34. Direct programming interface (DPI) (1800-2005 12.5, 26)

MINOR CHANGES: This clause comes entirely from 1800-2005 with no text from 1364-2005. Text added by editor as a result of the merge is in blue. Deleted text is in ~~red strike-through~~ with change bars. Mantis item changes made in this draft version are in purple with change bars.

This clause describes:

- DPI tasks and functions
- DPI layers
- Importing and exporting functions
- Importing and exporting tasks
- Disabling DPI tasks and functions

34.1 Overview

subclause from 1800-2005 26.1

This clause highlights the DPI and provides a detailed description of the SystemVerilog layer of the interface. The C layer is defined in Annex I.

DPI is an interface between SystemVerilog and a foreign programming language. It consists of two separate layers: the SystemVerilog layer and a foreign language layer. Both sides of DPI are fully isolated. Which programming language is actually used as the foreign language is transparent and irrelevant for the SystemVerilog side of this interface. Neither the SystemVerilog compiler nor the foreign language compiler is required to analyze the source code in the other's language. Different programming languages can be used and supported with the same intact SystemVerilog layer. For now, however, SystemVerilog defines a foreign language layer only for the C programming language. See Annex I for more details.

The motivation for this interface is two-fold. The methodological requirement is that the interface should allow a heterogeneous system to be built (a design or a testbench) in which some components can be written in a language (or more languages) other than SystemVerilog, hereinafter called the *foreign language*. On the other hand, there is also a practical need for an easy and efficient way to connect existing code, usually written in C or C++, without the knowledge and the overhead of PLI or VPI.

DPI follows the principle of a black box: the specification and the implementation of a component are clearly separated, and the actual implementation is transparent to the rest of the system. Therefore, the actual programming language of the implementation is also transparent, although this standard defines only C linkage semantics. The separation between SystemVerilog code and the foreign language is based on using functions as the natural encapsulation unit in SystemVerilog. By and large, any function can be treated as a black box and implemented either in SystemVerilog or in the foreign language in a transparent way, without changing its calls.

34.1.1 Tasks and functions

DPI allows direct inter-language function calls between the languages on either side of the interface. Specifically, functions implemented in a foreign language can be called from SystemVerilog; such functions are referred to as *imported functions*. SystemVerilog functions that are to be called from a foreign code shall be specified in export declarations (see 34.6 for more details). DPI allows for passing SystemVerilog data between the two domains through function arguments and results. There is no intrinsic overhead in this interface.

It is also possible to perform task enables across the language boundary. Foreign code can call SystemVerilog tasks, and native Verilog code can call imported tasks. An imported task has the same semantics as a native Verilog task: it never returns a value, and it can consume simulation time.

All functions used in DPI are assumed to complete their execution instantly and consume zero simulation time, just as normal SystemVerilog functions. DPI provides no means of synchronization other than by data exchange and explicit transfer of control.

Every imported task and function needs to be declared. A declaration of an imported task or function is referred to as an *import declaration*. Import declarations are very similar to SystemVerilog task and function declarations. Import declarations can occur anywhere where SystemVerilog task and function definitions are permitted. An import declaration is considered to be a definition of a SystemVerilog task or function with a foreign language implementation. The same foreign task or function can be used to implement multiple SystemVerilog tasks and functions (this can be a useful way of providing differing default argument values for the same basic task or function), but a given SystemVerilog name can only be defined once per scope. Imported task and functions can have zero or more formal **`input`**, **`output`**, and **`inout`** arguments. Imported tasks always return a void value and thus can only be used in statement context. Imported functions can return a result or be defined as void functions.

DPI is based entirely upon SystemVerilog constructs. The usage of imported functions is identical to the usage of native SystemVerilog functions. With few exceptions, imported functions and native functions are mutually exchangeable. Calls of imported functions are indistinguishable from calls of SystemVerilog functions. This facilitates ease of use and minimizes the learning curve. Similar interchangeable semantics exist between native SystemVerilog tasks and imported tasks.

