DPI Redux. Functionality. Speed. Optimization.

Rich Edelman
rich_edelman@mentor.com

Rohit Jain
rohit_jain@mentor.com

Hui Yin
hui_yin@mentor.com

Mentor Graphics, Fremont, CA

***Abstract*-SystemVerilog DPI-C is commonly used for many different things - all interfacing SystemVerilog with C. Although well used in places, it is not uniformly used, nor is it used always in optimal ways. This paper will revisit the DPI-C use model, suggest common usage and recommend preferred optimizations.**

INTRODUCTION

SystemVerilog DPI-C [1] is a powerful, simple and fast interface between C code and the SystemVerilog language. It is a powerful way for SystemVerilog to access any C library that is available. It is simple because C calls share the same function and task call semantics as a regular SystemVerilog function or task call. C calls and SystemVerilog calls can be used interchangeably. It is fast because it uses the regular C calling conventions (push the function arguments on the stack and jump). Some implementations may choose to copy the function arguments first, but the fastest implementations provide call-by-reference arguments without copying.

This paper will explore SystemVerilog DPI-C for the new user as well as the experienced user. After reading this paper, the reader should feel comfortable using SystemVerilog DPI-C to help solve their verification needs. A previous work [2] has many useful examples of using SystemVerilog DPI-C to achieve verification results. These examples are still valid, useful and should be used as a resource.

A. Imports and Exports

In SystemVerilog, a DPI-C call is either an import or an export. These definitions are from the point-of-view of SystemVerilog. An "import" function or task is C code that is IMPORTED, so that SystemVerilog may call it. An "export" function or task is SystemVerilog code that is EXPORTED, so that C code may call it.

B. Tasks and Functions

Tasks and functions are similar to each other. They may be used interchangeably in the remainder of the paper (due to laziness or error), but they are different. Both a task and function can take arguments - inputs, outputs and inouts. A function can return a value with the return statement, or by updating an output or inout argument. A task can only return a value by updating an output or inout argument. A function cannot consume simulation time. A task can consume simulation time.

C. Task and Function Arguments

Generally speaking, any SystemVerilog value can be passed across the DPI-C interface. Normally the best arguments to pass across the interface are values that have a direct representation in C. For example integers, shorts, bytes and real numbers. Passing a 4-state value to C is permitted, but our recommendation is that data on the C side is most optimal as 2-state values.

THE FIRST DPI-C IMPORT AND EXPORT

A. Top module

Our first example defines two C routines, one function and one task which perform the same basic operation – they add 'a' plus 'b' and return the result in 'c'. Additionally, they both print their SystemVerilog scope names and they both call a SystemVerilog function to print the scope and the time. The C task also calls a SystemVerilog task which consumes time (it waits for N ticks).

```
module top();

  function int sv_print_scope(input int value);
    $display("SV: @%2t: sv_print_scope(): %m [value=%0d]", $time, value);
    return value;
  endfunction

  task sv_consume_time(input int d);
    #d;
  endtask

  export "DPI-C" function sv_print_scope;
  export "DPI-C" task     sv_consume_time;

  import "DPI-C" context function int c_add_function(input int a, input int b, output int c);
  import "DPI-C" context task         c_add_task    (input int a, input int b, output int c);

  initial begin
    int c;
    int ret;

    #10;

    ret = c_add_function(10, 20, c);

    c_add_task(10, 20, c);
  end
endmodule
```

SV uses "%m" to print the scope

SV export to consume time. Programmable amount of time.

Function – cannot consume time

Task – can consume time

B. C Code

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

int
c_add_function(int a, int b, int *c) {
  const char *scope_name;

  *c = a + b;

  scope_name = svGetNameFromScope(svGetScope());
  vpi_printf(" C: %s.c_add_function(%d, %d, %d)\n", scope_name, a, b, *c);
  sv_print_scope(*c);
  return 0;
}

int
c_add_task(int a, int b, int *c) {
  const char *scope_name;

  *c = a + b;

  scope_name = svGetNameFromScope(svGetScope());
  vpi_printf(" C: %s.c_add_task    (%d, %d, %d)\n", scope_name, a, b, *c);
  sv_consume_time(10);
  sv_consume_time(10);
  sv_print_scope(*c);
```

'a' and 'b' are inputs. 'c' is an output

C uses svGetNameFromScope() to print the scope

Consume some time (10 ticks per call)

```
  sv_consume_time(10);
  return 0;
}
```

C. *Makefile*

Compilation and simulation is quite simple. When the SystemVerilog file containing the imports and exports is compiled, the dpiheader.h file is created.

```
rm -rf work
vlib work
vlog -dpiheader dpiheader.h t.sv t.c
vopt -o opt top
vsim -c opt -do "run -all; quit -f"
```

D. *dpiheader.h*

Some implementations have the ability to generate a header file from the SystemVerilog import and export statements. This header file embodies what the simulator thinks the interface is. By using this file in the C code, the C compiler will check interface mistakes.

```
int c_add_function(int a, int b, int* c);
int c_add_task(int a, int b, int* c);
int sv_consume_time(int d);
int sv_print_scope(int value);
```

AES ENCRYPTION

A. *AES SystemVerilog*

For a more interesting example, an AES [3][4][5] encryption C implementation can be found. This code can be compiled as-in, creating a shared library (aes.so). This code was compiled and used with SystemVerilog DPI-C calls with no changes from the original obtained from the internet.

```
gcc -Wall -Os -Wl,-Map,test.map -c aes.c -o aes.o –fPIC
gcc -o aes.so aes.o -shared
```

Once the shared library is built, the SystemVerilog can import those routines. Referencing the shared library is easy, using the –sv_lib argument.

```
rm -rf work
vlib work
vlog -dpiheader dpiheader.h t.sv
vopt -o opt top
vsim -c opt -do "run -all; quit -f" -sv_lib ../tiny-AES128-C-master/aes
```

B. *The SystemVerilog DPI-C test top*

The actual import statement is also easy – just one line. The test top creates the input array, the expected output array, the key and the iv.

