UVM and C – Perfect Together

Rich Edelman
Mentor, A Siemens Company
46871 Bayside Parkway, Fremont, CA 94538

***Abstract*- SystemVerilog[1] UVM[2] provides structure and rules for verification teams. It allows consistent results across many tests and the ability to share verification between teams. Many verification teams have access to verification suites made up of C code. This paper will discuss various ways to integrate C tests and verification suites with a normal UVM testbench.**

I. I. INTRODUCTION

This paper will demonstrate techniques and methods for using DPI-C along with a standard UVM Testbench. The C code will take the form of low level transaction generator, high level transaction generator, scoreboard and monitor. The UVM Testbench will be operating at the same time – for example the UVM tests may be streaming background traffic on the bus, while the C code is creating specific bus transactions that are under test.

II. THE PROBLEM WITH THE UVM AND C

The UVM is used quite widely for testbench creation, coverage collection and monitoring. Using SystemVerilog and UVM has become a way to improve productivity in verification teams.

In addition to the power of randomization and constraints that UVM offers naturally, verification teams have needs to create or reuse C programs. These C programs may generate stimulus, they may check golden results, they may collect statistical data. Using SystemVerilog DPI-C is the way to connect these two worlds together.

The problem is that using DPI-C can sometimes be hard, and the DPI code has a close connection to a “scope”. A scope can be a module instance, an interface instance, or the global root scope. These scopes offer the DPI a connection point for calling back and forth from SystemVerilog.

A UVM testbench has few if any of these kinds of scopes. A UVM testbench is a dynamic class based structure, not a static instance based structure.

III. A SOLUTION: THE VIRTUAL INTERFACE

There are a number of solutions for easily connecting the UVM and C. This paper will explore the simplest approach – leveraging the interface (virtual interface) that will be associated with an agent.

Background

A UVM testbench is frequently built with an agent attached to a SystemVerilog interface. The interface is connected to the DUT pins. For communication to the DUT, the UVM driver causes the interface pins to wiggle. For communication from the DUT, the UVM monitor collects pins wiggles. The interface instance is a perfect place to host our DPI-C calls.

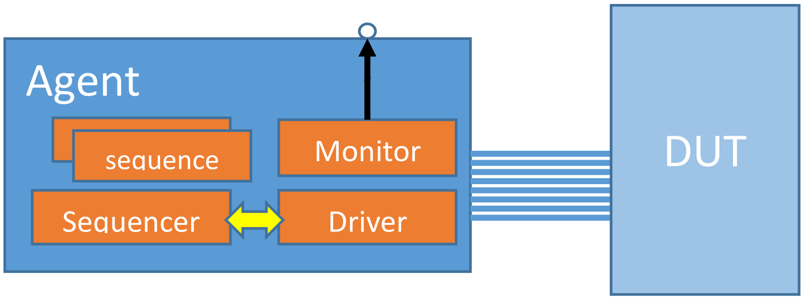

Figure 1 - Typical Agent Connected to DUT with Interface In Between

Adding C code using DPI-C in Figure 2

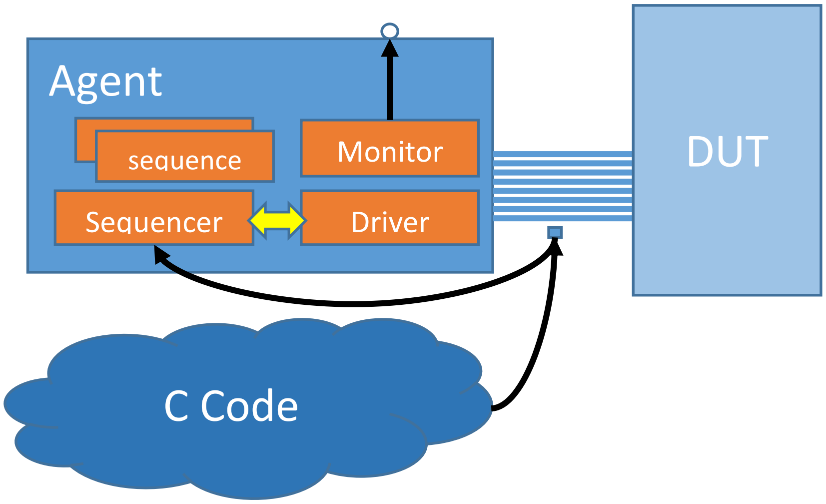

Figure 2 - Single Agent Connections to C Code

Many agents (Figure 3) can be connected to the C code – either threaded or not. The C code is not instance specific. There may be certain C code that is associated with certain interfaces (AHB vs AXI for example).

