



Having Your Cake and Eating It Too

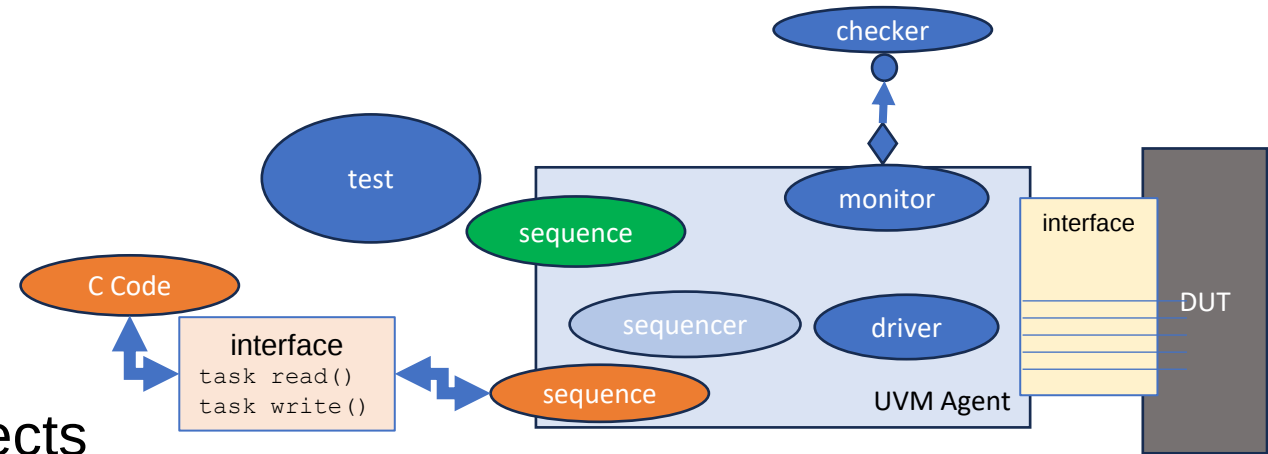
Programming UVM Sequences with C Code

Tomoki Watanabe & Rich Edelman
Siemens Japan & Siemens EDA



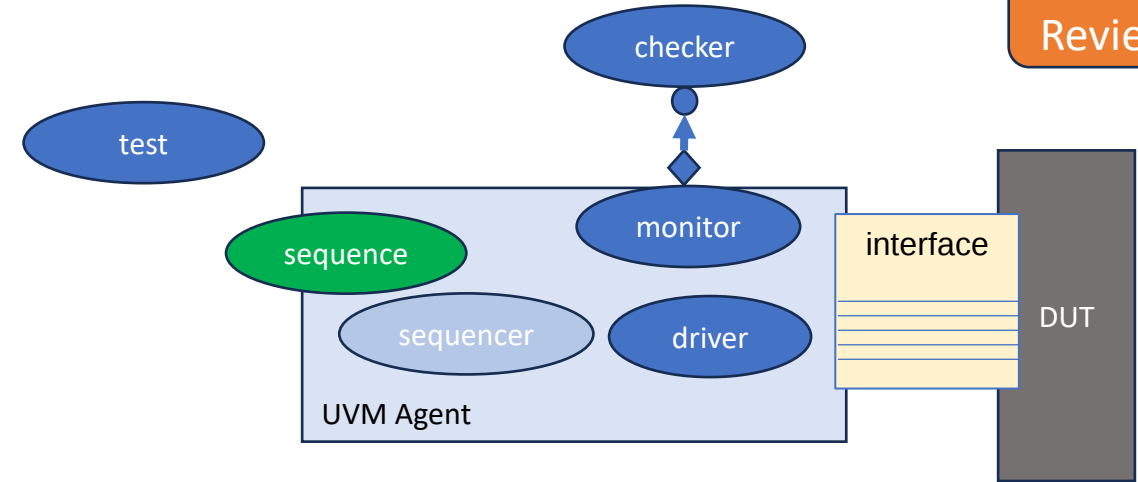
Motivation

- Reuse our UVM testbench as-is
- Add the ability to have C code
 - Generate traffic
 - Run a C test program from the architects
 - Reuse architects' performance C tests
- Solution #1:
 - Use DPI-C
 - Write C code calling tasks in SystemVerilog
 - But the tasks need to be aware of WHICH sequence is being used on which interface in the UVM testbench
- Problem:
 - We cannot “host” the DPI-C calls in a class
 - How to connect the class to C code.
- Solution #2:
 - The SystemVerilog Interface



The UVM

- Typical UVM
 - SystemVerilog
 - test
 - sequence
 - sequencer
 - driver
 - monitor
 - interface
 - checker
 - ...



What does a 'test' do? – It's a “coordinator of programs” to run

- Orchestrates starting “sequences”
- Checks results?
- Has a timeout?

What does a 'sequence' do? – It's a “program” to run

- Creates transactions
- Sends transactions to the sequencer and on to the driver
- Checks results?

