Chapter 12
Interfacing with C/C++

In Verilog, you can communicate with C routines using the Programming Language Interface. With the three generations of the PLI: TF (Task / Function), ACC (Access), and VPI (Verification Procedural Interface), you can create delay calculators, connect and synchronize multiple simulators, and add debug tools such as waveform displays. However the PLI's greatest strength is also its greatest weakness. If you just want to connect a simple C routine using the PLI, you need to write dozens of lines of code, and understand many different concepts such as synchronizing with multiple simulation phases, call frames, and instance pointers. Additionally, the PLI adds overhead to your simulation as it copies data between the Verilog and C domains, in order to protect Verilog data structures from corruption.

SystemVerilog introduces the Direct Programming Interface (DPI), an easier way to interface with C, C++, or any other foreign language. Once you declare or "import" the C routine with the `import` statement, you can call it as if it were any SystemVerilog routine. Additionally, your C code can call SystemVerilog routines. With the DPI you can connect C code that reads stimulus, contains a reference model, or just extends SystemVerilog with new functionality. Currently SystemVerilog only supports an interface to the C language. C++ code has to be wrapped to look like C.

If you have a SystemC model that does not consume time, and that you want to connect to SystemVerilog, you can use the DPI. SystemC models with time-consuming methods are best connected with the utilities built into your favorite simulator.

The first half of this chapter is data-centric and shows how you can pass different data types between SystemVerilog and C. The second half is control centric, showing how you can pass control back and forth between SystemVerilog and C. While the actual C code is trivial, with the factorial function, the Fibonacci series, and counters, they are easy to understand so you can quickly substitute your own code.

C. Spear and G. Tumbush, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*, DOI 10.1007/978-1-4614-0715-7_12,

12.1 Passing Simple Values

The first few examples in this chapter show you how to pass integral values between SystemVerilog and C, and the mechanics of how to declare routines and their arguments on both sides. Later sections show how to pass arrays and structures.

12.1.1 Passing Integer and Real Values

The most basic data type that you can pass between SystemVerilog and C is an `int`, the 2-state, 32-bit type. Sample 12.1 shows the SystemVerilog code that calls a C factorial routine, shown in Sample 12.2.

Sample 12.1 SystemVerilog code calling C factorial routine

```
import "DPI-C" function int factorial(input int i);

program automatic test;
  initial begin
    for (int i=1; i<=10; i++)
      $display("%0d! = %0d", i, factorial(i));
  end
endprogram
```

The `import` statement declares that a SystemVerilog routine `factorial` is implemented in a foreign language such as C. The modifier `"DPI-C"` specifies that this is a Direct Programing Interface routine, and the rest of the statement describes the routine arguments.

Sample 12.1 passes 32-bit signed values using the SystemVerilog `int` data type that maps directly to the C `int` type. The SystemVerilog int is always 32 bits, whereas the width of an int in C is operating system dependent. The C function in Sample 12.2 takes an integer as an input and so the DPI passes the argument by value.

Sample 12.2 C factorial function

```
int factorial(int i) {
  if (i<=1) return 1;
  else      return i*factorial(i-1);
}
```

12.1.2 The Import Declaration

The `import` declaration defines the prototype of the C task or function, but using SystemVerilog types. A C function with a return value is mapped to a SystemVerilog

function. A void C function can be mapped to a SystemVerilog task or void function. If the name of the imported C function conflicts with a SystemVerilog name, you can import the function with a new name. In Sample 12.3, the C function `expect` is mapped to the SystemVerilog name `fexpect`, as the name `expect` is a reserved keyword in SystemVerilog. The name `expect` becomes a global symbol, used to link with the C code, whereas `fexpect` is a local SystemVerilog symbol. In the second half of the example, the C function `stat` is given a new name in SystemVerilog, `file_exists`. SystemVerilog does not support overloading a routine, for example by importing `expect` once with a `real` argument and once with an `int`.

Sample 12.3 Changing the name of an imported function

```
program automatic test;

  // C function has same name as reserved keyword, change it
  import "DPI-C" \expect = function int fexpect();
  ...
    if (actual != fexpect()) $display("ERROR");
  ...

  // Change name of C function "stat" to "file_exists"
  import "DPI-C" stat = function int file_exists
             (input string fname, output int buff[1000]);
  initial begin
    int buff[1000];
    $display("file_exists(\"none.such\") = %0d",
             file_exists("none.such", buff));
  end
endprogram
```

You can import routines anywhere in your SystemVerilog code where you can declare a routine including inside programs, modules, interfaces, packages, and `$unit`, the compilation-unit space. The imported routine will be local to the declaration space in which it is declared. If you need to call an imported routine in several locations in your code, put the `import` statement in a package which you import where it is needed. Any changes to the `import` statements are localized to the package.

12.1.3 Argument Directions

Imported C routines can have zero or more arguments. By default the argument direction is `input` (data goes from SystemVerilog to C), but can also be `output` and `inout`. The direction `ref` is not supported. A function can return a simple value such as an integer or real number, or have no return value if you make it `void`. Sample 12.4 shows how to specify argument directions.

Sample 12.4 Argument directions

```
import "DPI-C" function int addmul (input int a, b,
                                    output int sum);
import "DPI-C" function void stop_model();
```

You can reduce the chances of bugs in your C code by declaring any input arguments as `const` as shown in Sample 12.5 so the C compiler will give an error for any write to an input.

Sample 12.5 C factorial routine with const argument

```
int factorial(const int i) {
  if (i<=1) return 1;
  else      return i*factorial(i-1);
}
```

12.1.4 Argument Types

Each variable that is passed through the DPI has two matching definitions: one for the SystemVerilog side, and one for the C side. It is your responsibility to use compatible types. The SystemVerilog simulator cannot compare the types as it is unable to read the C code. (The VCS compiler produces `vc_hdrs.h` and Questa creates `incl.h` with the C header for any routine that you have imported. You can use this file as a guide to matching the types.)

Table 12.1 shows the data type mapping between SystemVerilog and the inputs and outputs of C routines. The C structures are defined in the include file `svdpi.h`. Arrays mapping is discussed in Section 12.4 and 12.5, and structures are discussed in Section 12.6.

Table 12.1 Data types mapping between SystemVerilog and C

SystemVerilog	*C (input)*	*C (output)*
byte	char	char*
shortint	short int	short int*
int	int	int*
longint	long long int	long int*
shortreal	float	float*
real	double	double*
string	const char*	char**
string [N]	const char**	char**
bit	svBit or unsigned char	svBit* or unsigned char*
logic, reg	svLogic or unsigned char	svLogic* or unsigned char*
bit[N:0]	const svBitVecVal*	svBitVecVal*
reg[N:0] logic[N:0]	const svLogicVecVal*	svLogicVecVal*
unsized array[]	const svOpenArrayHandle	svOpenArrayHandle
chandle	const void*	void*

Note that some mappings are not exact. For example, a `bit` in SystemVerilog maps to `svBit` in C, which ultimately maps to `unsigned char` in the `svdpi.h` include file. As a result, you could write illegal values into the upper bits.

The LRM limits imported function results "small values", which include: `void`, `byte`, `shortint`, `int`, `longint`, `real`, `shortreal`, `chandle`, and `string`, plus single bit values of type `bit` and `logic`. A function cannot return a vector such as `bit [6:0]` as this would require returning a pointer to a `svBitVecVal` structure.

