Making Your DPI-C Interface a Fast River of Data

Rich Edelman
Mentor, A Siemens Business
Fremont, CA
rich_edelman@mentor.com

***Abstract*-SystemVerilog DPI-C enables functional verification teams to leverage C code for modeling, checking and utility functions. The simple "C" style call interface allows fast adoption and easy integration. This paper explains the workings of the integration and provides data type mapping examples and some hints on optimizing the calls for maximum performance.**

I. INTRODUCTION

The simplest usage of DPI forms the foundation for this paper. But SystemVerilog DPI-C [1][2] is anything but simple. It can be used in simple ways to achieve complex results. It can also be used in more complex ways for similar results. This paper will start simple, and then delve into some complex usages, leaving the most complex usages as an exercise for the reader.

In Figure 1, a SystemVerilog file is created with two C integration points. The so-called "integration points" are nothing more than C callable interfaces.

An 'export function' is declared on line 3, and defined on line 5. An 'import function' is declared on line 2. That import function will be defined in the C code below. An export function or task is an entry point that C (or SystemVerilog) code can call to enter into SystemVerilog. It is SystemVerilog "exporting" a call. An import function or task is an entry point that SystemVerilog (or C) can call to enter into C. It is SystemVerilog "importing" a call. They are simple C callable interfaces.

In the initial block on line 9, the import function (C code is called). That import function is implemented in Figure 2 below, on line 4.

```
 1 module top();
 2   import "DPI-C" context function void f_int_c(input int i);
 3   export "DPI-C"         function      f_int_sv;
 4
 5   function void f_int_sv(input int i);
 6     $display("Hello from f_int_sv(%0d)\n", i);
 7   endfunction
 8
 9   initial
10     f_int_c(1);
11 endmodule
```

Figure 1- SystemVerilog DPI-C passing an 'int'

In Figure 2, the C file defines and declares a function named 'f_int_c' which takes one argument (an integer). It prints a message and then calls the SystemVerilog function 'f_int_sv' with an argument.

Calling 'f_int_sv' on line 6 means that the value i+1 is pushed onto the stack, and the call is made to a C entry point – f_int_sv on line 5 above (Figure 1). A message is printed and then control is returned to the C code. Then the C code returns control to the SystemVerilog initial block and simulation ends.

```
1 #include <stdio.h>
2 #include "dpiheader.h"
3
4 void f_int_c(int i) {
5   printf("Hello from f_int_c(%0d)\n", i);
6   f_int_sv(i+1);
7 }
```

Figure 2 - C code which will be integrated using DPI-C

These two files – a SystemVerilog file and a C file are perhaps the simplest usage of DPI-C. They calls functions and pass an integer. And this is basic foundations of DPI-C. Data on the stack and function calls.

Compiling

In order to simulate these calls, compilation tools must be used on both files. Using the tools below, a special file called 'dpiheader.h' will be created. This file will contain the ANSI function call definitions, including the argument types. It is a very useful file to include in the C code – since it defines what the SystemVerilog simulator thinks are the interface datatypes. Any shared object compiled by gcc can be loaded for use in DPI-C. Please consult the proper documentation for other details about compilation and loading using other means.

```
vlog simplest.sv -dpiheader dpiheader.h
vlog simplest.c
vopt -o opt top
vsim -c opt -do "run -all; quit -f"
```

Figure 3 - Compile the SystemVerilog and C Code and Simulate

This simplest example has two function call definitions in dpiheader.h, listed below. This definition describes two functions which return void and each take an integer argument.

```
void f_int_c(int i);
void f_int_sv(int i);
```

Figure 4 - Helper file optionally created by SystemVerilog compilation

II. What is a C Callable Interface?

A C Callable interface for use here is just a C callable interface in any C program. When C code is compiled by a C compiler the object code is built to process arguments on the stack, and to return an optional return value.

When SystemVerilog creates the export call entry points or the import calls, it arranges the same stack and return value processing. It is building C callable interfaces, just like the C compiler.

To make a call to a C function, a stack of arguments is created, and the program control is set to the called function. The DPI-C interface is exactly that simple. When SystemVerilog DPI-C calls a C function, it is exactly the same as a C program calling that function. It is the same for a C function calling a SystemVerilog DPI-C exported function or task. It is the same as a SystemVerilog task or function making the same call.

Setup a stack and jump.

III. What else is in DPI-C?

Include files

SystemVerilog DPI-C contains constants, datatypes, access routines and other useful information. That information can be found in the include files that ship from the vendors. There should be a file named 'svdpi.h' with other supporting include files as needed.

Access routines

There are many access routines which do a variety of things. Feel free to use any of them, but the ones used in this paper and the associated examples are the easiest to understand and use. The svdpi.h contains hundreds of lines. It contains more than 90 constants, defines and access routines. The best advice on using these is to keep things as simple as possible. The access routines exist in part to help interpret the layout of complex structures that are passed back and forth. The best advice is to limit the complex structures that are passed back and forth.

IV. Passing Data

Passing native C data between SystemVerilog and C is relatively simple. It's just C data. A 32 bit 'int' in SystemVerilog is just a 32 bit 'int' in C. The recommendation is to only pass C data across interface, since that's what C is good at. But there are times when things like 4-state bit vectors might need to be passed. Or open arrays. Or other complex structures.

How is the data passed?

There are two ways that data is actually passed between SystemVerilog and C code.

The first is pass-by-copy. The data is copied. For example a SystemVerilog call to C would copy the data into a proper C-call-stack kind of area. Copying an integer or a small amount of data should not be a problem.

The second is pass-by-reference. There are two kinds of pass-by-reference that might be used.

The first is still pass-by-copy, the data is copied, but the item that is passed to the C side for example is a pointer. For example an array of structs might be copied, and then a pointer to them passed to C. Copying can be necessary because the simulator may not be representing the data in a C style way.

The second pass-by-reference is "real" pass-by-reference. In this case the simulator can simply pass a handle to C, with no copying needed. See the big data example below for more details (Figure 30).