What does a 'driver' do? – It's a “pin wiggler”

- Receives transactions
- Turns the transaction into pin wiggles on the interface
- Sends results back to the sequence via the transaction

DPI in the LRM

- Chapter H – “DPI C Layer” (33 pages)
- Chapter 35 – “Direct Programming Interface” (15 pages)

Using DPI-C

- Export SystemVerilog code for C to call
- Import C code for SystemVerilog to call
- Rules...
 - DPI-C can be defined inside

H.9.2 Context of imported and exported tasks and functions

DPI imported and exported tasks and functions can be declared in a **module**, **program**, **interface**, **package**, compilation unit scope, or **generate** declarative scope.

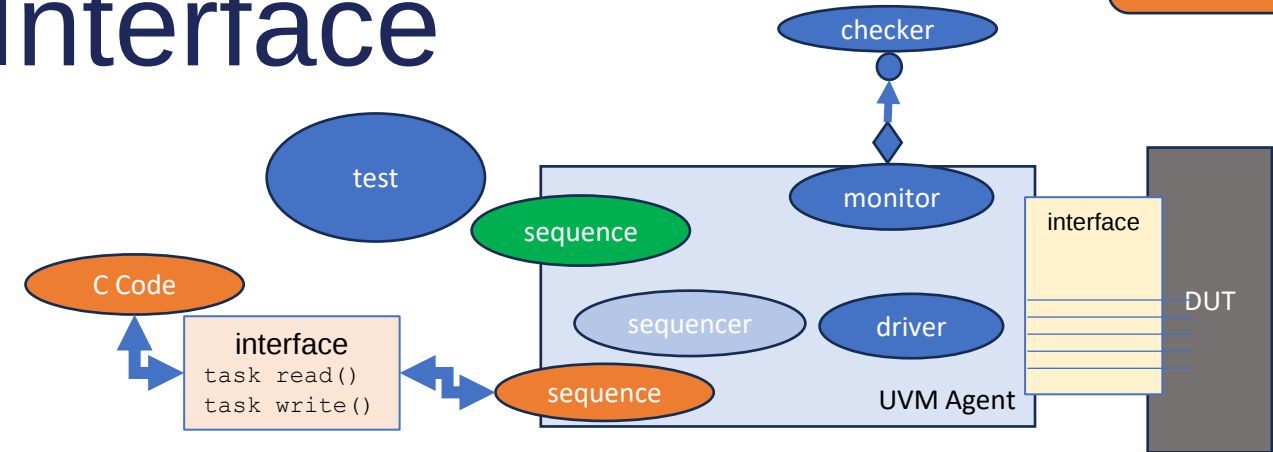
- But NOT in classes

The SystemVerilog Interface

```
interface zinterface();
  export "DPI-C" task read;
  export "DPI-C" task write;
  import "DPI-C" context task test_program1(...);

  task read(int index, int addr, output int data);
    ...
  endtask

  task write(int index, int addr, int data);
    ...
  endtask
endinterface
```



- SystemVerilog interfaces are closely related to modules
 - They can be instantiated - usually thought of as collections of wires
 - A “virtual interface” handle can be passed to a class
 - The class has a handle to the interface and can call tasks and functions in the interface
 - “Regular” tasks and functions can be defined inside them
 - DPI-C tasks and functions CAN be defined inside them

Running a Sequence

- The 'test' might start a sequence
 - And the body() task runs – 1000 transactions are created and “started”

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

  env e1;
  seq s1;

  function void build_phase(uvm_phase phase);
    e1 = env::type_id::create("e1", this);
  endfunction

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

    s1 = seq::type_id::create("s1");

    s1.start(e1.a.sqr);

    phase.drop_objection(this);
  endtask
endclass
```

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

  transaction t;

  task body();
    for (int i = 0; i < 1000; i++) begin
      t = transaction::type_id::create(name);
      start_item(t);
      finish_item(t);
    end
  endtask
endclass
```

Adding Specialized Tasks to the Sequence

```
class seq extends uvm_sequence#(transaction);  
    `uvm_object_utils(seq)  
  
    transaction t;  
  
    task body();  
        for (int i = 0; i < 1000; i++) begin  
            t = transaction::type_id::create(name);  
            start_item(t);  
            finish_item(t);  
        end  
    endtask
```

```
task read(bit [31:0] addr, output bit [31:0]data);  
    t = transaction::type_id::create("read");  
    t.rw = READ;  
    t.addr = addr;  
    t.data = 0;  
  
    start_item(t);  
    finish_item(t);  
  
    data = t.data;  
endtask  
  
task write(bit [31:0] addr, bit [31:0]data);  
    t = transaction::type_id::create("write");  
    t.rw = WRITE;  
    t.addr = addr;  
    t.data = data;  
  
    start_item(t);  
    finish_item(t);  
endtask  
endclass
```

