Pumping Up Test Development with Task Based, C-callable, UVM based Tests

Rich Edelman
Siemens EDA
Fremont, CA US

Didan Francis
Siemens India
Bangalore, IN

***Abstract*- UVM Testbenches continue to be both commonly used and dreaded by verification engineers. Enabling more test creation and more test writers all using the standard UVM interfaces will allow an increase in productivity. This paper will extend previous work that presents a simple task based, C callable interface for tests. This paper will demonstrate an actual implementation of task-based tests built on commonly used bus protocols. Task based tests can be called from any compatible "C callable" interface system, including C, C++, SystemC, Python, Rust, PSS, etc.**

I. INTRODUCTION

Verification teams use SystemVerilog UVM as a standard framework to write tests, collect coverage, and check results. But the UVM is hard to use, complicated and has a learning curve that many verification teams cannot consider. Additionally, there is a limited supply of UVM-aware verification engineers.

As proposed in this paper, a task-based test environment with a C callable interface will allow the inclusion of many more test writers, coverage collectors and checkers. This C callable interface is a SystemVerilog "DPI-C" interface with special properties that make the interface seamless. (See previous work [1]). A C callable interface makes all the C programmers at a company potential UVM test writers.

This paper suggests two teams working together on verification. First, a UVM aware team architecting the UVM environments, including sequence design and task design. And second, a team perhaps less UVM aware, but capable programmers who understand how to call the underlying test APIs. These test programmers probably wrote their tests first and worked with the system architects to prove out the design architecture. They may have already measured estimated performance for the design. These tests developed to prove out early architecture decisions can simply be re-used to prove out and verify the actual hardware implementation during simulation.

II. UVM BASICS AND PREVIOUS WORK

In this paper, the UVM model is as below. The DUT is connected to an interface. The interface may be connected to a monitor and a driver. The sequencer is connected to the driver and sends sequences to the driver. The sequences are "programs" that are designed to cause activity or respond to activity on the interface. The sequences in this case also has tasks. Those tasks send or respond to sequences. The tasks are exported via DPI-C to an interface which provide a way for C to call those tasks in a sequence.

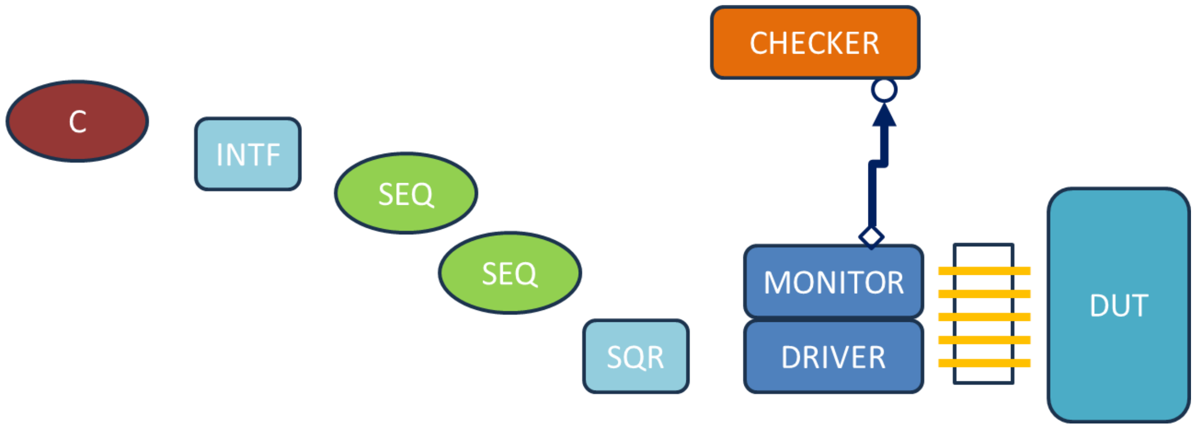


Figure 1 - UVM Bubble Diagram

The UVM provides a consistent way to interact with the DUT. Sequences can cause activity and monitor activity. The UVM has phases including the build_phase, connect_phase and the run_phase. The UVM has many other phases. Once the run_phase starts, the entire UVM is built and connected – the UVM component objects are complete. As the run_phase continues, many sequences may be created, and each sequence can create many sequence items. Those sequences are like "programs" or "tests", and the sequence items are like "the data" feeding each program or test.

