SystemVerilog DPI (Direct Programming Interface)

VCS 2006.06-SP2-2

Synopsys Global Technical Services

Agenda

Introduction: C Integration Support

DPI

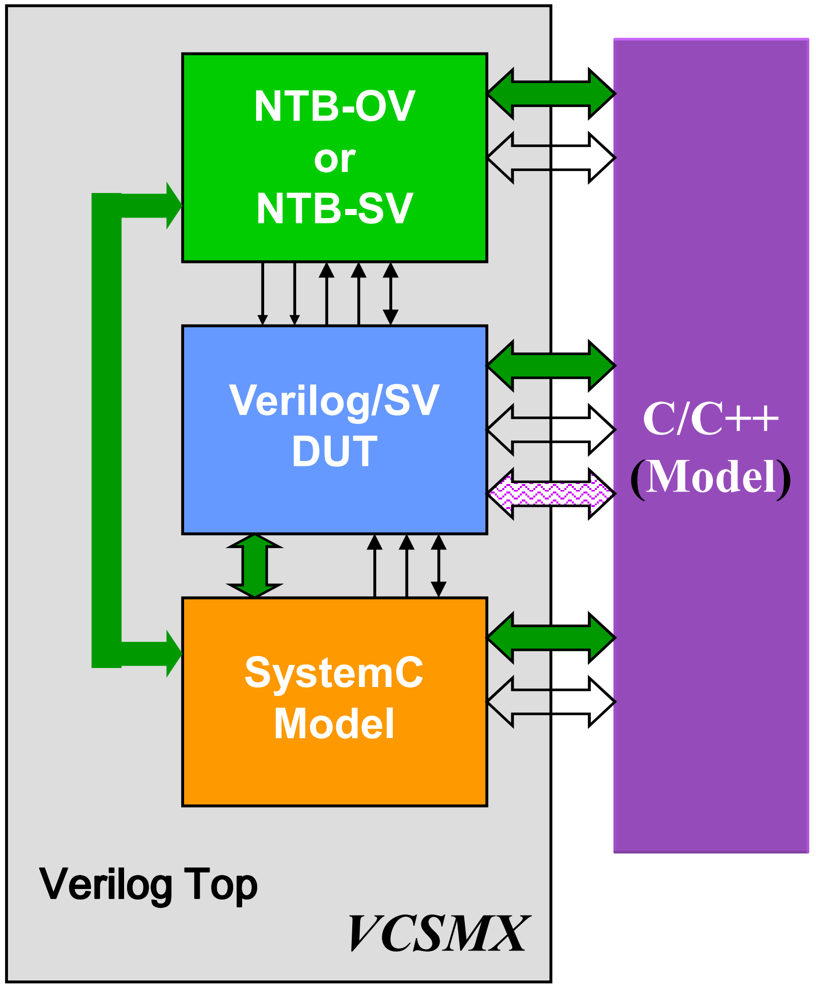
- **SystemVerilog Standard** to connect foreign-languages
  - Does not replace PLI/VPI
  - Restricted Data Types
  - Little Access to Sim Database
- Faster & Easier than PLI

PLI/VPI

- **IEEE** simulation API
  - Waveform dump
  - Netlist analysis

DirectC

- Proprietary VCS I/F
- For legacy Code. DPI for new applications



DPI: Direct Programming Interface

- **An interface between SystemVerilog and a foreign programming language: C or C++**
- **Simple interface to C models**
  - Allows SystemVerilog to call a C function just like any other native SystemVerilog function/task
  - Variables passed directly to/from C/C++
  - NO need to write PLI-like applications/wrappers
- **Why DPI?**
  - SystemVerilog users have C/C++ in designs and testbench that they want to reuse
  - DPI easily connects C/C++ code to SV without the overhead or complexity of VPI/PLI
- **Support both functions and tasks**
- **SystemVerilog can call C and C can call SV**

Quick Example: Import task from C

- **Import "DPI-C"**
  - SystemVerilog calling C/C++ task
  - C code must have:
    #include <svdpi.h>

```
// SystemVerilog code
program automatic test;
  import "DPI-C" context task c_test(input int addr);

  initial c_test(1000);

endprogram
```

```
// C code
#include <svdpi.h>

void c_test(int addr) {
   ...
}
```

```
> vcs -R -sverilog top.sv c_test.c
```

Quick Example : Export SV Task to C

- **Export "DPI-C"**
  - C calls SystemVerilog functions or blocking tasks

```
program automatic test;
  import "DPI-C" context task c_test(input int addr);
  initial c_test(1000);
  export "DPI-C" task apb_write;

  task apb_write(input int addr, data);
    ... @(posedge ready); ...
  endtask
endprogram
```

```
#include <svdpi.h>
extern void apb_write(int, int);

void c_test(int addr) {
  ...
  apb_write(addr, data);
 ...
}
```

```
> vcs -sverilog top.sv c_test.c
```

DPI vs. TLI

- **TLI**
  - **T**ransaction **L**evel **I**nterface in VCS for SystemC-SystemVerilog transaction level co-simulation
  - SystemC calling SystemVerilog interface functions/tasks for class methods
  - SystemVerilog call SystemC interface methods
    - Blocking or non-blocking
  - Under the hood, TLI is implemented with DPI
    - But easier to use
    - Automatically synchronizes between the SystemC domain and SystemVerilog

Agenda

Importing a C routine into SystemVerilog

- **Before you can use a C routine in SystemVerilog, you must <u>import</u> it**
  - This declares the C routine in the current scope
- **Now you can call the C routine as if it were a SystemVerilog task or function**
- **SystemVerilog data types are passed through the argument list and mapped into equivalent C structures**
- **If a C routine calls back into SystemVerilog, it should be declared as a task, otherwise a function**

