



** Verification Academy News **



Upcoming Live Events

- [Verification Academy Live \(https://siemens-disw.formstack.com/forms/register_academy_lunch_fremont_june_2023\)](https://siemens-disw.formstack.com/forms/register_academy_lunch_fremont_june_2023) - Fremont, CA | June 13th
- [Verification Academy Live \(https://siemens-disw.formstack.com/forms/register_academy_live_san_diego_june_2023\)](https://siemens-disw.formstack.com/forms/register_academy_live_san_diego_june_2023) - San Diego, CA | June 14th
- [Verification Academy Live \(https://siemens-disw.formstack.com/forms/register_academy_live_westford_june_2023\)](https://siemens-disw.formstack.com/forms/register_academy_live_westford_june_2023) - Westford, MA | June 14th
- [Verification Academy Live \(https://siemens-disw.formstack.com/forms/register_academy_live_huntsville_june_2023\)](https://siemens-disw.formstack.com/forms/register_academy_live_huntsville_june_2023) - Huntsville, AL | June 22nd

On-Demand Recordings

- [Efficient Interconnect Formal Verification for Complex, Large-scale Designs: A Comprehensive Workflow \(https://verificationacademy.com/sessions/efficient-interconnect-formal-verification-for-complex-large-scale-designs-a-comprehensive-workflow\)](https://verificationacademy.com/sessions/efficient-interconnect-formal-verification-for-complex-large-scale-designs-a-comprehensive-workflow)
- [The New Leader in Verification IP, Delivering First Silicon Success for Your Next SoC or 3DIC \(https://verificationacademy.com/sessions/the-new-leader-in-verification-ip-delivering-first-silicon-success-for-your-next-soc-or-3dic\)](https://verificationacademy.com/sessions/the-new-leader-in-verification-ip-delivering-first-silicon-success-for-your-next-soc-or-3dic)
- [Questa Verification IQ: Boost verification predictability and efficiency with Big Data \(https://verificationacademy.com/sessions/questa-verification-iq-boost-verification-predictability-and-efficiency-with-big-data\)](https://verificationacademy.com/sessions/questa-verification-iq-boost-verification-predictability-and-efficiency-with-big-data)

UVM Framework

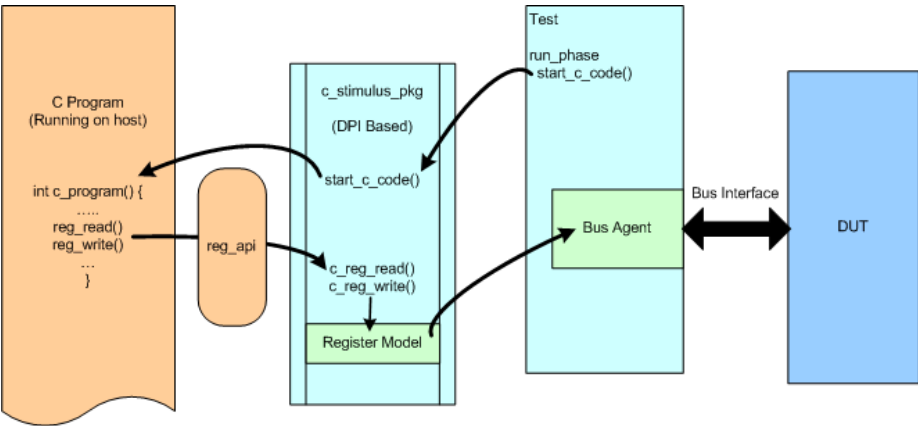
- [UVMF 2023.1 \(https://verificationacademy.com/news/uvm-framework-package-and-sessions-update\)](https://verificationacademy.com/news/uvm-framework-package-and-sessions-update) - New Release & Sessions

Verification Horizons

- [Verification Horizons \(https://verificationacademy.com/verification-horizons/march-2023-volume-19-issue-1/\)](https://verificationacademy.com/verification-horizons/march-2023-volume-19-issue-1/) - Read the March issue
- [Verification Horizons Blog \(https://blogs.sw.siemens.com/verificationhorizons/\)](https://blogs.sw.siemens.com/verificationhorizons/) - Read the latest posts