12.1.5 Importing a Math Library Routine

Sample 12.6 shows how you can call many functions in the C math library directly, without a C wrapper, thereby reducing the amount of code that you need to write. The Verilog `real` type maps to a C `double`.

Sample 12.6 Importing a C math function

```
import "DPI-C" function real fabs(input real r);
...
initial $display("fabs(0)=%f", fabs(-1.0));
```

12.2 Connecting to a Simple C Routine

Your C code might contain a simulation model, such as a processor, that is instantiated side by side with Verilog models. Or your code could be a reference model that is compared to a Verilog model at the transaction or cycle level. Many examples in this chapter show an 7-bit counter written in C or C++. Though very simple, the counter has the same parts as a complex model, with inputs, outputs, storage of internal values between calls, and the need to support multiple instances. The counter is 7 bits to show what happens when a hardware type does not match a C type.

12.2.1 A Counter with Static Storage

Sample 12.7 is the C code for an 7-bit counter. The count is stored in a static variable, as you might do if you wrote the model before thinking about simulation.

Sample 12.7 Counter routine using a static variable

```
#include <svdpi.h>

void counter7(svBitVecVal *o,
              const svBitVecVal *i,
              const svBit reset,
              const svBit load) {
  static unsigned char count = 0; // Static count storage

  if (reset)      count = 0;    // Reset
  else if (load) count = *i;   // Load value
  else           count++;      // Count
  count &= 0x7f;               // Mask off upper bit

  *o = count;
}
```

The `reset` and `load` signals are 2-state single bit signals, and so they are passed as `svBit` which reduces to `unsigned char`. Your code could declare the value either way, but play it safe by using the SystemVerilog DPI types. The input `i` is 2-state, and 7 bits wide, and is passed as `svBitVecVal`. Notice that it is passed as a `const` pointer, which means the underlying value can change, but you cannot change the value of the pointer, such as making it point to another value. Likewise, the `reset` and `load` inputs are also marked as `const`. In this example, the 7-bit counter value is stored in a `char`, so you have to mask off the upper bit.

The file `svdpi.h` contains the definitions for SystemVerilog DPI structures and methods. The C code examples in the rest of this chapter leave off the `#include` statements, unless they are important to the discussion.

Sample 12.8 shows a SystemVerilog program that imports and calls the C function for the 7-bit counter.

Sample 12.8 Testbench for an 7-bit counter with static storage

```
import "DPI-C" function void counter7(output bit [6:0] out,
                                      input bit [6:0] in,
                                      input bit reset, load);
program automatic counter;
  bit [6:0] out, in;
  bit       reset, load;

  initial begin
    $monitor("SV: out=%3d, in=%3d, reset=%0d, load=%0d\n",
             out, in, reset, load);
    reset = 0;                          // Default values
    load = 0;
    in = 126;
    counter7(out, in, reset, load);     // Apply default values

    #10 reset = 1;
    counter7(out, in, reset, load);     // Apply reset

    #10 reset = 0;
    load = 1;
    counter7(out, in, reset, load);     // Load in=126

    #10 load = 0;
    counter7(out, in, reset, load);     // Count
  end
endprogram
```

12.2.2 The Chandle Data Type

The `chandle` data type allows you to store a C or C++ pointer in your SystemVerilog code. A `chandle` variable is wide enough to hold a pointer on the machine where the code was compiled, i.e. 32- or 64-bits. The counter in Sample 12.7 works well as long as it is the only one in the design. You could wrap the `counter7` calls from Sample 12.8 in a module, and instantiate multiple copies in a design. However, since the counter value is stored in a C static, every instance shares a single value. If you need more than one instance of a module that calls C code, the C code needs to store its variables somewhere other than in static variables. A better way is to allocate storage, and pass a handle to it, along with the input and output signal values. Sample 12.9 shows a counter that stores the 7-bit count in the structure `c7`. This is overkill for a simple counter, but if you are creating a model for a larger device, you can build from this example.

Sample 12.9 Counter routine using instance storage

```
#include <svdpi.h>
#include <malloc.h>
#include <veriuser.h>

typedef struct {  // Structure to hold counter value
  unsigned char cnt;
} c7;

// Construct a counter structure
void* counter7_new() {
  c7* c = (c7*) malloc(sizeof(c7)); // Cast malloc value to c7
  c->cnt = 0;
  return c;
}

// Process the counter inputs
void counter7(c7 *inst,
              svBitVecVal* count,
              const svBitVecVal* i,
              const svBit reset,
              const svBit load) {

  if (reset)     inst->cnt = 0;  // Reset
  else if (load) inst->cnt = *i; // Load value
  else           inst->cnt++;    // Count
  inst->cnt &= 0x7f;             // Mask upper bit

  *count = inst->cnt;            // Write to output
  io_printf("C: count=%d, i=%d, reset=%d, load=%d\n",
            *count, *i, reset, load);
}
```

The routine `counter7_new` constructs the counter instance. This returns a `chandle` that must be passed into future calls to `counter7`. The counter value is stored in a struct of type `c7`. The function `counter7_new` calls `malloc` to allocate the struct, and casts the result into a local pointer `c`.

The C code uses the PLI task `io_printf` to display debug messages. The routine is helpful when you are debugging C and SystemVerilog code side-by-side as it writes to the same outputs, including log files, as `$display`, including the simulator's log file. The routine is defined in `veriuser.h`.

The testbench for this counter in Sample 12.10 differs from the static one in several ways. First, the counter must be constructed before it can be used. Next, the counter is called on a clock edge, rather than calling it in-line with the stimulus. For simplicity, the counter is invoked when the clock goes high, and stimulus is applied when the clock goes low, to avoid any race conditions.

Sample 12.10 Testbench for an 7-bit counter with per-instance storage

```
import "DPI-C" function chandle counter7_new();
import "DPI-C" function void counter7
       (input chandle inst,
        output bit [6:0] out,
        input  bit [6:0] in,
        input  bit  reset, load);

// Test two instances of the counter
program automatic test;

  bit [6:0] o1, o2, i1, i2;
  bit       reset, load, clk;
  chandle   inst1, inst2;     // Points to storage in C

  initial begin
    inst1 = counter7_new();
    inst2 = counter7_new();
    fork
      forever #10 clk = ~clk;
      forever @(posedge clk) begin
        counter7(inst1, o1, i1, reset, load);
        counter7(inst2, o2, i2, reset, load);
      end
    join_none

    reset = 0;                  // Initialize signals
    load = 0;
    i1 = 120;
    i2 = 10;

    @(negedge clk) load = 1;    // Load inputs
    @(negedge clk) load = 0;    // Count
    @(negedge clk) $finish;
  end
endprogram
```

12.2.3 Representation of Packed Values

The string `"DPI-C"`[1] specifies that you are using the canonical representation of packed values. This representation stores a SystemVerilog variable as a C array of one or more elements. A 2-state variable is stored using the type `svBitVecVal`. A 2-state array is stored with multiple elements of this type.

[1] Early versions of the LRM used “DPI” but this is now obsolete and should not be used.

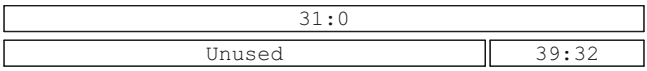

Fig. 12.1 Storage of a 40-bit 2-state variable

For performance reasons. the SystemVerilog simulator may not mask the upper bits after calling a DPI routine, and so the SystemVerilog variable could be corrupted. Make sure your C code treats these values properly.