Declaration of Imported Functions and Tasks

- **An imported task or function can be declared anywhere a native SV function or task can be declared**

```
import "DPI-C" pure function real sin(input real r); // math.h
```

- **Imported tasks and functions can have zero or more input, output and inout arguments. `ref` is not allowed.**

```
import "DPI-C" function void myInit();
```

  - Imported tasks always return a void value
  - Imported functions can return a "small" value or be a void function
    - void, byte, shortint, int, longint, real, shortreal, chandle, and string
    - Scalar values of type bit and logic

```
import "DPI-C" function int getStim(input string fname);
```

- **Map C routine name if it conflicts with existing SystemVerilog name**

```
// Map the C routine "test" to "c_test" in SystemVerilog.
import "DPI-C" test = task c_test();
```

Argument Passing

- Map arguments between SystemVerilog and C
  - Recommendation: use `int` whenever possible!
  - It is your responsibility to correctly declare compatible data types
  - The DPI does not check for type compatibility
- VCS produces `vc_hdrs.h` when you compile
  - Use it as a guide to see how types are mapped
- **Argument directions**

`input`	Input to C code
`output`	Output from C code (initial value undefined)
`inout`	Input and Output from C code
`ref`	Not supported by LRM. Use `inout` instead

- **Function return types (<32b)**
  - `(unsigned) int,  char*`

Argument Direction

- **Input argument**
  - Small values of formal input arguments passed by value
  - `int` -> `int`, `bit` -> `unsigned char`
  - Other types (`bit[7:0]`) are passed by reference
- **Inout and output arguments passed by reference**
- **Open arrays are always passed by a handle**
  - Shown in next section
- **Protection**
  - It's up to the C code to not modify input parameters
  - Use const to double check your C code
- **Common mistake – forgetting argument direction**

```
import "DPI-C" function void bug(int out);
```

A 32-bit 2-State Counter

```
program automatic test;
  bit [6:0] out, in;
  bit load;
  import "DPI-C" function void count32(output int o,
                                       input  int i,
                                       input  bit load);

  initial begin
    in = 42;  load = 1;
    count32(out, in, load);  // Load counter
    $display(out);
    load = 0;
    count32(out, in, load)
    $display(out);
  end
endprogram
```

```
#include <svdpi.h>
#include "vc_hdrs.h"
void count32(int                 *o,
             const int           i,
             const unsigned char load) {
  static unsigned char count;
  if (load) count = i;   // Load
  else      count++;     // Increment
  *o = count;            // Copy to output
}
```

What happens when this is instantiated more than once?

Argument Data Type Mapping

- Unpacked SystemVerilog types have a C-compatible representation.

SystemVerilog	C (input)	C (out/inout)
byte	char	char*
shortint	short int	short int*
int	int	int*
longint	long int	long int*
shortreal	float	float*
real	double	double*
string	const char*	char**
string[n]	const char**	char**

SystemVerilog	C (input)	C (out/inout)
bit	svBit (unsigned char)	svBit* (unsigned char)
logic, reg	svLogic	svLogic*
bit[N:0]	const svBitVecVal*	svBitVecVal*
reg[N:0] logic[N:0]	const svLogicVecVal*	svLogicVecVal*
Open array[ ] (import only)	const svOpenArrayHandle	svOpenArrayHandle
chandle	const void*	void*

A 7-bit 2-State Counter

```
program automatic test;
  bit [6:0] out, in;
  bit load;
  import "DPI-C" function void count7(output bit [6:0] o,
                                      input  bit [6:0] i,
                                      input  bit load);

  initial begin
    in = 42;  load = 1;
    count7(out, in, load);  // Load counter
    $display(out);
    load = 0;
    count7(out, in, load);
    $display(out);
  end
endprogram
```

Structure passed by ref

```
void count7(svBitVecVal       *o,
            const svBitVecVal *i,
            const svBit       load) {
  static unsigned char count;
  if (load) count = *i; // Load
  else      count++;    // Increment
  count &= 0x7f;        // Mask bits
  *o = count;           // Copy to output
}
```

What happens when this is instantiated more than once?

Type Mapping - Packed types

- **Packed types are stored with the canonical format:**
  - bit -> svBitVecVal, logic -> svLogicVecVal
- **A packed array is represented as an array of one or more elements**
  - Type svBitVecVal for 2-state and svLogicVecVal for 4-state
- **Each element is stored as a group of 32 bits.**
  - The first element of an array contains the 32 LSBs, next element contains the 32 more significant bits, and so on.
- **C macro `SV_PACKED_DATA_NELEMS(`*`width`*`)`**
  - Convert from bits to number of elements

```
typedef unsigned int svBitVecVal; /* (a chunk of) packed bit array */

typedef struct { unsigned int aval;
                 unsigned int bval;} svLogicVecVal;
              /* (a chunk of) packed logic array */
```

Passing a Bit Vectors with Odd Sizes

```
program automatic test;
  int eq32;             // length = 32
  bit [27:0] lt32;   // length < 32
  bit [35:0] gt32;   // length > 32
  import "DPI-C" function void disp3
            (input int x, y, bit[35:0] z);
  initial begin
    eq32 = 10;
    lt32 = 28'h12_3456;
    gt32 = 36'h1_2345_6789;
    disp3(eq32, lt32, gt32);
  end
endprogram
```

```
a is 10
b is 123456
c[0] is 23456789
c[1] is 1
```

```
#include <svdpi.h>
#include <veriuser.h>
#include “vc_hdrs.h”
void disp3(int a, int b,
           svBitVecVal c[SV_PACKED_DATA_NELEMS(36)]) {
  io_printf("a is %d\n",a);
  io_printf("b is %x\n",b);
  io_printf("c[0] is %x\n",c[0]);
  io_printf("c[1] is %x\n",c[1]); }
```