Home (/) > Cookbook (https://verificationacademy.com/cookbook) > UVM (https://verificationacademy.com/cookbook/UVM) > CBased Stimulus (https://verificationacademy.com/cookbook/CBasedStimulus)

Many hardware blocks are designed to interact with software using memory mapped registers. In the final implementation, the system level software, running on a CPU, reads and writes these registers via a bus interface on the hardware block. With UVM sequence based stimulus, accesses to these registers are made via a bus agent, sometimes in a directed way that emulates software accesses, sometimes using constrained random stimulus.



c_stimulus_pkg – Block Diagram

The UVM [register model \(/cookbook/Registers\)](/cookbook/Registers) is often used to raise the abstraction of the stimulus generated by these sequences.

However, there is often a requirement to develop c based test stimulus, the reasons for this include:

- A desire to develop device driver code early
- A requirement to have some directed tests that can be run at higher levels of integration, or potentially on a target device
- Additional software engineer resources are available to write tests

One way in which the c stimulus can be applied to the DUT is to insert a CPU or CPU model into a version of the testbench and then compile and execute the c as a program on the CPU. There is often a significant overhead involved with setting this additional testbench up, and then with simulating the CPU. This article describes a lighter weight alternative that can be used without having to change structure of an existing UVM testbench that contains one or more bus agents. The approach used is to add a C

register read/write API for use by C source code, which calls tasks in a SystemVerilog package via the SystemVerilog DPI mechanism to enable the C to make register accesses via the UVM testbench bus agents. The API enables c code to be compiled and then run on the host workstation during the simulation of a UVM environment. The package is called `c_stimulus_pkg` and comprises a light-weight C-API and two SystemVerilog packages.

Contents

- 1 Comparison with UVM-Connect
- 2 C Stimulus Package Overview
 - 2.1 Pre-Requisites:
 - 2.2 Theory Of Operation:
 - 2.2.1 UVM Side Of The `c_stimulus_pkg`
 - 2.2.2 The `c_stimulus_pkg` C API
 - 2.3 Starting The C Code
 - 2.4 Provision For Multiple Bus Targets
 - 2.5 Multiple C Threads
 - 2.6 More Than One C Based Test
 - 2.7 Handling Interrupts
 - 2.8 Package Download:
- 3 An Example Of Using The C Stimulus Package
 - 3.1 UVM Use Model
 - 3.2 Software Use Model
 - 3.3 Extending The Example To Run Another C Stimulus Test
- 4 Compilation and Simulation Process
- 5 Caveats
- 6 Example Download

Comparison with UVM-Connect

The C Stimulus package is not the same as the [UVM-Connect](http://verificationacademy.com/verification-methodology/uvm-connect) (<http://verificationacademy.com/verification-methodology/uvm-connect>) package.

The UVM Connect package encapsulates two main areas of functionality:

- TLM communication between UVM testbenches and SystemC via
 1. TLM 1 ports and exports
 2. TLM 1 analysis ports and exports
 3. TLM 2 sockets
- Providing a means for SystemC to call UVM functions

The primary purpose of the UVM-Connect package is to allow the user to mix SystemC and SystemVerilog components and stimulus. Although UVM-Connect is a very powerful solution, it does not provide a route to creating c or c++ programs that can access hardware registers.

The purpose of the C Stimulus package is to enable c-routines that communicate with hardware registers to access those registers in a DUT hooked up to a UVM bus agent within a UVM verification environment.

C Stimulus Package Overview

Pre-Requisites:

The C Stimulus package assumes the use of a register model in a UVM testbench. That register model should be integrated so that all the possible register accesses can be made via target bus agents.

If a register model is not available, then you will have to write one and integrate it. The process for doing this is described in the [register \(/cookbook/Doc/Glossary/Register_Model\)](#) article.