If you need to convert between bits and words, use the macro `SV_PACKED_DATA_NELEMS`. For example, to convert 40 bits to two 32-bit words (as seen in Fig. 12.1), `use SV_PACKED_DATA_NELEMS(40).`

12.2.4 4-State Values

Each 4-state bit in SystemVerilog is stored in the simulator using two bits known as `aval` and `bval`, as shown in Table 12.2

Table 12.2 4-state bit encoding

4-state value	*bval*	*aval*
0	0	0
1	0	1
Z	1	0
X	1	1

A single bit 4-state variable, such as `logic f`, is stored in an unsigned byte, with the `aval` bit stored in the least significant bit, and the `bval` in the next higher bit. So the value `1'b0` is seen as 0x0 in C, `1'b1` is 0x1, `1'bz` is 0x2, and `1'bx` is 0x3.

A 4-state vector such as `logic [31:0] lword` is stored using pairs of 32 bits, `svLogicVecVal`, which contains the `aval` and `bval` bits as shown in Figure 12.2. The 32-bit variable `lword` is stored in a single `svLogicVecVal`. Variables wider than 32-bits are stored in multiple `svLogicVecVal` elements, with the first element contains the 32 least significant bits, the next element contains the next 32 bits, up to the most significant bits. A 40-bit `logic` variable is stored as one `svLogicVecVal` for the least significant 32 bits, and a second for the upper 8 bits (Fig. 12.2). The unused 24-bits in this upper value are undetermined, and you are responsible for masking or extending the sign bit, as needed. The `svLogicVecVal` type is equivalent to `s_vpi_vecval`, which is used to represent 4-state types such as `logic` in the VPI.

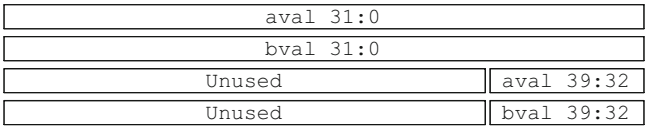

Fig. 12.2 Storage of a 40-bit 4-state variable

Beware of arguments declared without bit subscripts or those declared with a single bit. An argument declared as `input logic` a is stored in an `unsigned char`. The argument `input logic [0:0] b` is `svLogicVecVal`, even though it contains only a single bit.

Sample 12.11 shows the import statements for a 4-state counter. The only difference from Sample 12.10 is that the `bit` types are now `logic`.

Sample 12.11 Testbench for counter that checks for Z or X values

```
import "DPI-C" function chandle counter7_new();
import "DPI-C" function void counter7
      (input chandle inst,
       output logic [6:0] out,
       input  logic [6:0] in,
       input  logic  reset, load);
```

The counter previously shown in Sample 12.9 assumes all the inputs are 2-state. Sample 12.12 extends this code to check for Z and X values on `reset`, `load`, and `i`. The actual count is still kept as a 2-state value.

Sample 12.12 Counter routine that checks for Z and X values

```
// 4-state replacement for counter7 from Sample 12-9
void counter7(c7 *inst,
              svLogicVecVal* count,
              const svLogicVecVal* i,
              const svLogic reset,
              const svLogic load) {

  if (reset & 0x2) {  // Check just the bval bit of scalar
    io_printf("Error: Z or X detected on reset\n\n");
    return;
  }
  if (load & 0x2) {  // Check just the bval bit of scalar
    io_printf("Error: Z or X detected on load\n\n");
    return;
  }
  if (i->bval) {     // Check just the bval bits of 7-bit vector
    io_printf("Error: Z or X detected on i\n\n");
    return;
  }

  if (reset)     inst->cnt = 0;       // Reset
  else if (load) inst->cnt = i->aval; // Load value
  else           inst->cnt++;         // Count
  inst->cnt &= 0x7f;                  // Mask upper bit

  count->aval = inst->cnt;            // Write to output
  count->bval = 0;
}
```

If you want to force the simulation to terminate cleanly because of a condition found in an imported routine, you can call the VPI routine `vpi_control(vpiFinish, 0)`. This routine and constant are defined in the include file `vpi_user.h`. The value `vpiFinish` tells the simulator to execute a `$finish` after your imported routine returns.

12.2.5 Converting from 2-State to 4-State

If you have a DPI application that works with 2-state types and you want to convert it to work with 4-state types, follow the following guidelines.

On the SystemVerilog side, change the import declaration from using 2-state types such as `bit` and `int` to 4-state types such as `logic` and `integer`. Make sure you are using 4-state variables in the function call.

On the C side, switch the argument declarations from `svBitVecVal` to `svLogicVecVal`. Any reference to the arguments will have to use the `.aval` suffix to

correctly access the data. When you read from a 4-state variable, check the `bval` bits to see if there are any Z or X values. When you write to a 4-state variable, clear the `bval` bits unless you need to write Z or X values.

12.3 Connecting to C++

You can use the DPI to connect routines written in C or C++ to SystemVerilog. There are several ways your C++ code can communicate using the DPI, depending on your model's level of abstraction.

12.3.1 The Counter in C++

Sample 12.13 shows a C++ class for the 7-bit counter, with 2-state inputs. It connects to the SystemVerilog testbench in Sample 12.10 and the C++ wrapper code in Sample 12.14.

Sample 12.13 Counter class

```
class Counter7 {
public:
  Counter7();
  void counter7_signal(svBitVecVal* count,
                       const svBitVecVal* i,
                       const svBit reset,
                       const svBit load);
private:
  unsigned char cnt;
};

Counter7::Counter7() {
  cnt = 0;                    // Initialize counter
}

void Counter7::counter7_signal(svBitVecVal* count,
                               const svBitVecVal* i,
                               const svBit reset,
                               const svBit load) {
  if (reset)      cnt = 0;     // Reset
  else if (load)  cnt = *i;    // Load
  else            cnt++;       // Count
  cnt &= 0x7F;                 // Mask upper bit
  *count = cnt;
}
```

12.3.2 Static Methods

The DPI can only call a C or C++ function that is known at link time. As a result, your SystemVerilog code cannot call a C++ routine in an object as the object does not exist when the linker runs.

So what if you need to call a method in a C++ class? The solution, as shown in Sample 12.14, is to create a function with a fixed address, that then can communicate with the C++ dynamic objects and methods. The first routine, `counter7_new`, constructs an object for the counter and returns a handle to the object. The second static routine, `counter7`, calls the C++ method that performs the counter logic, using the object handle.

Sample 12.14 Static methods and linkage

```
extern "C" void* counter7_new()
{
  return new Counter7;
}

// Call a counter instance, passing the signal values
extern "C" void counter7(void* inst,
                         svBitVecVal* count,
                         const svBitVecVal* i,
                         const svBit reset,
                         const svBit load)
{
  Counter7 *c7 = (Counter7 *) inst;
  c7->counter7_signal(count, i, reset, load);
}
```

The `extern "C"` code tells the C++ compiler that the external information sent to the linker should use C calling conventions and not perform name mangling. You can put this before each routine that is called by SystemVerilog, or put `extern "C" { ... }` around a set of methods.

From the testbench point of view, the C++ counter looks the same as the counter that stored the value in per-instance storage, shown in Sample 12.9, so you can use the same testbench, Sample 12.10, for both.