This second pass-by-reference is much faster, since only a reference is needed to be copied.

Data Direction

For DPI-C the function call arguments can be "input", "output" or "inout". An input argument is passed in, an output argument passed out, and an inout argument passed in and out. In C, passing values out through the argument list is a pass-by-reference. The example in Figure 5 and Figure 6 is the same as the first example, but this time the arguments are input, output and inout.

```
module top();
  import "DPI-C" context function int f_int_c(input int i, output int o, inout int io);
  export "DPI-C"         function     f_int_sv;

  function int f_int_sv(input int i, output int o, inout int io);
    o = i + 1;
    io = o + 1;
    $display("@%t: Hello from f_int_sv(%0d, %0d, %0d)\n", $time, i, o, io);
    return io+1;
  endfunction
```

Figure 5 - SystemVerilog Passing 'ints' (inputs, outputs and inouts)

The C code below returns a value in argument 'o' using '*o = i +1'. The argument 'io' is both an input and an output. The output value of io is the output value of o plus one, in this example.

```
int f_int_c(int i, int *o, int *io) {
    int ret;
    *o = i + 1;
    *io = *o + 1;
    ret = f_int_sv(i, o, io);
    return *io+1;
}
```

Figure 6 - C Code Passing'ints' (inputs, outputs and inouts)

This function returns a value – in this case the return value is the same as the return value of io plus one.

Simple C Native Types

The two examples above are passing 'int' values. That's the recommended datatypes for DPI-C ease-of-use. The C native types map to SystemVerilog types as in the table below.

C Native Type	SystemVerilog type
char	byte
short int	shortint
int	int
long long	longint
double	real
float	shortreal
void *	chandle
const char *	String

Simple C native types are the simplest DPI-C interface data. Many DPI-C users use only simple native types with a small collection of calls. This is the simplest usage, easiest to debug and best performance. Simple datatypes, with a few calls. In benchmarking results, the DPI-C calls should execute like any other C code calls, with little or no extra overhead, aside from any copying needed. But like any function call interface, in order to be as fast as possible, the number of function calls should be limited. Calling DPI-C calls on every clock edge is not recommended – in the same way that doing most anything on every clock edge should be minimized.

But there are reasons to use bit vectors, arrays, structs and other SystemVerilog types. And there are reasons to make many function calls.

Using other C native types in DPI-C is left as an exercise – simply change the example above to have a different argument than 'int'. One of the other C native types.

Two-state types

In SystemVerilog, a two-state bit variable can represent a zero or a one. It is declared as a 'bit'. In svdpi.h, SV bits are declared as 'svBit', which is defined as a uint8_t. So a 2 state variable looks like an 8 bit value when it gets over to C. (This might be a good time to read svdpi.h to understand some of the typedefs and constant definitions).

```
module top();
  import "DPI-C" context function bit f_bit_c( input bit i, output bit o, inout bit io);
  export "DPI-C"         function     f_bit_sv;

  function bit f_bit_sv(input bit i, output bit o, inout bit io);
    o = i + 1;
    io = o + 1;
    $display("@%t: Hello from f_bit_sv(%0d, %0d, %0d)\n", $time, i, o, io);
    return io+1;
  endfunction

  bit i;
  bit o;
  bit io;
  bit ret;

  initial begin
    i = 1;
    o = 1;
    io = 1;

    ret = f_bit_c(i, o, io);
    ...
endmodule
```

Figure 7 – SV Code Passing 'bits' (2 state variables)

In the C code below, the variables 'i', 'o', and 'io' are SV bits represented as bytes. In the actual C code, 'o' and 'io' are not actually bytes, but are addresses to a byte – or a pointer to a byte. To return a value for the variable 'o', simple assign '*o' a value.

```
svBit f_bit_c(svBit i, svBit *o, svBit *io) {
  svBit ret;
  *o = i + 1;
  *io = *o + 1;
  printf("Hello from f_bit_c( i=%0d, o=%0d, io=%0d)\n", i, *o, *io);
  ret = f_bit_sv(i, o, io);
  printf("Hello return f_bit_sv(i=%0d, o=%0d, io=%0d) ret=%0d\n", i, *o, *io, ret);
  return *io+1;
}
```

Figure 8 - C Code Passing 'bits' (2 state variables)

Four-state types

In SystemVerilog, a four-state bit variable can represent a zero, a one, an ‘x’ or a ‘z’. It is declared as a ‘logic’ (or ‘reg’). In svdpi.h, SystemVerilog 4-state variables are declared as ‘svLogic’, which is defined as a uint8_t. So a 4 state variable looks like an 8 bit value when it gets over to C.

```
module top();
  import "DPI-C" context function logic f_logic_c(input logic i, output logic o, inout logic io);
  export "DPI-C"         function     f_logic_sv;

  function logic f_logic_sv( input logic i, output logic o, inout logic io);
      o = i + 1;
    io = 'z;
    return io+1;
  endfunction

  logic i;
  logic o;
  logic io;
  logic ret;

  initial begin
    i = 0;
    o = 1;
    io = 1;

    ret = f_logic_c(i, o, io);

    ...
  end
endmodule
```

Figure 9 – SV Code Passing ‘logic’ (4 state variables)

In the C code below, the variables ‘i’, ‘o’, and ‘io’ are SV logic variables represented as bytes. In the actual C code, ‘o’ and ‘io’ are not actually bytes, but are addresses to a byte – or a pointer to a byte. To return a value for the variable ‘o’, simple assign ‘*o’ a value.

```
  svLogic f_logic_c(svLogic i, svLogic *o, svLogic *io) {
    svLogic ret;
    *o = i + 1;
    *io = *o + 1;
    ret = f_logic_sv(i, o, io);
    return *io+1;
  }
```

Figure 10 - C Code Passing ‘logic’ (4 state variables)

For 2-state or 4-state variables, there are helper defines available. Zero and One are represented by their natural values. A value of 2 is a ‘z’ and a value of 3 is an ‘x’. These representations in DPI-C make passing 2-state and 4-state values easy in DPI-C.

```
  #define sv_0 0
  #define sv_1 1
  #define sv_z 2
  #define sv_x 3
```