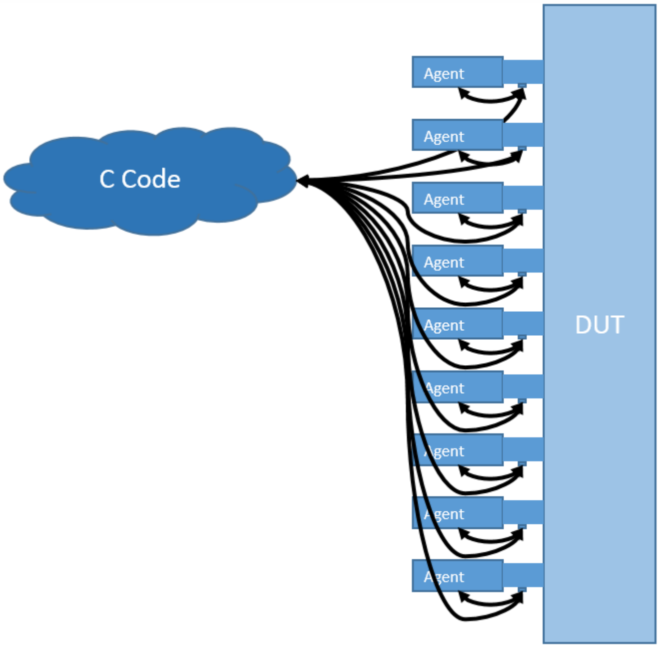

Figure 3 - C Code with many agents

The SystemVerilog Interface

The SystemVerilog Interface is a place to collect signals together that are thought of as a unit – like a bus. It can contain many other things, including modports and clocking blocks, other interfaces, etc. A SystemVerilog interface is “instanced” just like a module, and can be connected to.

For our purposes we are only concerned with the ability of the interface to provide a scope for hosting our DPI imports and exports.

```
typedef class sequencer;

interface my_interface(input clk);

  uvm_sequencer_base sqr;

  import "DPI-C" context function void c_datatype_array_of_10_int (
                                input int i[10], output int o[10], inout int io[10]);

  import "DPI-C" context function void c_hello(input int inst_id);
  import "DPI-C" context task          c_thread(input int id);

  export "DPI-C"         function      sv_hello;
  export "DPI-C"         task          sv_start_sequenceC;

  function void sv_hello(input int inst_id);
    sequencer my_sqr;
    $cast(my_sqr, sqr);
    $display("sv: Hello! from %m");
    my_sqr.sv_hello(inst_id);
  endfunction

  task sv_start_sequenceC(input int a, output int b);
    sequencer my_sqr;
    $cast(my_sqr, sqr);
    $display("sv: Hello! from %m");
    my_sqr.sv_start_sequenceC(a, b);
  endtask
endinterface
```

The interface above is a simple interface where an import and export are contained. It could contain many other items. The helper functions act to simply forward a call into the interface to the connected sequencer. Only simple wrappers are recommended in the interface. Keep the functionality in the sequencer or agent code.

The UVM Agent.

The agent we use in this solution is just a regular agent. There are no changes in the transactions or the driver or monitor, or the agent itself. Any sequence that calls DPI code will need some changes and the sequencer will need changes.

The agent does one new thing. It initializes the virtual interface handle in the sequencer and it initializes the sequencer handle in the virtual interface. This is the magic that connects the interface to the sequencer and the sequencer to the interface. The agent takes the responsibility to connect the interface and sequencer together.

From the interface a sequencer call can be made, and from the sequencer an interface call can be made.

```
class agent extends uvm_agent;
  `uvm_component_utils(agent)

  driver    d;
  sequencer sqr;

  virtual my_interface vif;



  function void build_phase(uvm_phase phase);
    d = driver::type_id::create("d", this);
    sqr = sequencer::type_id::create("sqr", this);
```

```
    vif.sqr = sqr;
    sqr.vif = vif;
  endfunction
  ...
endclass
```

The agent has the usual functionality, and in the build_phase, we add two additional lines.

```
    vif.sqr = sqr;
    sqr.vif = vif;
```

These two lines connect the virtual interface and the sequencer. This is the only addition or change from a “regular” agent description to a DPI-C enabled agent.