12.3.3 Communicating with a Transaction Level C++ Model

The previous C / C++ code examples were low-level models that communicated with the SystemVerilog at the signal level. This is not efficient; for example the counter is called every clock cycle, even if the data or control inputs have not changed. When you create models for complex devices such as processors and networking devices, communicate with them at the transaction level for faster simulations.

The C++ counter model in Sample 12.15 has a transaction-level interface, communicating with methods instead of signals and a clock.

Sample 12.15 C++ counter communicating with methods

```
class Counter7 {
public:
  Counter7();
  void count();
  void load(const svBitVecVal* i);
  void reset();
  int get();
private:
  unsigned char cnt;
};

Counter7::Counter7() {          // Initialize counter
  cnt = 0;
}

void Counter7::count() {        // Increment counter
  cnt = cnt + 1;
  cnt &= 0x7F;                  // Mask upper bit
}

void Counter7::load(const svBitVecVal* i) {
  cnt = *i;
  cnt &= 0x7F;                  // Mask upper bit
}

void Counter7::reset() {
  cnt = 0;
}

// Get the counter value in a pointer to a svBitVecVal
int Counter7::get() {
  return cnt;
}
```

The dynamic C++ methods such as `reset`, `load`, and `count` are wrapped in static methods that use the object handle, passed from SystemVerilog, as shown in Sample 12.16.

Sample 12.16 Static wrapper for C++ transaction level counter

```
#ifdef __cplusplus
extern "C" {
#endif

void* counter7_new() {
  return new Counter7;
}

void counter7_count(void* inst){
  Counter7 *c7 = (Counter7 *) inst;
  c7->count();
}

void counter7_load(void* inst, const svBitVecVal* i) {
  Counter7 *c7 = (Counter7 *) inst;
  c7->load(i);
}

void counter7_reset(void* inst) {
  Counter7 *c7 = (Counter7 *) inst;
  c7->reset();
}

int counter7_get(void* inst) {
  Counter7 *c7 = (Counter7 *) inst;
  return c7->get();
}

#ifdef __cplusplus
}
#endif
```

The OOP interface for the transaction level counter is carried up to the testbench. Sample 12.17 has the SystemVerilog import statements and a class to wrap the C++ object. This allows you to hide the C++ handle inside the class.

Note that the `counter7_get()` function returns an `int` (32-bit, signed) rather than `bit [6:0]`, as the latter would require returning a pointer to a `svBitVecVal`, as shown in Table 12.1. An imported function can not return a pointer. It can only return a value of type `void`, `byte`, `shortint`, `int`, `longint`, `real`, `shortreal`, `chandle`, and `string`, plus single bit values of type `bit` and `logic`.

Sample 12.17 Testbench for C++ model using methods

```
import "DPI-C" function chandle counter7_new();
import "DPI-C" function void counter7_count(input chandle inst);
import "DPI-C" function void counter7_load(input chandle inst,
                                           input bit [6:0] i);
import "DPI-C" function void counter7_reset(input chandle inst);
import "DPI-C" function int counter7_get(input chandle inst);

// Wrap the static C static wrapper functions with a
// SystemVerilog class to hide the C++ instance handle
class Counter7;
   chandle inst;

   function new();
      inst = counter7_new();
   endfunction

   function void count();
      counter7_count(inst);
   endfunction

   function void load(input bit [6:0] val);
      counter7_load(inst, val);
   endfunction

   function void reset();
      counter7_reset(inst);
   endfunction

   function bit [6:0] get();
      return counter7_get(inst);
   endfunction
endclass : Counter7

program automatic test;
  Counter7 c1;

  initial begin
    c1 = new;

    c1.reset();
    $display("SV: Post reset: counter1=%0d", c1.get());

    c1.load(126);
    if (c1.get() == 126)
      $display("Successful load");
    else
      $display("Error: load, expect 126, got %0d", c1.get());
```

```
    c1.count();  // count = 127
    if (c1.get() == 127)
      $display("Successful count");
    else
      $display("Error: load, expect 127, got %0d", c1.get());

    c1.count();  // count = 0
    if (c1.get() == 0)
      $display("Successful rollover");
    else
      $display("Error: rollover, exp 127, got %0d", c1.get());
   end

endprogram
```

12.4 Simple Array Sharing

So far you have seen examples of passing scalar and vectors between SystemVerilog and C. A typical C model might read an array of values, perform some computation, and return another array with the results.

12.4.1 Single Dimension Arrays - 2-State

Sample 12.18 shows a routine that computes the first 20 values in the Fibonacci series. It is called by the SystemVerilog code in Sample 12.19.

Sample 12.18 C routine to compute Fibonacci series

```
void fib(svBitVecVal data[20]) {
  int i;
  data[0] = 1;
  data[1] = 1;
  for (i=2; i<20; i++)
    data[i] = data[i-1] + data[i-2];
}
```

Note that in C, you could have alternatively declared the argument as a pointer, `*data` or an array, `data[20]`. In this example, they are interchangeable.

Sample 12.19 Testbench for Fibonacci routine

```
import "DPI-C" function void fib(output bit [31:0] data[20]);

program automatic test;
  bit [31:0] data[20];

  initial begin
    fib(data);
    foreach (data[i]) $display(i,,data[i]);
   end
endprogram
```

Notice that the array of Fibonacci values is allocated and stored in SystemVerilog, even though it is calculated in C. There is no way to allocate an array in C and reference it in SystemVerilog.

12.4.2 Single Dimension Arrays - 4-State

Sample 12.20 shows the Fibonacci C routine for a 4-state array with the testbench in Sample 12.21.

Sample 12.20 C routine to compute Fibonacci series with 4-state array

```
void fib(svLogicVecVal data[20]) {
  int i;
  data[0].aval = 1;    // Write to both aval
  data[0].bval = 0;    //      and bval
  data[1].aval = 1;
  data[1].bval = 0;
  for (i=2; i<20; i++) {
    data[i].aval = data[i-1].aval + data[i-2].aval;
    data[i].bval = 0;  // Don't forget to clear bval
  }
}
```

Sample 12.21 Testbench for Fibonacci routine with 4-state array

```
import "DPI-C" function void fib(output logic [31:0] data[20]);

program automatic test;
  logic [31:0] data[20];

  initial begin
    fib(data);
    foreach (data[i]) $display(i,,data[i]);
   end
endprogram
```

Section 12.2.5 describes how to convert a 2-state application to 4-state.

12.5 Open arrays

When sharing arrays between SystemVerilog and C, you have two options. For the fastest simulations, you can reverse-engineer the layout of the elements in SystemVerilog, and write your C code to use this mapping. This approach is fragile, meaning that you will have to rewrite and debug your C code if any of the array sizes change. A more robust approach is to use "open arrays", and their associated SystemVerilog routines to manipulate them. These allow you to write generic C routines that can operate on any size array.

12.5.1 Basic Open Array

Sample 12.22 and 12.23 show how to pass a simple array between SystemVerilog and C with open arrays. Use the empty square brackets `[]` in the SystemVerilog `import` statement to specify that you are passing an open array.