C macro to convert from bits to elements

`io_printf` prints to stdout & VCS log

Passing 4-state Values

```
program automatic test;
  logic scalar;
  logic [31:0] vec;
  import "DPI-C" function void show(input logic s,
                                    input logic [31:0] i);
  initial begin
    scalar = 'z;  vec = 32'h0000_0101;
    $display("SV: %h %h", scalar, vec);
    show(scalar, vec);

    scalar = 'x; vec = 32'h0000_zzxx;
    $display("SV: %h %h", scalar, vec);
 show(scalar, vec);
  end
endprogram
```

System Verilog	C: svLogic Scalar	C: svLogicVecVal vector Data aval	Control bval
0	0	0	0
1	1	1	0
Z	2	0	1
X	3	1	1

4-state value map

```
void show(const svLogic s,
          const svLogicVecVal *vec) {
  io_printf("C: scalar is %x\n",s);
  io_printf("C: vec=%x/%x\n", vec->aval, vec->bval);
}
```

```
SV: z 00000101
C: scalar is 2
C: i=101/0

SV: x 0000zzxx
C: scalar is 3
C: i=ff/ffff
```

Passing a Logic Vector > 32 bits

```
program automatic test;
  logic[63:0] c;
  import "DPI-C" function void disp64(input logic[63:0] c);
  initial begin
    c = 64'hzzzzzzzz_CafeDada;
    $display("SV: c=%h", c);
    disp64(c);
  end
endprogram
```

```
void disp64(const svLogicVecVal c[2]) {
  io_printf("C: c=%08x_%08x\n",
            c[0].aval, c[1].aval);
}
```

```
SV: c=zzzzzzzzcafedada
C: c=0_cafedada
```

c[31:0].aval cafedada
c[31:0].bval 0
c[63:32].aval 0
c[63:32].bval ffffffff

Agenda

Passing a Fixed-Sized Array

- If you use int type, it is very easy to pass arrays

```
program automatic test;
  int a[4][4];
  import "DPI-C" function void disp(input int a[4][4]);
  initial begin
    foreach (a[i,j])
      a[i][j] = i+j;
    disp(a);
  end
endprogram
```

```
void disp(const int a[4][4]) {
  int i, j;
  for(i=0; i<4; i++)
    for(j=0; j<4; j++)
       io_printf("C: a[%d][%d]=%d\n", i, j, a[i][j]);
}
```

Passing a Logic Array

```
program automatic test;
  logic [63:0] a[2];
  import "DPI-C" function void arr(
                              input logic a[2]);
  initial begin
    a[0] = 64'h12345678_ABCDEF0;
    a[1] = 'z;
    arr(a);
  end
endprogram
```

a[0][31:0].aval
a[0][31:0].bval
a[0][63:32].aval
a[0][63:32].bval
a[1][31:0].aval
a[1][31:0].bval
a[1][63:32].aval
a[1][63:32].bval

```
void arr(const svLogicVecVal *a[4]){
  int i;
  for(i=0;i<2;i++)
    io_printf("C: a[%0d]=%08x_%08x\n",
              i, a[i*2+1].aval, a[i*2].aval);
}
```

```
C: a[0]=12345678_abcdef0
C: a[1]=00000000_0000000
```

Passing a Structure

```
program automatic test;
   typedef struct {
      int a; int b;
   } mystruct;
   import "DPI-C" function void disp_s(inout mystruct s1);
   mystruct s1;
   initial begin
      s1.a = 10;
      s1.b = 20;
      disp_s(s1);
   end
endprogram
```

```
typedef struct {
   int a; int b;
}  mystruct;
void disp_s(mystruct *s1) {
   io_printf("s1.a is %d, s1.b is %d\n",s1->a,s1->b);
}
```

Passing a Packed Structure of Bytes

```
program automatic test;
   typedef struct packed {bit [7:0] r, g, b;} RGB;
   import "DPI-C" function void disp_p(input RGB p);
   initial begin
      RGB pixel;
      pixel.r=1; pixel.g=2; pixel.b=3;
      disp_p(pixel);
   end
endprogram
```

Be careful about big-endian (x86) vs. little endian (Sparc) byte order issues

```
typedef struct {
  unsigned char b, g, r;  // x86 big-endian
} *p_rgb;
void disp_p(p_rgb pixel) {
   io_printf("pixel: %x,%x,%x\n",
             pixel->r, pixel->g, pixel->b);
}
```

```
pixel: 1,2,3
```

Mapping Word Ranges Between Languages

- **Recommendation: Use arrays with LSB=0**

```
int two_val[99:0], one_val[100];
```

- **The natural order of elements for each dimension in the layout of an unpacked array is used**
  - C index 0 ←→ SV index min(L,R)
  - C index abs(L-R) ←→ SV index max(L,R)
    - SystemVerilog: a[2:8] ←→ C: a[7]
    - SystemVerilog: a[2] ←→ C: a[0], SystemVerilog: a[8] ←→ C: a[6]
- **Packed arrays are treated as one-dimensional**
  - An packed array with dimension size (i,j,k) is treated as single dimension of size(i*j*k)
    - One bit in SV myarray[l][m][n] ←→ C: myarray(n+m*k+l*j*k)
  - A packed array of [L:R] is normalized as [abs(L-R):0]
    - bit[5:2] a a[5]=1 a[4]=0 a[3]=1 a[2]=0 ←→ C a=10
    - bit[2:5] a a[5]=1 a[4]=0 a[3]=1 a[2]=0 ←→ C a=5

Passing a String

- **SystemVerilog strings are dynamically allocated**
  - Don’t change pointer on C side