```
module top();

  typedef byte unsigned uint8_t;

  uint8_t key[] = '{ 8'h2b, 8'h7e, ... 8'h4f, 8'h3c };
  uint8_t iv[]  = '{ 8'h00, 8'h01, ... 8'h0e, 8'h0f };

  uint8_t in[]  = '{ 8'h6b, 8'hc1, ... 8'h17, 8'h2a,
                     8'hae, 8'h2d, ... 8'h8e, 8'h51,
                     8'h30, 8'hc8, ... 8'h52, 8'hef,
```

Declaring and initializing the arrays.
The key and iv are 16 bytes each. (128 bits)
The ‘in’ and ‘out’ are 64 bytes each.

```
                    8'hf6, 8'h9f, ... 8'h37, 8'h10 };

  uint8_t out[] = '{ 8'h76, 8'h49, ... 8'h19, 8'h7d,
                     8'h50, 8'h86, ... 8'h78, 8'hb2,
                     8'h73, 8'hbe, ... 8'h95, 8'h16,
                     8'h3f, 8'hf1, ... 8'he1, 8'ha7 };
  uint8_t buffer[64];

  import "DPI-C" function void AES128_CBC_encrypt_buffer(
    output uint8_t buffer[64],
     input uint8_t in[64],
     input int size,
     input uint8_t key[16],
     input uint8_t iv[16]);

  initial begin
    AES128_CBC_encrypt_buffer(buffer, in, 64, key, iv);

    if (buffer != out)
      $display("FATAL: Mismatch");
    else
      $display(" INFO: Pass");
  end
endmodule
```

Easy one line import

Calling C

Comparing C output (buffer) without expected output (out)

C. AES C code

AES C code snippet, as downloaded, no changes. The entire C implementation is 880 lines and 200 lines of test program.

```
static void XorWithIv(uint8_t* buf)
{
  uint8_t i;
  for(i = 0; i < KEYLEN; ++i)
  {
    buf[i] ^= Iv[i];
  }
}

void AES128_CBC_encrypt_buffer(uint8_t* output,
                               uint8_t* input,
                               uint32_t length,
                               const uint8_t* key,
                               const uint8_t* iv)
{
  ...
  for(i = 0; i < length; i += KEYLEN)
  {
    XorWithIv(input);
    BlockCopy(output, input);
    state = (state_t*)output;
    Cipher();
    Iv = output;
    input += KEYLEN;
    output += KEYLEN;
  }
  ...
}
```

These arrays are simple.
A list of bytes or 32 bit words.

In the future, with the C implementation working, a SystemVerilog implementation can be developed, using the C code as a golden model.

One thing that makes the import easy is that the function arguments are simple integers or arrays of bytes. The arrays are fixed size. If the SystemVerilog interface had been an open array – for example, an array with variable size, then the native C implementation would not have worked. In that case a SystemVerilog DPI-C Open Array

Handle would be used to access the array data. Variable sized arrays (open arrays) do not prevent DPI-C from being used, but generally speaking they make things quite messy. It is our advice to try to use fixed size arrays whenever possible.

THREADED C CODE

For this example, a fractal [6][7] calculator implemented in Verilog and C is used. The SystemVerilog code starts the tests using the fork/join construct. This will cause threads to run. Without any further attention, each thread will run to completion, and then the next thread will run to completion, until all threads have completed.

In the diagram below when the BLUE thread is forked it will call C, then SV, etc. until it finishes. Then the GOLD thread will start, will call C and SV, etc, until it finishes. In order to get them to appear to be in parallel – some command to "yield" needs to be used in the SystemVerilog – for example a #DELAY or a wait() or a @(SIGNAL). (See 'hw_sync' below).

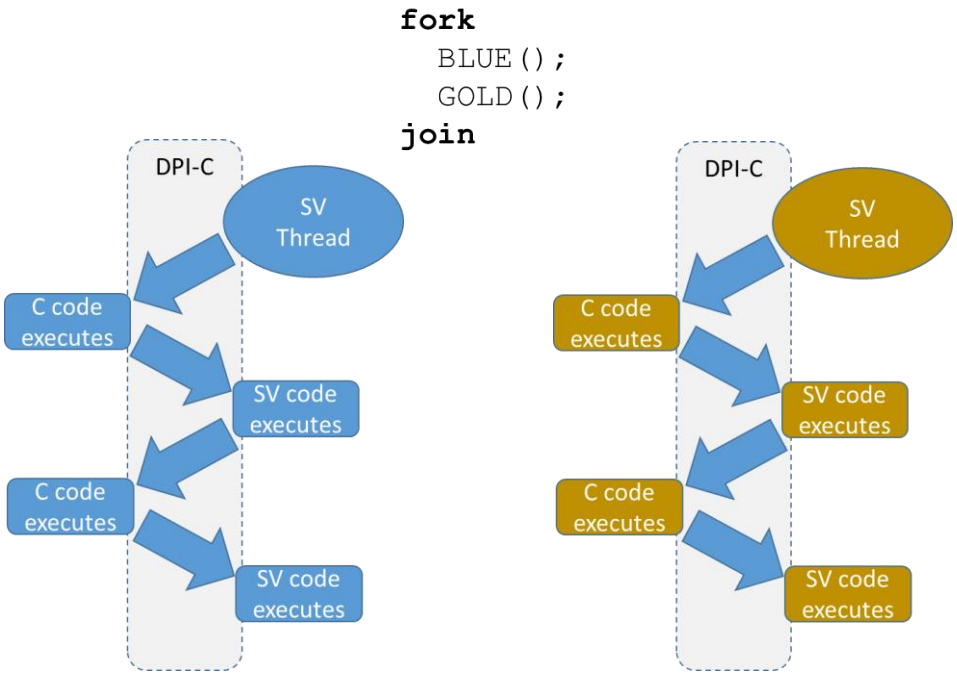

Figure 1: Two threads in SystemVerilog

A. Fractal SystemVerilog top level test

```
module m();
  task vl_mandel(input int xpos, ypos, width, height, real xstart, xend, ystart, yend);
    ...
  endtask
endmodule

module top();
  import "DPI-C" context task c_mandel(input int xpos, ypos, width, height,
                                       real xstart, xend, ystart, yend);
  initial begin
    fork
      begin m1.vl_mandel(0*(800+10), 0,          800, 800, xstart, xend, ystart, yend); end
      begin     c_mandel(1*(800+10), 0,          800, 800, xstart, xend, ystart, yend); end
```