The UVM Sequencer

The oft neglected and much maligned UVM Sequencer is center stage in this solution. It is the common place where we decided to “root” the DPI connection. In reality, any part of the agent could have been used. We root the DPI connection, so that we establish a one-to-one relationship between the interface and the DPI code. Or the agent and the DPI code. This way, managing threads and scope is free. There is nothing to manage, since the interface has a scope and we treat it as if it was part of the agent (really, it is).

First, a handle to the virtual interface is added to the sequencer. Then any helper functions are created. The helper functions could be located elsewhere, but this is a centralized place, and convenient.

```
class sequencer extends uvm_sequencer#(transaction);
  `uvm_component_utils(sequencer)

  virtual my_interface vif;

  ...

  function void sv_hello(int inst_id);
    $display("sv: Hello! from %s. inst_id=%0d", get_full_name(), inst_id);
  endfunction

  task c_start_threads();
    fork
      vif.c_thread(sqr_thread_id++);
      vif.c_thread(sqr_thread_id++);
      vif.c_thread(sqr_thread_id++);
      vif.c_thread(sqr_thread_id++);
    join
  endtask

  task sv_start_sequenceC(input int a, output int b);
    sequenceC seq;
    seq = sequenceC::type_id::create("seqC");
    seq.a = a;
    seq.start(this);
    b = seq.b;
  endtask
endclass
```

There are two kinds of helper functions. The helper function that calls C code (c_start_threads()) and the helper function that calls SV code (sv_start_sequenceC()).

The C example code uses the virtual interface handle to call the hosted C code. The scope is automatically set and managed. In the example here, 4 threads are started at once.

The SV example code creates a sequence and then starts it. Before starting it, any member variables or randomize calls can happen. After the sequence completes, any results can be copied out (seq.b). The idea that a C function can call into a UVM sequencer is very powerful. The C function call gets mapped by the helper functions into single or multiple sequence executions. This is a key part of this solution. Create sequences that perform useful functions and then call them from C code with helpers, using the interface as scope and the sequencer as a class based home for the helpers.

The UVM Sequence

The UVM Sequence in this solution has no changes, unless it needs to call C code. If it is to call C code, then the virtual interface handle is used.

```
typedef class sequencer;

class sequenceA extends uvm_sequence#(transaction);
  `uvm_object_utils(sequenceA)

  int a;
  int b;

  transaction t;
  virtual my_interface vif;

  task body();
    int  i[10];
    int  o[10];
    int io[10];
    sequencer sqr;

    $cast(sqr, m_sequencer);

    vif = sqr.vif;
    vif.c_hello(get_inst_id());

    vif.c_datatype_array_of_10_int (i, o, io);

    for (int i = 0; i < 100; i++) begin
      t = transaction::type_id::create($sformatf("t%0d", i));
      start_item(t);
      if (!t.randomize())
        `uvm_fatal("sequenceA", "Randomization Failed")
      finish_item(t);
    end
  endtask
endclass
```

. In this solution a sequence can retrieve the virtual interface handle from the sequencer it is running on.

```
    sequencer sqr;

    $cast(sqr, m_sequencer);

    vif = sqr.vif;
    vif.c_hello(get_inst_id());

    vif.c_datatype_array_of_10_int (i, o, io);
```

In the case of the call to c_datatype_array_of_10_int, the example C code is called directly from the UVM sequence, using the vif.

```
void
c_datatype_array_of_10_int(const int* i, int* o, int* io)
{
    int j;

    for (j = 0; j < 10; j++) {
      o[j] = i[j];
      io[j] = io[j]+1;
    }
}
```

This simple code just copies input to output and updates the input/output argument.

IV. C CODE

Simple Hello World, with Scope

```
void
c_hello(int inst_id)
{
    const char *scopeName;
    scopeName = svGetNameFromScope(svGetScope());
    printf(" c: Hello! from %s. inst_id=%0d\n", scopeName, inst_id);
    sv_hello(inst_id);
}
```

The simple hello world, is really more than a simple hello world. It is the way that C code is written and can be made to call SystemVerilog using DPI-C. In the SV DPI-C specification, there are a few API calls, like svGetScope() and svGetNameFromScope() that are sometimes useful. Normally you should not need any API calls. That's one of the things that makes DPI easy and powerful. You're just using C code. That's it.

When the SystemVerilog code wants to call C, then an 'import' call is used. When C code wants to call SystemVerilog, then an 'export' call is used.

In the code above, the c_hello() is an import, and sv_hello() is an export. In the C code, to call our SystemVerilog function or task, we simply call it.

C code could be written to perform any useful verification function – like reading a file of golden results, or generating stimulus or collecting statistics. Using our solution, additionally the C code has the power to call sequences. Any sequence that is available and has a helper function defined.