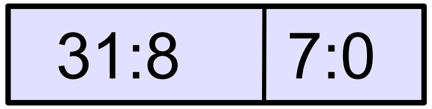
```
program automatic test;
  import "DPI-C" function void display(input string s);
  import "DPI-C" function void replace(inout string s);
  initial begin
       string s = “Forty-two";
       display(s);
       replace(s);
       $display("SV: '%s'", s);
   end
```

```
C: display 'Forty-two'
C: replace 'Life'
SV: 'Life'
```

```
void display(const char* s) {
  io_printf("C: %s '%s'\n", __FUNCTION__, s);
}

void replace(char** s) {
  strcpy(s, "Life");  // Copy of shorter string OK
  io_printf("C: %s '%s'\n", __FUNCTION__, *s);
}
```

SystemVerilog Storage: Under the Hood



- **All values are stored in 32-bit words**
  - A SystemVerilog byte wastes ¾ of space
  - So a C char array does NOT map to a SystemVerilog unpacked array of bytes
  - Don’t try to use SystemVerilog string as a null value will terminate the string
- **Make a packed array of bytes or int array**

```
import “DPI-C” task t(input int int_array[100]);
```

```
void display(const int ia[10]) {
  char* ba;  // Byte array
  ba = (char*) ia;
  ...
}
```

Agenda

Open Array

- Open arrays allow generic code to handle different size arrays. The size of the dimension (packed or unpacked) is unspecified: **[]**

```
import "DPI-C" function void foo(input int i [][]);
/* 2-dimensional unsized unpacked array of int */
int a_10x5 [11:20][6:2];
int a_64x8 [64:1][1:8];
foo(a_10x5);
foo(a_64x8);
```

Not a dynamic array

- Open arrays limited to a single packed dimension
- C code accesses an open array and its elements through system query and access functions
- Open arrays only work with imported methods, not exported methods

Open Array querying functions

```
/* h= handle to open array, d=dimension */
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);
```

Note:

1. If the dimension is 0, then the query refers to the packed part (which is one-dimensional) of an array. Dimensions > 0 refer to the unpacked part of an array.
2. int a[15:0] -> **svIncrement**(a,1)=1
   int a[0:15] -> **svIncrement**(a,1)=-1

Access functions

```
/* A pointer to the actual representation of the whole array of any type */
/* NULL if not in C layout */
void *svGetArrayPtr(const svOpenArrayHandle);

/* Total size in bytes or 0 if not in C layout */
/* Return a pointer to an element of the array or NULL if index outside the
range or null pointer */
int svSizeOfArray(const svOpenArrayHandle);

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

/* specialized versions for 1-, 2- and 3-dimensional arrays: */
void *svGetArrElemPtr1(const svOpenArrayHandle, int indx1);
void *svGetArrElemPtr2(const svOpenArrayHandle, int indx1, int indx2);
void *svGetArrElemPtr3(const svOpenArrayHandle, int indx1, int indx2, int indx3)
```

Simple Open Array

```
program automatic test;
   int a[3];
   import "DPI-C" function void disp(inout int h[]);
   initial begin
      foreach(a[i]) a[i]=i;
      foreach(a[i]) $display("SV: a[%0d]=%0d",i,a[i]);
      disp(a);
      foreach(a[i]) $display("SV after DPI: a[%0d]=%0d",i,a[i]);
   end
endprogram
```

```
void disp(svOpenArrayHandle h) {
  int *a, i, size;
  size = svSize(h, 1);
  a = (int*)svGetArrayPtr(h);
  for(i=0;i<size;i++) {
    io_printf("C: a[%d]=%d\n",i,a[i]);
    a[i] = size-i;
  }
}
```

```
SV: a[0]=0
SV: a[1]=1
SV: a[2]=2

C: a[0]=0
C: a[1]=1
C: a[2]=2

SV after DPI: a[0]=3
SV after DPI: a[1]=2
SV after DPI: a[2]=1
```

An Open Array with Multiple Dimensions

```
program automatic test;
  int a[6:4][2:3];
  import "DPI-C" function void disp2(inout int h[][]);
  initial begin
    foreach(a[i,j]) a[i][j]= i+j;
    foreach(a[i,j])
      $display("[%0d,%0d]=%0d",i,j,a[i][j]);
    disp2(a);
    foreach(a[i,j])
      $display("[%0d,%0d]=%0d",i,j,a[i][j]);
  end
endprogram
```

```
void disp2(svOpenArrayHandle h) {
  int *a, // Pointer so you can modify array
      i, j;
  for(i=svLow(h,1); i<=svHigh(h,1); i++)
    for(j=svLow(h,2); j<=svHigh(h,2); j++) {
      a = (int*)svGetArrElemPtr2(h,i,j);
      io_printf("C: a[%d][%d]=%d\n",i,j,*a);
      *a = i*j;
  }
}
```

```
SV: a[6][2]=8
SV: a[6][3]=9
SV: a[5][2]=7
SV: a[5][3]=8
SV: a[4][2]=6
SV: a[4][3]=7

C: a[4][2]=6
C: a[4][3]=7
C: a[5][2]=7
C: a[5][3]=8
C: a[6][2]=8
C: a[6][3]=9

SV: a[6][2]=12
SV: a[6][3]=18
SV: a[5][2]=10
SV: a[5][3]=15
SV: a[4][2]=8
SV: a[4][3]=12
...
```

