The Verilog PLI Is Dead (maybe)
Long Live The SystemVerilog DPI!

Stuart Sutherland
Sutherland HDL, Inc.
stuart@sutherland-hdl.com

ABSTRACT

In old England, when one monarch died a successor immediately took the throne. Hence the chant, "The king is dead—long live the king!". The Verilog Programming Language Interface (PLI) appears to be undergoing a similar succession, with the advent of the new SystemVerilog Direct Programming Interface (DPI). Is the old Verilog PLI dead, and the SystemVerilog DPI the new king? This paper addresses the question of whether engineers should continue to use the Verilog PLI, or switch to the new SystemVerilog DPI. The paper will show that the DPI can simplify interfacing to the C language, and has capabilities that are not possible with the PLI. However, the Verilog PLI also has unique capabilities that cannot be done using the DPI.

Table of Contents

1.0 Introduction

The Verilog Programming Language Interface (PLI) provides a mechanism for Verilog code to call functions written in the C programming language. The venerable Verilog PLI is one of the reasons the Verilog language has been so successful for hardware design. Using the PLI, third party companies and end-users can extend the capabilities of commercial Verilog simulators. Virtually every serious design project using Verilog has utilized the Verilog PLI.

Over its nearly 20 year history, the Verilog PLI has seen two major generations, often referred to as "PLI 1.0" and "PLI 2.0" (officially called the PLI-VPI). Accellera has recently introduced the "SystemVerilog" extensions to the IEEE Verilog standard, which includes a new way for Verilog code to call C and C++ functions. This new interface is called the *"Direct Programming Interface"* or *"DPI"*. Some engineers feel that the advent of the SystemVerilog DPI marks the demise of the old Verilog PLI. That is, that the DPI replaces the complex Verilog PLI with a simple and straight forward C language interface. Does SystemVerilog make the Verilog PLI obsolete? Can we all chant *"The Verilog PLI is dead—long live the SystemVerilog DPI!"* ?

Before the celebration begins, however, there are some important questions to answer. Is the SystemVerilog DPI really better than the Verilog PLI, or are there good reasons to continue to use the PLI? Are there applications that cannot be done using the SystemVerilog DPI, that will necessitate keeping the Verilog PLI alive?

2.0 An overview of the DPI

The SystemVerilog Direct Programming Interface (DPI) allows Verilog code to call the names of C functions as if the function were a native Verilog task or function (a task or function defined in the Verilog language). This is done by importing the C function name into the Verilog language, using a simple **`import`** statement. The import statement defines that the function uses the DPI interface, and contains a prototype of the function name and arguments. For example:

```
import "DPI" function real sin(real in); // sine function in C math library
```

This import statement defines the function name `sin` for use in Verilog code. The data type of the function return is a `real` value (double precision) and the function has one input, which is also a `real` data type. Once this C function name has been imported into Verilog, it can be called the same way as a native Verilog language function. For example:

```
always @(posedge clock) begin
   slope <= sin(angle);  // call the "sin" function from C
end
```

The imported C function must be compiled and linked into the Verilog simulator. This process will vary with different simulators. With the Synopsys VCS simulator, the process is very straightforward; the C source file name, or a pre-compiled object file name, is listed along with Verilog source code files as part of the `vcs` compile command.

With the SystemVerilog DPI, the Verilog code is unaware that it is calling C code, and the C function is unaware that it is being called from Verilog.

DPI compared to PLI. The ability to import a C function and then directly call the function using the DPI is much simpler than the Verilog PLI. With the PLI, users must define a *user-defined system task* or *user-defined system function*. The user-defined system task/function name must begin with a dollar sign ($). For example, the sine function above might be represented in Verilog with `$sine`. This function is then associated with a user-supplied C function known as a *calltf routine*. The calltf routine can then invoke the `sin` function from the C math library. The calltf routine, and any functions it calls, must then be compiled and linked into the Verilog simulator. Once these steps have been performed, the `$sine` system function can then be called from within Verilog in the same way as a regular function. When simulation executes the call to `$sine`, it will invoke the C calltf function that can then call the C `sin` function.

```
always @(posedge clock) begin
   slope <= $sine(angle);  // call the sine PLI application
end
```

At first glance, it might seem that within the Verilog code, there is very little difference between using the PLI function and an imported DPI function. Indeed, within the Verilog code, there is no significant difference; in both cases, the Verilog code is calling a task or function. Where the difference becomes apparent—and it is a significant difference—is in what it takes to define the PLI user-defined system task or system function. Using the DPI, the Verilog code can directly call the `sin` function from the C math library, directly pass inputs to the function, and directly receive a return value from the C function. Using the PLI, several steps are required to create a user-defined system task that indirectly calls and passes values to the sin function. This indirect process is described in more detail in the following section.

3.0 Verilog PLI capabilities

In order to fully discuss the strengths and weaknesses of the SystemVerilog DPI, it is necessary to understand the capabilities of the Verilog PLI, and the C libraries that are part of the PLI standard.

The Verilog PLI is a simulation interface. It provides run-time access to the simulation data structure, and allows user-supplied C functions a way to access information within the simulation data structure. This user-supplied C function can both read and modify certain aspects of the data structure. The Verilog PLI does not work with Verilog source code. It only works with a simulation data structure. The PLI is not designed for use with tools other than simulation, such as synthesis compilers.