Zombie Sequence

```
class zombie_seq extends seq;  
  `uvm_object_utils(zombie_seq)  
  
  transaction t;  
  bit done;  
  
  task body();  
  
    wait (done == 1);  
  
  endtask  
endclass
```

```
task read(bit [31:0] addr, output bit [31:0] data);  
  t = transaction::type_id::create("read");  
  t.rw = READ;  
  t.addr = addr;  
  t.data = 0;  
  
  start_item(t);  
  finish_item(t);  
  
  data = t.data;  
endtask  
  
task write(bit [31:0] addr, bit [31:0] data);  
  t = transaction::type_id::create("write");  
  t.rw = WRITE;  
  t.addr = addr;  
  t.data = data;  
  
  start_item(t);  
  finish_item(t);  
endtask  
endclass
```

This is the “API” that C coders will use – make it as needed – this is a very simple one

Running a Zombie Sequence

- The zombie sequence gets started by the test just like a “regular” sequence

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

  env e1;
  zombie_seq zs1;

  function void build_phase(uvm_phase phase);
    e1 = env::type_id::create("e1", this);
  endfunction

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

    zs1 = seq::type_id::create("s1");
    zs1.start(e1.a.sqr);

    phase.drop_objection(this);
  endtask
endclass
```

```
class zombie_seq extends seq;
  `uvm_object_utils(zombie_seq)

  transaction t;
  bit done;

  task body();

    wait (done == 1);

  endtask
```

Defining a DPI-C interface

- A special “trampoline” to bounce between C and SV

```
interface zinterface();  
    uvm_object registered_seq[int];
```

```
export "DPI-C" task read;  
export "DPI-C" task write;
```

Defined in SV – in this interface

```
import "DPI-C" context zinterface_start_test_program1 =  
    task start_test_program1(int index, string name, int start_addr);
```

...

Defined in C – this is the C test program

DPI-C interface 2

- A function is defined to add SV class handles to an array
- Look-up by simple index
- Each C thread has an index number

```
interface zinterface();  
    uvm_object registered_seq[int];  
    ...  
    function void register(int index, uvm_object seq);  
        zinterface_zombieseq zsq;  
        registered_seq[index] = seq;  
        $cast(zsq, seq);  
        zsq.vif = interface::self(); // Extension to the LRM  
    endfunction  
    ...
```

DPI-C interface 3

C Code

...

read()

...

write()

```
interface zinterface();  
    uvm_object registered_seq[int];  
    ...  
    task read(int index, int addr, output int data);  
        zinterface_zombieseq zsq;  
        $cast(zsq, registered_seq[index]);  
        zsq.read(addr, data);  
    endtask  
  
    task write(int index, int addr, int data);  
        zinterface_zombieseq zsq;  
        $cast(zsq, registered_seq[index]);  
        zsq.write(addr, data);  
    endtask  
endinterface
```

C Code calling the interface tasks

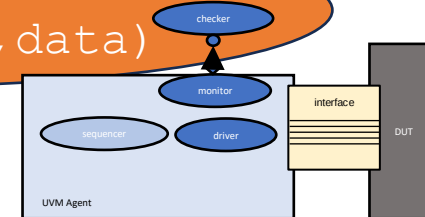
- This is just “DPI-C” export

C Code
`read(thread, addr, data)`

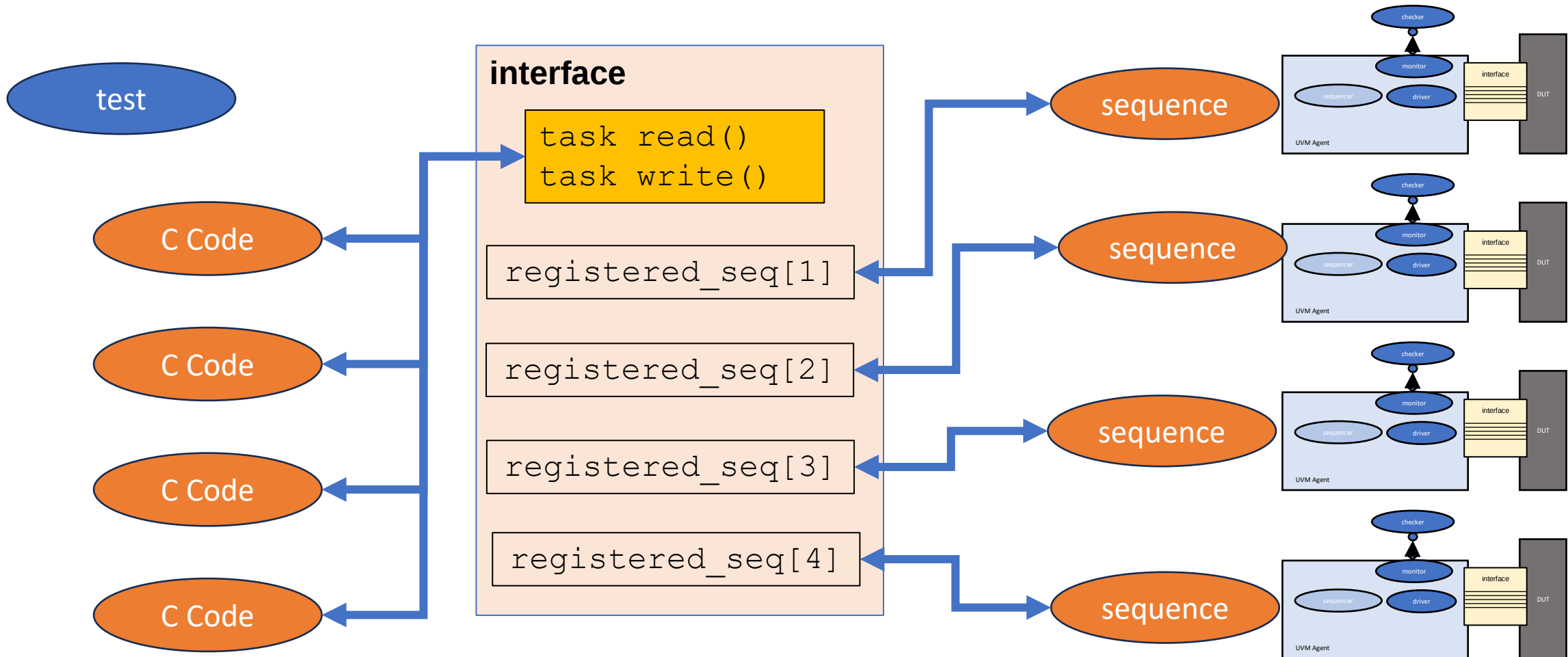
interface