An Open Array of Bit Elements

```
program automatic test;
  import "DPI-C" function void disp3(input bit[2:0] a[]);
  bit[2:0] a[8];
  initial begin
    foreach(a[i]) a[i] = 15+i;
    foreach(a[i]) $display("SV: a[%0d]=%0d",i, a[i]);
    disp3(a);
  end
endprogram
```

```
void disp3(const svOpenArrayHandle a) {
  svBitVecVal c;
  int low = svLow(a,1);
  int high = svHigh(a,1), i;
  for(i=low; i<=high; i++) {
    svGetBitArrElemVecVal(&c,a,i);
    io_printf("C: a[%d]=%d\n",i,c);
  }
}
```

```
SV: a[0]=7
SV: a[1]=0
SV: a[2]=1
SV: a[3]=2
SV: a[4]=3
SV: a[5]=4
SV: a[6]=5
SV: a[7]=6
C: a[0]=7
C: a[1]=0
C: a[2]=1
C: a[3]=2
C: a[4]=3
C: a[5]=4
C: a[6]=5
C: a[7]=6
```

An Open Array of Single Bits

```
program automatic test;
  import "DPI-C" function void disp4(bit a[]);
  bit a[5:2];  // 4 elements, one-bit each
  initial begin
    foreach(a[i]) a[i] = i;
    disp4(a);
  end
endprogram
```

```
void disp4(svOpenArrayHandle a) {
  svBit c;
  int low = svLow(a,1);
  int high = svHigh(a,1), i;
  io_printf("C: Incr=%d\n",svIncrement(a,1));
  for(i=low;i<=high;i++) {
    c = svGetBitArrElem(a,i);
    io_printf("C: a[%d]=%d\n",i,c);
  }
}
```

```
C: Incr=1
C: a[2]=0
C: a[3]=1
C: a[4]=0
C: a[5]=1
```

Canonical Open Array

```
program automatic test;
   import "DPI-C" function void disp5(inout bit [2:0] a[]);
   bit[2:0] a[8];
   initial begin
      for(int i=0;i<8;i++) a[i]= 15+i; // Wrap around 8
      disp5(a);
   end
endprogram
```

```
void disp5(svOpenArrayHandle a) {
   svBitVecVal c;
   int low = svLow(a,1);
   int high = svHigh(a,1), i;
   for(i=low;i<=high;i++) {
      svGetBitArrElem1VecVal(&c,a,i);
      io_printf("a[%d]=%d\n",i,c);
   }
   for(i=low;i<=high;i++) {
      c=i;
      svPutBitArrElem1VecVal(a,&c,i);
   }
}
```

```
a[0]=7
a[1]=0
a[2]=1
a[3]=2
a[4]=3
a[5]=4
a[6]=5
a[7]=6
```

Agenda

Quick Example : Export SV Task to C

- SystemVerilog call C methods, which can call back into SystemVerilog
- Every thread must have its own stack so it can be called concurrently. Use **`context`** in import declaration

```
program automatic test;
  import "DPI-C" context task c_test(input int addr);
  initial c_test(1000);
  export "DPI-C" task apb_write;

  task apb_write(input int addr, data)
    ... @(posedge ready); ...
  endtask

  initial
    another_thread();
endprogram
```

```
#include <svdpi.h>
extern void apb_write(int, int);

void c_test(int addr) {
   ...
  apb_write(addr, data);
  ...
}
```

Declaration of Exported Functions and Tasks

- **An exported task or function can be declared anywhere a native SV function or task can be declared**

- **An exported task or function declaration has only the routine name. There are <u>no</u> arguments or return type.**
  - *This is a very common coding mistake*

```
export "DPI-C" function my_func; // No return type,
export "DPI-C" task my_task;     // args, or even ()
```

- **Map SystemVerilog method name if it conflicts with existing C name**

```
// Map your SystemVerilog function "fread" to "sv_fread" in C
export "DPI-C" sv_fread = function fread;
```

Context Tasks and Functions

- **Context imported tasks and functions**
  - A DPI imported method might need to access or modify simulator data structures using PLI or VPI calls or call back into SystemVerilog with an exported task or function.
  - Extra info is saved when calling a context method, slowing down the call compared to calling a non-context method
- **By default imported tasks and functions are non-context**
  - Shall not access any data objects from SystemVerilog other than its actual arguments. Only the actual arguments can be affected (read or written) by its call
  - Not a barrier for simulator optimizations.
  - The effects of calling PLI or VPI functions or SystemVerilog tasks or functions are unpredictable and can crash

Exported Functions and Tasks Declaration

- **Export functions/tasks are always context**
- **If an imported task calls an exported task, the import must be context**
- **Class member functions/tasks cannot be exported, but all other SystemVerilog functions/tasks can be exported.**
  - Work-around shown in a later slide
- **Multiple export declarations are allowed with the same identifier, explicit or implicit, as long as they are in different scopes and have the equivalent type signature**
  - Multiple export declarations with the same *c_identifier* in the same scope are forbidden.
  - SystemVerilog tasks do not have return value types. The return value of an exported task is an *int* value that indicates if a disable is active or not on the current execution thread.
- **It is not legal to call an exported task from an imported <u>function</u>. Tasks must be called from a <u>task</u>.**

Context Tasks and Functions

- **There is an implicit scope for context tasks and functions**
  - The scope of the import declaration statement
  - Only tasks or functions defined and exported from the same scope as the import can be called directly.
  - To call any other exported SystemVerilog tasks or functions, the imported task or function must modify its current scope
- **The following functions allow you to retrieve and manipulate the current operational scope.**
  - You can only call these from a context imported method

```
svScope svGetScope();
svScope svSetScope(const svScope scope);
const char* svGetNameFromScope(const svScope);
svScope svGetScopeFromName(const char* scopeName);
```