3.1 How the PLI calls C functions

The Verilog PLI provides a set of C language functions that allow programmers to access the data structure of a Verilog simulation. These C functions are bundled into three PLI libraries, referred to as the *TF library*, the *ACC library* and the *VPI library*. The PLI access to the simulation data structure is dynamic, in that it occurs while simulation is running. The access is also bidirectional. Through the PLI, a programmer can both read information from the simulator's data structure, and modify information in the data structure.

The Verilog PLI allows programmers to extend the Verilog language though the creation of *system tasks and system functions*. The names of these user-defined system tasks and functions

must begin with a dollar sign ($). A task in Verilog is analogous to a subroutine. When a task is called, the simulator's instruction flow branches to a subroutine. Upon completion of the task, the instruction flow returns back to the instruction following the task call. Verilog tasks do not return values, but can have input, output and inout (bidirectional) formal arguments. A function in Verilog is analogous to functions in most languages. When a function is called, it executes a set of instructions, and then returns a value back to the statement that called the function. Verilog tasks and functions can be defined as part of the Verilog language.

Using the Verilog PLI, a programmer must first define a system task function name, such as `$sine`. The programmer then creates a C function referred to as a *calltf routine*, which will be associated with the `$sine` task/function name. When simulation executes the statement with the `$sine` system function call, the simulator calls the calltf routine associated with `$sine`. This calltf routine is a layer between `$sine` and the `sin` function in the C math library. The arguments to $sine are not passed directly to the calltf routine. Instead, the calltf routine must call special PLI functions from a PLI library to read the input argument of `$sine`. The calltf routine can then call the `sin` function, passing the input to the function. The calltf routine will receive the return value from the `sin` function, and then call another special PLI function to write this return value back to the `$sine` function. A final step when using the Verilog PLI is to bind the user-defined system task/function name with the user-defined calltf routine. This step is different for each simulator, and can range from editing a special table file to editing and compiling a complex C language file.

The PLI libraries allow a calltf routine to do more than just work with the arguments of a system task or function. The libraries also allow a calltf application to search for objects in the simulation data structure, modify delays and logic values in the data structure, and synchronize to simulation activity and simulation time.

3.2 The evolution of the Verilog PLI

The Verilog PLI was originally developed in the mid to late 1980s as a proprietary interface to the Gateway Design Automation Verilog-XL simulator product (now owned by Cadence Design Systems). Gateway developed the PLI to satisfy two fundamental needs: First was to allow Verilog-XL users a mechanism to use the C language for tasks such as file I/O. Second was to provide a mechanism for ASIC foundries to analyze the usage of their standard cell libraries in order to calculate accurate propagation delays within each cell instance. From these two fundamental intents, the usage of the Verilog PLI has expanded in many ways. Common applications of the Verilog PLI in today's engineering environments include: commercial and proprietary waveform viewers, commercial and proprietary design debug utilities, RAM/ROM program loaders, scan chain vector loaders, custom file readers and writers, power analysis, circuit tracing, C model interfaces, co-simulation environments, parallel process distribution, and much, much more. The ways in which the PLI can be used to extend Verilog simulators is limited only by the imagination of programmers.

The following paragraphs introduce the several generations of the Verilog PLI.

1985 — the TF interface. The first generation of the Verilog PLI is referred to as the *Task/ Function interface*, or *TF* interface. The TF interface comprises a set of C language functions defined in a library file called *verisuer.h*. These C functions are typically referred to as the *TF routines*. The primary purpose of the TF routines is to allow task/function arguments to be passed

to C functions. For example, a user-defined system task could be created to load data into a RAM model. In context, the usage of this user-defined system task might be:

```
initial begin
  $loadram(chip.ram.core, "ram_data.pgm"); // load the ram model
  ...
end
```

In this example, the `$loadram` system task has two arguments: the instance name of a RAM model storage array within the Verilog hierarchy, and the name of a file with the data to be loaded into the RAM. When a Verilog simulator executes the `$loadram` system task, it will branch to a user-defined C function associated with `$loadram`. That C function can then call functions from the TF library to read the arguments of `$loadram`, and act accordingly.

An important capability offered by the TF routines is the ability to synchronize PLI activity with the simulator's event scheduler. Event synchronization is a key part of many types of PLI applications, such as co-simulation applications, parallel processing applications, and interfacing to bus-functional C models.

There is a critical limitation of the TF interface. The library of TF routines can only access the arguments of a user-defined system task or function. In the preceding example, the name of the RAM model storage element must be passed to the PLI application as an argument to the system task. The `$loadram` application cannot search for RAM models anywhere in the simulation data structure, and then search for the storage elements within those RAM models.

A drawback of the TF interface is the mechanism for defining the system task/function name, function return types, and what C function(s) are associated with the task/function name. This mechanism is not standardized, which means every simulator has a different PLI interface mechanism. Most mechanisms require a knowledge of C programming, and require a good understanding of the PLI standard. Many engineers do not to take full advantage of the Verilog PLI in their design projects because of the complexity of how PLI applications are specified.