Sample 12.22 Testbench code calling a C routine with an open array

```
import "DPI-C" function void fib_oa(output bit [31:0] data[]);

program automatic test;
  localparam SIZE = 20;
  bit [31:0] data[SIZE], r;

  initial begin
    fib_oa(data, SIZE);
    foreach (data[i])
      $display(i,,data[i]);
   end
endprogram
```

Your C code references the open array with a handle of type `svOpenArrayHandle`. This points to a structure with information about the array such as the declared word range. You can locate the actual array elements with calls such as `svGetArrayPtr`. Note that `svSize()` is an open array query method, as described in the next section.

Sample 12.23 C code using a basic open array

```
void  fib_oa(const svOpenArrayHandle data_oa) {
  int i, *data;
  data = (int *) svGetArrayPtr(data_oa);
  data[0] = 1;
  data[1] = 1;
  for (i=2; i<=svSize(data_oa, 1); i++)
    data[i] = data[i-1] + data[i-2];
}
```

12.5.2 Open Array Methods

There are many DPI methods to access their contents and ranges, as defined in `svdpi.h`. These only work with open array handles declared as `svOpenArrayHandle`, not with pointers such as `svBitVecVal` or `svLogicVecVal`. The methods in Table 12.3 give you information about the size of an open array.

Table 12.3 Open array query functions

function	*Description*
`int svLeft(h, d)`	Left bound for dimension d
`int svRight(h, d)`	Right bound for dimension d
`int svLow(h, d)`	Low bound for dimension d
`int svHigh(h, d)`	High bound for dimension d
`int svIncrement(h, d)`	If left >= right, 1, else −1
`int svSize(h, d)`	Number of elements in dimension d: svHigh−svLow+1
`int svDimensions(h)`	Number of dimensions in open array
`int svSizeOfArray(h)`	Total size of array in bytes

In Table 12.3, the variable `h` is a `svOpenArrayHandle` and `d` is an `int`. The dimensions are numbered starting with `d=1`.

The functions in Table 12.4 return the locations of the C storage for the entire array or a single element.

Table 12.4 Open array locator functions

Function	*Returns pointer to*:
`void *svGetArrayPtr(h)`	storage for the entire array
`void *svGetArrElemPtr(h, i1, ...)`	an element in the array
`void *svGetArrElemPtr1(h, i1)`	an element in a 1-D array
`void *svGetArrElemPtr2(h, i1, i2)`	an element in a 2-D array
`void *svGetArrElemPtr3(h, i1, i2, i3)`	an element in a 3-D array

12.5.3 Passing Unsized Open Arrays

Sample 12.24 calls C code with a 2-dimensional array. The C code uses the `svLow` and `svHigh` methods to find the array ranges, which, in this example, don't follow the usual 0..size-1.

Sample 12.24 Testbench calling C code with multi-dimensional open array

```
import "DPI-C" function void mydisplay(inout int h[][]);

program automatic test;
  int a[6:1][8:3];      // Note word ranges are high:low
  initial begin
    foreach (a[i,j]) a[i][j] = i+j;
    mydisplay(a);
    foreach (a[i,j])
      $display("V: a[%0d][%0d] = %0d", i, j, a[i][j]);
  end
endprogram
```

This calls the C code in Sample 12.25 that reads the array using the open array methods. The routine `svLow(handle, dimension)` returns the lowest index number for the specified dimension. So `svLow(h,1)` returns 1 for the array declared with the range [6:1]. Likewise, `svHigh(h, 1)` returns 6. You should use `svLow` and `svHigh` with C `for` loops.

The methods `svLeft` and `svRight` return the left and right index from the array declaration, 6 and 1 respectively for the range [6:1]. At the center of Sample 12.25, the call `svGetArrElemPtr2` returns a pointer to an element in a two dimensional array.

Sample 12.25 C code with multi-dimensional open array

```
void mydisplay(const svOpenArrayHandle h) {
  int i, j;
  int lo1 = svLow(h, 1);
  int hi1 = svHigh(h, 1);
  int lo2 = svLow(h, 2);
  int hi2 = svHigh(h, 2);
  for (i=lo1; i<=hi1; i++) {
    for (j=lo2; j<=hi2; j++) {
      int *a = (int*) svGetArrElemPtr2(h, i, j);
      io_printf("C: a[%d][%d] = %d\n", i, j, *a);
      *a = i * j;
    }
  }
}
```

12.5.4 Packed Open Arrays in DPI

An open array in the DPI is treated as having a single packed dimension and one or more unpacked dimensions. You can pass an array with multiple packed dimensions, as long as they pack into an element that is the same size as a single element

of the formal argument. For example, if you have the formal argument `bit[63:0] b64[]` in the `import` statement, you could pass in the actual argument `bit [1:0][0:3][6:-1] bpack [9:1]`. Sample 12.26 shows the SystemVerilog code with packed open arrays.

Sample 12.26 Testbench for packed open arrays

```
import "DPI-C" function void view_pack(input bit [63:0] b64[]);

program automatic test;
  bit [1:0][0:3][6:-1] bpack[9:1];

  initial begin
    foreach(bpack[i]) bpack[i] = i;
    bpack[2] = 64'h12345678_90abcdef;

    $display("SV: bpack[2]=%h", bpack[2]);  // 64 bits
    $display("SV: bpack[2][0]=%h", bpack[2][0]); // 32 bits
    $display("SV: bpack[2][0][0]=%h", bpack[2][0][0]); // 8 bits

    view_pack(bpack);
  end
endprogram : test
```

Sample 12.27 C code using packed open arrays

```
void view_pack(const svOpenArrayHandle h) {
  int i;

  for (i=svLow(h,1); i<svHigh(h,1); i++)
    io_printf("C: b64[%d]=%llx\n",
              i, *(long long int *)svGetArrElemPtr1(h, i));
}
```

Notice that the C code in Sample 12.27 prints a 64-bit value using `%llx`, and casts the result from `svGetArrayElemPtr1` to `long long int`.

12.6 Sharing Composite Types

By this point you may wonder how to pass objects between SystemVerilog and C. The layout of class properties does not match between the two languages, so you cannot share objects directly. Instead, you must create similar structures on each

side, plus pack and unpack methods to convert between the two formats. Once you have all this in place, you can share composite types.

12.6.1 Passing Structures Between SystemVerilog and C

The following example shares a simple structure for a pixel made of three bytes packed into a word. Sample 12.28 shows the C structure. Notice that C treats a `char` as signed variable, which can give you unexpected results, so the structure marks the char as unsigned. The bytes are in reverse order from the SystemVerilog because this code was written for a Intel x86 processor that is little-endian, which means that the least significant byte is stored at a lower address than the most significant. A Sun SPARC is big endian, so the bytes are stored in the same order as in SystemVerilog: `r, g, b`.

Sample 12.28 C code to share a structure

```
typedef struct {
  unsigned char b, g, r;  // x86 litle-endian
//unsigned char r, g, b;  // SPARC format
} *p_rgb;

void invert(p_rgb rgb) {
  rgb->r = ~rgb->r;     // Invert the color values
  rgb->g = ~rgb->g;
  rgb->b = ~rgb->b;
  io_printf("C:  Invert rgb=%02x,%02x,%02x\n",
            rgb->r, rgb->g, rgb->b);
}
```

The SystemVerilog testbench in Sample 12.29 has a packed struct that holds a single pixel, and class to encapsulate the pixel operations. The `RGB_T` struct is packed so SystemVerilog will store the bytes in consecutive locations. Without the `packed` modifier, each 8-bit value would be stored in a separate word.