SystemVerilog Code with Two Contexts

```
module top;
   import "DPI-C" context function void c_display();
   export "DPI-C" task sv_display;
   m1 m1_inst();
   initial #1 c_display();
   task sv_display();
     $display("SV: top");
   endtask
endmodule

module m1;
   import "DPI-C" context function void c_display();
   export "DPI-C" task sv_display;
   initial c_display();
   task sv_display();
      $display("SV: m1");
   endtask
endmodule
```

```
C: c_display
SV: m1

C: c_display
SV: top
```

```
extern int sv_display();

void c_display() {
  io_printf("\nC: c_display\n");
  sv_display();
}
```

Setting Context Explicitly

```
module top;
   import "DPI-C" context function void c_display();
   export "DPI-C" task sv_display;
   m1 m1_inst();
   initial #1 c_display();
   task sv_display();     // Never used!
     $display("SV: top");
   endtask
endmodule

module m1;
   import "DPI-C" context function void c_display();
   import "DPI-C" context function void mygetscope();
   export "DPI-C" task sv_display;
   initial begin
      mygetscope();
      c_display();
   end
   task sv_display();
     $display("SV: m1");
   endtask
endmodule
```

```
C: c_display
SV: m1

C: c_display
SV: m1
```

```
extern int sv_display();
svScope myscope;

void mygetscope() {
  myscope = svGetScope();
}

void c_display() {
  svSetScope(myscope);
  io_printf("\nC: c_display\n");
  sv_display();
}
```

Calling an Exported Task from Context

```
module m1;
   task delay(output int t);
      #10; t = $stime;
   endtask

   export "DPI-C" task delay;
   import "DPI-C" context task dodelay(output int t);
   int myt;
   initial begin
     dodelay(myt);
   end
endmodule
```

```
extern int delay(int *t);
int dodelay(int *t) {
   delay(t);
   io_printf("t is %d\n",*t);
}
```

Note: blocking DPI task can only be called from context import DPI task

Calling Task in a Dynamic Object

```
program automatic test;
  class Dyn;
    function void stuff(int i);
      $display("In %m, i=%0d", i);
    endfunction
  endclass

  Dyn dq[$];

  import "DPI-C" context function void c_fcn(int i);

  initial begin
    Dyn d_obj;
    d_obj = new();         // Construct obj
    dq.push_back(d_obj); // Put in queue
    c_fcn(dq.size()-1);  // Call C with index
  end

  export "DPI-C" function sv_static;
  function void sv_static(int idx);
    dq[idx].stuff(idx);
  endtask
endprogram : test
```

```
extern void sv_static(int i);

void c_fcn(int i) {
  io_printf("c_stuff(%d)\n",i);
  sv_static(i);   // Call SV
}
```

Pure C Functions

- **Declare C functions as `pure` for more VCS optimizations**
  - Calls to pure functions can be removed by VCS or replaced with the values previously computed for the same values of the input arguments.
- **A function is `pure` when its result depend solely on the values of its input arguments**
  - Only non-void functions with no output or inout arguments can be specified as pure.
- **A pure function is assumed not to directly or indirectly (i.e., by calling other functions) perform the following:**
  - Perform any file operations
  - Read or write anything in the broadest possible meaning, including input/output, environment variables, objects from the operating system or from the program or other processes, shared memory, sockets, etc.
  - Access any persistent data, like global or static variables

Agenda

How to Compile and Debug with DPI

- **Compilation with debug of SystemVerilog code**

```
> vcs -sverilog test.sv test.c -debug
```

- **Enable the C/C++ source code debug with –g option**

```
> vcs -sverilog case1.sv case1.cpp -debug_all -CFLAGS -g
```

- **Include svdpi.h in C code for DPI structures**
- **Include vc_hdrs.h in C code**
  - Compile-time type checking of arguments
    - Does your import statement match the C code?
- **Include veriuser.h in C code for io_printf() definition**

Compilation & Linking C++ Code

1. Use `extern "C"` for all definitions in headers; protect it with `#ifdef __cplusplus`
2. SystemVerilog classes cannot be passed as arguments
3. Provide wrapper access functions for class members
4. Use a C++ compiler; vcs will automatically invoke a C++ compiler for extensions .cpp, .cxx and .cc

```
// SystemVerilog code
import "DPI-C" function void f();

...
endprogram
```

```
// C++ code
#include <svdpi.h>

#ifdef _cplusplus
  extern "C" void f()
#elsif
  void f()
#endif
  {
    ...
  }
```

Shared Object Files

- **How can I switch between different C tests without recompiling?**
  - Compile the C code into shared object files

```
> gcc -shared -fPIC test.c  -o test/libtest.so  # test 0
> gcc -shared -fPIC test1.c -o test1/libtest.so # test 1
> gcc -shared -fPIC test2.c -o test2/libtest.so # test 2
```