```
task read(int index, int addr, output int data);  
  zinterface_zombieseq zsq;  
  $cast(zsq, registered_seq[index]);  
  zsq.read(addr, data);  
endtask
```

sequence
`task read(addr, data)`



The testbench from the C point of view



The C code – The Test Program

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

```
int
zinterface_start_test_program1(int index, const char *name, int start_addr) {
    int addr, data;
```

```
    // Repeat 10 times, changing the data
```

Repeat 10 times, changing 'data' each time

```
    for (dataloops = 0; dataloops < 10; dataloops++) {
        // Repeat 10 times - writing 10, and reading 10
```

```
        for (loops = 0; loops < 10; loops++) {
```

```
            for (addr = start_addr; addr < start_addr+10; addr++) {
```

Cycle through the addresses, do 'write'

```
                data = ...;
```

```
                write(index, addr, data);
```

```
            for (addr = start_addr; addr < start_addr+10; addr++) {
```

Cycle through the addresses, do 'read' and compare

```
                read(index, addr, &data);
```

```
                if (data != addr + 1000 + dataloops) {
```

```
                    printf("C: ...ERROR READ (%0d, %0d) <%s> [wrote: %d, read %d]  \n",
```

```
                        addr, data, name, data, addr + 1000 + dataloops);
```

```
                }
```

```
            }
```

```
            start_addr += 10;
```

```
        }
    }
    return 0;
```

```
interface zinterface();
export "DPI-C" task read;
export "DPI-C" task write;
import "DPI-C" context zinterface_start_test_program1 =
    task start_test_program1(int index, string name, int start_addr);
```

The C code – The Test Program - Executing

Pseudo-code

Given a start_address, do the following steps:

Repeat 10 times with different 'data' values

- Repeat 10 times
 - Repeat 10 times
 - WRITE data at 'address'
 - increment 'address' by 1
 - Reset address
- Repeat 10 times
 - READ data at 'address'
 - Compare Actual == Expected
 - increment 'address' by 1
- Increment start_address by 10

```
# C: ...executed WRITE(200, 1200) <thread2>
# C: ...executed WRITE(201, 1201) <thread2>
# C: ...executed WRITE(202, 1202) <thread2>
# C: ...executed WRITE(203, 1203) <thread2>
# C: ...executed WRITE(204, 1204) <thread2>
# C: ...executed WRITE(205, 1205) <thread2>
# C: ...executed WRITE(206, 1206) <thread2>
# C: ...executed WRITE(207, 1207) <thread2>
# C: ...executed WRITE(208, 1208) <thread2>
# C: ...executed WRITE(209, 1209) <thread2>
# C: ...executed READ (200, 1200) <thread2>
# C: ...executed READ (201, 1201) <thread2>
# C: ...executed READ (202, 1202) <thread2>
# C: ...executed READ (203, 1203) <thread2>
# C: ...executed READ (204, 1204) <thread2>
# C: ...executed READ (205, 1205) <thread2>
# C: ...executed READ (206, 1206) <thread2>
# C: ...executed READ (207, 1207) <thread2>
# C: ...executed READ (208, 1208) <thread2>
# C: ...executed READ (209, 1209) <thread2>
# C: ...executed WRITE(210, 1210) <thread2>
# C: ...executed WRITE(211, 1211) <thread2>
# C: ...executed WRITE(212, 1212) <thread2>
# C: ...executed WRITE(213, 1213) <thread2>
# C: ...executed WRITE(214, 1214) <thread2>
# C: ...executed WRITE(215, 1215) <thread2>
# C: ...executed WRITE(216, 1216) <thread2>
# C: ...executed WRITE(217, 1217) <thread2>
# C: ...executed WRITE(218, 1218) <thread2>
# C: ...executed WRITE(219, 1219) <thread2>
# C: ...executed READ (210, 1210) <thread2>
# C: ...executed READ (211, 1211) <thread2>
# C: ...executed READ (212, 1212) <thread2>
```

Just thread2