Figure 11- Defines for 0, 1, x and z

Arrays of C Native Types

The example below passes a fixed size (10) array of integers. The SystemVerilog code initializes the input, output and inout arrays and calls ‘f_array_of_int_c’.

```
module top();
  typedef int array_of_int[10];

  import "DPI-C" context function int f_array_of_int_c( input array_of_int i,
                                                        output array_of_int o,
                                                         inout array_of_int io);
  export "DPI-C"         function     f_array_of_int_sv;

  function bit f_array_of_int_sv( input array_of_int i,
                                 output array_of_int o,
                                  inout array_of_int io);
    foreach (o[x])
      o[x] = i[x] + 1;
    foreach (io[x])
      io[x] = o[x] + 1;
    return 1;
  endfunction

  array_of_int i;
  array_of_int o;
  array_of_int io;
  int ret;

  initial begin
    foreach (i[x])
      i[x] = 10;
    foreach (o[x])
      o[x] = 100;
    foreach (io[x])
      io[x] = 1000;

    ret = f_array_of_int_c(i, o, io);
    #10;
    ret = f_array_of_int_c(i, o, io);
    #10;
    ret = f_array_of_int_c(i, o, io);
  end
endmodule
```

Figure 12 - Passing a fixed sized array of integers

The C code iterates across the fixed sized array (it assumes the array is of size 10). Output values are assigned as are inout values. Then the SystemVerilog function 'f_array_of_int_sv' is called. The SystemVerilog function increments the output and inout values, iterating across all elements (using foreach), Then the stack is unwound, returning to C, which then returns to SystemVerilog.

```
int f_array_of_int_c(const int *i, int *o, int *io) {
  int ret;
  int x;

  for (x = 0; x < 10; x++) {
    o[x] = o[x] + i[x] + 1;
    io[x] = io[x] + o[x] + 1;
  }
    ret = f_array_of_int_sv(i, o, io);
  return ret;
}
```

Figure 13 - C Code passing fixed sized arrays of integers

Arrays of Two-state Types – packed

A packed arrays of bits is another way to say 'int' if the size of the array is 32. A packed arrays of bits is more simply just called a bit vector. In Figure 14 a typedef (BV_t) is created for convenience. This is an 8 bit packed array of bits. The SystemVerilog code initializes the arguments, and calls C. Then C increments the output and inout values and calls SystemVerilog. Again incrementing, then unwinding the stack with returns.

```
module top();
  typedef bit [7:0] BV_t;
  import "DPI-C" context function int f_bitvector_c( input BV_t i,
                                                     output BV_t o,
                                                      inout BV_t io);
  export "DPI-C"         function     f_bitvector_sv;

  function int f_bitvector_sv( input BV_t i, output BV_t o, inout BV_t io);
    o = i + 1;
    io = o + 1;
    return io+1;
  endfunction

  BV_t i;
  BV_t o;
  BV_t io;
  BV_t ret;

  initial begin
    i = 10;
    o = 100;
    io = 1000;

    ret = f_bitvector_c(i, o, io);
    ...
  end
endmodule
```

Figure 14 - Passing Packed Arrays of Bits – Bitvectors

svBitVecVal is defined in svdpi.h as uint32_t. A 32 bit integer. The C code below is passed the address of an integer. A bitvector is an address to an integer in C. 32 bit vectors are easy. Longer vectors are just multiple 32 bit integers.

```
int f_bitvector_c(const svBitVecVal *i, svBitVecVal *o, svBitVecVal *io) {
  svBitVecVal ret;
  *o = *i + 1;
  *io = *o + 1;
  printf("Hello from f_bitvector_c( i=%0d, o=%0d, io=%0d)\n", *i, *o, *io);
  ret = f_bitvector_sv(i, o, io);
  return *io+1;
}
```

Arrays of Four-state Types – packed

Four state types are 0,1,x,z. Packed arrays of four-state types are logic vectors. The DPI-C representation of a logic vector is the same as the PLI/VPI. See the typedef below. A pair of 32 bit integers.

```
typedef struct t_vpi_vecval {
    uint32_t aval;
    uint32_t bval;
} s_vpi_vecval;
typedef s_vpi_vecval svLogicVecVal;
```

Figure 15 - Four state logic vector definition

A logic vector 32 or fewer bits is represented by two 32 bit integers in C. (Longer vectors are chunks of additional pairs of integers). The value of the logic vector is the combination of the aval and bval.

aval	bval	4 state value
0	0	0
1	0	1
0	1	z
1	1	x

Figure 16 below is a simple example with an 8 bit logic vector. The value '01xz01xz' is passed to C, printed and values updated.

```
module top();
  typedef logic [7:0] LV_t;

import "DPI-C" context function int f_logicvector_c( input LV_t i, output LV_t o, inout LV_t io);
export "DPI-C"         function     f_logicvector_sv;

  function int f_logicvector_sv( input LV_t i, output LV_t o, inout LV_t io);
    o = i;
    for(int n = 0; n < 8; n++)
      io[n] = o[8-1-n];
    $display("@%t: Hello from f_logicvector_sv(%b, %b, %b)", $time, i, o, io);
    return 1;
  endfunction

  LV_t i;
  LV_t o;
  LV_t io;
  LV_t ret;

  initial begin
     i = 8'b01xz01xz;
     o = 8'b01xz01xz;
    io = 8'b01xz01xz;
    $display("@%t: Hello from top i=%b, o=%b, io=%b) (before)", $time, i, o, io);
    ret = f_logicvector_c(i, o, io);
    $display("@%t: Hello from top i=%b, o=%b, io=%b) (after)", $time, i, o, io);

  endendmodule
```