```
      begin     c_mandel(2*(800+10), 0*(300+50), 300, 300, xstart, xend, ystart, yend); end
      begin m2.vl_mandel(2*(800+10), 1*(300+50), 300, 300, xstart, xend, ystart, yend); end
      begin m3.vl_mandel(2*(800+10), 2*(300+50), 300, 300, xstart, xend, ystart, yend); end
      begin     c_mandel(2*(800+10), 3*(300+50), 100, 100, xstart, xend, ystart, yend); end
    join
    $finish;
  end
endmodule
```

In the code above, there are six calls to either vl_mandel() or c_mandel(). These are SystemVerilog or C tasks which calculate the Mandelbrot set. Each task will draw in its own window. Each window will be drawn completely, one after the other. If it is desirable to have the windows draw at the same time (in unison – each window drawing one row, then each window drawing the next row, etc), then a simple synchronization can be used. At a regular interval, each thread can yield (executing a wait), by using a pound delay (#n), or an event wait (@). When the thread executes the wait it allows another thread to continue until IT executes the wait. In this way the threads each take turns. (A form of cooperative multithreading). For this example, each thread is synchronizing using the routine 'hw_sync()' after calculating and drawing an entire row of pixels.

The exported SystemVerilog DPI-C routine named 'hw_sync' is used to synchronize all the threads, and allow "cooperative multi-processing".

```
  export "DPI-C" task hw_sync;

  always begin sync_clk = 0;  #1; sync_clk = 1;  #1; end

  task automatic hw_sync(input int count);
    repeat(count)
      @(posedge sync_clk);
  endtask
```

In order to implement the drawing, the C code simply calls the C drawing library. For the Verilog code, imports are defined, and the imports are called. The C drawing library called by the C code, is exactly the same as the imported C calls that the Verilog is calling.

B. Imports for Drawing

A thin API layer was developed around the EGGX [7] library for drawing in an X11 environment. That library is imported for SystemVerilog, and is also available as link targets from C.

```
  import "DPI-C" function void draw_finish(input int win);
  import "DPI-C" function void draw_flush (input int win);
  import "DPI-C" function void draw_clear (input int win);
  import "DPI-C" function void draw_title (input int win, string title);

  import "DPI-C" function int  draw_init  (input int xpos, xypos, width, height);
  import "DPI-C" function void draw_pixel (input int win, x, y, n, minlimit, maxlimit);
```

C. Fractal Verilog code

The Verilog code to draw the fractal is quite similar to the C code. The Verilog code calls the imported routines for drawing: draw_init(), draw_title(), draw_flush(), draw_pixel(), draw_finish() and draw_clear(). The algorithm is relatively simple. For each pixel, decide what color the pixel should be according to whether it is in the set or not and how close it is to meeting the "escape" condition. For our purposes, Verilog and C produce a pretty picture, and demonstrate how to draw (x, y) pixels from Verilog and C.

```
task vl_mandel(input int xpos, input int ypos, input int width, input int height,
  input real xstart, input real xend, input real ystart, input real yend);

  string label;

  win = draw_init(xpos, ypos, width, height);
  label = $sformatf("VL(win=%0d) IMG(%g, %g) (%g, %g)",
          win, xstart, ystart, xend, yend);
  draw_title(win, label);

  xstep = (xend - xstart)/width;
  ystep = (yend - ystart)/height;

  yr = ystart;
  for (y = 0; y < height; y++) begin
    hw_sync(1);
    draw_flush(win);
    xr = xstart;
    for (x = 0; x < width; x++) begin
      n = get_mandel(xr, yr);
      draw_pixel(win, x, y, n, 1, 1000);
      xr += xstep;
    end
    yr += ystep;
  end
  draw_finish(win);
  draw_clear(win);
endtask
```

draw_init() initializes a new drawing window

hw_sync(1) causes this thread to yield

Draw a pixel at (x, y) of color ‘n’

D. Fractal C Code

Notice the C code is almost identical to the Verilog code. The C code has no need for the SystemVerilog imported draw library – it is linked to the actual C library directly. The C code does call the hw_sync export routine in order to participate in the cooperative multi-processing. Each call to hw_sync will cause a wait for the @(posedge clk), allowing all such cooperating tasks to get executed in unison.

```
int
c_mandel(
  int xpos, int ypos,
  int width, int height,
  double xstart, double xend, double ystart, double yend)
{
  double xr, yr, xstep, ystep;
  int n, x, y;
  int win;
  char label[1024];

  win = draw_init(xpos, ypos, width, height);
  sprintf(label, " C(win=%d) IMG(%g, %g) (%g, %g)",
          win, xstart, ystart, xend, yend);
  draw_title(win, label);

  xstep = (xend - xstart)/width;
  ystep = (yend - ystart)/height;

  yr = ystart;
  for(y = 0; y < height; y++) {
    hw_sync(1);
    draw_flush(win);
    xr = xstart;
    for(x = 0; x < width; x++) {
      n = get_mandel(win, xr, yr);
      draw_pixel(win, x, y, n, 1, LIMIT);
      xr += xstep;
    }
    yr += ystep;
  }
```

hw_sync(1) causes this thread to yield