```
# C: ...executed READ (125, 1134) <thread1>
# C: ...executed READ (126, 1135) <thread1>
# C: ...executed READ (127, 1136) <thread1>
# C: ...executed READ (208, 1217) <thread2>
# C: ...executed WRITE(410, 1419) <thread4>
# C: ...executed WRITE(331, 1340) <thread3>
# C: ...executed READ (128, 1137) <thread1>
# C: ...executed READ (129, 1138) <thread1>
# C: ...executed WRITE(411, 1420) <thread4>
# C: ...executed READ (209, 1218) <thread2>
# C: ...executed WRITE(130, 1139) <thread1>
# C: ...executed WRITE(333, 1342) <thread3>
# C: ...executed WRITE(210, 1219) <thread2>
# C: ...executed WRITE(412, 1421) <thread4>
# C: ...executed WRITE(334, 1343) <thread3>
# C: ...executed WRITE(131, 1140) <thread1>
# C: ...executed WRITE(413, 1422) <thread4>
# C: ...executed WRITE(211, 1220) <thread2>
# C: ...executed WRITE(335, 1344) <thread3>
# C: ...executed WRITE(414, 1423) <thread4>
# C: ...executed WRITE(336, 1345) <thread3>
# C: ...executed WRITE(132, 1141) <thread1>
# C: ...executed WRITE(212, 1221) <thread2>
```

All the threads

Changes to the original test

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

```
env e1, e2, e3, e4;
seq s1, s2, s3, s4;
```

```
zinterface_zombieseq zs1, zs2, zs3, zs4;
```

```
virtual zinterface zif;
```

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

```
function void build_phase(uvm_phase phase);
  e1 = env::type_id::create("e1", this);
  e2 = env::type_id::create("e2", this);
  e3 = env::type_id::create("e3", this);
  e4 = env::type_id::create("e4", this);
endfunction
```

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

```
s1 = seq::type_id::create("s1");
s2 = seq::type_id::create("s2");
s3 = seq::type_id::create("s3");
s4 = seq::type_id::create("s4");
```

```
if (!uvm_config_db#(virtual zinterface)::get(
  this, "*", "zinterface", zif))
  `uvm_fatal(get_type_name(),
    "cannot find ZINTERFACE INSTANCE")
```

Declare the zombie
sequence handles
and the virtual
interface

Lookup the interface
in the config db

Construct the
zombie sequences

```
zs1 = zinterface_zombieseq::type_id::create("zs1");
zs2 = zinterface_zombieseq::type_id::create("zs2");
zs3 = zinterface_zombieseq::type_id::create("zs3");
zs4 = zinterface_zombieseq::type_id::create("zs4");
```

Register with the
interface

```
zif.register(1, zs1);
zif.register(2, zs2);
zif.register(3, zs3);
zif.register(4, zs4);
```

Start the zombie
sequences

```
fork
  zs1.start(e1.a.sqr);
  zs2.start(e2.a.sqr);
  zs3.start(e3.a.sqr);
  zs4.start(e4.a.sqr);
join_none
```

Start the C code test
programs

```
fork
  s1.start(e1.a.sqr);
  s2.start(e2.a.sqr);
  s3.start(e3.a.sqr);
  s4.start(e4.a.sqr);
join
```

```
fork
  zs1.start_test_program1(1, "thread1", 100);
  zs2.start_test_program1(2, "thread2", 200);
  zs3.start_test_program1(3, "thread3", 300);
  zs4.start_test_program1(4, "thread4", 400);
join
```

```
phase.drop_objection(this);
endtask
endclass
```

Changes to the original top

- Instantiate the special zombie interface for the 'zinterface' API
- Put the virtual interface handle into the config db

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

import ip_pkg::*;

module top();
    memory_interface memory_interface_instance();

    zinterface I_zinterfacel();

    initial begin
        uvm_config_db#(virtual memory_interface)::set(
            uvm_root::get(), "*", "memory_interface", memory_interface_instance);

        uvm_config_db#(virtual zinterface)::set(
            uvm_root::get(), "*", "zinterface", I_zinterfacel);

        run_test();
    end
endmodule
```