Figure 16 - Passing Logic Vectors

This is where things can start to get tricky. Using C for 4 state logic vectors is an unnatural usage for C. It can be done, but it is best to minimize the amount of code and isolate it helper functions – for example 'bits()' and 'print_logicvector()' below. This level of bit twiddling, is best left on the SystemVerilog side. The code in Figure 17 makes many assumptions about vector size and other things. More general support is left as an exercise.

```
  char *bits(uint32_t a) {  // Turn a BIT vector into a string of ASCII 01.
      char *eightbitstringvalue;
      int i;
      uint32_t val;
      int bit;
      char result;

      eightbitstringvalue = (char *)malloc(9);
      val = a;

      eightbitstringvalue[8] = 0;
      for (i = 0; i < 8; i++) {
        bit = val & 1;
             if (bit == 0) result = '0';
        else if (bit == 1) result = '1';
        eightbitstringvalue[8-1-i] = result;
        val = val >> 1;
      }
      return eightbitstringvalue;
  }

  char *print_logicvector(svLogicVecVal *a) { // Return a string of 01xz (examine aval, bval)
      char *eightbitstringvalue;
      int i;
      uint32_t aval;
      int abit;
      uint32_t bval;
      int bbit;
```

```
    char result;

    eightbitstringvalue = (char *)malloc(9);
    aval = a->aval;
    bval = a->bval;

    eightbitstringvalue[8] = 0;
    for (i = 0; i < 8; i++) {
      abit = aval & 1;
      bbit = bval & 1;
           if ((abit == 0) && (bbit == 0)) result = '0';
      else if ((abit == 1) && (bbit == 0)) result = '1';
      else if ((abit == 0) && (bbit == 1)) result = 'z';
      else if ((abit == 1) && (bbit == 1)) result = 'x';
      eightbitstringvalue[8-1-i] = result;
      aval = aval >> 1;
      bval = bval >> 1;
    }
    return eightbitstringvalue;
}

int f_logicvector_c(const svLogicVecVal *i, svLogicVecVal *o, svLogicVecVal *io) {
  int ret;

  printf("Hello from f_logicvector_c(i=%s, o=%s, io=%s) aval\n",
    bits(i->aval), bits(o->aval), bits(io->aval));
  printf("Hello from f_logicvector_c(i=%s, o=%s, io=%s) bval\n",
    bits(i->bval), bits(o->bval), bits(io->bval));
  printf("Hello from f_logicvector_c(i=%s, o=%s, io=%s)\n",
    print_logicvector((svLogicVecVal*)i), print_logicvector(o), print_logicvector(io));

  o->aval = i->aval;
  o->bval = 255;
  io->aval = o->aval;
  io->bval = 255;

  ret = f_logicvector_sv(i, o, io);
  return 1;
}
```

Figure 17- C code handling 4 state logic vector

The C code code has become difficult. Bit vector manipulations and interpreting 4-state values is not something that C is good at. Those things can be done, it is just not as easy as SystemVerilog.

```
# @0: Hello from top             i=01xz01xz, o=01xz01xz, io=01xz01xz) (before)
#     Hello from f_logicvector_c(i=01100110, o=11111111, io=01100110) aval
#     Hello from f_logicvector_c(i=00110011, o=11111111, io=00110011) bval
#     Hello from f_logicvector_c(i=01xz01xz, o=xxxxxxxx, io=01xz01xz)
# @0: Hello from f_logicvector_sv( 01xz01xz,   01xz01xz,    zx10zx10)
# @0: Hello from top             i=01xz01xz, o=01xz01xz, io=zx10zx10) (after)
```

Figure 18 - Output from simulation

The astute reader may notice that the C code prints the “output value” of ‘o’ as all x bits. But those output values haven’t been set – so the C code is just printing whatever the initial value was of the variable put on the stack. Those output values are NOT passed into C. They are passed out. This is the same for all output variables – not just logic vectors.

Structs

Passing structs is quite straightforward. The dpiheader.h even contains a useful struct definition for the C code to include. The C struct below is generated automatically from the SystemVerilog code.

```
typedef struct {
    svBit b;
    svLogic l;
```

```
    int i;
    int64_t li;
} my_struct;
```

Figure 19 - C struct definition in dpiheader.h

The SystemVerilog code below simply assigns values and calls C.

```
module top();

  typedef struct {
    bit b;
    logic l;
    int i;
    longint li;
  } my_struct;

  import "DPI-C" context function int
    f_my_struct_c( input my_struct i, output my_struct o, inout my_struct io);

  export "DPI-C"         function     f_my_struct_sv;

  function bit f_my_struct_sv( input my_struct i, output my_struct o, inout my_struct io);
    o.b = i.b + 1;
    o.l = i.l + 1;
    o.i = i.i + 1;
    o.li = i.li + 1;
    io.b = o.b + 1;
    io.l = o.l + 1;
    io.i = o.i + 1;
    io.li = o.li + 1;
    return 1;
  endfunction

  my_struct i;
  my_struct o;
  my_struct io;
  int ret;

  initial begin
    i.b = 0;
    i.l = 1;
    i.i = 10;
    i.li = 10;
    o.b = 0;
    o.l = 1;
    o.i = 100;
    o.li = 100;
    io.b = 0;
    io.l = 1;
    io.i = 1000;
    io.li = 1000;

    ret = f_my_struct_c(i, o, io);
    ...
  end
endmodule
```

Figure 20 - Passing structs in SystemVerilog

The C code below simply assigns values and increments values and calls SystemVerilog. The arguments are pointers to structs.

```
int f_my_struct_c(const my_struct *i, my_struct *o, my_struct *io) {
  int ret;

  o->b = 1;
  o->l = 1;
  o->i = o->i + i->i + 1;
  o->li = o->li + i->li + 1;
```

```
  io->b = 1;
  io->l = 1;
  io->i = io->i + o->i + 1;
  io->li = io->li + o->li + 1;

  ret = f_my_struct_sv(i, o, io);
  return ret;
}
```