  - Build VCS with a shared object file

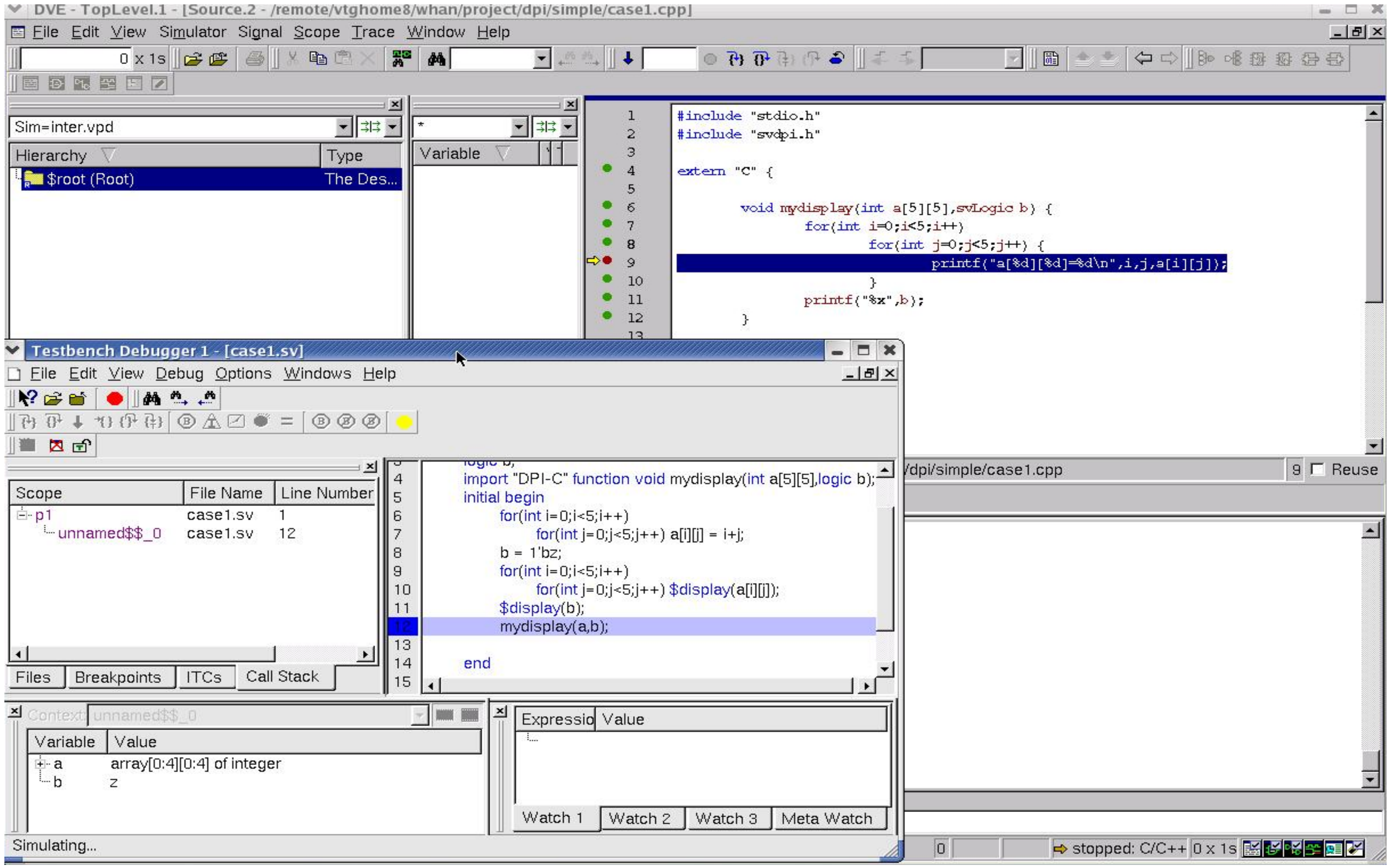
```
> vcs -sverilog model.v test/libtest.so
```

  - Point to shared object file and run the simulation

```
> setenv LD_LIBRARY_PATH test
> simv   # Run with test1.so
> setenv LD_LIBRARY_PATH test
> simv   # Run with test1.so
> setenv LD_LIBRARY_PATH test
> simv   # Run with test1.so
```

Debug with DVE

- **DVE supports SV design/testbench and C/C++ integrated debug**


DVE - TopLevel.1 - [Source.2 - /remote/vtghome8/whan/project/dpi/simple/case1.cpp]
File Edit View Simulator Signal Scope Trace Window Help
Sim=inter.vpd
Hierarchy
Type
$root (Root)
The Des...
Variable
Testbench Debugger 1 - [case1.sv]
File Edit View Debug Options Windows Help
/dpi/simple/case1.cpp
Reuse
stopped: C/C++
0 x 1s
Files
Breakpoints
ITCs
Call Stack
Watch 1
Watch 2
Watch 3
Meta Watch
Simulating...

```
#include "stdio.h"
#include "svdpi.h"

extern "C" {

      void mydisplay(int a[5][5],svLogic b) {
            for(int i=0;i<5;i++)
                  for(int j=0;j<5;j++) {
                        printf("a[%d][%d]=%d\n",i,j,a[i][j]);
                  }
            printf("%x",b);
      }
```

```
      logic b;
      import "DPI-C" function void mydisplay(int a[5][5],logic b);
      initial begin
            for(int i=0;i<5;i++)
                  for(int j=0;j<5;j++) a[i][j] = i+j;
            b = 1'bz;
            for(int i=0;i<5;i++)
                  for(int j=0;j<5;j++) $display(a[i][j]);
            $display(b);
            mydisplay(a,b);

      end
```

Scope	File Name	Line Number
p1	case1.sv	1
unnamed$$_0	case1.sv	12

Context: unnamed$$_0

Variable	Value
a	array[0:4][0:4] of integer
b	z

Expressio	Value

Debugging Runtime Issues

- **gdb or ddd**
  - Attach gdb to running simv at T=0 “gdb –p <pid>”
  - Set breakpoints and watchpoints
- **Valgrind (www.valgrind.org) for Linux**
  - Invalid memory reads/writes
  - Memory leaks
- **Purify**
  - Memory leak / corruption detection

gdb introduction

- `list`: list the source code
- `break` *`method`*: set a breakpoint on a C method
- `break` *`file:line#`* : set breakpoint in file at line number
- `info break`: display current breakpoints
- `run` *`[args]`*: start program execution with command line args
- `c`: continue execution
- `whatis` *`var`*: display the data type of the variable
- `print` *`*var`*: display the structure referenced by var
- `set output-radix 16`: set output radix to hex