1989 — the ACC interface. The second generation of the Verilog PLI introduced another library of C functions referred to *access routines*, or *ACC* routines. The library is defined in a separate file called *acc_user.h*. The ACC routines are an add-on to the TF routines, not a replacement and were created to meet the needs of ASIC vendors. In the early years of Verilog-based design, most ASIC vendors used proprietary delay calculators to increase the timing accuracy of simulations. These delay calculators would use the Verilog PLI to search the simulation data structure and find each instance of an ASIC cell. For each cell, the ACC routines then provided a way to search for the output fanout. Other ACC routines could access information stored within the cell models such as capacitance, resistance and intrinsic delays. Using this information, a PLI application could calculate accurate delays for each cell instance, and then modify the simulation data structure with the calculated delays.

This dynamic, PLI-based delay calculation methodology was eventually replaced by more static methods and Standard Delay Format (SDF) files. However, the ability of the ACC routines to search for objects within the simulation data structure, to follow design connectivity, and to read the current values of objects, is still widely used for a variety of PLI applications. For example, using the ACC routines, a PLI application could search the simulation data structure to find all

scan chains in a design, and then parallel load test vectors into the scan flip-flops. Waveform displays and graphical debug utilities often use the ACC routines to find all nets and/or variables in a portion or all of a design. These utilities can then read the values of the nets and variables, display them, or trace the design connectivity back to find the origin of the value.

The ability to search the simulation data structure is a major advantage of the ACC routines compared to the TF routines. But the of the library of ACC routines have a limitation that is important to note. The ACC routines can only search the simulation data structure for structural objects in the design. Structural objects include instances of modules and primitives, and connections between modules and primitives. The ACC library cannot access the RTL and behavioral portions of a design.

The ACC interface uses the TF interface mechanism for defining a user-defined system task/ function name, and associating that name with user-defined C functions. This complex, simulator-specific interface mechanism is another disadvantage of the ACC interface.

1995 — the VPI interface. The third generation of the Verilog PLI is named the *Verilog Procedural Interface*, or *VPI*. The VPI routines are a complete superset of the older TF and ACC routines, and are designed to completely replace the older routines. The VPI routines are defined in a third C function library, called *vpi_user.h*. With the VPI library, users are given full access to the entire simulation data structure. This includes the arguments of system tasks and functions (replacing the TF routines) and the structural objects of a design (replacing the ACC routines). In addition, the VPI library can access all RTL and behavioral statements that make up a design. The VPI library can also access the simulator's event scheduler, and add or remove scheduled events. The VPI library both replaces and enhances the TF routines that synchronize PLI applications to simulation time.

The VPI interface requires the creation of a user-defined system task or user-defined system function, the same as with the TF and ACC interfaces. The manner in which the system task/ function is defined is different than that of the older interfaces, and overcomes some of the disadvantages of the older interfaces. However, the definition is still complex and requires a good understanding of the PLI standard.

The full capabilities of the VPI routines are beyond the scope of this paper. Suffice it to say that using the VPI library, a PLI application has full, unlimited access to the simulation data structure. The VPI routines provide an extremely powerful interface to the simulation data structure. The ways in which a PLI application can interact with simulation is virtually boundless.

2003 — the SystemVerilog DPI interface. The newest generation of a procedural interface between Verilog and C is the SystemVerilog *Direct Programming Interface*, or *DPI*. The capabilities of the DPI are explained in more detail in subsequent sections of the paper.

3.3 Verilog PLI standards — Where they've been and where they're heading

The generations of the PLI described above come under a number of Verilog standards.

OVI (now Accellera) PLI 1.0. The original definitions of the TF and ACC interfaces were proprietary to Gateway Design Automation, as was the original Verilog language. In 1989,

Gateway was acquired by Cadence Design Systems, and in 1990, Cadence placed the Verilog language and PLI into the public domain. A not-for-profit organization called Open Verilog International (OVI) was formed to promote the use of Verilog. OVI labeled the 1990 public domain Verilog as Verilog 1.0 and PLI 1.0.

OVI PLI 2.0. In 1993, OVI released version 2.0 of the Verilog language and PLI. PLI 2.0 was a completely new procedural interface that was radically different than PLI 1.0 (the TF and ACC libraries). PLI 2.0 in its original form was not backward compatible with PLI 1.0 applications, and therefore was not widely adopted by EDA companies with Verilog simulators. After releasing the Verilog 2.0 standard, OVI donated Verilog to the IEEE, to become an IEEE standard.

IEEE 1364-1995. In 1995, the first IEEE version of the Verilog standard was released. This version included OVI's PLI 1.0 plus a complete rewrite of OVI's PLI 2.0 so that it was backward compatible. This rewrite became the VPI library.

IEEE 1364-2001. In 2001, the IEEE released a significant number of enhancements to the Verilog language. The VPI library was enhanced to provide access to the new language features within simulation data structures. The 1364-2001 Verilog standard states that the older TF and ACC interfaces are included solely for documentation and backward compatibility. The older libraries were not enhanced to support new language features.

Accellera SystemVerilog 3.1. In May of 2003, Accellera (formerly OVI) released the SystemVerilog 3.1 standard. SystemVerilog is not a new language. It is a significant extension to the IEEE 1364-2001 Verilog language. The SystemVerilog standard includes the Direct Programming Interface (DPI).