Theory Of Operation:

The C-Stimulus package uses the UVM register model to make accesses to DUT hardware registers via a thin DPI layer. On the software side, a c program makes a hardware register access using an address and a data argument, this access is converted to a UVM register read() or write() call by the `c_stimulus_pkg`.

The c stimulus is written as normal c, including the `reg_api.h` header file which is supplied as part of the `c_stimulus_pkg`. The UVM test is responsible for starting the c stimulus.

UVM Side Of The `c_stimulus_pkg`

In order to use the package, the UVM testbench needs to assign a valid handle to the register model before starting the c stimulus at the beginning of the `run_phase`. A function call is provided in the package to make this easier for the user:

```
//
// function: set_c_stimulus_register_block
//
// Sets the register model handle to the UVM environment register
// model so that c based register accesses can use the register model
//
//
function void set_c_stimulus_register_block(uvm_reg_block rm);
```

The package contains three tasks which are exported via the SystemVerilog DPI so that they are available to the c-side reg_api layer:

- c_reg_read()
- c_reg_write()
- wait_1ns() - Hardware delay - Wait for n * 1 ns

These tasks are intended to be only used by the reg_api layer.

```
//
// task: c_reg_read
//
// Reads data from register at address
//
task automatic c_reg_read(input int address, output int data);

//
// task: c_reg_write
//
// Writes data to register at address
//
task automatic c_reg_write(input int address, input int data);

//
// task: wait_1n
//
// Wait for n * 1ns
//
task wait_1ns(int n = 1);
```

When either of the read or write methods is called from C code, they go through the following process:

- Get the handle for the register to be accessed via a lookup in the register model using the get_register_from_address() method
- Call a reg.read() or reg.write() method using the register handle
- In the case of a read, return the read data

Pseudo code for the read case is shown in the code snippet below:

```

//
// function: get_register_from_address
//
// Uses the register model to make an lookup of the register
// associated with the address passed to the function.
//
// Returns a handle to the addressed register
//
function uvm_reg get_register_from_address(int address);
    uvm_reg_map reg_maps[$];
    uvm_reg found_reg;

    if(register_model == null) begin
        `uvm_error("c_reg_read", "Register model not mapped for the c_stimulus package")
    end

    register_model.get_maps(reg_maps);
    foreach(reg_maps[i]) begin
        found_reg = reg_maps[i].get_reg_by_offset(address);
        if(found_reg != null) begin
            break;
        end
    end

    return found_reg;

endfunction: get_register_from_address

task automatic c_reg_read(input int address, output int data);
    uvm_reg_data_t reg_data;
    uvm_status_e status;
    uvm_reg read_reg;

    read_reg = get_register_from_address(address);
    if(read_reg == null) begin
        `uvm_error("c_reg_read", $sformatf("Register not found at address: %0h", address))
        data = 0;
        return;
    end
    read_reg.read(status, reg_data);

    data = reg_data;

endtask: c_reg_read

```

The c_stimulus_pkg C API

The C API for the c_stimulus_pkg is defined in a header file called the reg_api.h. It is very light-weight and contains only 4 functions:

```

// reg_api.h
//
//
// function: reg_read
//
// Returns data from register address
//
int reg_read(int address);

//
// function: reg_write
//
// Writes data to register address
//
void reg_write(int address, int data);

//
// function: register_thread
//
// Called to register a non-default c thread with
// the c_stimulus_pkg context
//
void register_thread();

//
// function: hw_wait_1ns
//
// Hardware delay in terms of 1ns increments
//
void hw_wait_1ns(int n);

```

Starting The C Code

The read and write functions have to be called from the C program. In order to start the C program, a DPI context task needs to be called from SystemVerilog and a default for this is provided within the C Stimulus package - `start_c_code()`. The C code needs to implement a function call with the same name, either with the c program or a call to a function that contains the c program. Note that the normal name of a c program, "main", should not be used. The UVM testbench should call the `start_c_code()` function during an active UVM phase such as the `run_phase` in order to start C execution.