Figure 21 - Passing structs in C

Arrays of Structs

This example has a more complicated struct, and passes arrays of them.

```
module top();

  typedef struct {
    bit[ 1:0] twobits;
    reg[ 1:0] tworegs;
    bit[ 6:0] sevenbits;
    reg[ 6:0] sevenregs;
    bit[31:0] thirtytwobits;
    reg[31:0] thirtytworegs;
    bit[64:0] sixtyfourbits;
    reg[64:0] sixtyfourregs;
    byte b;
    shortint si;
    int i;
    longint li;
  } my_struct2;
  typedef my_struct2 array_of_structs[10];
  import "DPI-C" context function int f_array_of_structs_c(
                                        input int size,
                                        inout array_of_structs io);

  array_of_structs io;
  int ret;

  initial begin
    int counter;
    int size;
    foreach (io[x]) begin
      io[x].tworegs = 1 + counter++;
      io[x].twobits = 1 + counter++;
      io[x].b  = 10 + counter++;
      io[x].si = 100 + counter++;
      io[x].i  = 1000 + counter++;
      io[x].li = 10000 + counter++;
      size++;
    end

    $display("@%t: Hello from top io=%0p)\n", $time, io);
    ret = f_array_of_structs_c(size, io);
    ...
  end
endmodule
```

Using dpiheader.h, the struct definition can be mentally unpacked, and is quite interesting. It is instructive to discover the array sizes – as a way to explore the data layout from DPI-C for vectors. That is left as an exercise for the reader.

```
#ifndef MTI_INCLUDED_TYPEDEF_my_struct2
typedef struct {
    svLogicVecVal tworegs[      SV_PACKED_DATA_NELEMS(2)];
    svBitVecVal   twobits[      SV_PACKED_DATA_NELEMS(2)];
```

```
    svBitVecVal   sevenbits[    SV_PACKED_DATA_NELEMS(7)];
    svLogicVecVal sevenregs[    SV_PACKED_DATA_NELEMS(7)];
    svBitVecVal   thirtytwobits[SV_PACKED_DATA_NELEMS(32)];
    svLogicVecVal thirtytworegs[SV_PACKED_DATA_NELEMS(32)];
    svBitVecVal   sixtyfourbits[SV_PACKED_DATA_NELEMS(65)];
    svLogicVecVal sixtyfourregs[SV_PACKED_DATA_NELEMS(65)];
    char b;
    short si;
    int i;
    int64_t li;
} my_struct2;
#endif

int f_array_of_structs_c(int size, my_struct2* io) {
  int x, y;
  char *c;
  char cc;
  my_struct2 *p;

  for (x = 0; x < size; x++) {
    p->b++;
    p->si++;
    p->i++;
    p->li++;
      printf("Hello from f_array_of_structs_c( io[%0d]=(%0d,%0d,%0d,%0ld))\n",
        x, p->b, p->si, p->i, p->li);
    p++;
  }
  return 0;
}
```

Figure 22 - C code with a struct containing many types

The array of structs in C is an array of pointers to each struct. It is quite easy to manage.

Return Values for functions and tasks

Functions in SystemVerilog DPI-C can return any C native type, also known as SystemVerilog "small values". (void, byte, shortint, int, longint, real shortreal, chandle and string). See SystemVerilog LRM section 25.5.5 "Function result".

SystemVerilog DPI-C tasks always return an 'int'. That 'int' indicates whether the task was disabled. Returning non-zero means it was disabled. Anyone using this part of DPI-C certainly must be bored by this paper. The API sv_IsDisabledState() may be of interest. See section 35.9 "Disabling Tasks and Functions" in the LRM, and generally section 35 "Direct programming interface". Disabling threads is fraught.

Timing-less calls - functions

Most examples in this paper are simple, and are implemented with functions. This makes them timing-less. They could have been implemented as tasks without any timing constructs. There are two differences between tasks and functions. A task in SystemVerilog has no return value. A function can have a return value. The second difference is that a task *can* cause the simulator to yield using a wait, a #delay or other synchronization construct. The task can cause simulation time to pass. A function cannot cause simulation time to pass. (DPI-C tasks C code do return a value – for disable detection. SystemVerilog tasks have this maintained a different way).

In SystemVerilog, a function (timing-less) cannot call a task (possibly containing timing). There are certain execptions for fork/join_none. For SystemVerilog DPI-C, functions cannot call tasks. In the case that a chain of DPI-C calls will block, then that chain must be task based. Functions cannot call tasks, but tasks can call functions.

Tasks are not required to consume time, but in normal usage a task that did not consume time would instead be implemented as a function.

Time-consuming calls - tasks

The code below simply copies an 'int' to C and SystemVerilog. But there are two threads running, and when t_int_sv is called the simulator will yield – blocking for 10 time tics. When the first thread runs, it will hit the #10, and block. Then the second thread will run and hit the #10 and block. Then the threads will wake up and finish.

```
module top();
  import "DPI-C" context task t_int_c(input string threadname,
                                      input  int i,
                                      output int o,
                                      inout  int io);
  export "DPI-C"         task t_int_sv;

  task automatic t_int_sv(input string threadname, input int i,
                                        output int o, inout int io);
    o = i + 1;
    io = o + 1;
    #10;
  endtask

  initial begin // Thread 1
    int i;
    int o;
    int io;

    i = 10;
    o = 100;
    io = 1000;

    t_int_c("1", i, o, io);
    $display("1: @%t: Hello from top i=%0d, o=%0d, io=%0d)\n", $time, i, o, io);
    ...
  end

  initial begin // Thread 2
    int i;
    int o;
    int io;

    i = 20;
    o = 200;
    io = 2000;

    t_int_c("2", i, o, io);
    $display("2: @%t: Hello from top i=%0d, o=%0d, io=%0d)\n", $time, i, o, io);
  end
endmodule
```

Figure 23 - SV blocking DPI-C code with two threads

The code above has two threads – two initial blocks. Each is calling t_int_c. Therefore, t_int_c must the written in a thread-safe way. It has become threaded code. It is threaded because it will be active, and then suspend, yielding to another call of the same function.

```
int t_int_c(const char *threadname, int i, int *o, int *io) {
  int ret;
  *o = i + 1;
  *io = *o + 1;
  printf("%s:Hello t_int_c( i=%0d, o=%0d, io=%0d) [before SV call]\n", threadname, i, *o, *io);
  ret = t_int_sv(threadname, i, o, io);
  printf("%s:Hello t_int_c( i=%0d, o=%0d, io=%0d) [after SV call]\n", threadname, i, *o, *io);
  return 0;  // < return non-zero is a fail
}
```

