/src \
$(OVM_REGISTER_HOME)/src/ovm_register_pkg. sv`
4. Compiling the register and address map definitions  
`vl og auto/stopwatch_register_pkg. sv`
5. Compiling the top level  
`vl og +i ncdi r+$(OVM_HOME)/src t. sv`
6. Compiling the RTL  
`vl og dut/stopwatch_rtl. sv
vl og dut/stopwatch_rtl_wrapper. sv`

Once everything is compiled, simulation is a straightforward 'vsim' run.

```
vsim -c top -do "run -all; quit -f"
```

Filename: examples/registers/00\_stopwatch/auto/stopwatch\_register\_pkg.sv

```
//*****
//-----
// Copyright 2007-2008 Mentor Graphics Corporation
// All Rights Reserved Worldwide
//
// Licensed under the Apache License, Version 2.0 (the
// "License"); you may not use this file except in
// compliance with the License. You may obtain a copy of
// the License at
//
// http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in
// writing, software distributed under the License is
// distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
// CONDITIONS OF ANY KIND, either express or implied. See
// the License for the specific language governing
// permissions and limitations under the License.
//-----
/* ***** */
/* THIS IS AUTOMATICALLY GENERATED CODE */
/* ***** */

package stopwatch_register_pkg;

import ovm_pkg::*;
import ovm_register_pkg::*;

typedef bit[31:0] bit32_t;

typedef struct packed {
    bit [31:7] padding;
    bit [3:0] stride;
    bit updown;
    bit upper_limit_reached;
    bit lower_limit_reached;
} stopwatch_csr_t;

class stopwatch_value extends ovm_register #(bit32_t);
    function new(string name, ovm_component p);
        super.new(name, p);
        // READ-ONLY under normal circumstances.
        WMASK = 'b0;
    endfunction
endclass

class stopwatch_reset_value extends ovm_register #(bit32_t);
    function new(string name, ovm_component p);
        super.new(name, p);
    endfunction
endclass

class stopwatch_upper_limit extends ovm_register #(bit32_t);
    function new(string name, ovm_component p);
        super.new(name, p);
    endfunction
endclass
```

```

class stopwatch_lower_limit extends ovm_register #(bit32_t);
  function new(string name, ovm_component p);
    super.new(name, p);
  endfunction
endclass

class stopwatch_csr extends ovm_register #(stopwatch_csr_t);

  covergroup c;
    stride: coverpoint data.stride;
    updown: coverpoint data.updown;
    upper_limit_reached: coverpoint data.upper_limit_reached;
    lower_limit_reached: coverpoint data.lower_limit_reached;
  endgroup

  function void sample();
    c.sample();
  endfunction

  function new(string name, ovm_component p);
    super.new(name, p);
    c = new();
    // All bits writable, except the "limits-reached" bits.
    WMASK = 'b1111_1_0_0;
  endfunction
endclass

class stopwatch_memory extends ovm_register #(bit32_t);
  function new(string name, ovm_component p);
    super.new(name, p);
  endfunction
endclass

class stopwatch_register_file extends ovm_register_file;

  stopwatch_value      stopwatch_value_reg;

  stopwatch_reset_value stopwatch_reset_value_reg;
  stopwatch_upper_limit stopwatch_upper_limit_reg;
  stopwatch_lower_limit stopwatch_lower_limit_reg;

  stopwatch_csr        stopwatch_csr_reg;

  stopwatch_memory      stopwatch_memory_reg[8];

  function new(string name, ovm_component p);
    super.new(name, p);
  endfunction

  function void build();
    ovm_report_info("stopwatch_register_file", "build()");
    super.build();

    // -----
    // Construct the registers
    // -----
    stopwatch_value_reg      = new("VALUE", this);
    stopwatch_reset_value_reg = new("RESET_VALUE", this);
    stopwatch_upper_limit_reg = new("UPPER_LIMIT", this);
    stopwatch_lower_limit_reg = new("LOWER_LIMIT", this);
    stopwatch_csr_reg        = new("CSR", this);

    foreach ( stopwatch_memory_reg[i] )