Sample 12.29 Testbench for sharing structure

```
typedef struct packed { bit [ 7:0] r, g, b; } RGB_T;
import "DPI-C" function void invert(inout RGB_T pstruct);

program automatic test;

class RGB;
  rand bit [ 7:0] r, g, b;
  function void display(input string prefix="");
    $display("%sRGB=%x,%x,%x", prefix, r, g, b);
  endfunction : display

  // Pack the class properties into a struct
  function RGB_T pack();
    pack.r = r; pack.g = g; pack.b = b;
  endfunction : pack

  // Unpack a struct into the class properties
  function void unpack(input RGB_T pstruct);
    r = pstruct.r; g = pstruct.g; b = pstruct.b;
  endfunction : unpack
endclass : RGB

  initial begin
    RGB pixel;
    RGB_T pstruct;

    pixel = new;
    repeat (5) begin
      `SV_RAND_CHECK(pixel.randomize()); // Create random pixel
      pixel.display("\nSV: before "); // Print it
      pstruct = pixel.pack();         // Convert to a struct
      invert(pstruct);                // Call C to invert bits
      pixel.unpack(pstruct);          // Unpack struct to class
      pixel.display("SV: after  ");   // Print it
    end
  end
endprogram
```

12.6.2 Passing Strings Between SystemVerilog and C

Using the DPI, you can pass strings from C back to SystemVerilog. You might need to pass a string for the symbolic value of a structure, or get a string representing the internal state of your C code for debug.

The easiest way to pass a string from C to SystemVerilog is for your C function to return a pointer to a static string as shown in Sample 12.30. The string must be

declared as `static` in C, and not as a local string. Non-static variables are stored on the stack and are reclaimed when the function returns.

Sample 12.30 Returning a string from C

```
char *print(p_rgb rgb) {
  static char s[12];
  sprintf(s, "%02x,%02x,%02x", rgb->r, rgb->g, rgb->b);
  return s;
}
```

A danger with static storage is that multiple concurrent calls could end up sharing storage. For example, a SystemVerilog `$display` statement that is printing several pixels might call the above `print` routine multiple times. Depending on how the SystemVerilog compiler orders these calls, later calls to `print()` could overwrite results from earlier calls, unless the SystemVerilog compiler makes a copy of the string. Note that a call to an imported routine can never be interrupted by the SystemVerilog scheduler. Sample 12.31 stores the strings in a heap to support concurrent calls.

Sample 12.31 Returning a string from a heap in C

```
#define PRINT_SIZE 12
#define MAX_CALLS 16
#define HEAP_SIZE PRINT_SIZE * MAX_CALLS

char *print(p_rgb rgb) {
  static char print_heap[HEAP_SIZE + PRINT_SIZE];
  char *s;
  static int heap_idx = 0;
  int nchars;

  s = &print_heap[heap_idx];
  nchars = sprintf(s, "%02x,%02x,%02x",
                   rgb->r, rgb->g, rgb->b);
  heap_idx += nchars + 1;          // Don't forget null!
  if (heap_idx > HEAP_SIZE)
    heap_idx = 0;
  return s;
}
```

12.7 Pure and Context Imported Methods

Imported methods are classified as `pure`, `context`, or generic. A `pure` function calculates its output strictly based on its inputs, with no outside interactions. Specifically, a `pure` function does not access any global or static variables, perform any file operations, or interact with anything outside the function such as the operating

system, processes, shared memory, sockets, etc. The SystemVerilog compiler may optimize away calls to a `pure` function if the result is not needed, or replace the call with the results from a previous call with the same arguments. The `factorial` function in Sample 12.5, and the `sin` function in 12.6 are both `pure` functions as their result is only based on their inputs. Sample 12.32 shows how to import a pure function.

Sample 12.32 Importing a pure function

```
import "DPI-C" pure function int factorial(input int i);
import "DPI-C" pure function real sin(input real in);
```

An imported routine may need to know the context of where it is called so it can call a PLI TF, ACC, or VPI methods, or a SystemVerilog task that has been exported. Use the `context` attribute for these methods as shown in Sample 12.33.

Sample 12.33 Imported context tasks

```
import "DPI-C" context task call_sv(bit [31:0] data);
```

An imported routine may use global storage, so it is not `pure`, but might not have any PLI references, so it does not need the overhead of a `context` routine. Sutherland (2004) uses the term "generic" for these methods as the SystemVerilog LRM does not have a specific name. By default, an imported routine is generic, as are many of the examples in this chapter.

There is overhead invoking a `context` imported routine as the simulator needs to record the calling context, so only declare a routine as `context` if needed. On the other hand, if a generic imported routine calls an exported task or a PLI routine that accesses SystemVerilog data objects, the simulator could crash.

A context-aware PLI routine is one that needs to know where it was called from so that it can access information relative to that location.

12.8 Communicating from C to SystemVerilog

The examples so far have shown you how to call C code from your SystemVerilog models. The DPI also allows you to call SystemVerilog routines from C code. The SystemVerilog routine can be a simple task to record the result from an operation in C, or a time-consuming task representing part of a hardware model.

12.8.1 A simple Exported Function

Sample 12.34 shows a module that imports a context function, and exports a System-Verilog function.

Sample 12.34 Exporting a SystemVerilog function

```
module block;
  import "DPI-C" context function void c_display();
  export "DPI-C" function sv_display;  // No type or args

  initial c_display();

  function void sv_display();
    $display("SV: in sv_display");
  endfunction
endmodule : block
```

The `export` declaration in Sample 12.34 looks naked because the LRM forbids putting a return value declaration or any arguments. You can't even give the usual empty parentheses. This information in the `export` declaration would duplicate the information in the function declaration at the end of the module and could thus become out of sync if you ever changed the function.

Sample 12.35 shows the C code that calls the exported function.

Sample 12.35 Calling an exported SystemVerilog function from C

```
extern void sv_display();

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

This example prints the line from the C code, followed by the `$display` output from the SystemVerilog, as shown in Sample 12.36.

Sample 12.36 Output from simple export

```
C:  in c_display
SV: in sv_display
```

12.8.2 C function Calling SystemVerilog Function

While the majority of your testbench should be in SystemVerilog, you may have legacy testbenches in C or other languages, or applications that you want to reuse. This section creates a SystemVerilog memory model that is stimulated by C code that reads transactions from an external file.

The first version of the memory model, shown in Sample 12.38 and 12.37, is coded with just functions, so everything runs in zero time. The C code in

Sample 12.37 opens the file, reads a command, and calls the exported function. Error checking has been removed for compactness.

Sample 12.37 C code to read simple command file and call exported function

```
#include <svdpi.h>
#include <stdio.h>
extern void mem_build(int);

void read_file(char *fname){
  char cmd;
  FILE *file;

  file = fopen(fname, "r");
  while (!feof(file)) {
    cmd = fgetc(file);
    switch (cmd)
      {
      case 'M': {
        int hi;
        fscanf(file, "%d", &hi);
        mem_build(hi);
        break;
      }
    }
  }
  fclose(file);
}
```

The SystemVerilog code calls the C task `read_file` which opens a file. The only command in the file sets the memory size, so the C code calls an exported function.

Sample 12.38 SystemVerilog module for simple memory model