DPI Header Files & Examples

- **DPI header files**

  `$VCS_HOME/include/svdpi.h`

- **VCS generated prototypes for C routines**

  `vc_hdrs.h`

- **SystemVerilog examples:**

  `$VCS_HOME/doc/examples/sv/dpi/`

  Directory contains following two subdirectories:

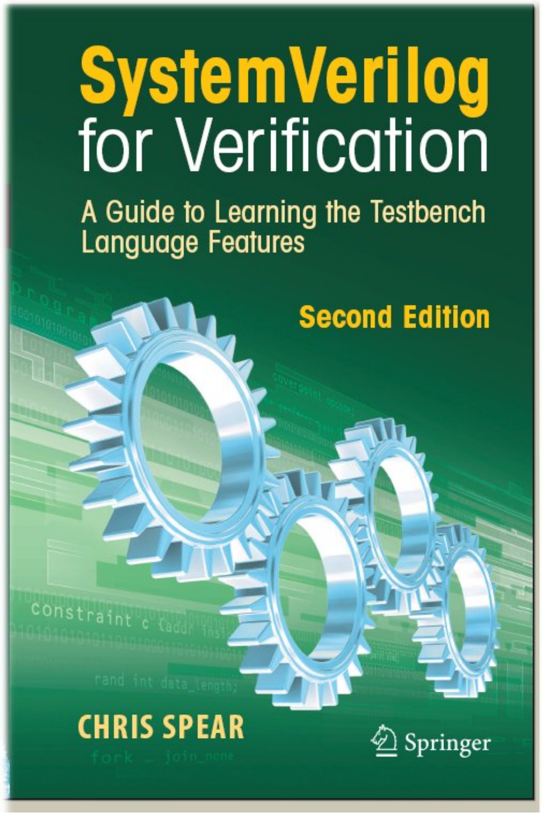
  `export_fun` – DPI export function for SV

  `import_fun` – DPI import function for SV

For more information

SystemVerilog for Verification

- **Based on IEEE P1800-2009 Standard**
  - New chapter on DPI with over 50 examples
  - Verification-specific language features
  - Constrained random stimulus generation
  - Functional coverage
  - Great intro to VMM
- **Available June 2008**



Appendix

- Memory management
- TF routines that should not called in DPI
- VCS and the LRM
- Difference between DPI and DPI-C

Memory management

- **The memory spaces owned and allocated by the foreign code and SystemVerilog code are disjoint**
  - SystemVerilog code can only use memory declared in its own arrays. It can not use blocks of memory from C
  - C code must not free the memory allocated by SystemVerilog code
  - SystemVerilog code must not free the memory allocated by the C code
- **SystemVerilog strings are dynamically allocated**
  - You can change the contents (same length or shorter), but don’t change the pointers

Calling tf routines from DPI code

tf routines that can be called from DPI context:

io_printf
mc_scan_plusargs
tf_gettime

tf routines that should not be called from DPI context:

tf_asynchon	tf_spname
tf_asynchoff	tf_ispname
tf_itestpvc_flag	tf_expr_eval
tf_testpvc_flag	tf_exprinfo
tf_imovepvc_flag	tf_nodeinfo
tf_copypvc_flag	tf_propagatep
tf_icopypvc_flag	tf_evaluatep
tf_movepvc_flag	tf_strgetp
tf_imovepvc_flag	tf_getp
tf_getpchange	tf_getscalarp
tf_synchronize	tf_getlongp
tf_rosynchronize	tf_getcstringp
tf_controller	tf_getrealp
tf_setlongdelay	tf_putp_direct
tf_setrealdelay	tf_putp
tf_clearalldelays	tf_putlongp
tf_typep	tf_strlongdelputp
tf_sizep	tf_strrealdelputp
tf_mipname	tf_setworkarea
tf_mipid	tf_getworkarea
tf_imipid	tf_mdanodeinfo

VCS and the LRM

- **In VCS 2006.06**
  - SystemVerilog structures are only supported in import DPI as inout/output argument until 2006.06-SP1
- **VCS 2008.03 will support a majority of the DPI**
  - See release notes for more info
  - Note that the SystemVerilog LRM has changed significantly between 2005 and 2009

Appendix: Difference between DPI and DPI-C

- **"DPI" is defined as the keyword for direct programming interface in SystemVerilog LRM 3.1a**
- **"DPI" and "DPI-C" are defined in SystemVerilog LRM IEEE-P1800-2005**
- **VCS(2006.06, 2006.06-SP1 and 2008.03) implements both of them**
    - Stick with "DPI-C" for portable code

```
import "DPI-C" task c_test(input int addr);

program automatic top;
  initial c_test(1000);
endprogram
```

Appendix: Difference between DPI and DPI-C

- **Bit vector or logic vector related system functions**
  - DPI: xxxxxVec32
    - svPutBitArrElemVec32
  - DPI-C: xxxxxVecVal
    - svPutBitArrElemVecVal

Appendix: Difference between DPI and DPI-C

- **Canonical representations**
  - DPI
    - svBitVec32
    - svLogicVec32
      - c are control bits
      - d are value bits
  - DPI-C
    - svBitVecVal
    - svLogicVecVal
      - aval are value bits
      - bval are control bits

svdpi.h

```
/* (a chunk of) packed bit array */
typedef unsigned int svBitVec32;

/* (a chunk of) packed logic array */
typedef struct {
         unsigned int c;
         unsigned int d;
} svLogicVec32;
```

```
/* (a chunk of) packed bit array */
typedef uint32_t svBitVecVal;

/* (a chunk of) packed logic array */
typedef struct vpi_vecval {
         uint32_t aval;
         uint32_t bval;
} s_vpi_vecval, *p_vpi_vecval;
typedef s_vpi_vecval svLogicVecVal;
```