```

```

        stopwatch_memory_reg[i] =
            new( $sprintf("MEM[%0d]", i), this);

// -----
// Add the registers to the register file
// -----
add_register(
    stopwatch_value_reg.get_full_name(),
    1, stopwatch_value_reg);
add_register(
    stopwatch_reset_value_reg.get_full_name(),
    2, stopwatch_reset_value_reg);
add_register(
    stopwatch_upper_limit_reg.get_full_name(),
    3, stopwatch_upper_limit_reg);
add_register(
    stopwatch_lower_limit_reg.get_full_name(),
    4, stopwatch_lower_limit_reg);

add_register(
    stopwatch_csr_reg.get_full_name(),
    5, stopwatch_csr_reg);

add_register(
    stopwatch_memory_reg[0].get_full_name(),
    6, stopwatch_memory_reg[0]);
add_register(
    stopwatch_memory_reg[1].get_full_name(),
    7, stopwatch_memory_reg[1]);
add_register(
    stopwatch_memory_reg[2].get_full_name(),
    8, stopwatch_memory_reg[2]);
add_register(
    stopwatch_memory_reg[3].get_full_name(),
    9, stopwatch_memory_reg[3]);
add_register(
    stopwatch_memory_reg[4].get_full_name(),
    10, stopwatch_memory_reg[4]);
add_register(
    stopwatch_memory_reg[5].get_full_name(),
    11, stopwatch_memory_reg[5]);
add_register(
    stopwatch_memory_reg[6].get_full_name(),
    12, stopwatch_memory_reg[6]);
add_register(
    stopwatch_memory_reg[7].get_full_name(),
    13, stopwatch_memory_reg[7]);
endfunction
endclass

//
// The actual register map for this system.
//
class stopwatch_register_map extends ovm_register_map;
    stopwatch_register_file sw1;
    stopwatch_register_file sw2;

    function new(string name, ovm_component p);
        super.new(name, p);
    endfunction

    function void build();

```

```

    ovm_report_info("stopwatch_register_map", "build()");
    super.build();

    sw1 = new("sw1", this);
    sw2 = new("sw2", this);

    sw1.build();
    sw2.build();

    add_register_file(sw1, 'h1000);
    add_register_file(sw2, 'h2000);
endfunction
endclass

//
// A class to automatically load a register map.
//
class register_map_auto_load;

    // Triggers factory registration of this default
    // sequence. Can be overridden by the user using
    // "default_auto_register_test".
    register_sequence_all_registers
        #(ovm_register_transaction,
          ovm_register_transaction) dummy;

    static bit loaded = build_register_map();

    static function bit build_register_map();

        stopwatch_register_map register_map;

        register_map = new("register_map", null);

        register_map.build();

        set_config_string("*",
            "default_auto_register_test",
            "register_sequence_all_registers_REQ_RSP");
        set_config_object("*",
            "register_map",
            register_map, 0);
        return 1;
    endfunction

endclass
endpackage