Provision For Multiple Bus Targets

Most block level testbenches only deal with a single bus interface, but with more complex DUTs, there may be several bus interfaces which are used to access the registers. Provided the different bus masters are in the register model's register map and the UVM testbench supports register based accesses to all the DUT interfaces, then the person writing the c code only has to worry about reading and writing from the registers at the correct addresses.

Multiple C Threads

In some circumstances, there may be a requirement to run multiple C threads and this can easily be accommodated by the user making calls to the additional C program threads via his own DPI imports in his own package. The test package is usually the most convenient place to do this.

Note that:

- The DPI imports must be "context" imports
- If a DPI import is defined, then there must be a matching c function declared, otherwise there will be at least an elaboration warning

```
// Example of importing a c test routine via the DPI
import "DPI-C" context task a_c_test_routine;
```

On the c side a `register_thread()` method should be called at the beginning of a c-thread. This is used to register the DPI context of the c-thread as the `c_stimulus_pkg`.

```
// In the UVM test:
//
task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    fork
        start_c_code(); // Default c thread
        a_c_function(); // Additional c thread
        // HW Stimulus:
        v_seq.start(null);
    join

    phase.drop_objection(this);
endtask: run_phase

//
// On the c-side:
//

int a_c_test_routine() {
    // Declare variables

    register_thread(); // Must be called

    // Rest of thread code

    return 0;
}
```

More Than One C Based Test

The default package assumes that only one c based test will be written and that the c-side will be started using `start_c_code()`. However, if more than one c based test is required, then one of three strategies can be used:

- Recompile the c code before running each testcase in order to make sure that `start_c_code()` calls the correct c-side function.
- Import a new c function for the c-side of each new testcase in the test package - using the same approach as outlined in the section describing multiple threads.
- Compile each c code into a separate shared object using the advanced DPI compilation flow and only load that shared object.

The recommended approach is to add a new c function to wrap the thread for a new c based test and to add a DPI import for the c side function into the test package.

Handling Interrupts

In the "real world" a hardware interrupt causes a CPU execution thread to suspend, and then jump to some interrupt handler code to service the interrupt and then return to the execution thread once the handler routine has completed.


In order to simulate the effect of a hardware interrupt, an additional package should be used - `isr_pkg`. This provides a task called `interrupt_service_routine`, this raises a flag that blocks other c threads from accessing the hardware registers until the `interrupt_service_routine` completes.

On the c side, a function called start_isr() is used to wrap the interrupt service routine. This function should NOT call the register_thread() method.

This is an approximation to what would happen with a real CPU, since the main c thread will continue to execute until it blocks on a register access.

Package Download:

The C Stimulus package can be downloaded here:

Download a complete working example related to this page content
 (tarball: [C_stimulus_pkg.tgz](https://verificationacademy.com/cookbook/download?file=https://s3.amazonaws.com/courses.verification.academy/C_stimulus_pkg.tgz&download_refer=/cookbook/cbasedstimulus) (https://verificationacademy.com/cookbook/download?file=https://s3.amazonaws.com/courses.verification.academy/C_stimulus_pkg.tgz&download_refer=/cookbook/cbasedstimulus))

An Example Of Using The C Stimulus Package

The following example is an adaptation of the SPI testbench used as one of the cookbook register examples. The example shows how to use a c routine to test the DUT, and how to use a c interrupt service routine to handle interrupts. It uses the same testbenchfiles as the original example, but the test package adds imports of the C Stimulus and the ISR packages and a new test class which starts the c routine during its run phase.

UVM Use Model

In order to use the c interface, a UVM test has to assign the register model handle in the c_stimulus_pkg package to the testbench register model and then call the c thread(s) to execute, this is achieved using the set_c_stimulus_register_block() method.

If there is only one c thread to execute, then the default call to the c thread is start_c_code() - this should be matched by a function in the c application of the same name. If multiple c threads are used, then a call to each of these should be declared in a package (possibly the test package) as DPI imports.