Accellera SystemVerilog 3.1a (planned). This version will finalize the specification of SystemVerilog in preparation to donate SystemVerilog to the IEEE 1364 Verilog standard. The target donation date is June of 2004. SystemVerilog 3.1a will include extensions to the Verilog VPI library to support the SystemVerilog extensions to the Verilog language.

IEEE 1364-2005/2006 (planned). The IEEE has targeted to release its next version of the Verilog standard in 2005 or 2006. It is expected that this version of the IEEE standard will include the SystemVerilog extensions to Verilog, as well as several other enhancements currently being defined by the 1364 working group.

The IEEE 1364 working group is considering a proposal to deprecate the TF and ACC libraries (PLI 1.0), and remove them from the next 1364 Verilog standard. These libraries are considered obsolete. They do not support the Verilog-2001 enhancements, and will not support the SystemVerilog and other enhancements to Verilog that will be in the next IEEE Verilog standard.

4.0 A more detailed look at the SystemVerilog DPI

As introduced at the beginning of this paper, the key feature of the DPI is that it allows Verilog code to directly call C functions, without the complexity of creating and defining a system task or function name and an associated calltf routine. With the DPI, values can be directly passed to the C functions, and received directly back from the C functions. These direct calls and directly passing values to and from C functions do not require the use of any procedural interface libraries.

This makes the DPI much more straightforward and easy to use than the PLI's TF, ACC, and VPI interfaces.

The DPI standard has its origins in two proprietary interfaces, the VCS *DirectC* interface from Synopsys, and the SystemSim *Cblend* interface from Co-Design Automation (later acquired by Synopsys). These two proprietary interfaces were originally developed to work with their respective simulator, and not as a standard that would work with any simulator. The Accellera SystemVerilog standards committee merged the capabilities of the two donated technologies together, and defined the DPI interface semantics in such a way as to ensure the DPI could work with any Verilog simulator.

4.1 The DPI import declaration

The DPI import declaration defines a prototype of the C function name, arguments and function return type. A C function can be imported as either a Verilog task, or as a Verilog function. A task in Verilog can input and output arguments, but does not return a value. Tasks are called as programming statements from Verilog procedural code. Functions differ from tasks in that they have a return value, as well as input and output arguments. Functions can be called anywhere an expression can be used in Verilog code.

Two example import declarations are:

```
import "DPI" function real sin(real in); // sine function in C math library

import "DPI" task file_write(string data); // user-supplied C function
```

The DPI import declaration allows a local name to be used to represent the C function name. This local name can be useful if the C function name would conflict with other names in the Verilog model. For example, the `sin` C function above could be given the name in Verilog of `sine_func`:

```
import "DPI" sine_func = function real sin(real in);
```

The DPI import declaration can be placed anywhere a native Verilog function can be defined. This includes modules, interfaces, program blocks, clocking blocks, packages, and the compilation-unit space. The imported task or function name is local to the scope in which it is imported. It is legal to import the same C function in multiple scopes, such as in multiple modules, as long as each DPI import declaration of the same C function name has exactly the same prototype. If an imported C function is to be used in multiple scopes, a good coding style is to define the import declaration in a SystemVerilog package. The package can be used in any number of modules, interfaces and clocking blocks, without having to duplicate the DPI import declaration.

4.2 Function formal arguments

Imported C functions can have any number of formal arguments, including none. By default, each formal argument is assumed to be an input into the C function. The DPI import declaration can override this default, and explicitly declare each formal argument as an **`input`**, **`output`** or bidirectional **`inout`** argument. In the following example, a square root function is defined to have two arguments: a double precision input, and a 1-bit output value that represents an error flag.

```
import "DPI" function real sqrt(input real base, output bit error);
```

From the Verilog code perspective, input and output formal arguments appear the same as with a task or function defined in the Verilog language. That is, the behavior is as if input argument values are copied into the task or function when it is called, output argument values are copied out of the task or function when it returns, and inout arguments are copied in at the call, and copied out at the return. This copy-in and copy-out behavior does not impose the actual implementation of the DPI by Verilog simulators. It merely describes the effect that is seen within the Verilog code. Implementation might use pointers or other methods to optimize simulation performance.

A formal argument that is declared as an input can only have values passed into the C function. The C function should not modify its copy of the argument value. This can be assured by declaring the input arguments of the C function as **`const`** variables.

4.3 Function return values

The C function that is imported as a function into Verilog can have any return value type that is legal in the C language, such a **`char`**, **`int`**, **`short`**, **`float`**, **`double`**, **`void`**, or a pointer. SystemVerilog extends Verilog by adding a **`void`** data type and a special **`chandle`** data type for importing C functions that return a pointer data type. The C pointer can be saved in a **`chandle`** variable, and passed back to other imported C functions as a function argument.

4.4 Data type restrictions

On the Verilog side, the DPI restricts the data types that can be passed to a C function's formal arguments, and that can be used as the return types of imported functions. The legal data types are: **`void`**, **`logic`**, **`bit`**, **`byte`**, **`shortint`**, **`int`**, **`longint`**, **`real`**, **`shortreal`**, **`chandle`**, and **`string`**. These data types can also be used as function return types.