Instantiate the
interface

Put the interface in
the config database

Results – 4 “zombie” threads / streams

Signal Name	Values C1	15950	16000	16050	16100	16150	16200														
Transaction		Transaction																			
uvvm_test_top.e1.a.sqr.zs1	S0'h00000004	read	write	write	write	write	write	write	write	write	write	read	read	read	read	read	read	read	read	write	
id	3'd0	2	6	1	5	1	4	0	3	7	3	5	2	0	3	7	3	0	3	6	
serial_number	32'd4608	4546	4550	4553	4557	4561	4564	4568	4571	4575	4579	4581	4586	4592	4595	4599	4603	4608	4611	4614	
rw	READ	READ	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	READ	READ	READ	READ	READ	READ	READ	WRITE	
addr	32'd175	169	170	171	172	173	174	175	176	177	178	179	170	171	172	173	174	175	176	177	
data	32'd1175	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1170	1171	1172	1173	1174	1175	1176	1177	
delay	32'd4	4	5	5	6	4	7	6	8	9	7	8	5	4	6	8	8	4	4	8	
uvvm_test_top.e2.a.sqr.zs2	S0'h00000008	read	read	read	read	read	write	write	write	write	write	write	write	write	write	write	read	read	read	read	
id	3'd7	1	5	2	6	2	6	2	5	1	7	3	7	4	0	4	7	4	0	4	
serial_number	32'd4607	4545	4549	4554	4558	4562	4566	4570	4573	4577	4583	4587	4591	4596	4600	4604	4607	4612	4616	4620	
rw	READ	READ	READ	READ	READ	READ	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	READ	READ	READ	READ	
addr	32'd270	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	270	271	272	273	
data	32'd1270	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279	1270	1271	1272	1273	
delay	32'd8	4	4	9	9	6	8	5	5	9	7	6	5	9	6	7	8	9	5	4	
uvvm_test_top.e3.a.sqr.zs3	S0'h00000004	read	read	read	write	write	write	write	write	write	write	write	write	write	read	read	read	read	read	read	
id	3'd2	0	4	0	4	0	5	1	6	2	6	1	5	1	5	2	6	2	5	1	
serial_number	32'd4610	4548	4552	4556	4560	4565	4569	4574	4578	4582	4585	4589	4593	4597	4602	4606	4610	4613	4617	4621	
rw	READ	READ	READ	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	READ	READ	READ	READ	READ	READ	
addr	32'd372	367	368	369	370	371	372	373	374	375	376	377	378	379	370	371	372	373	374	375	
data	32'd1372	1367	1368	1369	1370	1371	1372	1373	1374	1375	1376	1377	1378	1379	1370	1371	1372	1373	1374	1375	
delay	32'd4	6	4	8	8	7	8	7	7	7	4	7	4	5	8	9	7	4	4	6	
uvvm_test_top.e4.a.sqr.zs4	S0'h00000008	read	read	read	read	read	read	write	write	write	write	write	write	write	write	write	read	read	read	read	
id	3'd1	6	3	7	3	7	3	7	4	0	4	0	4	6	2	6	1	5	1	7	
serial_number	32'd4609	4547	4551	4555	4559	4563	4567	4572	4576	4580	4584	4588	4590	4594	4598	4601	4605	4609	4615	4619	
rw	READ	READ	READ	READ	READ	READ	READ	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	WRITE	READ	READ	READ	
addr	32'd470	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	470	471	472	
data	32'd1470	1464	1465	1466	1467	1468	1469	1470	1471	1472	1473	1474	1475	1476	1477	1478	1479	1470	1471	1472	
delay	32'd8	6	9	7	7	8	7	5	5	6	7	6	4	5	9	4	7	8	7	9	