Figure 24 - C code - increment arguments and call SV export task

The C code just increments the output and inout arguments and calls the SystemVerilog task, t_int_sv. In Figure 23, t_int_sv increments the arguments and delays simulation by 10 ticks (#10). That causes this thread to suspend and the other thread can run.

```
  # 1:      Hello from t_int_c( i=10, o=11, io=12) [before SV call]
  # 1: @ 0: Hello from t_int_sv(10, 11, 12) [before]
  # 2:      Hello from t_int_c( i=20, o=21, io=22) [before SV call]
```

```
# 2: @ 0: Hello from t_int_sv(20, 21, 22) [before]
# 1: @10: Hello from t_int_sv(10, 11, 12) [after]
# 1:      Hello from t_int_c( i=10, o=11, io=12) [after SV call]
# 1: @10: Hello from top i=10, o=11, io=12)
# 2: @10: Hello from t_int_sv(20, 21, 22) [after]
# 2:      Hello from t_int_c( i=20, o=21, io=22) [after SV call]
# 2: @10: Hello from top i=20, o=21, io=22)
```

Figure 25 - Output from two threads and blocking DPI-C

These kinds of blocking calls use SV 'tasks' – they are allowed to block. Any C code used must be thread safe and calls back into SystemVerilog tasks should call 'automatic' tasks for safety.

Getting, setting and printing the scope

```
module OtherModule();
  export "DPI-C"         function      f_scopetest_sv;
  function void f_scopetest_sv();
    $display("@%0t: %m:: Hello", $time);
  endfunction
endmodule

module top();
  import "DPI-C" context function void f_scopetest_c();
  export "DPI-C"         function      f_scopetest_sv;

  OtherModule other_module_instance();

  function void f_scopetest_sv();
    $display("@%0t: %m:: Hello", $time);
  endfunction

  initial begin
    f_scopetest_c();
    ...
endmodule
```

Figure 26 - SystemVerilog code with two module instances

The C code below is using the scope API in DPI-C (svGetScope()) to retrieve the module instance scope and set the module instance scope (svSetScope()). The original module instance scope of the C call is the module instance scope of the C call site. This is where the C call originated.

When the scope is changed, and then a call is made from C to SV, the NEW scope is used as the location (scope) of the call. In the example below, the scope is changed to 'top.other_module_instance'. When the call 'f_scopetest_sv' is executed the actual code that gets executed is the f_scopetest_sv() in the module OtherModule.

Then the C code changes the scope back to the original, and issues the f_scopetest_sv() call, this time executing the f_scopetest_sv() in the module top. (The original call site for the original C call).

```
void f_scopetest_c() {
  char *scopeName;
  svScope scope; // svdpi.h: void*

  scope=svGetScope();
  scopeName = svGetNameFromScope(scope);
  printf("Hello from f_scopetest_c(), scope=%s\n", scopeName);

  f_scopetest_sv();

  // Try OTHER scopes
  scope = svGetScopeFromName("top.other_module_instance");
  svSetScope(scope);
  f_scopetest_sv();

  // Go back to THIS scope
  scope = svGetScopeFromName(scopeName);
  svSetScope(scope);
```

```
    f_scopetest_sv();
  }
```

Figure 27 - C Code which changes scopes and make DPI-C calls

Using user-data or not

The DPI-C has a capability to map 'user data' as a way to associate C or C++ data structures (objects) with SystemVerilog scopes. It is quite powerful. But it has its own complexities. Most kinds of cases can be easily handled simpler ways. (See chandle below). See the LRM for further details.

Using chandle

One alternate to user-data is to use chandles. SystemVerilog chandles are simply special 'void *' handles, which can be returned from C and passed around in SystemVerilog and passed back to C.

```
module top();
  import "DPI-C" context function chandle f_chandle_c(input chandle i,
                                                      output chandle o,
                                                       inout chandle io);
  export "DPI-C"         function         f_chandle_sv;

  function chandle f_chandle_sv( input chandle i, output chandle o, inout chandle io);
    o = i;
    io = o;
    return i;
  endfunction

  chandle i;
  chandle o;
  chandle io;
  chandle ret;

  initial begin
    ret = f_chandle_c(i, o, io);
    ...
  end
endmodule
```

Figure 28 - SystemVerilog chandles

The C code below will call malloc to create a structure, when 'i' is NULL. That new struct is initialized. The 'o' and 'io' values are set and an SV function is called. The SV simply assigns handles and returns.

A real application might be written in SystemVerilog, which calls a DPI-C function to create a 'struct' or object. That object (chandle) is returned and stored by the module instance. Now each time the module instance uses the DPI-C API, it passes in the chandle to operate on. (for example: ACCESS_ROUTINE(handle))

```
typedef struct {
  int i;
  int j;
} my_c_object_t;

my_c_object_t static_object;

void * f_chandle_c(void * i, void * *o, void * *io) {
  int ret;
    my_c_object_t stack_object;
  my_c_object_t *p, *pi, *po, *pio;;

  if (i == NULL) {
    i = malloc(sizeof(my_c_object_t)); // heap object
    p = i;
    p->i = 1;
    p->j = 2;
  } else {
    p = i;
```

```
    }
    p->i = p->i + 1;
    p->j = p->j + 10;
    *o = i;
    p = *o;
    p->i = p->i + 1;
    p->j = p->j + 10;
    *io = *o;
    p = *io;
    p->i = p->i + 1;
    p->j = p->j + 10;

    pi = i;
    po = *o;
    pio = *io;
    printf("Hello from f_chandle_c( i=(%0d,%0d), o=(%0d,%0d), io=(%0d,%0d))\n",
        pi->i, pi->j, po->i, po->j, pio->i, pio->j);
    ret = f_chandle_sv(i, o, io);
    printf("Hello return f_chandle_c( i=(%0d,%0d), o=(%0d,%0d), io=(%0d,%0d))\n",
        pi->i, pi->j, po->i, po->j, pio->i, pio->j);
    return *io;
  }
```

Figure 29 - C code creating structs and chandles

Note that this example creates an "object" – a struct on the C side, and passes handles back to SystemVerilog, but the SystemVerilog code has no idea what those 'void *' chandles actually contain.