```

Filename examples/registers/00\_stopwatch/t.sv

```
//-----
// Copyright 2007-2008 Mentor Graphics Corporation
// All Rights Reserved Worldwide
//
// Licensed under the Apache License, Version 2.0 (the
// "License"); you may not use this file except in
// compliance with the License. You may obtain a copy of
// the License at
//
// http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in
// writing, software distributed under the License is
// distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
// CONDITIONS OF ANY KIND, either express or implied. See
// the License for the specific language governing
// permissions and limitations under the License.
//-----

import ovm_pkg: *;
import ovm_register_pkg: *;

// The automatically generated register description.
import stopwatch_register_pkg: *;

`include "util.svh"

//
// Custom built code to translate between classes
// and pin wiggles
//
// bus_request --> DUT pin wiggles --> bus_response
//
class auto_test
  #(type REQ = ovm_sequence_item,
    type RSP = ovm_sequence_item)// com
```



```

//      request and response is interfaced
//      with the DUT.
//
task do_operation(REQ req, output RSP rsp);
    int chunk;
    bit[31:0] mapped_address;

    // Create a response.
    rsp = new();
    rsp.copy_req(req);
    rsp.set_id_info(req);
    rsp.status = PASS; // Default is PASS.

    ovm_report_info("AutoTest MEMREQ", req.convert2string());

    // Do the simple address mapping here.
    // Each 1000 bytes is one chunk
    //
    // Note: Each virtual interface above manages
    //       one chunk. So each virtual interface
    //       hangs a device for an address space
    //       of 'h1000
    chunk = req.address / 'h1000;
    mapped_address = req.address % 'h1000;
    assert((chunk==1)|| (chunk==2)) else
        $fatal("chunk failed");

    // Actually "execute" the transaction.
    case (req.op)
        READ: sw_if[chunk].read(mapped_address,  rsp.data);
        WRITE: sw_if[chunk].write(mapped_address, req.data);
    endcase
    ovm_report_info("AutoTest MEMRSP", rsp.convert2string());
endtask
endclass

//
// The environment
// contains
//   1. a register environment
//   2. a translator between class-based
//       transactions and pin wiggles
//
class my_env
    extends ovm_env;

    ovm_register_env register_env;

    auto_test #(bus_request, bus_response)
        m_auto_test;

    function new(string name, ovm_component p);
        my_report_server my_report_server;
        //ovm_global_report_server gs;

        super.new(name, p);
        register_env = new("register_env", this);
        m_auto_test = new("auto_test", this);

        // Make my report server!
        my_report_server = new();
        my_report_server.name_width = 28;
        my_report_server.id_width = 20;
        // Hook it up.

```

```

        //gs = get_server();
        //gs.set_server(my_report_server);
        m_rh.m_glob.set_server(my_report_server);
    endfunction

    function void build();
        set_config_int("*", "count", 0);
        set_config_string("*",
            "default_auto_register_test",
            "register_sequence_all_registers_REQ_RSP");
    endfunction

    // Called from the top-level module to
    // connect to the real interface.
    function void assign_vi (
        virtual interface stopwatch_if sw_if_1,
        virtual interface stopwatch_if sw_if_2);
        m_auto_test.assign_vi (sw_if_1, sw_if_2);
    endfunction

    function void connect();
        register_env.bus_transport_port.connect(
            m_auto_test.transport_export);
        m_auto_test.ap.connect(
            register_env.bus_rsp_analysis_export);
    endfunction

    function void report();
        ovm_register_map register_map;
        // Some general debug routines.
        m_auto_test.print();
        register_env.m_register_map.display_address_map();
    endfunction
endclass

module top;
    bit clk = 0;

    // Our hardware
    // There are two instances of the hardware, and
    // each has its own interface.
    stopwatch_if sw_if_1(clk);
    stopwatch_if sw_if_2(clk);

    // The hardware instances.
    stopwatch_wrapper sw1(sw_if_1);
    stopwatch_wrapper sw2(sw_if_2);

    // The testbench environment
    my_env env = new("env", null);

    initial begin
        // Connect the class based testbench
        // to the hardware.
        env.assign_vi (sw_if_1, sw_if_2);

        // Run the test
        run_test();
    end

    // Hardware clock
    always
        #10 clk = ~clk;
endmodule

```

*Text below not updated for this release. These sections are now mostly replaced by using the new built-in classes in ovm\_register\_pkg. If you want to extend, enhance or replace the built-in classes, the sections below may help you.*