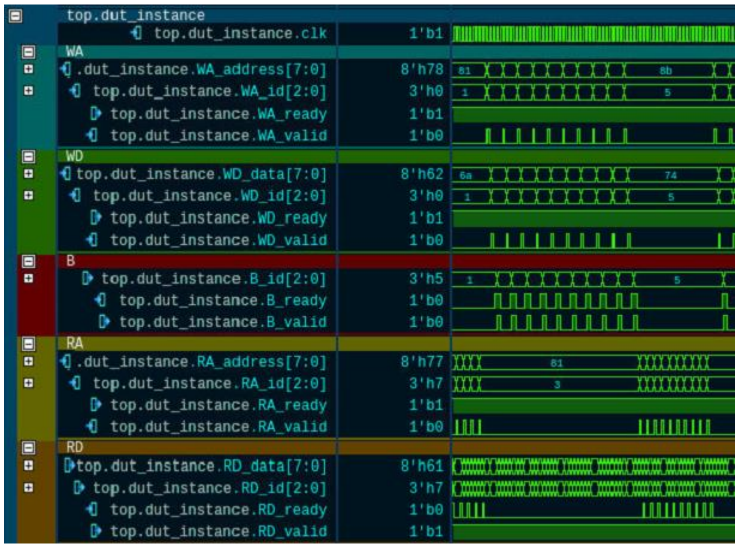
By extending this capability to C, C programs or tests can be involved in the same stream of activities. The C program and tests are running in parallel – at the same time as the normal UVM sequences. Because these programs are running together – in parallel – the productivity of the test writer is improved. For example, the critical C program or test can run while "cross-traffic" modeled by sequences is running at the same time on the same interface.

III. Overview of Demonstration Interface

The demonstration bus interface is a simple Ready/Valid scheme with 5 sub-channels (RA, RW, WA, WD and B). This is an out-of-order interface. It's purpose is simply to be an interface that sequences can cause activity on.

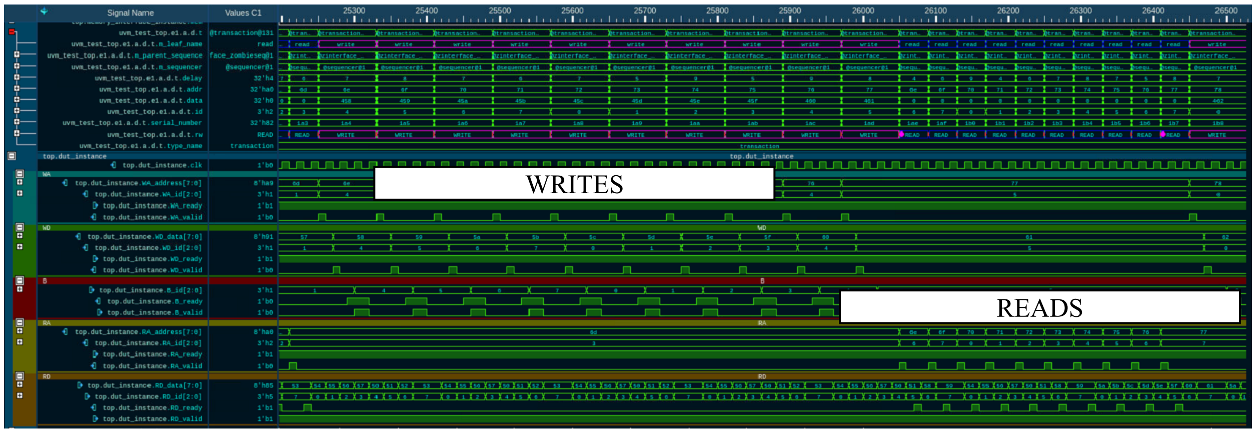
```
interface dut_interface(input clk);
  reg [7:0] WA_address;
  reg [2:0] WA_id;
  reg WA_ready;
  reg WA_valid;

  reg [7:0] WD_data;
  reg [2:0] WD_id;
  reg WD_ready;
  reg WD_valid;

  reg [2:0] B_id;
  reg B_ready;
  reg B_valid;

  reg [7:0] RA_address;
  reg [2:0] RA_id;
  reg RA_ready;
  reg RA_valid;

  reg [7:0] RD_data;
  reg [2:0] RD_id;
  reg RD_ready;
  reg RD_valid;
endinterface
```