As an example, the run_phase for a simple example test would be:

```
// From inside a test `included into a package containing the following imports:
import c_stimulus_pkg::*;
import isr_pkg::*;

// This task starts the c program that then calls back into
// the UVM simulation
//
// It also monitors the interrupt line from the SPI block
// and calls the interrupt service routine when it is asserted
//
task spi_c_int_test::run_phase(uvm_phase phase);
    spi_tfer_seq spi_seq = spi_tfer_seq::type_id::create("spi_seq");

    phase.raise_objection(this, "Test Started");
    `uvm_info("run_phase", "starting c code", UVM_LOW)

    set_c_stimulus_register_block(spi_rm); // Assign the register model handle

    fork
        start_c_code(); // Start the c-side test routine
        // Respond to SPI transfers:
        begin
            forever begin
                spi_seq.BITS = 0;
                spi_seq.rx_edge = 0;
                spi_seq.start(m_env.m_spi_agent.m_sequencer);
                spi_rm.ctrl_reg.char_len.get(spi_seq.BITS);
                spi_rm.ctrl_reg.rx_neg.get(spi_seq.rx_edge);
                spi_seq.start(m_env.m_spi_agent.m_sequencer);
            end
        end
        begin
            forever begin
                m_env_cfg.wait_for_interrupt();
                interrupt_service_routine(); // Start the c-side interrupt service routine
            end
        end
    join_any
    `uvm_info("run_phase", "c code finished", UVM_LOW)
    phase.drop_objection(this, "Test Finished");

endtask: run_phase
```

This test also illustrates the use of an interrupt. In the run_phase() method, an interrupt line is monitored via a configuration monitoring method (see the article on [signal_wait \(/cookbook/Stimulus/Signal_Wait\)](#) for more details). When an interrupt is detected, the interrupt service routine is called.

Software Use Model

The c code used in this example #includes the reg_api.h file, so that it can call the UVM register access methods. The c test code and the interrupt service routine are wrapped by the start_c_code() and start_isr() functions which are the default DPI imports supported by the c_stimulus_pkg and the isr_pkg.

```

#include "spi_regs.h" // Defines for register offsets etc
#include "reg_api.h" // UVM C stimulus register layer API

int int_flag = 0;

void spi_int_test() {
    int no_chars = 1;
    int format = 0;
    int divisor = 2;
    int slave_select = 1;
    int control = 0;
    int i = 0;
    int data_0 = 0x12345678;
    int data_1 = 0x87654321;
    int data_2 = 0x90901212;
    int data_3 = 0x5a6b7c8d;
    int status;
    int data;

    reg_write(DIVIDER, divisor);

    while(i < 10) {
        int_flag = 0;
        control = no_chars + (format << 9) + 0x3000;
        reg_write(CTRL, control);
        reg_write(SS, slave_select);
        reg_write(TX0, data_0);
        reg_write(TX1, data_1);
        control = control + 0x100;
        reg_write(CTRL, control);
        while(int_flag == 0) {
            status = reg_read(CTRL);
            data = reg_read(SS);
        }
        no_chars = no_chars++;
        format = format++;
        if(format == 8) {
            format = 0;
        }
        slave_select = slave_select << 1;
        if(slave_select == 0x100) {
            slave_select = 1;
        }
        i++;
    }
}

void spi_isr() {
    int status;
    int rx_data0;
    int rx_data1;
    int rx_data2;
    int rx_data3;

    status = reg_read(CTRL);
    reg_write(SS, 0x0);
    rx_data0 = reg_read(RX0);
    rx_data1 = reg_read(RX1);
    rx_data2 = reg_read(RX2);
    rx_data3 = reg_read(RX3);
    int_flag = 1;
}

int start_c_code () {
    spi_int_test();
    return 0;
}

int start_isr () {
    spi_isr();
    return 0;
}