---

## Building a register testcase

The get\_register\_array() method is also very useful for register testing. One example could be in the case where we need to ensure that all registers in a register map, can be written to, and read from in a random order.

```
task test();
  ovm_register_base list_of_registers[];
  ovm_register_base r;

  int i;

  // Get a list of all register in the map
  list_of_registers = register_map.get_register_array();

  // Shuffle the register list
  list_of_registers.shuffle();

  // iterate through the register list
  foreach (list_of_registers[i]) begin
    // Randomize the data value
    assert(this.randomize());
    r = list_of_registers[i];
    write_read(r.get_full_name(), data);
  end
endtask
```

In the example above by using the get\_register\_array() method combined with SystemVerilog array methods, shuffle(), we are able to get register access in random order. The access are done by the convenience task - write\_read().

```
task write_read(string name, int data);
  int expected_data;

  write(name, data);
  read(name, expected_data);

  if (data != expected_data) begin
    ovm_report_error("MISMATCH",
      $psprintf("Register %s: FAIL Expected (%h), read (%h)",
        name, expected_data, data));
  end
  else begin
    ovm_report_message("MATCH",
      $psprintf("Register %s: OK Expected (%h), read (%h)",
        name, expected_data, data));
  end
endtask
```

See the stopwatch example shipped with the register package for full details on the write() and read() tasks.

## Building a register aware driver

By using the register name as a key we have raised the abstraction level of our verification environment. However in order for the register tests to be useful on the real bus, some component in our verification environment must map the register names to actual bus addresses and then issue bus transactions.

This mapping of register name to address can be conveniently done in the pin or bus drivers. A “translator” accepts register transactions and turns them into bus transactions, which are then turned into bus activity on the wires by the bus driver.

In the following code, we use the `lookup_register_address_by_name()` to get a register address from the register map given the register name. Once found, a typical bus level transaction is created and sent downstream to cause pin level activity:

```
task write_to_bus(string name, int data);
    bus_transaction req, rsp;
    bit valid_address;

    req = new();

    // Get the register address from the register name
    req.addr = register_map.lookup_register_address_by_name(name,
        valid_address);
    ...
    // ensure that it's all ok
    assert(valid_address);

    req.data = data;

    // send bus transaction to bus
    put(req);
    ...
endtask
```

## Building a register aware monitor

On the monitor side of our verification environment – similar to the driver side, we are moving between registers transactions and bus transactions. On the driver side we are moving from registers to busses – on the monitor side we are moving from busses to registers.

The monitor gets an address from the bus, and then must use a register map to find the register that is located at that address.

As the `lookup_register_by_address()` provides register access based on the address, we can use it to convert an address to a register name, that then can be used in the environment

```
class register_monitor extends ovm_subscriber #(bus_transaction);
...
function void write(input bus_transaction t);
    register_transaction out_t = new();
    ovm_register_base register;

    // Find the register handle from the address.
    register = register_map.lookup_register_by_address(t.addr);

    out_t.name = register.get_full_name();

    // Map transaction type and data fields
    ...

    // Publish the converted transaction
    ap.write(out_t);
endfunction
...
endclass
```

## Building a register aware scoreboard

Building a register aware score is easy using the register package. One of many reasons is that a register instantiated in the verification environment acts as a mirror of the actual DUT register, making it simple to compare register data.

In the following code we update the register data in the verification environment on each write, and on a read response we compare the data read back with the bit masking applied, the code for the READ\_RSP could also use the built-in bus\_read() method.

```
virtual function void
  my_scoreboard_function(register_transaction t);

  ovm_register_base r;

  // Use the register name to get the register handle from the map
  r = register_map.lookup_register_by_name(t.name);

  case (t.register_transaction_type)

    WRITE: begin
      // Mask and write the register data
      r.set_data32(t.data);
    end

    READ_RSP: begin
      if (r.compare_data(bv)) begin
        // update register with WMASK
        write_without_notify(bv);
      end
      else begin
        // Error. Data doesn't match.
        ovm_report_error("..");
      end
    end
  endfunction
```