```
  draw_finish(win);
  draw_clear(win);
  return 0;
}
```

E. Verilog and C generated images

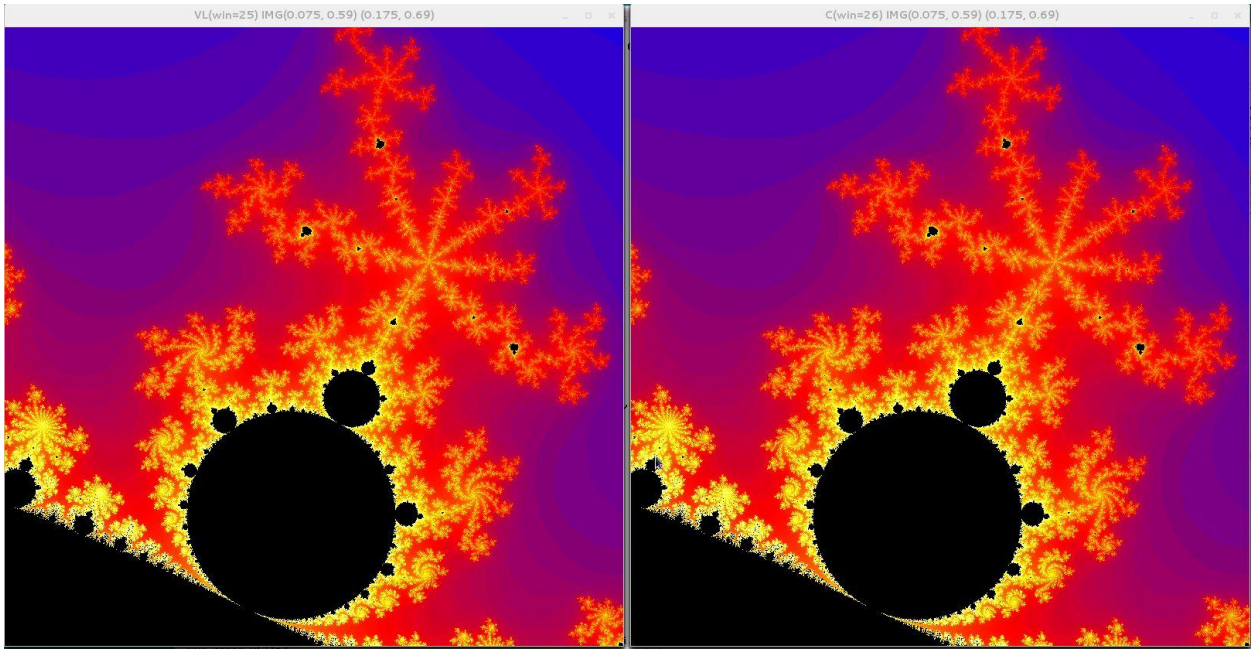

Figure 2: Mandelbrot - Verilog and C code

The left image above was generated from the Verilog implementation "VL(win=25) IMG(0.075, 0.59) (0.175, 0.69):, consisting of 640,000 total dots after doing 130,241,256 inner loop evaluations. The right image was generated from the C implementation (and has the same statistics). They draw in a second or two on a remote virtual Linux machine. (CentOS 7).

EXAMPLE CODE – PRIME NUMBER SEARCH

Prime number [8] search is used to demonstrate a simple Pthread [9] implementation. The "highest" number and the "size of each slice" are the two inputs. There will be "highest/sliceSize" threads created. For demonstration we searched from 0 to 1,000,000,000 with slice sizes of 1,000,000,000 (1 thread), 500,000,000 (2 threads), 250,000,000 (4 threads) and 100,000,000 (10 threads). The experiments finished in 60 seconds, 30 seconds, 15 seconds and 15 seconds on a 4-CPU Centos 7 remote virtual machine, each finding 50,847,534 prime numbers.

```
module top();
  import "DPI-C" function int eratosthenesBlockwise(longint lastNumber, longint sliceSize);
  ...
  found = eratosthenesBlockwise(lastNumber, sliceSize);
...
endmodule
```

```
// http://create.stephan-brumme.com/eratosthenes/
// https://computing.llnl.gov/tutorials/pthreads/

#include <pthread.h>
```

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

typedef struct {
  long from;
  long to;
  long found;
} from_to_t;

void *LaunchThread(void *v) {
  from_to_t *from_to;
  from_to = (from_to_t *)v;
  from_to->found = eratosthenesOddSingleBlock(from_to->from, from_to->to);
}

int eratosthenesBlockwise(long lastNumber, long sliceSize)
{
  pthread_t threads[100];
  from_to_t from_to[100];
  int nthreads = 0;
  void *status;
  int found;

  // each slices covers ["from" ... "to"], incl. "from" and "to"
  for (long from = 2; from <= lastNumber; from += sliceSize)
  {
    long to = from + sliceSize;
    if (to > lastNumber)
      to = lastNumber;
    from_to[nthreads].from = from;
    from_to[nthreads].to = to;
    from_to[nthreads].found = 0;
    rc = pthread_create(&threads[nthreads], NULL, LaunchThread, &from_to[nthreads]);
    nthreads++;
  }

  found = 0;
  for (int j = 0; j < nthreads; j++) {
    pthread_join(threads[j], &status); // Effectively a wait on thread 'j'
    found += from_to[j].found;
  }

  return found;
}

int eratosthenesOddSingleBlock(const long from, const long to)
{
  const long memorySize = (to - from + 1) / 2;
  // initialize
  char* isPrime = malloc(memorySize);
  for (long i = 0; i < memorySize; i++)
    isPrime[i] = 1;
  for (long i = 3; i*i <= to; i+=2)
  {
    ...
    // skip numbers before current slice
    long minJ = ((from+i-1)/i)*i;
    if (minJ < i*i)
      minJ = i*i;
    // start value must be odd
    if ((minJ & 1) == 0)
      minJ += i;
    // find all odd non-primes
    for (long j = minJ; j <= to; j += 2*i)
    {
      long index = j - from;
      isPrime[index/2] = 0;
    }
  }
  // count primes in this block
  int found = 0;
```

The data to be passed to each pthread

The pthread that gets started. It in turns starts the real "worker thread" with the proper data.

The entry point from C. The inputs are the highest number (when to stop), and the size of slices for parallelization.

Fill up the data to be passed to each thread (from and to). The number of primes found is set to 'found' when this thread finishes.

Get the results from the harvested thread data

The worker thread

```
  for (long i = 0; i < memorySize; i++)
    found += isPrime[i];
  // 2 is not odd => include on demand
  if (from <= 2)
    found++;
  free(isPrime);
  return found;
}
```

Pthreads are used here as workers that a SystemVerilog import call starts. That import call does NOT return until all the pthread worker threads have completed. Once they all complete, the answer is returned to SystemVerilog. This is a very straightforward usage of pthreads. For more complex usages, care must be taken. Figure 3 illustrates pthreads completing and returning. Figure 4 illustrates the more complex arrangement where the pthreads remain running and communicate via a shared data structure.