Verilog vectors of **`reg`**, **`logic`** and **`bit`** data types can also be passed into and out of imported C functions. How these vectors are represented in C can be complex, and is beyond the scope of this paper. In brief, one way in which simple Verilog vectors can be represented in C is as an array of integers, where each integer represents 32 bits of the Verilog vector. The 4-state logic of Verilog **`reg`** and **`logic`** data types are encoded as a pair of integers in C. The mapping of Verilog vectors into arrays of integers makes it more difficult to work with Verilog vectors within C.

Verilog structures and arrays can also be passed into and out of imported C functions. When using structures and arrays, the DPI imposes several limitations. These limitations are too involved for the scope of this paper, but are explained in the SystemVerilog Language Reference Manual. Unpacked Verilog structures and arrays that use data types compatible with C are easy to pass in and out of C. They are represented as equivalent structures and arrays in C. More complex Verilog structures and arrays are more difficult to work with using the DPI interface, as there is no equivalent representation of this data in the C language.

Caution! The DPI places the burden on the user of the DPI to match the data types used in the C language with equivalent data types on the Verilog side (at the import statement). For example, an **`int`** on the C function side should be declared as an **`int`** in the DPI import prototype. A **`double`** on C function side should be declared as a **`real`** in the DPI import statement, etc. The DPI does not provide a mechanism for the C function to test what type of value is on the Verilog side. The C function simply reads or writes to its arguments, unaware that it was actually called from the

Verilog language. If the Verilog prototype does not match the actual C function, the C function might read or write erroneous values.

This strict and potentially unforgiving declaration requirement is very different than with the Verilog PLI. The PLI provides mechanisms for the C function to test the data types of system task/ function arguments. The PLI application can then adapt how values are read or written based on the data types used on the Verilog side. In addition, The PLI functions that read or write values perform automatic conversions of values from one data type to another.

4.5 Pure, context and generic C functions

The DPI allows C functions to be classified as *pure functions*, *context functions*, or *generic functions*.

Pure C functions. With a pure function, the results of the function must depend solely on values that are passed into the function through formal arguments. An advantage of pure functions is that simulators may be able to perform optimizations that improve simulation performance. A pure function cannot use global or static variables, cannot perform any file I/O operations, cannot access operating system environment variables, and cannot call functions from the Verilog PLI libraries. Only non-void functions with no output or inout arguments can be specified as pure. Pure functions cannot be imported as a Verilog task. An example of declaring an imported C function as pure is:

```
import "DPI" pure function real sin(real in); // function in C math library
```

Context C functions. A context C function is aware of the Verilog hierarchy scope in which the function is declared. This allows an imported C function to call functions from the PLI TF, ACC or VPI libraries, which makes it possible for DPI functions to take advantage of PLI features such as writing to the simulator's log file and files that are opened from within Verilog source code. An example declaration of a context-dependent task is:

```
import "DPI" context task print(input int file_id, input bit [127:0] data);
```

Generic C functions. This paper refers to a C function that is not explicitly declared as either pure or context as a *generic function*. (The SystemVerilog standard does not have a special term for functions that are neither pure nor context). A generic C function can be imported as either a Verilog function or a Verilog task. The task or function can have input, output and inout arguments. Functions can have a return value, or be declared as void. A generic C function is not allowed to call Verilog PLI functions, and should not access any data other than its actual arguments, and should only make changes to its the actual arguments.

Caution! It is the user's responsibility to correctly declare an imported function pure or context. By default, DPI functions are assumed to be generic. A call to a C function that was incorrectly declared as pure may return incorrect or inconsistent results, and can cause unpredictable run-time errors, even crashing the simulation. Similarly, if a C function accesses the Verilog PLI libraries or other API libraries, and that function is not declared as a context function, unpredictable simulation results can occur, or the simulation may crash.

4.6 Referencing PLI libraries from DPI functions

A C function that is imported as either a context function or a context task is allowed to call functions from the Verilog TF, ACC and VPI PLI libraries. This has at least two important advantages: First, by utilizing the PLI libraries, a DPI function can access information within the simulation data structure that was not passed into the imported C function. For example, a DPI function could write to the simulator's log file, or to files that had been opened by the Verilog code. Second, the DPI import methodology is much simpler than the complex PLI mechanism of defining a system task/function name. Using the DPI, a PLI application can be imported as a Verilog-like function, without the complexity of defining a system task/function name (e.g. $sine shown earlier in this paper) and then having to bind the system task/function name into the Verilog simulator.

Limitations. There is an important limitation to calling PLI library routines, however. Each instance of a PLI user-defined system task or function has a unique ID (referred to as an instance handle). If, for example, the same user-defined system task is used in two different modules, each usage is completely unique within the simulation data structure. A large number of routines in the PLI libraries depend on this instance-specific characteristic. Calls to a DPI function do not have unique instance handles. If the same DPI function is called from two or more locations in the Verilog source code, there is no way within the function to distinguish one call from another.

Another important difference between the DPI and the PLI lies in what the two interfaces classify as the task or function scope. A context DPI application uses the scope where the task or function DPI import statement is declared; *not* the scope from where the task or function is called. This matches the Verilog language, where a task or function scope in which the task or function is defined. The scope context in the PLI, on the other hand, is the scope from which a system task or system function is called. The PLI libraries expect context scope to be the scope of the call, not the DPI import declaration. This limits which functions a DPI application can call from the TF, ACC and VPI PLI libraries.