DPI Performance Discussion

Performance of DPI-C was mentioned earlier. DPI-C call performance should be comparable to any C code calling any other C code, with one exception – copying data. DPI-C context task calls can also have a small overhead due to process management. But any such overhead is small.

For large data, on the other hand, the overhead of copying can be quite large, sometimes dominating run time. Generally, it is easy to avoid these copying overheads – don't pass large amounts of data across the SV-C boundary frequently. In the case that large amounts of data must be passed across frequently, with some care the overhead can be completely eliminated.

Under special circumstances some optimizations can be performed. For example, for a large array of integers, use an import function with input or inout arguments. This means that a large data set can be passed as input to C. What about output from C? The inout argument can be used for this. See the User Manual of your tool for advice and other details.

Optimizing data for performance

The best performance may be obtained from an array of integers used in an import function input or inout arguments.

```
typedef int big_data[1000];

module top();

  import "DPI-C" function int f_big_data_c(input int size, input big_data i, inout big_data io);

  big_data i;
  big_data io;
  int ret;

  initial begin
    repeat (10_000_000) begin
      ret = f_big_data_c(1000, i, io);
      #10;
    end
  end
endmodule
```

Figure 30 - SystemVerilog solution to Big Data and Performance

The SystemVerilog code above defines an array of 1000 elements. It calls the DPI-C import function 10 million times. Using a non-optimized version can be 2-5x slower.

```
int f_big_data_c(int size, const int *i, int *io) {
  int x;
  for (x = 0; x < size; x++) {
    io[x] = i[x] + x;
  }
  return 0;
}
```

It's always a good idea to double check any DPI-C arguments, in case there are optimization opportunities.

Two Dimensional Arrays

What about 2-dimensional arrays? They benefit from the same optimization speedups.

In the example below, the module instance initializes the large array and then some fancy algorithm might process the large dataset – perhaps doing machine learning training, or other analysis of the dataset. When the fancy algorithm finishes, an event 'check_results' is triggered and a DPI-C call is made to check the results.

The results of the fancy algorithm could be passed to the C checker using an input or inout argument.

In the optimized version a simple handle is passed. In the unoptimized version the data is copied and a handle to the copy is passed.

```
typedef int big_data_2d[100][10];
module top();
  import "DPI-C" function int f_big_data_2d_c(input int sizeX,  input int sizeY,
                                              input big_data_2d i,
                                              inout big_data_2d io);
  big_data_2d i;
  big_data_2d io;
  int ret;
  event check_results;

  always @(check_results) begin
    f_big_data_2d_c(100, 10, i, io);
  end

  initial begin
    int k;
    foreach (i[x,y])
      i[x][y] = k++;
    foreach (i[x,y])
      io[x][y] = k++;
    repeat (10_000_000) begin
      // Perform some fancy algorithm...
      #1000;
      -> check_results;
    end
  $display("@%t: Hello from top i=%0p, io=%0p)\n", $time, i, io);
  end
endmodule
```

Figure 31 - SystemVerilog Big Data 2D array with a checker

```
typedef int big_data_2d_t[100][10];

int f_big_data_2d_c_(int sizeX, int sizeY, const big_data_2d_t i, big_data_2d_t io) {
  int x;
  int y;
  for (x = 0; x < sizeX; x++) {
    for (y = 0; y < sizeY; y++) {
      printf("BIG_DATA_INFO: i[%0d][%0d] = %0d (io=%0d)\n", x, y, i[x][y], io[x][y]);
      io[x][y] = i[x][y] + x + y;
```

```
    }
  }
  return 0;
}

int f_big_data_2d_c(int sizeX, int sizeY, const int *i, int *io) {
  return f_big_data_2d_c_(sizeX, sizeY, i, io);
}
```

Figure 32- C code checker for the fancy algorithm

Notice that the C code is written quite naturally using a one-line wrapper. The ‘i' and ‘io’ arguments are defined arrays of the big_data_2d_t data type. Inside the C code, some checking or further calculation can occur. Notice that the ‘io’ argument is being updated (values are output). When the C code returns, those values are not copied back in the optimized case. The C is changing the values in place.

V. INTERESTING USE MODELS

Recording C Data in Waveforms

C variables below, recorded in the wave database alongside other testbench and DUT signals. This example starts a fancy C algorithm, and inserts “probes” to record some interesting data using DPI-C and interface instances.

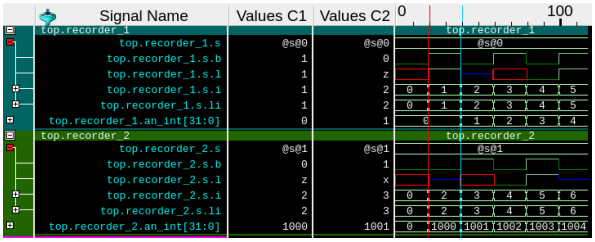

Figure 33 - Recording C code variables with DPI-C

```
interface recorder();
  typedef struct {
    bit b;
    logic l;
    int i;
    longint li;
  } my_struct;

  my_struct s;
  int an_int;

  export "DPI-C" function record_an_int;
  function void record_an_int(input int i);
    an_int = i;
  endfunction

  export "DPI-C" function record_my_struct;
  function void record_my_struct(input my_struct i);
    s = i;
  endfunction
endinterface

module top();
  reg clk;

  recorder recorder_1();
  recorder recorder_2();

  import "DPI-C" context task run_c_code1();
  import "DPI-C" context task run_c_code2();

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