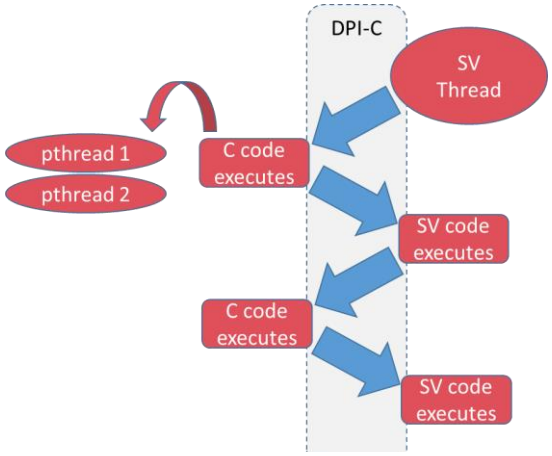

Figure 3: Pthreads started, compute, end

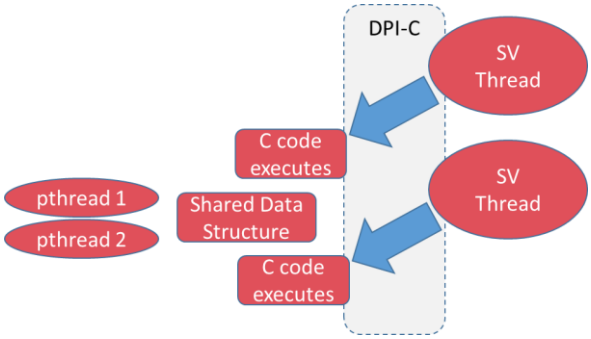

Figure 4: Pthreads running - communicating via shared data

DPI-C SystemC Model Integration in HDL design

In today's complex designs, it is common to have SystemC [10] models as part of IP integration or functionality reuse. These SystemC models are written at a high level of abstraction and are self-contained.

A. Stand-alone SystemC and RTL model verification

In order to verify this high level SystemC model against the RTL there are many possible solutions. However, depending on the model, this checking can be difficult to implement. In a simple case, the SystemC model receives

some input and produces some output, perhaps with its own clock, operating "alone" in its own environment. The RTL model can take the same input and produce output, perhaps with its own clock, operating "alone" in its own environment. The two outputs (SystemC and RTL) can be compared. Figure 5 illustrates this kind of stand-alone verification.

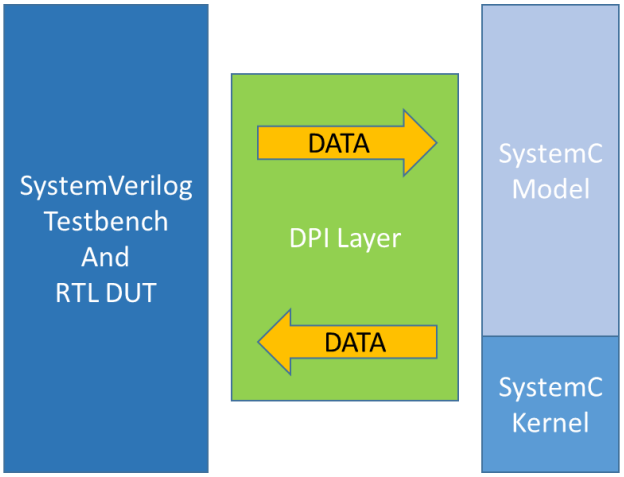

Figure 5: SystemC – SystemVerilog RTL – Two separate models

In this approach a DPI-C layer can be created to peek() and poke() data into the SystemC model. For example, the SystemVerilog testbench could load the 4 registers on the SystemC model using a DPI-C API to copy the register values from the SystemVerilog testbench into the SystemC model. Then the SystemVerilog testbench could tell the SystemC model to begin calculating and wait for an answer. The answer is returned to the SystemVerilog testbench. Now the SystemVerilog testbench provides the same register values to the RTL DUT, and waits for the output. The two outputs are compared.

With a simple model this approach can be successfully managed. (Load Registers, Go, Get Answer). With a more complex model, this approach can become unwieldy. Some issues are:

User may need to manage full SystemC kernel library themselves

User may need to manage synchronization of HDL design and SystemC Models themselves

User may need to manage kernel function calls inside DPI layer themselves

B. Instantiate SystemC in SystemVerilog

In this approach, user can directly instantiate the SystemC model inside the HDL design. This is the easiest model to manage, and allows for a shared clock, and other hardware synchronizations. It may require the SystemC model to be re-architected or designed from the beginning with this use model in mind.

```
module top();
  sc_model SC_INSTANCE(…);
  vl_model VL_INSTANCE(…);
endmodule
```

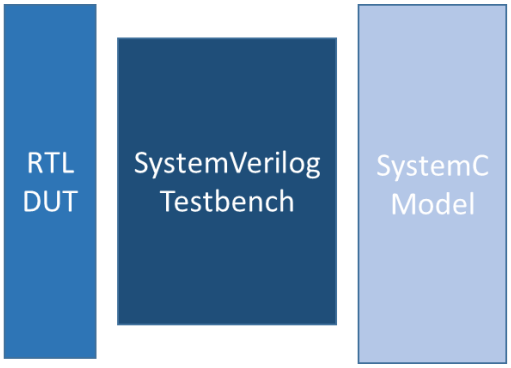

Figure 6: SystemC – SystemVerilog RTL – Direct Instantiation

This approach eliminates the downsides of the previous approach and makes SystemC models easy to use and work with. This is the preferred approach.

All industry-level simulators provide integrated support for SystemC and HDL usage. Hence, direct SystemC instantiation is supported by all simulators. There may be some simulator-specific differences in how they allow the user to write the direct SystemC instantiations, but these differences are often very easy to handle and only require one time setup in the code.

SUMMARY

This paper has covered many subjects related to using DPI-C successfully. We hope it has inspired the reader to try some of them. All example code is available from the authors.

Keep DPI-C calls simple. Let C code do the "integer" work, and let SystemVerilog code do the 4-state logic work. Be careful about passing large arrays across the language boundary. Design your arrays like "arrays of ints". That way they will be passed by reference and not copied. Multi-dimensional arrays will make things messier. Keep in mind your threading needs. Threaded code is possible with DPI-C and easy to write using pthreads. But you must manage the DPI-C call semantics if you have threads that survive across DPI-C calls.