5.0 Exporting Verilog tasks and functions

In addition to importing functions from C, the DPI allows Verilog tasks and functions to be exported to C (or potentially other foreign languages). This allows code within the C language to execute code written in Verilog. Through a combination of imported C functions and exported Verilog tasks and functions, data can be easily passed between Verilog and C, and modified by whichever language is best suited for the design need.

An export declaration is similar to a DPI import declaration, except that only the name of the Verilog task or function is specified. The formal arguments of the Verilog task or function are not listed as part of the DPI export declaration. For example:

```
export "DPI" adder_function;
```

Optionally, a different name can be given to the task or function within the C language, as in:

```
export "DPI" adder = adder_function;  // called "adder" within C
```

A Verilog task or function can only be exported from the same scope in which the task or function

is defined, and only one DPI export declaration for a task or function is allowed. The formal arguments of an exported task or function must adhere to the same data type rules as with DPI import declarations.

In Verilog, a task can call other functions or tasks, but a function can only call other functions. This restriction is also true for exported tasks or functions. An exported Verilog function can only be called from a C function that has been imported as a context function or context task. An exported Verilog task can only be called from a C function that is imported as a context task.

Both exported functions and exported tasks have a return value that is unique to the DPI. The return value of an exported task or function is an `int` value, which indicates whether or not a disable statement is active on the current task execution.

The ability to import C functions and export Verilog tasks and functions enables representing a design with some parts modeled in Verilog, and other parts modeled in C (perhaps as SystemC models). For example, a design flow might begin with much of the design represented at a very high level of abstraction using the C language. As the design flow progresses, portions of these C models might be re-coded at a more detailed RTL level using Verilog. These RTL models can still interact with the abstract C models by importing the C functions representing those models. Conversely, the more detailed RTL models, if represented as exported Verilog functions, can be called from the abstract C models. The complete design can remain intact as portions of a design are represented first in one language and then in another.

An important advantage of exporting Verilog tasks is that tasks can consume simulation time through the use of nonblocking assignments, event controls, delays, and wait statements. This provides a way for a DPI-based C function to synchronize activity with simulation time. When a C function calls an exported Verilog task that consumes time, execution of the C function will stall until the Verilog task completes execution and returns back to the calling C function.

The ability for C functions to call Verilog tasks and functions is a powerful capability that is unique to the DPI. There is no equivalent to exporting tasks and functions in the Verilog PLI standard.

6.0 Using the DPI to interface to C++, SystemC, and other languages

The SystemVerilog standard only defines the DPI interface for the C language. But, the DPI is designed to be an extensible interface that can also support other languages. The DPI standard defines two layers: the *SystemVerilog layer* and the *foreign language layer.* The SystemVerilog layer contains the DPI import and export declarations and the rules for calling imported foreign functions from Verilog code. This layer will look the same, regardless of what foreign language is being called. The foreign language layer is defined for the C language in the SystemVerilog 3.1a standard. Definitions for additional foreign language layers could be added as part of future versions of the SystemVerilog or IEEE 1364 Verilog standards. Proprietary foreign language layers could also be defined for other languages. Since the SystemVerilog layer is independent of the foreign language layer, it is transparent to Verilog as to what foreign language an imported function is defined in.

Using the DPI with C++ and SystemC. The DPI is designed to interface Verilog code with the C programming language. However, imported tasks and functions can be written in C++, as long as C linkage conventions are observed at the language boundary. SystemC is based on C++. Imported SystemC functions into Verilog must follow the same safe coding conventions as when calling C++ functions.

7.0 Does the SystemVerilog DPI replace the Verilog PLI?

It is important to determine if the DPI is a superset of the Verilog PLI, and can therefore be used as a replacement of the PLI.

There are three important features of the DPI that differentiate the DPI from the TF, ACC and VPI generations of the Verilog PLI.

- The DPI removes the PLI's complexity of having to define a system task or system function name, and then associate a calltf C function with that system task/function name. With the DPI, the name of a C function can be called directly from the Verilog source code.
- The DPI provides a mechanism whereby C functions can call Verilog language tasks and functions. There is no equivalent functionality in the Verilog PLI.
- The DPI import declaration contains a prototype of the imported function arguments. Simulator compilers can use this prototype to perform strong type checking on the arguments of each call to the imported function. The PLI does not define prototypes of system tasks and functions, but does provide a mechanism for programmers to add custom syntax checking routines that are called for each instance of the system task or function.

To a certain extent, the DPI interface provides similar capabilities to the PLI TF library. Both the TF library and the DPI interface allow Verilog code to call C functions, pass values to those C functions, and receive values back from the C functions. With both interfaces, the values received back can be written to formal arguments or as a function return (or both). Both interfaces also allow the C function to called as either a function or a task in Verilog code. The DPI is *not* a superset of the TF interface, however. There are capabilities in the TF interface that cannot be done using the DPI, such as synchronizing to simulation time and events. The ACC and VPI libraries also provide important capabilities that cannot be performed with the DPI.