34.1.2 Data types

SystemVerilog data types are the sole data types that can cross the boundary between SystemVerilog and a foreign language in either direction (i.e., when an imported function is called from SystemVerilog code or an exported SystemVerilog function is called from a foreign code). It is not possible to import the data types or directly use the type syntax from another language. A rich subset of SystemVerilog data types is allowed for formal arguments of import and export functions, although with some restrictions and with some notational extensions. Function result types are restricted to small values, however (see 34.4.5).

Formal arguments of an imported function can be declared as open arrays as specified in 34.4.6.1. Mantis 1307

34.1.2.1 Data representation

DPI does not add any constraints on how SystemVerilog-specific data types are actually implemented. Optimal representation can be platform dependent. The layout of 2- or 4-state packed structures and arrays is implementation and platform dependent.

The implementation (representation and layout) of 4-state values, structures, and arrays is irrelevant for SystemVerilog semantics and can only impact the foreign side of the interface.

34.2 Two layers of the DPI

DPI consists of two separate layers: the SystemVerilog layer and a foreign language layer. The SystemVerilog layer does not depend on which programming language is actually used as the foreign language. Although different programming languages can be supported and used with the intact SystemVerilog layer, SystemVerilog defines a foreign language layer only for the C programming language. Nevertheless, SystemVerilog code shall look identical and its semantics shall be unchanged for any foreign language layer. Different foreign languages can require that the SystemVerilog implementation shall use the appropri-

ate function call protocol and argument passing and linking mechanisms. This shall be, however, transparent to SystemVerilog users. SystemVerilog requires only that its implementation shall support C protocols and linkage.

34.2.1 DPI SystemVerilog layer

The SystemVerilog side of DPI does not depend on the foreign programming language. In particular, the actual function call protocol and argument passing mechanisms used in the foreign language are transparent and irrelevant to SystemVerilog. SystemVerilog code shall look identical regardless of what code the foreign side of the interface is using. The semantics of the SystemVerilog side of the interface is independent from the foreign side of the interface.

This clause does not constitute a complete interface specification. It only describes the functionality, semantics, and syntax of the SystemVerilog layer of the interface. The other half of the interface, the foreign language layer, defines the actual argument passing mechanism and the methods to access (read/write) formal arguments from the foreign code. See Annex I for more details.

34.2.2 DPI foreign language layer

The foreign language layer of the interface (which is transparent to SystemVerilog) shall specify how actual arguments are passed, how they can be accessed from the foreign code, how SystemVerilog-specific data types (such as **`logic`** and **`packed`**) are represented, and how they are translated to and from some predefined C-like types.

The data types allowed for formal arguments and results of imported functions or exported functions are generally SystemVerilog types (with some restrictions and with notational extensions for open arrays). Users are responsible for specifying in their foreign code the native types equivalent to the SystemVerilog types used in imported declarations or export declarations. Software tools, like a SystemVerilog compiler, can facilitate the mapping of SystemVerilog types onto foreign native types by generating the appropriate function headers.

The SystemVerilog compiler or simulator shall generate and/or use the function call protocol and argument passing mechanisms required for the intended foreign language layer. The same SystemVerilog code (compiled accordingly) shall be usable with different foreign language layers, regardless of the data access method assumed in a specific layer. Annex J defines the DPI foreign language layer for the C programming language.

34.3 Global name space of imported and exported functions

Every task or function imported to SystemVerilog must eventually resolve to a global symbol. Similarly, every task or function exported from SystemVerilog defines a global symbol. Thus the tasks and functions imported to and exported from SystemVerilog have their own global name space of linkage names, different from compilation-unit scope name space. Global names of imported and exported tasks and functions must be unique (no overloading is allowed) and shall follow C conventions for naming; specifically, such names must start with a letter or underscore, and they can be followed by alphanumeric characters or underscores. Exported and imported tasks and functions, however, can be declared with local SystemVerilog names. Import and export declarations allow users to specify a global name for a function in addition to its declared name. Should a global name clash with a SystemVerilog keyword or a reserved name, it shall take the form of an escaped identifier. The leading backslash (\ ) character and the trailing white space shall be stripped off by the SystemVerilog tool to create the linkage identifier. After this stripping, the linkage identifier so formed must comply with the normal rules for C identifier construction. If a global name is not explicitly given, it shall be the same as the SystemVerilog task or function name. For example:

Mantis 758

```
export "DPI-C" foo_plus = function \foo+ ; // "foo+" exported as "foo_plus"
export "DPI-C" function foo; // "foo" exported under its own name
import "DPI-C" init_1 = function void \init[1] (); // "init_1" is a linkage name
import "DPI-C" \begin = function void \init[2] (); // "begin" is a linkage name
```

The same global task or function can be referred to in multiple import declarations in different scopes or/and with different SystemVerilog names (see 34.4.4).

Multiple export declarations are allowed with the same *c_identifier*, explicit or implicit, as long as they are in different scopes and have the equivalent type signature (as defined in 34.4.4 for imported tasks and functions). Multiple export declarations with the same *c_identifier* in the same scope are forbidden.

It is possible to use the deprecated `"DPI"` version string syntax in an import or export declaration. This syntax indicates that the SystemVerilog 2-state and 4-state packed array argument passing convention is to be used (see I.12). In such cases, all declarations using the same *c_identifier* shall be declared with the same DPI version string syntax.

34.4 Imported tasks and functions

The usage of imported functions is similar as for native SystemVerilog functions.

34.4.1 Required properties of imported tasks and functions—semantic constraints

This subclause defines the semantic constraints imposed on imported tasks or functions. Some semantic restrictions are shared by all imported tasks or functions. Other restrictions depend on whether the special properties **`pure`** (see 34.4.2) or **`context`** (see 34.4.3) are specified for an imported task or function. A SystemVerilog compiler is not able to verify that those restrictions are observed; and if those restrictions are not satisfied, the effects of such imported task or function calls can be unpredictable.

34.4.1.1 Instant completion of imported functions

Imported functions shall complete their execution instantly and consume zero simulation time, similarly to native functions.

NOTE—Imported tasks can consume time, similar to native SystemVerilog tasks.

34.4.1.2 input, output, and inout arguments

Imported functions can have **`input`**, **`output`**, and **`inout`** arguments. The formal **`input`** arguments shall not be modified. If such arguments are changed within a function, the changes shall not be visible outside the function; the actual arguments shall not be changed.

The imported function shall not assume anything about the initial values of formal **`output`** arguments. The initial values of **`output`** arguments are undetermined and implementation dependent.

The imported function can access the initial value of a formal **`inout`** argument. Changes that the imported function makes to a formal **`inout`** argument shall be visible outside the function.

34.4.1.3 Special properties pure and context

Special properties can be specified for an imported task or function as **`pure`** or as **`context`** (see also 34.4.2 or 34.4.3).

A function whose result depends solely on the values of its input arguments and with no side effects can be specified as **pure**. This can usually allow for more optimizations and thus can result in improved simulation performance. Subclause 34.4.2 details the rules that must be obeyed by pure functions. An imported task can never be declared pure.

An imported task or function that is intended to call exported tasks or functions or to access SystemVerilog data objects other than its actual arguments (e.g., via VPI or PLI calls) must be specified as **context**. Calls of context tasks and functions are specially instrumented and can impair SystemVerilog compiler optimizations; therefore, simulation performance can decrease if the **context** property is specified when not necessary. A task or function not specified as **context** shall not read or write any data objects from SystemVerilog other then its actual arguments. For tasks or functions not specified as **context**, the effects of calling PLI, VPI, or exported SystemVerilog tasks or functions can be unpredictable and can lead to unexpected behavior; such calls can even crash. Subclause 34.4.3 details the restrictions that must be obeyed by noncontext tasks or functions.

If neither the pure nor the context attribute is used on an imported task or function, the task or function shall not access SystemVerilog data objects; however, it can perform side effects such as writing to a file or manipulating a global variable.