Threaded Code

C code is not normally thread-safe. In SystemVerilog, it is quite easy to create threads and threaded applications. But these threaded applications must be "co-operative". When a SystemVerilog "thread" starts, it has control of the single compute. The only way for a different thread to gain control is for the current thread to give up control or yield.

Yielding can take many forms. If a SystemVerilog thread executes a #delay, or a wait() or a @(posedge clk), then that thread will yield and the next thread will be able to gain control and run.

In the sequencer code above, there is a helper function which starts C threads

```
  task c_start_threads();
    fork
      vif.c_thread(sqr_thread_id++);
      vif.c_thread(sqr_thread_id++);
      vif.c_thread(sqr_thread_id++);
      vif.c_thread(sqr_thread_id++);
    join
```

```
    endtask
```

When this code executes, 4 threads get created. Each of them runs in turn. (Until the running one yields. Only one can run at a time).

In the example code, the task c_thread() calls the helper function sv_start_sequenceC(). This will create a sequence and run it. In that sequence, transactions will be created and randomized, and then start_item() and finish_item() will send them to the driver and on to the interface pins and the device-under-test.

But the C code doesn't need to worry about those details. The C code is a piece of code which calls built in helper functions. The C code and the helper functions must be written in a thread-safe way. For this example, we used a global variable named 'jj', which is not thread-safe – its value will change from the time a thread goes to sleep, to the time the thread wakes up. While a thread sleeps, if the state changes for that thread, then the code is not thread-safe.

```
int jj; // Not a thread safe variable.

int c_thread(int id)
{
    int j;  // A thread safe variable.
    int a, b;

    j = 1;
    jj = 1;
    a = id;
    sv_start_sequenceC(a, &b);
    printf("a=%0d, b=%0d\n", a, b);
    if (j != 1)  printf("Error: 1 mismatch j\n");   j = 2;
    if (jj != 1) printf("Error: 1 mismatch jj\n"); jj = 2;

    sv_start_sequenceC(a, &b);
    printf("a=%0d, b=%0d\n", a, b);
    if (j != 2)  printf("Error: 2 mismatch j\n");   j = 3;
    if (jj != 2) printf("Error: 2 mismatch jj\n"); jj = 3;

    sv_start_sequenceC(a, &b);
    printf("a=%0d, b=%0d\n", a, b);
    if (j != 3)  printf("Error: 3 mismatch j\n");   j = 0;
    if (jj != 3) printf("Error: 3 mismatch jj\n"); jj = 0;

    return 0;
}
```

An example of yielding, using old-school initial blocks is below

```
module top();
  initial
    for (int i = 0; i < 4; i++)
      #0 $display("Ping");
  initial
    for (int i = 0; i < 4; i++)
      #0 $display("Pong");
  final
    $display("@%0t: done", $time);
endmodule

Output
# Ping
# Pong
# Ping
# Pong
# Ping
# Pong
# Ping
# Pong
```

```
module top();
  initial
   for (int i = 0; i < 4; i++)
       $display("Ping");
  initial
   for (int i = 0; i < 4; i++)
       $display("Pong");
  final
     $display("@%0t: done", $time);
 endmodule

 Output
 # Ping
 # Ping
 # Ping
 # Ping
 # Pong
 # Pong
 # Pong
 # Pong
```

Using SystemVerilog fork/join yields the same result.

```
module top();
  initial
    fork
      for (int i = 0; i < 4; i++)
            #0 $display("Ping");
      for (int i = 0; i < 4; i++)
            #0 $display("Pong");
    join
  final
        $display("@%0t: done", $time);
endmodule

Output
# Ping
# Pong
# Ping
# Pong
# Ping
# Pong
# Ping
# Pong
```

```
module top();
  initial
    fork
      for (int i = 0; i < 4; i++)
               $display("Ping");
      for (int i = 0; i < 4; i++)
               $display("Pong");
    join
  final
        $display("@%0t: done", $time);
endmodule

Output
# Ping
# Ping
# Ping
# Ping
# Pong
# Pong
# Pong
# Pong
```