```

Extending The Example To Run Another C Stimulus Test

The same testbench can be extended to run with another c test routine by adding a DPI import for the c test routine to the test package, and by adding another test class to the package:

```
// In the spi_test_lib_pkg:

// DPI Imports for c based routines:
import "DPI-C" context task spi_c_poll_test_routine();

// C based tests:
`include "spi_c_int_test.svh"
`include "spi_c_poll_test.svh"

// The run method of spi_c_poll_test
//
// This task starts the c program that then calls back into
// the UVM simulation
//
task spi_c_poll_test::run_phase(uvm_phase phase);
    spi_tfer_seq spi_seq = spi_tfer_seq::type_id::create("spi_seq");
    uvm_reg_data_t reg_data;

    phase.raise_objection(this, "Test Started");
    `uvm_info("run_phase", "starting c code", UVM_LOW)

    set_c_stimulus_register_block(spi_rm);

    fork
        spi_c_poll_test_routine(); // Calling the imported C routine
        // Respond to SPI transfers:
        begin
            forever begin
                spi_seq.BITS = 0;
                spi_seq.rx_edge = 0;
                spi_seq.start(m_env.m_spi_agent.m_sequencer);
                spi_seq.BITS = spi_rm.ctrl_reg.char_len.get();
                spi_seq.rx_edge = spi_rm.ctrl_reg.rx_neg.get();
                spi_seq.start(m_env.m_spi_agent.m_sequencer);
            end
        end
    join_any
        `uvm_info("run_phase", "c code finished", UVM_LOW)
        phase.drop_objection(this, "Test Finished");

endtask: run_phase
```

The corresponding c test routine needs to make the register_thread() API call at the beginning:

```
#include "spi_regs.h" // Defines for register offsets etc
#include "reg_api.h" // DPI Register Hardware access layer API

int spi_c_poll_test_routine() {
    int no_chars = 1;
    int format = 0;
    int divisor = 2;
    int slave_select = 1;
    int control = 0;
    int i = 0;
    int data_0 = 0x12345678;
    int data_1 = 0x87654321;
    int data_2 = 0x90901212;
    int data_3 = 0x5a6b7c8d;
    int status;
    int data;

    register_thread(); // To register this thread with the c_stimulus_pkg DPI context

    reg_write(DIVIDER, divisor);

    //
    // etc
    //

    return 0;
}
```

Compilation and Simulation Process

The compilation process for the Package and the c code, using Questa vlog compiler, is as follows:

- Compile the c_stimulus_pkg.sv file, and if required, the isr_pkg.sv, generating a DPI header file

- Compile the test package for any UVM tests that are relying on c-side threads other than the default, generating a DPI header
- Compile the reg_api.c file
- Compile the application c code

When the simulation is invoked, then Questa will automatically create the necessary shared object used in simulation.

To **generate** the **DPI** header file **for** the **package**:

```
vlog $(C_STIMULUS_PKG_HOME)/c_stimulus_pkg.sv -dpiheader sv_dpi.h
vlog $(C_STIMULUS_PKG_HOME)/isr_pkg.sv -dpiheader sv_dpi.h
```

If there is a non-**default** c test routine in a test **package**:

```
vlog +incdir+$(TEST_PKG_HOME) $(TEST_PKG_HOME)/test_pkg.sv -dpiheader sv.dpi.h
```

To compile the reg_api:

```
vlog +incdir+$(C_STIMULUS_PKG_HOME) $(C_STIMULUS_PKG_HOME)/reg_api.c
```

To compile the c thread code:

```
vlog +incdir+$(C_CODE_HOME) $(C_CODE_HOME)/my_c_code.c -ccflags -I$(C_STIMULUS_PKG_HOME)
```

To run the simulation loading the **package**:

```
vsim top_tb +UVM_TESTNAME=spi_c_int_test
```

Caveats

The c side of the API is implemented as function calls where the address of the register is passed as an argument. The easiest way to abstract these addresses is to use #defines in a header file.


There are alternative methods of implementing a register interface API which involve using an array of structs to map the registers into memory space, register accesses then take place by using pointers to these structs. Unfortunately, this approach cannot be used with the API provided. In order to be able to support this style of interface it would be necessary to either add a compiler option or implement a memory management unit that would throw an exception which would allow the access to be taken care of using the simple API provided. Both of these options are outside the scope of the solution provided.