REFERENCES

[1] SystemVerilog Language Reference Manual
[2] "Using SystemVerilog DPI Now", DVCON 2005, Rich Edelman, Doug Warmke
[3] "A HARDWARE IMPLEMENTATION OF THE ADVANCED ENCRYPTION STANDARD (AES) ALGORITHM USING SYSTEMVERILOG", Bahram Hakhamaneshi, Master's Project, http://www.ecs.csus.edu/wcm/cias/caeiae%20pdfs/a%20hardware%20implementation.pdf
[4] Tiny AES 128, kokke, https://github.com/kokke/tiny-AES128-C
[5] SHA-3, KECCAK, http://opencores.org/project,sha3 and https://github.com/gvanas/KeccakCodePackage/blob/master/Standalone/CompactFIPS202/Keccak-readable-and-compact.c
[6] Mandelbrot Set, https://en.wikipedia.org/wiki/Mandelbrot_set
[7] EGGX Graphics Library, 0.93r5, http://www.ir.isas.jaxa.jp/~cyamauch/eggx_procall/
[8] Parallel Sieve Of Eratosthenes, http://create.stephan-brumme.com/eratosthenes/
[9] Posix Threads Programming, https://computing.llnl.gov/tutorials/pthreads/
[10] SystemC, Accellera, http://accellera.org/downloads/standards/systemc

APPENDIX A – SYSTEMVERILOG TEST FILE

This is a sample file that defines a few SystemVerilog DPI-C import functions with input and output arguments. This example helps explain what the C layout is for various data types. The next appendix is the corresponding C code which implements the imports.

```
typedef bit  [31:0] bv_t;
typedef logic[31:0] lv_t;

typedef struct {
  bit a, b, c;
} zzz_t;

typedef struct {
  int x;
  real y;
  zzz_t z;
} xyz_t;

typedef struct packed {
  bit a, b, c;
} zzz_packed_t;

typedef struct packed {
  int x, y;
  zzz_packed_t z;
} xyz_packed_t;

typedef enum int {
  RED=1, GREEN=2, YELLOW=3
} stoplight_t;


import "DPI-C" function bit f_bv_t      (input bv_t         i, output bv_t         o);
import "DPI-C" function bit f_xyz       (input xyz_t        i, output xyz_t        o);
import "DPI-C" function bit f_xyz_packed(input xyz_packed_t i, output xyz_packed_t o);

import "DPI-C" function bit f_stoplight_t(input stoplight_t i, output stoplight_t  o);

import "DPI-C" function int f_bv_t_array(input bv_t i_array[10], output bv_t o_array[10]);
import "DPI-C" function int f_int_array (input int  i_array[10], output int  o_array[10]);
import "DPI-C" function bit f_bit_array (input bit  i_array[10], output bit  o_array[10]);

import "DPI-C" function bit f_xyz_array       ( input xyz_t        i_array[10],
                                                output xyz_t        o_array[10]);
import "DPI-C" function bit f_xyz_packed_array( input xyz_packed_t i_array[10],
                                                output xyz_packed_t o_array[10]);

import "DPI-C" function int f_bv_t_openarray(input bv_t i_array[], output bv_t o_array[]);
import "DPI-C" function int f_int_openarray (input int  i_array[], output int  o_array[]);
import "DPI-C" function bit f_bit_openarray (input bit  i_array[], output bit  o_array[]);

import "DPI-C" function bit f_xyz_openarray       ( input xyz_t        i_array[],
                                                    output xyz_t        o_array[]);
import "DPI-C" function bit f_xyz_packed_openarray( input xyz_packed_t i_array[],
                                                    output xyz_packed_t o_array[]);

module top();
  bv_t i_bv_t, o_bv_t, ret_bv_t;

  int i_int, o_int, ret_int;
  bit i_bit, o_bit, ret_bit;

  xyz_packed_t i_xyz_packed;
  xyz_packed_t o_xyz_packed;

  int i_array[10];
```

```
  int o_array[10];

  int i_openarray[];
  int o_openarray[];

  int i_array_1[10:2];
  int o_array_1[10:2];
  int i_array_2[2:10];
  int o_array_2[2:10];

  stoplight_t i_stoplight;
  stoplight_t o_stoplight;

  lv_t i_lv_t;
  lv_t o_lv_t;

  integer i_integer, o_integer;

  initial begin
    i_bv_t = 1;
    ret_bv_t = f_bv_t(i_bv_t, o_bv_t);
    $display("i_bv_t=%0d, o_bv_t=%0d, ret=%0d", i_bv_t, o_bv_t, ret_bv_t);

    i_int = 1;
    ret_int = f_int(i_int, o_int);
    $display("o_int=%0d, ret=%0d", o_int, ret_int);

    i_bit = 1;
    ret_bit = f_bit(i_bit, o_bit);
    $display("o_bit=%0d, ret=%0d", o_bit, ret_bit);

    ret_bit = f_xyz_packed(i_xyz_packed, o_xyz_packed);
    $display("i=%p, o=%p, ret=%0d", i_xyz_packed, o_xyz_packed, ret_bit);

    foreach (i_array[i])
      i_array[i] = i+1;

    foreach (o_array[i])
      o_array[i] = 0;

    ret_bit = f_int_array(i_array, o_array);
    $display("i=%p, o=%p, ret=%0d", i_array, o_array, ret_bit);

    i_openarray = new[5];
    o_openarray = new[5];

    foreach (i_openarray[i])
      i_openarray[i] = i+1;

    foreach (o_openarray[i])
      o_openarray[i] = 0;

    ret_bit = f_int_openarray(i_openarray, o_openarray);
    $display("i=%p, o=%p, ret=%0d", i_openarray, o_openarray, ret_bit);

    foreach (i_array_1[i])
      i_array_1[i] = i+1;

    ret_bit = f_int_openarray(i_array_1, o_array_1);
    $display("i=%p, o=%p, ret=%0d", i_array_1, o_array_1, ret_bit);

    foreach (i_array_2[i])
      i_array_2[i] = i+1;

    ret_bit = f_int_openarray(i_array_2, o_array_2);
    $display("i=%p, o=%p, ret=%0d", i_array_2, o_array_2, ret_bit);

    i_stoplight = RED;
    ret_bit = f_stoplight_t(i_stoplight, o_stoplight);
    $display("i=%p, o=%p, ret=%0d", i_stoplight, o_stoplight, ret_bit);
    i_stoplight = YELLOW;
```

```
    ret_bit = f_stoplight_t(i_stoplight, o_stoplight);
    $display("i=%p, o=%p, ret=%0d", i_stoplight, o_stoplight, ret_bit);
    i_stoplight = GREEN;
    ret_bit = f_stoplight_t(i_stoplight, o_stoplight);
    $display("i=%p, o=%p, ret=%0d", i_stoplight, o_stoplight, ret_bit);

    i_lv_t = '0;
    ret_bit = f_lv_t(i_lv_t, o_lv_t);
    $display("i=%p, o=%p, ret=%0d", i_lv_t, o_lv_t, ret_bit);

    i_lv_t = '1;
    ret_bit = f_lv_t(i_lv_t, o_lv_t);
    $display("i=%p, o=%p, ret=%0d", i_lv_t, o_lv_t, ret_bit);

    i_lv_t = 'z;
    ret_bit = f_lv_t(i_lv_t, o_lv_t);
    $display("i=%p, o=%p, ret=%0d", i_lv_t, o_lv_t, ret_bit);

    i_lv_t = 'x;
    ret_bit = f_lv_t(i_lv_t, o_lv_t);
    $display("i=%p, o=%p, ret=%0d", i_lv_t, o_lv_t, ret_bit);

    i_integer = 1024;
    ret_bit = f_integer(i_integer, o_integer);
    $display("i=%p, o=%p, ret=%0d", i_integer, o_integer, ret_bit);

  end
endmodule
```