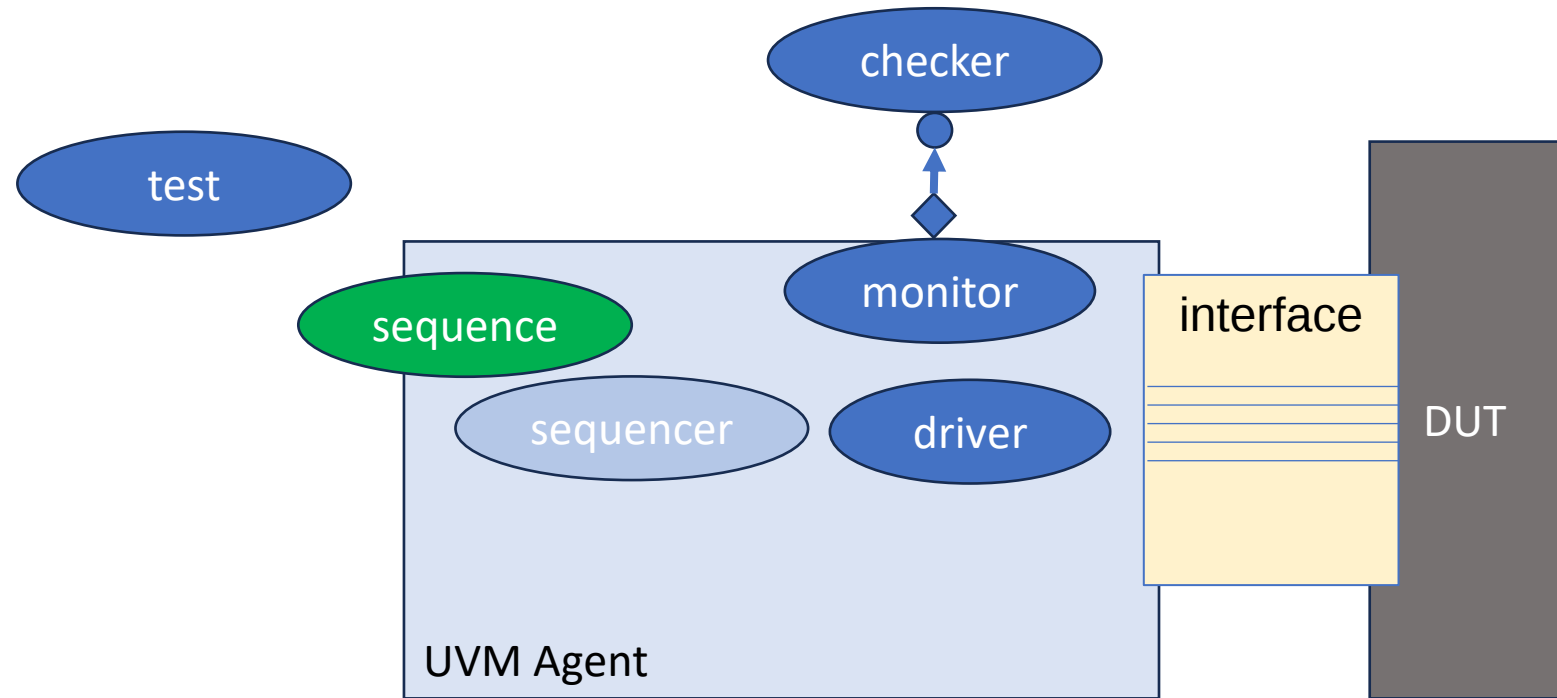
Conclusion

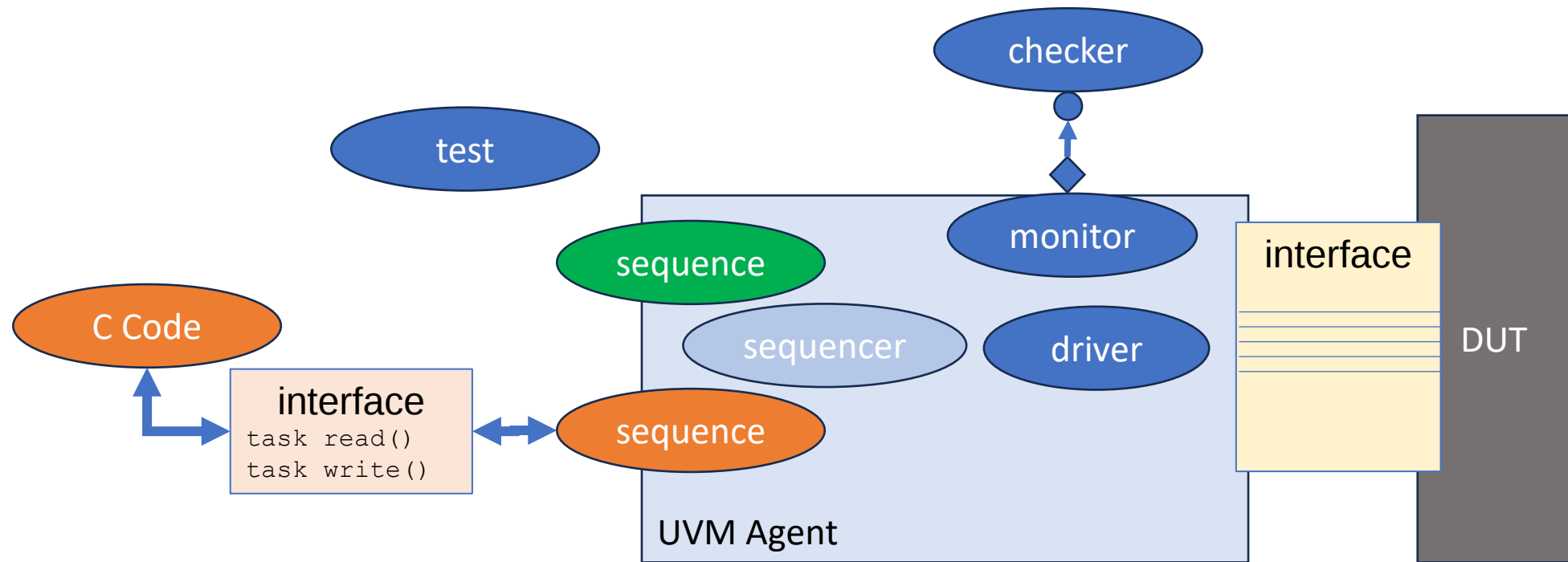
- A solution was shared to demonstrate easy additions to an existing UVM TB that will enable easy C code integration
- Standard SystemVerilog coding – LRM compliant
- Enables C coders access to write tests
- You can have BOTH a UVM testbench and a C test program

Questions?