Example Download

To download the example that illustrates:

1. The use of c stimulus
2. Interrupt handling using c stimulus
3. The use of an additional c based thread for another test case







Download a complete working example related to this page content

 (tarball: [uvm_c_stimulus_bl_example.tgz](https://verificationacademy.com/cookbook/download?file=https://s3.amazonaws.com/courses.verification.academy/Uvm_c_stimulus_bl_example.tgz&download_refer=/cookbook/cbasedstimulus) (https://verificationacademy.com/cookbook/download?file=https://s3.amazonaws.com/courses.verification.academy/Uvm_c_stimulus_bl_example.tgz&download_refer=/cookbook/cbasedstimulus))

Previous (<https://verificationacademy.com/cookbook/messaging/sequencemessaging>)






Next (<https://verificationacademy.com/cookbook/registers>)

CBasedStimulus Discussion

SOLVED	TITLE	REPLIES	VIEWS	POSTED	UPDATED	ACTIONS
	The Cbased stimulus example .tgz is broken. (/forums/uvm/cbased-stimulus-example-tgz-broken)	1	1,194	9 years 2 months ago (https://verificationacademy.com/forums/uvm/cbased-stimulus-example-tgz-broken) by dylan	9 years 2 months ago (https://verificationacademy.com/forums/uvm/cbased-stimulus-example-tgz-broken#answer-40368) by gordon	  
	Systemverilog DPI-C (/forums/uvm/systemverilog-dpi-c)	4	583	7 months 6 days ago (https://verificationacademy.com/forums/uvm/systemverilog-dpi-c) by Guy S	7 months 3 days ago (https://verificationacademy.com/forums/uvm/systemverilog-dpi-c#answer-109546) by Guy S	  

[Ask a Question \(/ask-a-question?forum=23&tag=Cookbook%3A%20CBasedStimulus\)](#)

The Verification Methodology Cookbook content is provided by Mentor Graphics' Verification Methodology Team. Please [contact us \(mailto:vmdoc@mentor.com?subject=verification-methodology-cookbook-feedback-CBasedStimulus\)](#) with any enquiries, feedback, or bug reports.

Siemens Digital Industries Software	Cloud	Community	About Us	VA - Contact Us
#TodayMeetsTomorrow (https://twitter.com/search?q=%23todaymeetstomorrow)	(https://www.sw.siemens.com/en-US/digital-transformation/cloud/)	(https://community.sw.siemens.com/blog/)(https://blogs.sw.siemens.com/)	(https://www.sw.siemens.com/en-US/)	(https://verificationacademy.com/PLM - Contact Us
    	(https://www.sw.siemens.com/en-US/plm/)	Online Store	(https://www.sw.siemens.com/enus.html?ref=footer)	(https://www.plm.automation.siemens.com/global/en/your-success/events/)
	(https://www.mendix.com/)	(https://www.dex.siemens.com/)	US/careers/)	EDA - Contact Us
	Electronic Design Automation		Events	(https://resources.sw.siemens.com/)
	(https://eda.sw.siemens.com/en-US/)			(https://www.plm.automation.siemens.com/global/en/your-success/events/)
	MindSphere		News and Press	Worldwide Offices
	(https://www.plm.automation.siemens.com/global/en/products/mindsphere/)	(https://www.plm.automation.siemens.com/global/en/our-story/offices/global/)		
	Design, Manufacturing and		story/newsroom/)	Support Center
	PLM Software		Customer Stories	(https://support.sw.siemens.com/)
	(https://www.plm.automation.siemens.com/global/en/)		(https://www.plm.automation.siemens.com/global/en/our-story/customers/)	Give us Feedback
	View all Portfolio		Partners	(https://webtac.industrysoftware.com/)
	(https://www.sw.siemens.com/en-US/portfolio/)			(https://www.plm.automation.siemens.com/global/en/our-story/partners/)
			Trust Center	
			(https://www.sw.siemens.com/en-US/trust-center)	