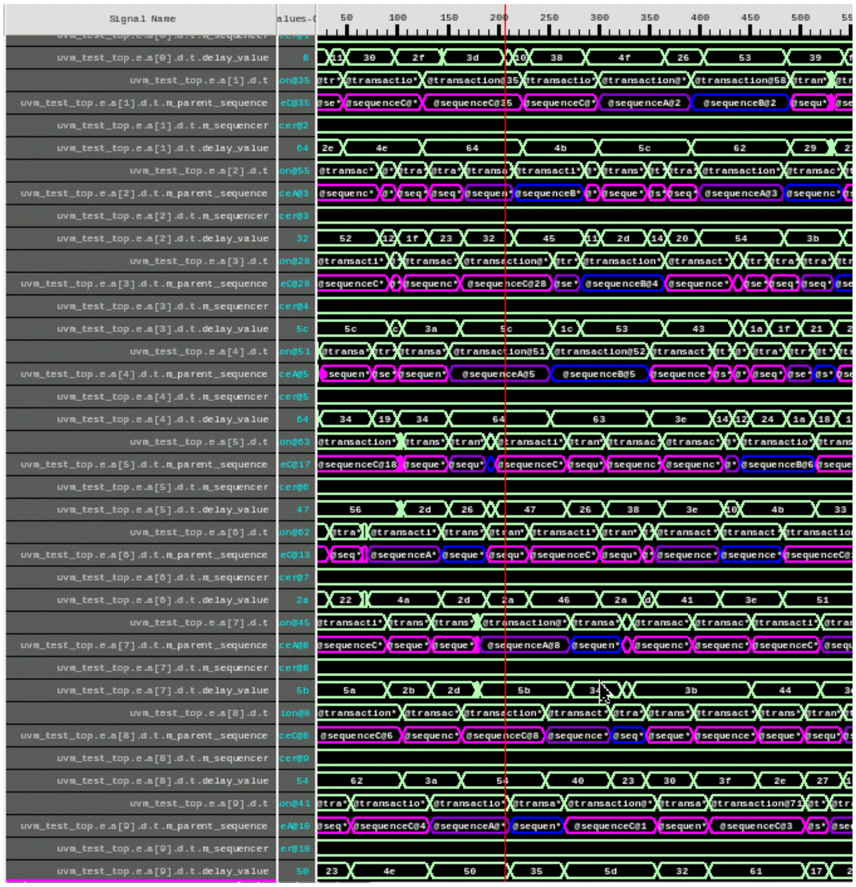

Figure 4 - Pink for C sequences, Blue and Purple for SV sequences

In Figure 4 the example code is running, and for each agent there is a line of colored class handles. The colors represent the type of sequence that is running at that moment on that agent. Pink for C generated sequences and Blue and Purple for SystemVerilog generated sequences. They all take turns – all background, or “normal” traffic is sequenced on the interface, as well as the new C based sequences.

This ability to seamlessly integrate the UVM background traffic and the C generated traffic key a key point in this solution.

V. THE C CODE

The goal of the C code is to interact with the device under test in some way. Either by generating input or monitoring output or some of both. Interacting with the DUT is under control of the UVM, so any interaction from C must follow the rules. Using the UVM sequencer and sequences is an easy an efficient way to do this.

All operations that might be imagined in C need to be broken down into sub-processing with each sub-process represented by a UVM sequence. For example, a large data transfer from C to the DUT would be implemented as a call to a “large-transfer” sequence, or many calls to a byte-wise transfer sequence.

C Bus transfers are tests which cause a transfer or transaction on a bus. For example, a READ or WRITE. Each call to or from C is a bus transfer. The C call in turn creates a bus transfer sequence and executes it.

There are many ways C code can be used, all of which are beyond the scope of this paper to describe in detail. Some examples include:

Stimulus Generator

Simply a stimulus generator. Data will be created on the C side and sent to the SV side with a DPI-C call.

Data checker

A golden model written in C. Data from the DUT will be monitored and moved to the C side by calling an import task or function.

Bus transfer generator

A program on the C side which issues bus READs and WRITEs. The C program has little knowledge that it is running in a SystemVerilog simulation.

VI. CONCLUSION

Using DPI-C is easy and powerful. Using a SystemVerilog interface, many of the integration and connection issues can be eliminated. Using the techniques outlined above large, threaded C tests can be created easily. Please contact the author for downloadable source code to get started and experiment with DPI-C.

VII. REFERENCES

[1] SystemVerilog Language Reference Manual, http://standards.ieee.org/getieee/1800/download/1800-2012.pdf

[2] UVM Language Reference Manual, http://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf

[3] *"DPI Redux. Functionality. Speed. Optimization.",* DVCON 2017, Rich Edelman, Rohit Jain, Hui Yin.