```
  task tictoc(int times);
    repeat (times)
      @(posedge clk);
  endtask

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

  initial
    run_c_code1();
  initial
    run_c_code2();
  initial
    #1000 $finish(2);
endmodule
```

Figure 34 - Simple SystemVerilog Recorder

The SystemVerilog code above takes a quick and dirty approach, building an export function for each kind of "recording probe" and hosting those calls in a SystemVerilog interface. This interface gets instantiated somewhere and that instance hierarchy name is used from within the C code. This solution is acceptable for a small number of probes, but other more complicated solutions will scale better. Those are left as an exercise.

In the C code below, the C algorithm is running. At a good "probe point", scope is set to the instance hierarchy of the proper recorder, and a call to the exported function is made. In the first case below, effectively, a call is made to "top.recorder_1.record_my_struct(&s)" and "top.recorder_1.record_an_int(k)".

The helper function push_scope and pop_scope are ripe for further optimizations and simplifications. But the concept remains the same. The C code is instrumented with a call to DPI-C in a way that causes signal changes to be recognized and recorded in the SystemVerilog.

```
svScope push_scope(char *scopeName) {
  svScope scope, oldscope;

  oldscope = svGetScope();
  scope = svGetScopeFromName(scopeName);
  svSetScope(scope);
  return oldscope;
}

void pop_scope(svScope oldscope) {
  svSetScope(oldscope);
}

int run_c_code1 () {
    svScope scope, oldscope;

    my_struct s;
    int k;

    // Initialize 's'
    ...
    for (k = 0; k < 100; k++) {
      // Update 's'
      ...
      tictoc(1);
      oldscope = push_scope("top.recorder_1");
        record_my_struct(&s); // Record C Code Variables
        record_an_int(k);
      pop_scope(oldscope);
    }
  return 0;
}

int run_c_code2 () {
    svScope scope, oldscope;
```

```
    my_struct s;
    int k;

    // Initialize 's'
    ...
    for (k = 0; k < 100; k++) {
      // Update 's'
      ...
      tictoc(1);
      oldscope = push_scope("top.recorder_2");
        record_my_struct(&s); // Record C Code Variables
        record_an_int(k+1000);
      pop_scope(oldscope);
    }
  return 0;
}
```

Figure 35 - C code with a fancy algorithm - in need of waveform recording

Open Arrays – Using some DPI-C API calls

Open arrays in DPI-C are simply arrays that have a dynamic size. They are new'ed on the SystemVerilog side. This is a great time to use the DPI-C access routines.

```
typedef int openarray_2d[][];

module top();
  import "DPI-C" function int f_openarray_2d_c(
                                          input  openarray_2d i,
                                          output openarray_2d o,
                                          inout  openarray_2d io);
  openarray_2d i;
  openarray_2d o;
  openarray_2d io;
  int ret;
  int X, x;
  int Y, y;
  int ki, ko, kio;

  function void print_openarray_2d(string name, openarray_2d a);
    ...
  endfunction

  initial begin
    repeat (10) begin
      X = $urandom_range(10, 5);
      Y = $urandom_range(14, 11);
      i = new[X];
      o = new[X];
      io = new[X];
      // Fill
      for (x = 0; x < X; x++) begin
        i[x] = new[Y];
        o[x] = new[Y];
        io[x] = new[Y];
      end

      for (x = 0; x < X; x++) begin
        for (y = 0; y < Y; y++)  begin
           i[x][y] = 12340000 + ki++;
           o[x][y] = 23450000 + ko++;
          io[x][y] = 34560000 + kio++;
        end
      end


      print_openarray_2d("i", i);
      print_openarray_2d("o", o);
      print_openarray_2d("io", io);
```

```
      ret = f_openarray_2d_c(i, o, io);
      #10;
    end
  end
endmodule
```

Figure 36 - SystemVerilog DPI-C and Open Array of integers

The C code below calls svGetArrayPtr on each argument, to get the start of the array data. Then svSize is used for the first dimension and the second dimension (X and Y). An "int *" pointer (p_i) is used to step through each array element. This level of details with the C compiler pointer arithmetic and data layout is likely the limit for a normal DPI-C user. Further in-depth layout and coding can handle many other layouts, but at the expense of clarity and complexity. As that complex code is written, rethink the plan to perhaps handle the details in SystemVerilog.

```
void print_openarray_2d(char *name, svOpenArrayHandle a) {
  ...
}

int f_openarray_2d_c( const svOpenArrayHandle i, const svOpenArrayHandle o,
                                                 const svOpenArrayHandle io) {

  int count;

  int X, x;
  int Y, y;
  int *p_i, *p_o, *p_io;

  print_openarray_2d("i", i);
  print_openarray_2d("o", o);

  p_i = svGetArrayPtr(i);
  p_o = svGetArrayPtr(o);
  p_io = svGetArrayPtr(io);

  X = svSize(i, 1);;
  Y = svSize(i, 2);;

  count = 0;
  for (x = 0; x < X; x++) {
    for (y = 0; y < Y; y++) {
      *p_i = *p_i + 1; // i[x][y]
      *p_o = 0;        // o[x][y]
      p_i++;
      p_o++;
    }
  }
  // Set o[2][3] = 23;
  p_o = svGetArrayPtr(o);
  *(p_o + Y*2 + 3) = 23; // *(p + Y*x + y)

  print_openarray_2d("o", o);
  print_openarray_2d("io", io);

  return 0;
}
```

Figure 37 - C Code for Open Arrays

VI. Conclusion

SystemVerilog DPI-C remains a powerful way to combine C and C++ with SystemVerilog providing functionality including modeling, checking, verification, and user interface functions. This paper shares examples and explanations about datatypes, scopes, threaded execution, chandles and much more to enable a path the either getting started on DPI-C or to extending an existing implementation.

I hope you enjoy DPI-C as much as I do. Please contact the author for complete source code for the examples.

VII. REFERENCES

[1] SystemVerilog LRM, “1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language”, https://ieeexplore.ieee.org/document/8299595

[2] Using SystemVerilog Now With DPI, DVCON 2005, Rich Edelman