34.4.1.4 Memory management

The memory spaces owned and allocated by the foreign code and SystemVerilog code are disjoined. Each side is responsible for its own allocated memory. Specifically, an imported function shall not free the memory allocated by SystemVerilog code (or the SystemVerilog compiler) nor expect SystemVerilog code to free the memory allocated by the foreign code (or the foreign compiler). This does not exclude scenarios where foreign code allocates a block of memory and then passes a handle (i.e., a pointer) to that block to SystemVerilog code, which in turn calls an imported function (e.g., C standard function `free`) that directly or indirectly frees that block.

NOTE—In this last scenario, a block of memory is allocated and freed in the foreign code, even when the standard functions `malloc` and `free` are called directly from SystemVerilog code.

34.4.1.5 Reentrancy of imported tasks

A call to an imported task can result in the suspension of the currently executing thread. This occurs when an imported task calls an exported task, and the exported task executes a delay control, event control, or wait statement. Thus it is possible for an imported task's C code to be simultaneously active in multiple execution threads. Standard reentrancy considerations must be made by the C programmer. Some examples of such considerations include safely using static variables and ensuring that only thread-safe C standard library calls (MT safe) are used.

34.4.1.6 C++ exceptions

It is possible to implement DPI imported tasks and functions using C++, as long as C linkage conventions are observed at the language boundary. If C++ is used, exceptions must not propagate out of any imported task or function. Undefined behavior will result if an exception crosses the language boundary from C++ into SystemVerilog.

34.4.2 Pure functions

A **pure** function call can be safely eliminated if its result is not needed or if the previous result for the same values of input arguments is available somehow and can be reused without needing to recalculate. Only nonvoid functions with no **output** or **inout** arguments can be specified as **pure**. Functions specified as **pure** shall have no side effects whatsoever; their results need to depend solely on the values of their input argu-

ments. Calls to such functions can be removed by SystemVerilog compiler optimizations or replaced with the values previously computed for the same values of the input arguments.

Specifically, a pure function is assumed not to directly or indirectly (i.e., by calling other functions) perform the following:

— Perform any file operations.
— Read or write anything in the broadest possible meaning, including input/output, environment variables, objects from the operating system or from the program or other processes, shared memory, sockets, etc.
— Access any persistent data, like global or static variables.

If a pure function does not obey the above restrictions, SystemVerilog compiler optimizations can lead to unexpected behavior, due to eliminated calls or incorrect results being used.

34.4.3 Context tasks and functions

Some DPI imported tasks or functions require that the context of their call be known. It takes special instrumentation of their call instances to provide such context; for example, an internal variable referring to the "current instance" might need to be set. To avoid any unnecessary overhead, imported task or function calls in SystemVerilog code are not instrumented unless the imported task or function is specified as **`context`**.

The SystemVerilog context of DPI export tasks and functions must be known when they are called, including when they are called by imports. When an import invokes the `svSetScope` utility prior to calling the export, it sets the context explicitly. Otherwise, the context will be the context of the instantiated scope where the import declaration is located. Because imports with diverse instantiated scopes can export the same task or function, multiple instances of such an export can exist after elaboration. Prior to any invocations of `svSetScope`, these export instances would have different contexts, which would reflect their imported caller's instantiated scope.

The concept of *call chains* is useful for understanding how context works as control weaves in and out of SystemVerilog and another language through a DPI interface. A DPI import call chain is a series of task or function invocations that begin with a call from SystemVerilog into a task or function that is defined in a DPI-supported language and is declared in a DPI import declaration. The import call chain consists of successive calls to routines in the imported language. One of those routines becomes the last routine in the import call chain when it calls a SystemVerilog exported task or function. The import call chain can also end by simply unwinding without calling any SystemVerilog export. Once entered, an exported SystemVerilog task or function can transfer control to new import chains by invoking imports and, when exiting, can return control to its caller in the original import call chain. Mantis 1488

The following behavior characterizes context mechanics for imported call chains:

— The actions below determine an import call chain's context value:
  — When a SystemVerilog task or function calls a DPI context import, a context for the import call chain is created that is equal to the instantiated scope of the import declaration.
  — When a routine in an import call chain invokes `svSetScope` with a legal argument, the call chain's context is set to the indicated scope.
  — When a call from an import call chain to an exported SystemVerilog task or function finishes and returns to the chain, the call chain's context is set equal to its value when the call to the export was made.
— Detecting when control moves across the language boundary between SystemVerilog and an imported language is critical for simulators managing DPI context. Therefore, if user code circumvents unwinding an export call chain back to its import chain caller (e.g., by using C `setjmp`/`longjmp` constructs) the results are undefined.