The interface zoomed in in a waveform window



```
top.dut_instance
    top.dut_instance.clk            1'b1
WA
  .dut_instance.WA_address[7:0]     8'h78
  top.dut_instance.WA_id[2:0]       3'h0
  top.dut_instance.WA_ready         1'b1
  top.dut_instance.WA_valid         1'b0
WD
  top.dut_instance.WD_data[7:0]     8'h62
  top.dut_instance.WD_id[2:0]       3'h0
  top.dut_instance.WD_ready         1'b1
  top.dut_instance.WD_valid         1'b0
B
  top.dut_instance.B_id[2:0]        3'h5
  top.dut_instance.B_ready          1'b0
  top.dut_instance.B_valid          1'b0
RA
  .dut_instance.RA_address[7:0]     8'h77
  top.dut_instance.RA_id[2:0]       3'h7
  top.dut_instance.RA_ready         1'b1
  top.dut_instance.RA_valid         1'b0
RD
  top.dut_instance.RD_data[7:0]     8'h61
  top.dut_instance.RD_id[2:0]       3'h7
  top.dut_instance.RD_ready         1'b0
  top.dut_instance.RD_valid         1'b1
```

Many WRITE phases followed by READ phases – see the C test program.



IV. TWO WAYS TO BUILD A TASK BASED OUT-OF-ORDER TEST

API design is an art. This paper presents two API levels to implement communication. The low-level API can be used to double check latency between channel phases for example. The high-level API is simpler to use, and probably most useful to those writing complex tests from the C side.

These are the same APIs that are in use from the hardware design and architecture/testing phase before any implementation was available.

Detailed low-level API

The communication channel has 5 sub-channels, which can individually be called. This level API is likely less used from a C callable external program, but it will be demonstrated for the purpose of latency and throughput calculations.

```
task send_WA(input reg[2:0] id, reg[7:0]address);
  WA_address = address;
  WA_id = id;
  WA_valid = 1;
  forever begin
    @(posedge clk);
    if ( WA_ready == 1) begin
      // xfer
      WA_valid = 0;
      @(negedge clk);
      return;
    end
  end
endtask

task send_WD(input reg[2:0] id, reg[7:0]data);
  WD_data = data;
  WD_id = id;
  WD_valid = 1;
  forever begin
    @(posedge clk);
    if ( WD_ready == 1) begin
      // xfer
      WD_valid = 0;
      @(negedge clk);
      return;
    end
  end
endtask
```

Higher-Level API

From the “test program”, the details of each sub-channel may not be important. A simpler higher-level API with simple READ and WRITE calls will be built on top of the lower-level sub-channels. For example, an algorithm in C to calculate the secondary radiation exposure due to treating a cancer with radiation. That algorithm will load memory with constants and measurements and then ask the hardware to calculate the dosage at each surrounding point. Then the C test will read the memory back into the C side and check the results.

Higher level APIs can be created – WRITE_PACKET, READ_PACKET, WRITE_BLOCK, READ_BLOCK.

```
task write(input reg[2:0] id, reg[7:0]address, reg[7:0]data);
  send_WA(id, address);
  send_WD(id, data);
  wait_B(id);
endtask

task read(input reg[2:0] id, reg[7:0]address, output reg[7:0]data);
  send_RA(id, address);
  wait_RD(id, data);
endtask
```

The higher level APIs may simply reuse the lower level APIs – as above, or they may have more detailed access to the very lowest levels of communication (the wires).