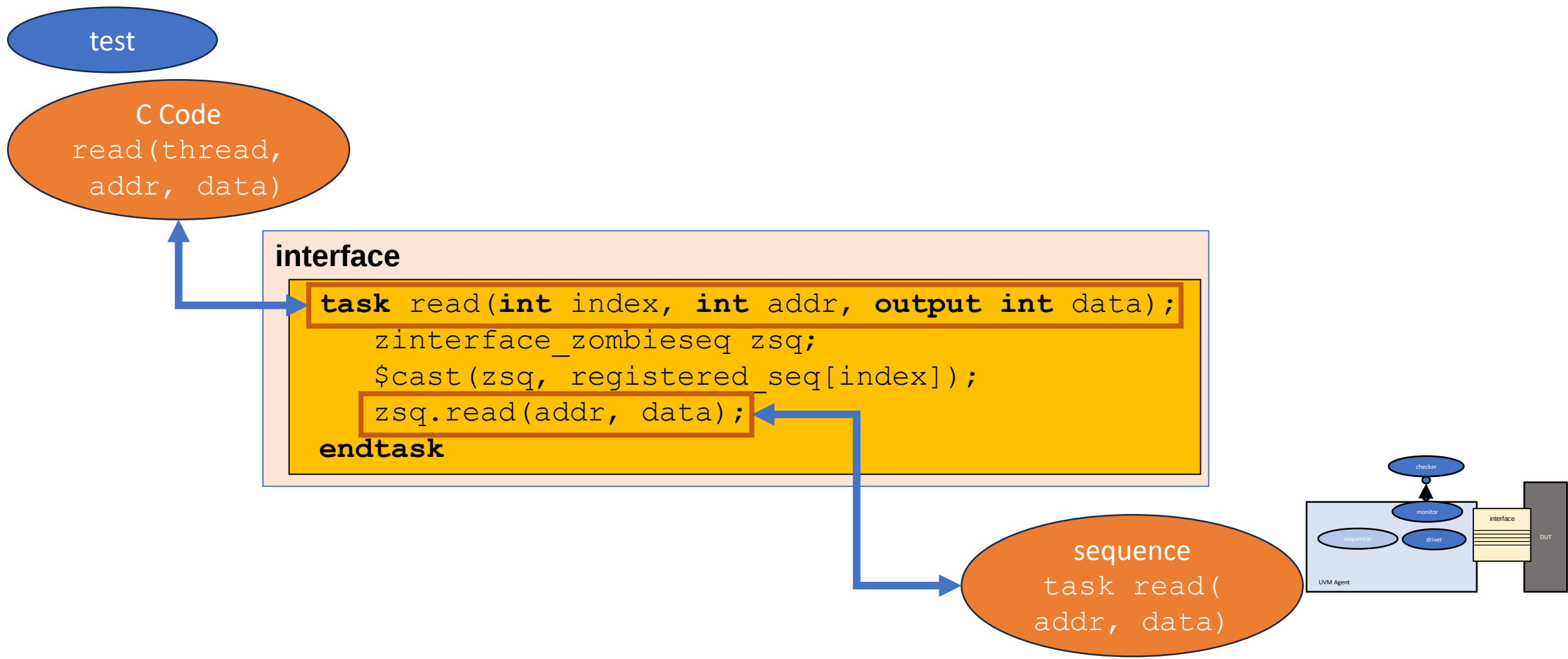
- Source code is available – please email rich.edelman@siemens.com

EXTRA SLIDES - duplicates





The Sequence Task calling the Interface Task Calling the C code



Use an existing UVM TB

A sequence uses `start_item()` and `finish_item()` to send transactions to the driver

```
task body();  
    start_item(t);  
    finish_item(t);  
endtask
```

UVM Sequence

```
task run_phase(uvm_phase phase);  
    forever begin  
        seq_item_port.get_next_item(t);  
        send_to_bus(t);  
        seq_item_port.item_done();  
    end  
endtask
```

UVM Driver

Add a C Test program

A sequence uses `start_item()` and `finish_item()` to send transactions to the driver

```
int
zinterface start test program1(int index, const char *name, int start_addr) {
    write(index, addr, data);
    read(index, addr, &data);
}
```

C Test Program

```
task body();
    start_item(t);
    finish_item(t);
endtask
```

UVM Sequence

Goal: Write a C program that can send transactions to the driver

```
task run_phase(uvm_phase phase);
    forever begin
        seq_item_port.get_next_item(t);
        send_to_bus(t);
        seq_item_port.item_done();
    end
endtask
```

UVM Driver

Connect sequence tasks

A sequence uses `start_item()` and `finish_item()` to send transactions to the driver

```
int
```

```
zinterface_start_test_program1(int index, const char *name, int start_addr) {  
    write(index, addr, data);  
    read(index, addr, &data);  
}
```

C Test Program

A C program that can send transactions to the driver

```
task body();  
    start_item(t);  
    finish_item(t);  
endtask
```

```
task read(...);  
    start_item(t);  
    finish_item(t);  
endtask
```

```
task write(...);  
    start_item(t);  
    finish_item(t);  
endtask
```

UVM Sequence

```
task run_phase(uvm_phase phase);  
    forever begin  
        seq_item_port.get_next_item(t);  
        send_to_bus(t);  
        seq_item_port.item_done();  
    end  
endtask
```

UVM Driver