VIII. APPENDIX

The C Code

```
  #include <stdio.h>
  #include "dpiheader.h"

  void
  c_hello(int inst_id)
  {
      const char *scopeName;
      scopeName = svGetNameFromScope(svGetScope());
      printf("  c:  Hello!  from  %s.  inst_id=%0d\n",
scopeName, inst_id);
      sv_hello(inst_id);
  }

  int jj; // Not a thread safe variable.

  int c_thread(int id)
  {
      int j;  // A thread safe variable.
      int a, b;

      j = 1;
      jj = 1;
      a = id;
      sv_start_sequenceC(a, &b);
      printf("a=%0d, b=%0d\n", a, b);
      if (j != 1)  printf("Error: 1 mismatch j\n"); j
= 2;
      if (jj != 1) printf("Error: 1 mismatch jj\n"); jj
= 2;

      sv_start_sequenceC(a, &b);
      printf("a=%0d, b=%0d\n", a, b);
      if (j != 2)  printf("Error: 2 mismatch j\n"); j
= 3;
      if (jj != 2) printf("Error: 2 mismatch jj\n"); jj
= 3;

      sv_start_sequenceC(a, &b);
      printf("a=%0d, b=%0d\n", a, b);
      if (j != 3)  printf("Error: 3 mismatch j\n"); j
= 0;
      if (jj != 3) printf("Error: 3 mismatch jj\n"); jj
= 0;
```

```
      return 0;
  }

  /* --------------------------------- */

  void
  c_datatype_2d_array_of_int(
      const int* i,
      int* o,
      int* io)
  {
  }

  void
  c_datatype_array_of_10_int(
      const int* i,
      int* o,
      int* io)
  {
      int j;

      for (j = 0; j < 10; j++) {
        o[j] = i[j];
        printf("datatype::  C:  i  array[%d]=%d\n",  j,
i[j]);
        printf("datatype::  C:  o  array[%d]=%d\n",  j,
o[j]);
        printf("datatype::  C:io  array[%d]=%d\n",  j,
io[j]);
        io[j] = io[j]+1;
        printf("datatype::  C:io  array[%d]=%d\n",  j,
io[j]);
      }
  }

  void
  c_datatype_bit(
      svBit i,
      svBit* o,
      svBit* io)
  {
  }

  void
```

```
c_datatype_bit2(
    const svBitVecVal* i,
    svBitVecVal* o,
    svBitVecVal* io)
{
}

void
c_datatype_bit33(
    const svBitVecVal* i,
    svBitVecVal* o,
    svBitVecVal* io)
{
}

void
c_datatype_enum(
    const svLogicVecVal* i,
    svLogicVecVal* o,
    svLogicVecVal* io)
{
}

void
c_datatype_logic(
    svLogic i,
    svLogic* o,
    svLogic* io)
{
}

void
c_datatype_logic2(
    const svLogicVecVal* i,
    svLogicVecVal* o,
    svLogicVecVal* io)
{
}

void
c_datatype_logic33(
    const svLogicVecVal* i,
    svLogicVecVal* o,
    svLogicVecVal* io)
{
}

void
c_datatype_openarray_of_int(
    const svOpenArrayHandle i,
    const svOpenArrayHandle o,
    const svOpenArrayHandle io)
{
}

void
c_datatype_real(
    double i,
    double* o,
    double* io)
{
}

void
c_datatype_shortreal(
    float i,
    float* o,
    float* io)
{
}

void
c_datatype_struct(
    const struct_t* i,
    struct_t* o,
    struct_t* io)
{
}

void
c_datatype_struct_packed(
    const svLogicVecVal* i,
    svLogicVecVal* o,
    svLogicVecVal* io)
{
}
```