Some of key differences between the SystemVerilog DPI and the Verilog PLI capabilities are:

- The types of formal arguments that a DPI task or function can have is limited to the basic Verilog and SystemVerilog variable data types that can have logic values. In contrast, Verilog PLI system task/function arguments can include not only Verilog and SystemVerilog variables, but also net data types, module instance names, full hierarchy scope names, named events, constants, concatenations, and null (empty) arguments.
- Tasks and functions imported through the DPI must have a fixed number of arguments, which is specified in the DPI import prototype. Each call to a DPI task function must pass the exact number of arguments given in the prototype. PLI system tasks and functions can have a variable number of arguments. For example, a system task can be written that writes the current value of the variables listed in its arguments to a file. The same system task definition can be called with one argument, two arguments, or 10 arguments. With the DPI, a different C

function and import statement would need to be written for different numbers of arguments.

- DPI applications cannot schedule writing values into simulation at a future simulation time. Verilog PLI applications can schedule value changes to transpire at any future time.
- DPI applications cannot synchronize with other events within the simulation event queue, whereas PLI applications can synchronize to execute before any other event at a specific simulation time, before or after nonblocking assignments are executed within a specific moment of simulation, or at the end of all events at a specific simulation time.
- DPI applications cannot synchronize to events such as a change of value on some variable or net. PLI applications can be synchronized to value changes on any data type, the start of simulation, the end of simulation, when an interactive debug mode is entered, and in several other ways.
- DPI applications cannot override simulation events. PLI applications can force any variable or net to any value, overriding all other simulation activity for that object. PLI applications can also cancel future scheduled events.
- DPI applications cannot directly analyze the simulator data structure. A DPI application can indirectly perform some analysis by calling functions from the Verilog PLI libraries to access the simulation data structure. However, this indirect access is more limited than what a PLI application can access.

The following table compares the more prominent features of the DPI, TF, ACC and VPI interfaces:

Table 1: SystemVerilog DPI capabilities versus Verilog PLI

DPI Interface	TF Interface	ACC Interface	VPI Interface
Directly call C functions from Verilog code	Indirectly call C functions from Verilog code by associating the C function with a user-defined system task or system function name		
Directly pass input values to C functions	Indirectly pass input values to C functions by accessing the system task/ function arguments through a PLI library		
Directly receive values back from C functions as a function return value	Indirectly receive values back from C functions as a system function return value		
Directly modify function argument values	Indirectly modify function argument values though a PLI library		
Fixed number of task/ function arguments	Variable number of task/function arguments		
Only certain data types allowed as task/function arguments	Any data type is a legal task/function argument, plus instance names, hierarchical paths, complex expressions, and null arguments		

Table 1: SystemVerilog DPI capabilities versus Verilog PLI (Continued)

DPI Interface	TF Interface	ACC Interface	VPI Interface
User must select compatible data types on the Verilog and C sides; incorrect specification can cause errors	PLI applications can check what data type is passed as an argument. Values are read/written using PLI routines that automatically convert Verilog values of any type to a specified C data type		
Uses prototypes to check correctness of calls to tasks and functions	Uses a user-defined syntax checking routine to check correctness of calls to system tasks and functions. The routine can check for more than just the number and compatibility of arguments		
Function returns are limited to small values (1 to 64 bits)	Function returns are limited to 1 to 64 bits	Function returns can be any size (1 to an unlimited number of bits)	
C function can call Verilog functions and tasks	(no equivalent)		
Concurrent calls to tasks use the C stack	(no equivalent — concurrent calls are not possible)		
(no equivalent)	Schedule value changes in Verilog at future simulation times		
(no equivalent)		Modify propagation delays of objects	
(no equivalent)	Synchronize to simulator's event scheduler		Synchronize to simulator's event scheduler
(no equivalent)	Automatic C function callback for simulator actions (save, restart, stop, finish, etc.)		Automatic C function callback for simulator actions (save, restart, stop, finish, etc.)
(no equivalent)	Pass module and primitive instance names to C functions		
(no equivalent)		Traverse design hierarchy	
(no equivalent)		Access structural objects anywhere in design hierarchy	
(no equivalent)			Access procedural code anywhere in design hierarchy
(no equivalent)	Automatic C function callbacks for Verilog logic value changes or strength level changes on system task/function arguments		
(no equivalent)		Automatic C function callbacks for Verilog logic value changes on any object in the data structure	
(no equivalent)			Automatic C function callbacks at execution of procedural statements
(no equivalent)	Automatic work area for preserving user data over simulation time for each instance of a call to a system task/function		

8.0 Conclusions

8.1 When to use the DPI

The SystemVerilog DPI opens a new door to integrating Verilog code with C code. Using a DPI import declaration, a C function can be made to look as if it were a native Verilog function. Once imported, a C function can be called directly from any place a native Verilog function can be called. Verilog logic values can be passed directly to the C function as inputs, and C function returns or output arguments can be passed directly back to Verilog. The DPI eliminates the Verilog PLI overhead of creating system task/function names and indirectly passing values in and out of C functions through complex PLI libraries.

A major benefit of the DPI interface is that, within Verilog code, it is completely transparent as to whether a task or function call is invoking a native Verilog task or function, or invoking code written in a foreign language. This makes it very easy to interchange the actual definition of a task or function with native Verilog code or with C code. For example, a portion of a design could have both a Verilog representation and a SystemC representation. Using the DPI, it can be a simple process to switch which representation is used in a simulation.