V. DESIGNING A UVM DRIVER TO SUPPORT THIS TESTING ENVIRONMENT

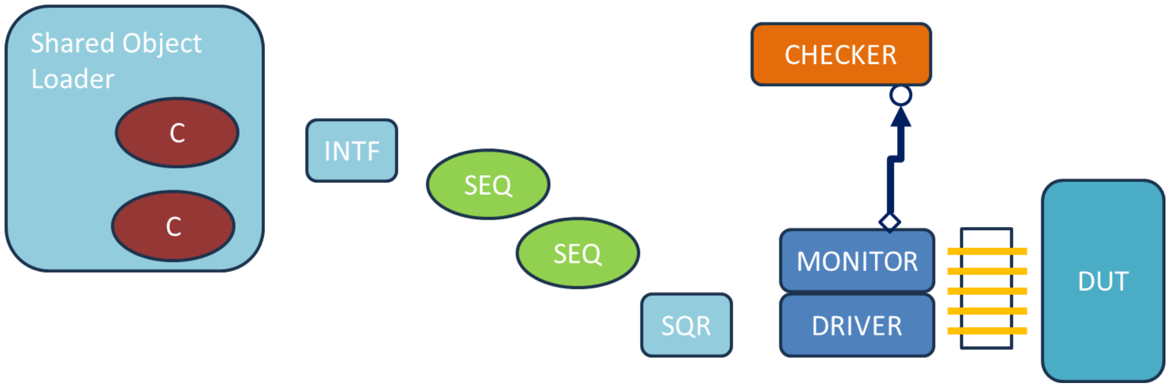
The driver uses the interface to communicate – by calling tasks in the interface. It may consider driving stimulus, but also obtaining results. An out-of-order driver needs to consider the delayed response. That detail is beyond the scope of this paper, but included in the working demonstration code.

The UVM driver below receives a transaction from a sequence, and then depending on the RW field will call the appropriate API. The API is implemented in the interface. The interface for this demonstration bus is about 150 lines of code.

```
task run_phase(uvm_phase phase);
  `uvm_info(get_type_name(), "...starting", UVM_MEDIUM)
  forever begin
    seq_item_port.get_next_item(t);
    case (t.rw)
       READ: dvif.read(t.id, t.addr, t.data);
      WRITE: dvif.write(t.id, t.addr, t.data);
         RA: dvif.send_RA(t.id, t.addr);
         RD: dvif.wait_RD(t.id, t.data);
         WA: dvif.send_WA(t.id, t.addr);
         WD: dvif.send_WD(t.id, t.data);
          B: dvif.wait_B(t.id);
    endcase
    seq_item_port.item_done();
  end
endtask
```

VI. USING THE C-CALLABLE INTERFACES

The UVM diagram is extended below to allow for dynamic loading of C code – tests. The shared object loader uses dlopen() to load a shared object and run it. That share object calls the tasks that have been defined for such purpose by the UVM architects.



C Calls

A C test can be written using “READ(address, data)” or at the lower level “READ_ADDRESS(address)” and “READ_DATA(data)”.

Rust, Python and other external tools, PSS

Any language that is "C callable" and can be compiled into a "normal" shared object file can be used to interface with the task based API.

For the demonstration example a simple C program is compiled into shared objects – testprogram1.so and testprogram2.so. They get loaded from SystemVerilog and run using the API below.

```
// LOAD & RUN
zif.sv_load_shared_object("./testprogram1.so");

// LOAD & RUN
zif.sv_load_shared_object("./testprogram2.so");
```

VII. WRITING MANY TESTS – THE MAGIC OF DYNAMIC LOADING FOR DPI-C

Building Many Tests

The C tests can be written and tested standalone on the abstract model used in early architectural development. Those tests can be compiled into shared objects and linked with simulation. The shared object loader will load and run them.

Easily Using the Many Tests

The SystemVerilog "DPI-C" standard defines how to load shared objects and calling conventions. The C code will define entry points for starting threads (ABC_BUS_CONNECTIVITY_CHECK(), for example). The demonstration code will show how many different shared objects can be used to run many external tests.

In the example below, a shared object loader is implemented, and linked in the SystemVerilog simulation.

From SystemVerilog, calling "c_load_shared_object("testprogram1.so") will cause that shared object to be loaded and run. With some small extensions the loader could be made more general – loading the shared object and calling a named function call rather than a fixed name.