APPENDIX B – C TEST FILE

This is a sample C file which defines imports functions with a variety of input and output arguments. The SystemVerilog code in the previous appendix calls this code.

```
#include "svdpi.h"
#include "svdpi_src.h"
#include "dpiheader.h"
#include "vpi_user.h"

typedef struct {
  svBit a, b, c;
} zzz_t;

typedef struct {
  int x;
  double y;
  zzz_t z;
} xyz_t;

svBit f_bit(svBit i, svBit* o) {...}

svBit f_bit_array(
    const svBit* i_array,
          svBit* o_array) {...}

svBit f_bit_openarray(
    const svOpenArrayHandle i_array,
    const svOpenArrayHandle o_array) {...}

int f_bv_t(
    const svBitVecVal* i,
          svBitVecVal* o) {
  vpi_printf(" i=%0d[0x%x]\n",  i, i);
  vpi_printf("*i=%0d\n", *i);
  *o = *i;
  return *i;
}

int f_lv_t(
    const svLogicVecVal* i,
          svLogicVecVal* o) {
  vpi_printf(" i    =%0d[0x%x]\n",  i, i);
  vpi_printf("i.aval=%0d[0x%x]\n",
    i->aval, i->aval);
  vpi_printf("i.bval=%0d[0x%x]\n",
    i->bval, i->bval);
  *o = *i;
  return 0;
}

int f_bv_t_array(
    const svBitVecVal* i_array,
          svBitVecVal* o_array) {...}

int f_bv_t_openarray(
    const svOpenArrayHandle i_array,
    const svOpenArrayHandle o_array) {...}

int f_int(int i, int* o) {
  *o = i;
  return 0;
}

int f_integer(
    const svLogicVecVal* i,
          svLogicVecVal* o) {
```

```
  vpi_printf(" i    =%0d[0x%x]\n",  i, i);
  vpi_printf("i.aval=%0d[0x%x]\n",
    i->aval, i->aval);
  vpi_printf("i.bval=%0d[0x%x]\n",
    i->bval, i->bval);
  *o = *i;
  return 0;
}

int f_int_array(
    const int* i_array, int* o_array) {
  int i;
  for (i = 0; i < 10; i++) {
    o_array[i] = i_array[i];
  }
  return 0;
}

int f_int_openarray(
    const svOpenArrayHandle i_array,
    const svOpenArrayHandle o_array) {
  int j, n;
  int *i, *o;
  n = svSize(i_array, 1);
  i = svGetArrayPtr(i_array);
  o = svGetArrayPtr(o_array);
  for (j = 0; j < n; j++) {
    o[j] = i[j];
  }
  return 0;
}
```

```
svBit f_stoplight_t(int i, int* o) {
  vpi_printf("stoplight(%0d)\n", i);
  *o = i;
  return 0;
}

svBit f_xyz(const xyz_t* i, xyz_t* o) {...}

svBit f_xyz_array(
    const xyz_t* i_array,
          xyz_t* o_array) {...}

svBit f_xyz_openarray(
    const svOpenArrayHandle i_array,
    const svOpenArrayHandle o_array) {...}

svBit f_xyz_packed(
    const svBitVecVal* i,
          svBitVecVal* o) {...}

svBit f_xyz_packed_array(
    const svBitVecVal* i_array,
          svBitVecVal* o_array) {...}

svBit
f_xyz_packed_openarray(
    const svOpenArrayHandle i_array,
    const svOpenArrayHandle o_array) {
  return 0;
}
```