The simple and direct nature of the SystemVerilog DPI makes it ideal for calling functions from standard C libraries, such as the C math library, or from user-defined libraries. The DPI provides a straight-forward mechanism to represent a large design partially as abstract C (or SystemC) models, and partially as more detailed Verilog behavioral, RTL or gate-level models.

The DPI is an ideal interface to use when the C function is working with data exchanged directly with Verilog through function arguments and return values. However, the DPI does not provide direct access to the internals of a simulation data structure. This limits what the DPI can be used for in comparison to the Verilog PLI. It is important to note, though, that many of these limitations of the SystemVerilog DPI can be overcome by using the DPI and Verilog PLI together. A DPI based application, if imported as a content task or context function, can then call functions from the Verilog PLI libraries. In this way, users can have the simplicity of the DPI import mechanism, and still have access to some aspects of the simulation data structure.

8.2 When to use the PLI

The DPI is not a replacement for the Verilog PLI. The PLI has full access to the internal simulation data structure, whereas a DPI based application, even when calling functions from the PLI libraries, does not have access to the full simulation data structure. The PLI libraries also provide a mechanism for Verilog PLI applications to synchronize to simulation activity in a variety of ways. DPI-based applications cannot directly synchronize to simulation activity. The indirect nature of the Verilog PLI, with its set of specialized libraries, also serves as a protecting layer between user-developed C programs and the simulation data structure.

The Verilog PLI is still the best, and only, procedural interface for applications such as waveform displays, graphical debug utilities, and other applications that need to access and analyze the contents of the simulation data structure. The PLI is also required for applications such as co-simulation environments that need to synchronize event scheduling with the Verilog simulator.

8.3 Can the PLI 1.0 standard be deprecated?

The venerable Verilog PLI is 19 years old. It has undergone three major generations of PLI libraries, under multiple standards. The oldest generations of PLI libraries are the TF and ACC libraries, which are often referred to as the PLI 1.0 standard. The IEEE 1364 Verilog standard considers these libraries to be obsolete, but, thus far, the IEEE has maintained these older libraries for documentation and backward compatibility. All functionality from these libraries has been replaced by a more powerful VPI library, often called the PLI 2.0 standard. The next IEEE 1364 standard may deprecate the older TF and ACC libraries.

The TF interface does have one advantage over the ACC and VPI interfaces. PLI applications that use only the library of TF routines can only access Verilog data structure information that is passed in as a system task/function argument. This limited access to the simulation data structure means that simulators can determine at compilation time exactly what data structure information will be accessed by the PLI application. This allows simulators to better optimize the simulation data structure and improve the performance of the simulator.

A DPI pure function has the same limited access. It can only read the Verilog values that are passed in as function inputs. Therefore, a key advantage of using the TF interface is now also available through the DPI interface.

8.4 The correct chant

It cannot be stated, *"the Verilog PLI is dead...long live the SystemVerilog DPI."* While the DPI will play an important role in some types of Verilog design, there are many capabilities unique to the Verilog PLI, in particular the VPI portion of the PLI.

The true chant is, ***"the PLI 1.0 is dead...long live PLI VPI and the SystemVerilog DPI!"***

9.0 References

[1] *"IEEE Std. 1364-2001 standard for the Verilog Hardware Description Language"*, IEEE, Pascataway, New Jersey, 2001. ISBN 0-7381-2827-9.

[2] *"SystemVerilog 3.1: Accellera's Extensions to Verilog"*, Accellera, Napa, California, 2003. Available at www.accellera.org.

[3] *"SystemVerilog 3.1a, draft 4: Accellera's Extensions to Verilog"*, Accellera, Napa, California, 2004.

[4] S. Sutherland, *"The Verilog PLI Handbook, second edition"*, Kluwer Academic Publishers, Boston, Massachusetts, 2002. ISBN 0-7923-7658-7.

10.0 Glossary of acronyms

ACC — Access interface; the second generation of the Verilog PLI.

DPI — Direct Programming Interface; part of the Accellera SystemVerilog extensions to the Verilog language.

OVI — Open Verilog International; A not-for-profit organization to promote Verilog standards and the use of Verilog. In 2001 merged with VHDL International (VI) to form Accellera.

PLI — Programming Language Interface; part of the IEEE Verilog language.

PLI 1.0 — A nickname for the TF and ACC interfaces. Originally an OVI standard.

PLI 2.0 — A nickname for the VPI interface. Originally an OVI standard.

TF — Task/Function interface; the oldest generation of the Verilog PLI.

VPI — Verilog Programming Interface; the most current generation of the Verilog PLI.

11.0 About the author

Mr. Stuart Sutherland is a member of the Accellera SystemVerilog technical subcommittee that is defining SystemVerilog, and is the technical editor of the SystemVerilog Reference Manual. He is also a member of the IEEE 1364 Verilog Standards Group, where he serves as co-chair of the PLI task force. Mr. Sutherland is an independent Verilog consultant, and specializes in providing expert training on the Verilog HDL, SystemVerilog and PLI. He can be reached by e-mail at stuart@sutherland-hdl.com.