— Whether a DPI import call-chain is a context chain or not is determined by whether the chain's first task or function was declared with the **`context`** keyword in the import declaration that governs its invocation; when an import declared with the context keyword appears in a call chain position other than the first position it has no effect on whether the chain has the context characteristic or not.

— Whether a DPI import call chain has the context characteristic or not cannot be dynamically changed after the initial call into the chain is made.

— The context characteristic adheres to the calling-chain, not to an individual imported task or function; thus, the same imported task or function can appear in both context and non-context call chains.

For the sake of simulation performance, an imported task or function call shall not block SystemVerilog compiler optimizations. An imported task or function not specified as **`context`** shall not access any data objects from SystemVerilog other than its actual arguments. Only the actual arguments can be affected (read or written) by its call. Therefore, a call of a noncontext task or function is not a barrier for optimizations. A context imported task or function, however, can access (read or write) any SystemVerilog data objects by calling PLI/VPI or by calling an export task or function. Therefore, a call to a context task or function is a barrier for SystemVerilog compiler optimizations.

Only calls of context imported tasks or functions are properly instrumented and cause conservative optimizations; therefore, only those tasks or functions can safely call all tasks or functions from other APIs, including PLI and VPI functions or exported SystemVerilog tasks or functions. For imported tasks or functions not specified as **`context`**, the effects of calling PLI or VPI functions or SystemVerilog tasks or functions can be unpredictable; and such calls can crash if the callee requires a context that has not been properly set. However, declaring an import context task or function does not automatically make any other simulator interface automatically available. For VPI access (or any other interface access) to be possible, the appropriate implementation-defined mechanism must still be used to enable these interface(s). Realize also that DPI calls do not automatically create or provide any handles or any special environment that can be needed by those other interfaces. It is the user's responsibility to create, manage, or otherwise manipulate the required handles or environment(s) needed by the other interfaces.

Context imported tasks or functions are always implicitly supplied a scope representing the fully qualified instance name within which the import declaration was present. This scope defines which exported SystemVerilog tasks or functions can be called directly from the imported task or function; only tasks or functions defined and exported from the same scope as the import can be called directly. To call any other exported SystemVerilog tasks or functions, the imported task or function shall first have to modify its current scope, in essence performing the foreign language equivalent of a SystemVerilog hierarchical task or function call.

Special DPI utility functions exist that allow imported task or functions to retrieve and operate on their scope. See Annex I for more details.

34.4.4 Import declarations

Each imported task or function shall be declared. Such declaration are referred to as *import declarations*. The syntax of an **`import`** declaration is similar to the syntax of SystemVerilog task or function prototypes (see 13.5).

Imported tasks or functions are similar to SystemVerilog tasks or functions. Imported tasks or functions can have zero or more formal **`input`**, **`output`**, and **`inout`** arguments. Imported functions can return a result or be defined as void functions. Imported tasks always return an **`int`** result as part of the DPI disable protocol and, thus, are declared in foreign code as **`int`** functions (see 34.7 and 34.8).

---

dpi_import_export ::= *// from A.2.6*

```
    import dpi_spec_string [ dpi_function_import_property ] [ c_identifier = ] dpi_function_proto ;
  | import dpi_spec_string [ dpi_task_import_property ] [ c_identifier = ] dpi_task_proto ;
  | export dpi_spec_string [ c_identifier = ] function function_identifier ;
  | export dpi_spec_string [ c_identifier = ] task task_identifier ;
dpi_spec_string ::= "DPI-C" | "DPI"
dpi_function_import_property ::= context | pure
dpi_task_import_property ::= context
dpi_function_proto[8,9] ::= function_prototype
dpi_task_proto[9] ::= task_prototype
function_prototype ::= function function_data_type function_identifier ( [ tf_port_list ] )
task_prototype ::= task task_identifier ( [ tf_port_list ] )                    // from A.2.7
```

8) dpi_function_proto return types are restricted to small values, per 34.4.5.