```
module memory;
  import "DPI-C" context function void read_file(string fname);
  export "DPI-C" function mem_build;  // No type or args

  initial
    read_file("mem.dat");

  int mem[];

  function void mem_build(input int size);
    mem = new[size];     // Allocate dynamic memory elements
  endfunction

endmodule : memory
```

Notice that in Sample 12.38, the `export` statement does not have any arguments as this information is already in the function declaration.

The command file is trivial, with one command to construct a memory with 100 elements as shown in Sample 12.39.

Sample 12.39 Command file for simple memory model

```
M 100
```

12.8.3 C Task Calling SystemVerilog Task

A real memory model has operations such as read and write that consume time, and thus must be modeled with tasks.

Sample 12.40 shows the SystemVerilog code for the second version of the memory model. It has several improvements compared to Sample 12.38. There are two new tasks, `mem_read` and `mem_write`, which respectively take 20ns and 10ns to complete. The imported routine `read_file` is now a SystemVerilog task as it is calling other tasks that consume time. The `import` statement now specifies that `read_file` is a context task, as the simulator needs to create a separate stack when it is called.

Sample 12.40 SystemVerilog module for memory model with exported tasks

```
module memory;
  import "DPI-C" context task read_file(string fname);
  export "DPI-C" task mem_read;
  export "DPI-C" task mem_write;
  export "DPI-C" function mem_build;

  initial read_file("mem.dat");

  int mem[];

  function void mem_build(input int size);
    mem = new[size];
  endfunction

  task mem_read(input int addr, output int data);
    #20 data = mem[addr];
  endtask

  task mem_write(input int addr, input int data);
    #10 mem[addr] = data;
  endtask
endmodule : memory
```

The C code in Sample 12.41 primarily expands the case statement that decodes commands and calls the exported tasks, which are declared as `extern int` according to the LRM.[2]

Sample 12.41 C code to read command file and call exported function

```
extern int mem_read(int, int*);
extern int mem_write(int, int);
extern void mem_build(int);

void read_file(const char *fname) {
  char cmd;
  FILE *file;

  file = fopen(fname, "r");
  while (!feof(file)) {
    cmd = fgetc(file);
    switch (cmd) {
      case 'M': {
        int hi;
        fscanf(file, "%d ", &hi);
        mem_build(hi);
        break;
      }

      case 'R': {
        int addr, data, exp;
        fscanf(file, "%d %d ", &addr, &exp);
        mem_read(addr, &data);
        if (data != exp)
          io_printf("C: Data=%d, exp=%d\n", data, exp);
        break;
      }

      case 'W': {
        int addr, data;
        fscanf(file, "%d %d ", &addr, &data);
        mem_write(addr, data);
        break;
      }
    }
  }
  fclose(file);
}
```

The command file in Sample 12.42 has new commands that write two locations, and then reads back one of them, and includes the expected value.

[2] VCS declared exported tasks as void functions in C.

Sample 12.42 Command file for simple memory model

```
M 100
W 12 34
W 99 8
R 12 34
```

12.8.4 Calling Methods in Objects

You can export SystemVerilog methods, except for those defined inside a class. This restriction is similar to the restriction of importing static C methods, as shown in Section 12.3.2 as objects do not exist when SystemVerilog elaborates your code. The solution is to pass a reference to the object between the SystemVerilog and C code. However, unlike a C pointer, a SystemVerilog handle cannot be passed through the DPI. You can instead have an array of handles, and pass the array index between the two languages.

The following examples build on the previous versions of the memory. The SystemVerilog code in Sample 12.44 has a class that encapsulates the memory. Now you can have multiple memories, each in a separate object. The command file in Sample 12.43 creates two memories, M0, and M1. Then it performs several writes to initialized locations in both memories, and lastly tries to read back the values. Notice that location 12 is used for both memories.

Sample 12.43 Command file for exported methods with OOP memories

```
M0 1000
M1 2000
W0 12 34
W1 12 88
W0 99 18
R1 22 44
R0 12 34
R1 12 88
```

The SystemVerilog code in Sample 12.44 constructs a new object for every `M` command in the file. The exported function `mem_build` calls the `Memory` constructor. It then stores the handle to the `Memory` object in a SystemVerilog queue, and returns the queue index, `idx`, to the C code as shown in Sample 12.45. The handles are stored in a queue so you can dynamically add new memories. The exported tasks `mem_read` and `mem_write` now have an additional argument, the index of the memory handle in the queue.

Sample 12.44 SystemVerilog module with memory model class

```
module memory;
  import "DPI-C" context task read_file(string fname);
  export "DPI-C" task mem_read;
  export "DPI-C" task mem_write;
  export "DPI-C" function mem_build;

  initial read_file("mem.dat");  // Call C code to read file


  class Memory;
    int mem[];

    function new(input int size);
      mem = new[size];
    endfunction

    task mem_read(input int addr, output int data);
      #20 data = mem[addr];
    endtask

    task mem_write(input int addr, input int data);
      #10 mem[addr] = data;
    endtask : mem_write
  endclass : Memory

  Memory memq[$];           // Queue of Memory objects

  // Construct a new memory instance & push on the queue
  function void mem_build(input int size);
    Memory m;
    m = new(size);
    memq.push_back(m);
  endfunction

  // idx is the index of the memory handle in memq
  task mem_read(input int idx, addr, output int data);
    memq[idx].mem_read(addr, data);
  endtask

  task mem_write(input int idx, addr, input int data);
    memq[idx].mem_write(addr, data);
  endtask

endmodule : memory
```

Sample 12.45 C code to call exported tasks with OOP memory

```
extern int mem_read(int, int, int*);
extern int mem_write(int, int, int);
extern void mem_build(int);

void read_file(char *fname) {
  char cmd;
  int idx;
  FILE *file;

  file = fopen(fname, "r");
  while (!feof(file)) {
    cmd = fgetc(file);
    fscanf(file, "%d ", &idx);
    switch (cmd)
      {
      case 'M': {
        int hi;
        fscanf(file, "%d ", &hi);
        mem_build(hi);
        break;
      }

      case 'R': {
        int addr, data, exp;
        fscanf(file, "%d %d ", &addr, &exp);
        mem_read(idx, addr, &data);
        if (data != exp)
          io_printf("C: Error Data=%d, exp=%d\n", data, exp);
        break;
      }

      case 'W': {
        int addr, data;
        fscanf(file, "%d %d ", &addr, &data);
        mem_write(idx, addr, data);
        break;
      }
    }
  }
  fclose(file);
}
```

12.8.5 The Meaning of Context

The context of an imported routine is the location where it was defined, such as `$unit`, module, program, or package scope, just like a normal SystemVerilog routine. If you import a routine in two different scopes, the corresponding C code executes in the context of where the `import` statement occurred. This is similar to defining a SystemVerilog `run()` task in each of two separate modules. Each task accesses variables in its own module, with no ambiguity.

Sample 12.46 shows that if you add a second module to Sample 12.34 that imports the same C code and exports its own function, the C routine will call different SystemVerilog methods, depending on the context of the import and export statements.

Sample 12.46 Second module for simple export example

```
module top;
  import "DPI-C" context function void c_display();
  export "DPI-C" function sv_display;

  block b1();
  initial c_display();

  function void sv_display();
    $display("SV: In %m");
  endfunction
endmodule : top


module block;
  import "DPI-C" context function void c_display();
  export "DPI-C" function sv_display;

  initial c_display();

  function void sv_display();
    $display("SV: In %m");
  endfunction
endmodule : block
```

The output in Sample 12.47 shows that one C routine calls two separate SystemVerilog methods, depending on where the C routine was called.

Sample 12.47 Output from simple example with two modules