The SystemVerilog Interface

```
typedef enum logic [1:0] {
  GO,         // All transfers complete for this id.
  READ,       // Read 32 bits.
  WRITE,      // Write 32 bits.
  WRITE_REAL  // Write a REAL.
} op_t;

typedef struct {
  int x;
  byte y;
} simple_struct_t;

typedef struct {
  int a;
  bit b;
  simple_struct_t simple_struct;
  simple_struct_t simple_struct10[10];
  bit   [10:0] eleven_bits;
  logic [10:0] eleven_logics;
  bit   [10:0] eleven_bits3[3];
  logic [10:0] eleven_logics4[4];
} struct_t;

typedef struct packed {
  int a;
  bit b;
  bit   [10:0] eleven_bits;
  logic [10:0] eleven_logics;
} struct_packed_t;

typedef class sequencer;
interface my_interface(input clk);
```

```
    import "DPI-C" context function void c_datatype_enum         (input op_t         i, output op_t         o,
inout op_t         io);

    import "DPI-C" context function void c_datatype_bit          (input bit          i, output bit          o,
inout bit          io);
    import "DPI-C" context function void c_datatype_logic        (input logic        i, output logic        o,
inout logic        io);
    import "DPI-C" context function void c_datatype_bit2         (input bit    [1:0] i, output bit    [1:0] o,
inout bit    [1:0] io);
    import "DPI-C" context function void c_datatype_bit33        (input bit   [32:0] i, output bit   [32:0] o,
inout bit   [32:0] io);
    import "DPI-C" context function void c_datatype_logic2       (input logic  [1:0] i, output logic  [1:0] o,
inout logic  [1:0] io);
    import "DPI-C" context function void c_datatype_logic33      (input logic [32:0] i, output logic [32:0] o,
inout logic [32:0] io);

    import "DPI-C" context function void c_datatype_struct        (input struct_t        i, output struct_t
o, inout struct_t        io);
    import "DPI-C" context function void c_datatype_struct_packed(input struct_packed_t i, output struct_packed_t
o, inout struct_packed_t io);

    import "DPI-C" context function void c_datatype_real         (input real      i, output real      o, inout
real      io);
    import "DPI-C" context function void c_datatype_shortreal    (input shortreal i, output shortreal o, inout
shortreal io);

    import "DPI-C" context function void c_datatype_array_of_10_int (input int i[10], output int o[10], inout
int io[10]);
    import "DPI-C" context function void c_datatype_openarray_of_int(input int i[],   output int o[],   inout
int io[]);

    import "DPI-C" context function void c_datatype_2d_array_of_int (input int i[10][5], output int o[10][5],
inout int io[10][5]);

    //import "DPI-C" context function void c_datatype_queue_of_int(input int i[$], output int o[$], inout int
io[$]);
    //import "DPI-C" context function void c_datatype_associative_array_of_int(input int i[int], output int
o[int], inout int io[int]);

    uvm_sequencer_base sqr;

    import "DPI-C" context function void c_hello(input int inst_id);
    import "DPI-C" context task          c_thread(input int id);

    export "DPI-C"         function      sv_hello;
    export "DPI-C"         task          sv_start_sequenceC;

    function void sv_hello(input int inst_id);
          sequencer my_sqr;
          $cast(my_sqr, sqr);
      $display("sv: Hello! from %m");
          my_sqr.sv_hello(inst_id);
    endfunction

    task sv_start_sequenceC(input int a, output int b);
          sequencer my_sqr;
          $cast(my_sqr, sqr);
      $display("sv: Hello! from %m");
          my_sqr.sv_start_sequenceC(a, b);
    endtask

  endinterface

  typedef virtual my_interface viflist_t[10];
```