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

#include "mti.h"
#include "dpiheader.h"

int
c_load_shared_object(const char *name) {
    void *handle;
    int (*zinterface_start_test_program)(int, char *, int);

    handle = dlopen(name, RTLD_LAZY|RTLD_GLOBAL|RTLD_NOW);
    if (!handle) {
      // …Error handing
    }
    dlerror(); // Clear any existing error.

    zinterface_start_test_program = dlsym(handle, "zinterface_start_test_program");
    zinterface_start_test_program(1, "thread1", 100);
    return 0;
}
```

VIII. The C Test Program

The C test program below takes three arguments. The first – the index is dictated by the connection scheme from C to interface to sequence class. Then the name is used as a debug aid. Finally the 'addr' argument is the first address that should be "tested". Any set of arguments could be designed.

This test program is quite simple.
It writes 10 times to incrementing addresses. At each location writing a different value. Then it reads 10 times, and repeats. As it reads back the value, it checks to make sure the value read was the value that was expected to be written. A self-checking test.

```
int
zinterface_start_test_program(int index, const char *name, int start_addr) {
    int addr, data;
    int original_start_addr;
    int dataloops, loops;

    printf("test1 go()\n");

    original_start_addr = start_addr;

    // Repeat 10 times, changing the data
    for (dataloops = 0; dataloops < 10; dataloops++) {
      // Repeat 10 times - writing 10, and reading 10
      start_addr = original_start_addr;
      for (loops = 0; loops < 10; loops++) {
        for (addr = start_addr; addr < start_addr+10; addr++) {
          data = addr + dataloops;
          SV_write(index, addr, data);
        }
        for (addr = start_addr; addr < start_addr+10; addr++) {
          SV_read(index, addr, &data);

          if (data != addr + dataloops) {
            printf("C: ...ERROR  READ (%0d, %0d) <%s> [wrote: %d, read %d]  \n",
              addr, data, name, data, addr + 1000 + dataloops);
          }
        }
        start_addr += 10;
      }
    }
    printf("test1 done()\n");
    return 0;
}
```

Advanced Usage

Advanced testbenches use things like 'restart', 'checkpoint/restore', and other techniques to make tests more efficient. These are standard techniques used for many years.

Using C in this environment just means that the test needs to be "restartable" – that shouldn't be a problem. An additional feature for the ideas in this paper is that when doing a restart a different test could be run. So a test flow might be

```
Run
Restart with different test (load a different .so)
Run
Restart with a different test (load a different .so)
Repeat
```

IX. Putting It All Together

Compilation and simulation of the system is simple. Compile the memory, the DUT interface and the DUT itself. Then the support package and the top and the special “zinterface” that enables the connection of C to class based tasks. Commands below:

```
vlog mem.sv
vlog dut_interface.sv
vlog dut.sv
vlog ip_pkg.sv
vlog t.sv zinterface.sv -dpiheader dpiheader.h
```

Compile the special dynamic loader.

```
vlog -ccflags -g dlopen.c
```

Compile the two test programs. The two test programs exist as .so files – shared objects that will be loaded by the test.

```
gcc -shared -fPIC -I.../include -o testprogram1.so testprogram1.c
gcc -shared -fPIC -I.../include -o testprogram2.so testprogram2.c
```

Optimize and simulate.

```
qopt -o opt top -debug,livesim +designfile
qsim -c opt +UVM_TESTNAME=test -do "run -all; quit -f"
```

Debug as needed

```
qsim design.bin qwave.db
```

X. Conclusion

Task based testing has been used for years. This paper extends previous work on a C callable interface into SystemVerilog UVM class-based code, specifically into UVM Sequences. This easy-to-use interface is extended by this paper - introducing practical matters concerned with designing an API at the appropriate abstraction level and any supporting UVM code, including the UVM driver. Additionally, this easy C callable interface is available to use with other languages and interfaces, including Rust, Python and PSS. Any language that supports C callable and loadable objects.

Finally, a system is demonstrated which can run many different compiled code shared objects. This system is completely compatible with the UVM. “Normal” UVM sequences can be run in parallel on the C callable interfaces without any problem. All transactions are managed by the UVM sequencer.

The examples are open-source, using standard UVM constructs and are available from the author.

Note: This paper is not simply DPI-C calling tasks in SystemVerilog. It has many other features, including calling “directly” from C to Class based code and vice-versa. It does this through an interface. Additionally, the task-based implementations can use any constructs at their disposal, including randseq, randomize, $urandom_range, etc. The task-based system is easy to understand and extend for both new and experienced users.

XI. References

[1] DVCON Japan 2024, “Having Your Cake and Eating It Too - Programming UVM Sequences with DPI-C”, Rich Edelman, [Contact author for a copy]