```
C:  in c_display
SV: In top.b1.sv_display
C:  in c_display
SV: In top.sv_display
```

12.8.6 Setting the Scope for an Imported Routine

Just as your SystemVerilog code can call a routine in the local scope, an imported C routine can call a routine outside its default context. Use the routine `svGetScope` to get a handle to the current scope, and then use that handle in a call to `svGetScope` to make the C code think it is inside another context. Sample 12.48 shows the C code for two methods. The first, `save_my_scope( )`, saves the scope of where it was called from the SystemVerilog side. The second routine, `c_display()`, sets its scope to the saved one, prints a message, then calls your function, `sv_display()`.

Sample 12.48 C code getting and setting context

```
extern void sv_display();
svScope my_scope;

void save_my_scope() {
  my_scope = svGetScope();
}

void c_display() {
  // Print the current scope
  io_printf("\nC: c_display called from scope %s\n",
            svGetNameFromScope(svGetScope()));

  // Set a new scope
  svSetScope(my_scope);
  io_printf("C: calling %s.sv_display\n",
            svGetNameFromScope(svGetScope()));
  sv_display();
}
```

The C code calls `svGetNameFromScope()` that returns a string of the current scope. The scope is printed twice, once with the scope where the C code was first called from, and again with the scope that was previously saved. The routine `svGetScopeFromName()` takes a string with a SystemVerilog scope and returns a pointer to a `svScope` handle that can be used with `svSetScope()`.

In the SystemVerilog code in Sample 12.49, the first module, `block`, calls a C routine that saves the context. When the module `top` calls `c_display()`, the routine sets scope back to `block`, and so it calls the `sv_display()` routine in the `block` module, not the `top` module.

Sample 12.49 Modules calling methods that get and set context

```
module block;
  import "DPI-C" context function void c_display();
  import "DPI-C" context function void save_my_scope();
  export "DPI-C" function sv_display;

  function void sv_display();
    $display("SV: In %m");
  endfunction : sv_display

  initial begin
    save_my_scope();
    c_display();
  end

endmodule : block


module top;
  import "DPI-C" context function void c_display();
  export "DPI-C" function sv_display;

  function void sv_display();
    $display("SV: In %m");
  endfunction : sv_display

  block b1();

  initial #1 c_display();

endmodule : top
```

This produces the output shown in Sample 12.50.

Sample 12.50 Output from svSetScope code

```
C: c_display called from top.b1
C: Calling top.b1.sv_display
SV: In top.b1.sv_display

C: c_display called from top
C: Calling top.b1.sv_display
SV: In top.b1.sv_display
```

You could use this concept of scope to allow a C model to know where it was instantiated from, and differentiate each instance. For example, a memory model may be instantiated several times, and needs to allocate unique storage for every instance.

12.9 Connecting Other Languages

This chapter has shown the DPI working with C and C++. With a little work, you can connect other languages. The easiest way is to call the Verilog `$system()` task. If you need the return value from the command, use the Unix `system()` function and the `WEXITSTATUS` macro. The SystemVerilog code in Sample 12.51 calls a C wrapper for `system()`.

Sample 12.51 SystemVerilog code calling C wrapper for Perl

```
import "DPI-C" function int call_perl(string s);

program automatic perl_test;
  int ret_val;
  string script;

  initial begin
    $value$plusargs("script=%s", script);
    $display("SV: Running '%0s'", script);
    ret_val = call_perl(script);
    $display("SV: Perl script returned %0d", ret_val );
  end
endprogram : perl_test
```

Sample 12.52 is the C wrapper that calls system() and translates the return value.

Sample 12.52 C wrapper for Perl script

```
#include <svdpi.h>
#include <stdlib.h>
#include <wait.h>

int call_perl(const char* command) {
  int result = system(command);
  return WEXITSTATUS(result);
}
```

Sample 12.53 is a Perl script that prints a message and returns a value.

Sample 12.53 Perl script called from C and SystemVerilog

```
#!/usr/local/bin/perl
print "Perl: Hello world!\n" ;
exit (3)
```

Now you can run the Unix command in Sample 12.54 to run the simulation and call the `hello.pl` script.

Sample 12.54 VCS command line to run Perl script

```
> simv +script="perl hellp.pl"
```

12.10 Conclusion

The Direct Programing Interface allows you to call C routines as if they are just another SystemVerilog routine, passing SystemVerilog types directly into C. This has less overhead than the PLI, which builds argument lists, and always has to keep track of the calling context, not to mention the complexity of having up to four C routines for every system task.

Additionally, with the DPI, your C code can call SystemVerilog routines, allowing external applications to control simulation. With the PLI you would need trigger variables and more argument lists, and you have to worry about subtle bugs from multiple calls to time-consuming tasks.

The most difficult part of the DPI is mapping SystemVerilog types to C, especially if you have structures and classes that are shared between the two languages. If you can master this problem, you can connect almost any application to SystemVerilog.

12.11 Exercises

1. Create a C function, `shift_c`, that has two input arguments: a 32-bit unsigned input value `i` and an integer for the shift amount `n`. The input `i` is shifted `n` places. When `n` is positive, values are shifted left, when `n` is negative, shifted right, and when `n` is 0, no shift is performed. The function returns the shifted value. Create a SystemVerilog module that calls the C function and tests each feature. Provide the output.
2. Expand Exercise 1 to add a third argument to `shift_c`, a load flag `ld`. When `ld` is true, `i` is shifted by `n` places and then loaded into an internal 32-bit register. When `ld` is false, the register is shifted `n` places. The function returns the value of the register after these operations. Create a SystemVerilog module that calls the C function and tests each feature. Provide the output.
3. Expand Exercise 2 to create multiple instances of the `shift_c` function. Each instance in C needs a unique identifier, so use the address where the internal register is stored. Print this address along with the arguments when the function `shift_c` is called. Instantiate the function twice, and call each instance twice. Provide the output.

4. Expand the C code from Exercise 3 to display the total number of times the `shift_c` function has been called, even if the function is instantiated more than once.
5. Expand Exercise 4 to provide the ability to initialize the stored value at instantiation.
6. Expand Exercise 5 to encapsulate the `shift_c` function in a class.
7. For the code in Sample 12.24 and 12.25, what is returned by the following open array methods?

```
svLeft(h, 1);
svLeft(h, 2);
svRight(h, 1);
svRight(h, 2);
svSize(h, 1);
svSize(h, 2);
svDimensions(h);
svSizeOfArray(h);
```

8. Modify Exercise 1 so that instead of shifting the value in C, the function calls an exported SystemVerilog void function named `shift_sv` that does the shifting.
9. Expand Exercise 8 to call the SystemVerilog function `shift_sv` for two different SystemVerilog objects as demonstrated in Section 12.8.4 of the text. Assume the SystemVerilog function `shift_build` has been exported to the C code.
10. Expand Exercise 8 to:
    a. Create a SystemVerilog class `Shift` containing the function `shift_sv` that stores the result in a class-level variable, and a `shift_print` function that displays the stored result.
    b. Define and export SystemVerilog function `shift_build`.
    c. Support the creation of multiple `Shift` objects with the handles to these objects stored in a queue.
    d. Create a testbench that constructs multiple `Shift` objects. Demonstrate that each object holds a separate result after performing calculations.