The UVM Testbench

```
  import uvm_pkg::*;
  `include "uvm_macros.svh"

  `include "interface.svh"

  class transaction extends uvm_sequence_item;
    `uvm_object_utils(transaction)
    rand int delay_value;
    constraint delay_value_constraint {
      delay_value >=   0;
      delay_value <= 100;
    }
```

```
  function new(string name = "transaction");
    super.new(name);
  endfunction

  function string convert2string();
    return $sformatf("delay_value=%0d", delay_value);
  endfunction
endclass

typedef class sequencer;

class sequenceA extends uvm_sequence#(transaction);
  `uvm_object_utils(sequenceA)
  function new(string name = "sequenceA");
    super.new(name);
  endfunction

  int a;
  int b;

  transaction t;
  virtual my_interface vif;

  task body();
    sequencer sqr;
    $cast(sqr, m_sequencer);

    vif = sqr.vif;
    vif.c_hello(get_inst_id());

    begin // Datatype checks...

      int  i[10];
      int  o[10];
      int io[10];

      for (int j = 0; j < 10; j++) begin
         i[j] =     j+1;
        io[j] = 10*(j+1);
      end
      vif.c_datatype_array_of_10_int (i, o, io);
      for (int j = 0; j < 10; j++) begin
        $display("datatype::SV: o[%0d]=%0d", j,  o[j]);
        $display("datatype::SV:io[%0d]=%0d", j, io[j]);
      end
    end

    for (int i = 0; i < 100; i++) begin
      t = transaction::type_id::create($sformatf("t%0d", i));
      start_item(t);
      if (!t.randomize())
        `uvm_fatal("sequenceA", "Randomization Failed")
      finish_item(t);
      b = t.delay_value;
    end
  endtask
endclass

class sequenceB extends sequenceA;
  `uvm_object_utils(sequenceB)
  function new(string name = "sequenceB");
    super.new(name);
  endfunction
endclass

class sequenceC extends sequenceB;
  `uvm_object_utils(sequenceC)
  function new(string name = "sequenceC");
    super.new(name);
  endfunction
endclass

class sequencer extends uvm_sequencer#(transaction);
  `uvm_component_utils(sequencer)
  virtual my_interface vif;
  function new(string name = "sequencer", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  int sqr_thread_id = 1;
```

```
  // ---------------
  // Helper Routines
  // ---------------

  function void sv_hello(int inst_id);
    $display("sv: Hello! from %s. inst_id=%0d", get_full_name(), inst_id);
  endfunction

  task c_start_threads();
    fork
      vif.c_thread(sqr_thread_id++);
      vif.c_thread(sqr_thread_id++);
      vif.c_thread(sqr_thread_id++);
      vif.c_thread(sqr_thread_id++);
    join
  endtask

  task sv_start_sequenceC(input int a, output int b);
    sequenceC seq;
    seq = sequenceC::type_id::create("seqC");
    seq.a = a;
    seq.start(this);
    b = seq.b;
  endtask
endclass

class driver extends uvm_driver#(transaction);
  `uvm_component_utils(driver)
  function new(string name = "driver", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  transaction t;

  task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(t);
      `uvm_info(get_type_name(), t.convert2string(), UVM_MEDIUM)
      #(t.delay_value);
      seq_item_port.item_done();
    end
  endtask
endclass

class agent extends uvm_agent;
  `uvm_component_utils(agent)

  driver      d;
  sequencer sqr;

  virtual my_interface vif;

  function new(string name = "agent", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    d = driver::type_id::create("d", this);
    sqr = sequencer::type_id::create("sqr", this);
    vif.sqr = sqr;
    sqr.vif = vif;
  endfunction

  function void connect_phase(uvm_phase phase);
    d.seq_item_port.connect(sqr.seq_item_export);
  endfunction
endclass

class env extends uvm_env;
  `uvm_component_utils(env)

  agent a[10];

  viflist_t viflist;

  function new(string name = "env", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    if (!uvm_config_db#(viflist_t)::get( this, "", "viflist", viflist))
```

```
      `uvm_fatal(get_type_name(), "VIF lookup failed")
    for (int i = 0; i < 10; i++) begin
      a[i] = agent::type_id::create($sformatf("a[%0d]", i), this);
      a[i].vif = viflist[i];
    end
  endfunction
endclass

class test extends uvm_test;
  `uvm_component_utils(test)
  env e;

  sequenceA seqA[10];
  sequenceB seqB[10];

  function new(string name = "test", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    e = env::type_id::create("e", this);
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    e.viflist[0].c_hello(get_inst_id());
    e.viflist[1].c_hello(get_inst_id());

    for (int i = 0; i < 10; i++)
       seqA[i] = sequenceA::type_id::create($sformatf("seqA[%0d]", i));

    for (int i = 0; i < 10; i++)
       seqB[i] = sequenceB::type_id::create($sformatf("seqB[%0d]", i));

    for (int i = 0; i < 10; i++)
      fork
        int j = i;
        seqA[j].start(e.a[j].sqr);
        seqB[j].start(e.a[j].sqr);
      join_none

    for (int i = 0; i < 10; i++)
      fork
        int j = i;
        e.a[j].sqr.c_start_threads();
      join_none

    wait fork;
    phase.drop_objection(this);
  endtask
endclass

module top();
  reg clk;

  my_interface interface0(clk);
  my_interface interface1(clk);
  my_interface interface2(clk);
  my_interface interface3(clk);
  my_interface interface4(clk);
  my_interface interface5(clk);
  my_interface interface6(clk);
  my_interface interface7(clk);
  my_interface interface8(clk);
  my_interface interface9(clk);

  viflist_t viflist;

  initial begin
    viflist[0] = interface0;
    viflist[1] = interface1;
    viflist[2] = interface2;
    viflist[3] = interface3;
    viflist[4] = interface4;
    viflist[5] = interface5;
    viflist[6] = interface6;
    viflist[7] = interface7;
    viflist[8] = interface8;
    viflist[9] = interface9;
```

```
    uvm_config_db#(viflist_t)::set( null, "*", "viflist", viflist);

    run_test();
  end

  always begin
    #10; clk = 0;
    #10; clk = 1;
  end
endmodule
```