APPENDIX C – SVDPI.H

This is a sanitized version of the SystemVerilog LRM svdpi.h. It defines useful types, macros and access routines. Generally speaking, these routines are used infrequently. This is because most DPI-C code deals with C-style data, and the macros and access routines are not needed.

```
#define SV_CANONICAL_SIZE(WIDTH) (((WIDTH)+31)>>5)

/* Number of chunks required to represent the given width packed array */
#define SV_PACKED_DATA_NELEMS(WIDTH) (((WIDTH) + 31) >> 5)

/*
 * Since the contents of the unused bits is undetermined,
 * the following macros can be handy.
 */
#define SV_MASK(N) (~(-1 << (N)))

#define SV_GET_UNSIGNED_BITS(VALUE, N) \
    ((N) == 32 ? (VALUE) : ((VALUE) & SV_MASK(N)))

#define SV_GET_SIGNED_BITS(VALUE, N) \
    ((N) == 32 ? (VALUE) : \
    (((VALUE) & (1 << (N))) ? ((VALUE) | ~SV_MASK(N)) : ((VALUE) & SV_MASK(N))))


/* macros for declaring variables to represent the SystemVerilog */
/* packed arrays of type bit or logic */
/* WIDTH= number of bits,NAME = name of a declared field/variable */

#define SV_BIT_PACKED_ARRAY(WIDTH,NAME) \
    svBitVec32 NAME [SV_CANONICAL_SIZE(WIDTH)]

#define SV_LOGIC_PACKED_ARRAY(WIDTH,NAME) \
    svLogicVec32 NAME [SV_CANONICAL_SIZE(WIDTH)]
```

```
/* canonical representation */
#define sv_0 0
#define sv_1 1
#define sv_z 2
#define sv_x 3

typedef unsigned char  uint8_t;
typedef unsigned short int uint16_t;
typedef unsigned int   uint32_t;

typedef uint8_t  svScalar;
typedef svScalar svBit;
typedef svScalar svLogic;

typedef struct t_vpi_vecval {
    uint32_t aval;
    uint32_t bval;
} s_vpi_vecval, *p_vpi_vecval;

typedef s_vpi_vecval svLogicVecVal;
typedef uint32_t     svBitVecVal;

const char* svDpiVersion(void);

typedef void* svScope;
typedef void* svOpenArrayHandle;

 svBit svGetBitselBit(const svBitVecVal* s, int i);
 void  svPutBitselBit(      svBitVecVal* d, int i, svBit s);

 svLogic svGetBitselLogic(const svLogicVecVal* s, int i);
 void    svPutBitselLogic(      svLogicVecVal* d, int i, svLogic s);

 void svGetPartselBit  (svBitVecVal* d, const svBitVecVal* s, int i, int w);
 void svPutPartselBit  (svBitVecVal* d, const svBitVecVal  s, int i, int w);

 void svGetPartselLogic(svLogicVecVal* d, const svLogicVecVal* s, int i, int w);
 void svPutPartselLogic(svLogicVecVal* d, const svLogicVecVal  s, int i, int w);

 void svCopyVecValBit  (svBitVecVal* d, const svBitVecVal* s, int w);
 void svCopyVecValLogic(svLogicVecVal* d, const svLogicVecVal* s, int w);

 int svLeft      (const svOpenArrayHandle h, int d);
 int svRight     (const svOpenArrayHandle h, int d);
 int svLow       (const svOpenArrayHandle h, int d);
 int svHigh      (const svOpenArrayHandle h, int d);

 int svIncrement (const svOpenArrayHandle h, int d);
 int svSize      (const svOpenArrayHandle h, int d);
 int svDimensions(const svOpenArrayHandle h);

 void* svGetArrayPtr  (const svOpenArrayHandle);
 int   svSizeOfArray  (const svOpenArrayHandle);
 void* svGetArrElemPtr(const svOpenArrayHandle, int indx1, ...);

 void* svGetArrElemPtr1(const svOpenArrayHandle, int indx1);
 void* svGetArrElemPtr2(const svOpenArrayHandle, int indx1, int indx2);
 void* svGetArrElemPtr3(const svOpenArrayHandle, int indx1, int indx2, int indx3);

 void svPutBitArrElemVecVal(const svOpenArrayHandle d, const svBitVecVal* s, int indx1, ...);
 void svPutBitArrElem1VecVal(const svOpenArrayHandle d, const svBitVecVal* s, int indx1);
 void svPutBitArrElem2VecVal(const svOpenArrayHandle d, const svBitVecVal* s, int indx1,
                                                                              int indx2);
 void svPutBitArrElem3VecVal(const svOpenArrayHandle d, const svBitVecVal* s, int indx1,
                                                                              int indx2,
                                                                              int indx3);

 void svPutLogicArrElemVecVal (const svOpenArrayHandle d, const svLogicVecVal* s, int indx1,
                                                                                  ...);
```

```
void svPutLogicArrElem1VecVal(const svOpenArrayHandle d, const svLogicVecVal* s, int indx1);
void svPutLogicArrElem2VecVal(const svOpenArrayHandle d, const svLogicVecVal* s, int indx1,
                                                                                  int indx2);
void svPutLogicArrElem3VecVal(const svOpenArrayHandle d, const svLogicVecVal* s, int indx1,
                                                                                  int indx2,
                                                                                  int indx3);

void svGetBitArrElemVecVal (svBitVecVal* d, const svOpenArrayHandle s, int indx1, ...);
void svGetBitArrElem1VecVal(svBitVecVal* d, const svOpenArrayHandle s, int indx1);
void svGetBitArrElem2VecVal(svBitVecVal* d, const svOpenArrayHandle s, int indx1, int indx2);
void svGetBitArrElem3VecVal(svBitVecVal* d, const svOpenArrayHandle s, int indx1, int indx2,
                                                                                   int indx3);

void svGetLogicArrElemVecVal (svLogicVecVal* d, const svOpenArrayHandle s, int indx1, ...);
void svGetLogicArrElem1VecVal(svLogicVecVal* d, const svOpenArrayHandle s, int indx1);
void svGetLogicArrElem2VecVal(svLogicVecVal* d, const svOpenArrayHandle s, int indx1,
                                                                           int indx2);
void svGetLogicArrElem3VecVal(svLogicVecVal* d, const svOpenArrayHandle s, int indx1, int indx2,
                                                                           int indx3);

svBit svGetBitArrElem (const svOpenArrayHandle s, int indx1, ...);
svBit svGetBitArrElem1(const svOpenArrayHandle s, int indx1);
svBit svGetBitArrElem2(const svOpenArrayHandle s, int indx1, int indx2);
svBit svGetBitArrElem3(const svOpenArrayHandle s, int indx1, int indx2, int indx3);

svLogic svGetLogicArrElem (const svOpenArrayHandle s, int indx1, ...);
svLogic svGetLogicArrElem1(const svOpenArrayHandle s, int indx1);
svLogic svGetLogicArrElem2(const svOpenArrayHandle s, int indx1, int indx2);
svLogic svGetLogicArrElem3(const svOpenArrayHandle s, int indx1, int indx2, int indx3);

void svPutLogicArrElem (const svOpenArrayHandle d, svLogic value, int indx1, ...);
void svPutLogicArrElem1(const svOpenArrayHandle d, svLogic value, int indx1);
void svPutLogicArrElem2(const svOpenArrayHandle d, svLogic value, int indx1, int indx2);
void svPutLogicArrElem3(const svOpenArrayHandle d, svLogic value, int indx1, int indx2,
                                                                  int indx3);

void svPutBitArrElem (const svOpenArrayHandle d, svBit value, int indx1, ...);
void svPutBitArrElem1(const svOpenArrayHandle d, svBit value, int indx1);
void svPutBitArrElem2(const svOpenArrayHandle d, svBit value, int indx1, int indx2);
void svPutBitArrElem3(const svOpenArrayHandle d, svBit value, int indx1, int indx2, int indx3);

svScope svGetScope(void);
svScope svSetScope(const svScope scope);

const char* svGetNameFromScope(const svScope);
svScope     svGetScopeFromName(const char* scopeName);

int   svPutUserData(const svScope scope, void* userKey, void* userData);
void* svGetUserData(const svScope scope, void* userKey);

int svGetCallerInfo(const char** fileName, int* lineNumber);

int   svIsDisabledState(void);
void svAckDisabledState(void);
```