9) Formals of dpi_function_proto and dpi_task_proto cannot use pass by reference mode and class types cannot be passed at all; for the complete set of restrictions see 34.4.6.

Syntax 34-1—DPI import declaration syntax (excerpt from Annex A)

An import declaration specifies the task or function name, function result type, and types and directions of formal arguments. It can also provide optional default values for formal arguments. Formal argument names are optional unless argument binding by name is needed. An import declaration can also specify an optional task or function property. Imported functions can have the properties context or pure; imported tasks can have the property **`context`**.

Because an import declaration is equivalent to defining a task or function of that name in the SystemVerilog scope in which the import declaration occurs, and thus multiple imports of the same task or function name into the same scope are forbidden.

NOTE—This declaration scope is particularly important in the case of imported context tasks or functions (see 34.4.3); for noncontext imported tasks or functions the declaration scope has no other implications other than defining the visibility of the task or function.

The *dpi_spec_string* can take values `"DPI-C"` and `"DPI"`. `"DPI"` is used to indicate that the deprecated SystemVerilog packed array argument passing semantics is to be used. In this semantics, arguments are passed in actual simulator representation format rather than in canonical format, as is the case with `"DPI-C"`.

Use of the string `"DPI"` shall generate a compile-time error. The error message shall contain the following information:

— `"DPI"` is deprecated and should be replaced with `"DPI-C"`.

— Use of the `"DPI-C"` string may require changes in the DPI application's C code.

For more information on using deprecated `"DPI"` access to packed data, see I.12.

The *c_identifier* provides the linkage name for this task or function in the foreign language. If not provided, this defaults to the same identifier as the SystemVerilog task or function name. In either case, this linkage name must conform to C identifier syntax. An error shall occur if the *c_identifier*, either directly or indirectly, does not conform to these rules.

For any given *c_identifier* (whether explicitly defined with *c_identifier*= or automatically determined from the task or function name), all declarations, regardless of scope, must have exactly the same type signature.

The signature includes the return type and the number, order, direction, and types of each and every argument. The type includes dimensions and bounds of any arrays or array dimensions. The signature also includes the **pure**/**context** qualifiers that can be associated with an import definition, and it includes the value of the *dpi_spec_string*. Mantis 924

It is permitted to have multiple declarations of the same imported or exported task or function in different scopes; therefore, argument names and default values can vary, provided the type compatibility constraints are met.

A formal argument name is required to separate the packed and the unpacked dimensions of an array.

The qualifier **ref** cannot be used in import declarations. The actual implementation of argument passing depends solely on the foreign language layer and its implementation and shall be transparent to the SystemVerilog side of the interface.

The following are examples of external declarations.

Mantis 758

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

// from standard math library
import "DPI-C" pure function real sin(real);

// from standard C library: memory management
import "DPI-C" function chandle malloc(int size); // standard C function
import "DPI-C" function void free(chandle ptr); // standard C function

// abstract data structure: queue
import "DPI-C" function chandle newQueue(input string name_of_queue);

// Note the following import uses the same foreign function for
// implementation as the prior import, but has different SystemVerilog name
// and provides a default value for the argument.
import "DPI-C" newQueue=function chandle newAnonQueue(input string s=null);
import "DPI-C" function chandle newElem(bit [15:0]);
import "DPI-C" function void enqueue(chandle queue, chandle elem);
import "DPI-C" function chandle dequeue(chandle queue);

// miscellanea
import "DPI-C" function bit [15:0] getStimulus();
import "DPI-C" context function void processTransaction(chandle elem,
                                    output logic [64:1] arr [0:63]);
import "DPI-C" task checkResults(input string s, bit [511:0] packet);
```

34.4.5 Function result

An imported function declaration must explicitly specify a data type or void for the type of the function's return result. Function result types are restricted to small values. The following SystemVerilog data types are allowed for imported function results:

— **void**, **byte**, **shortint**, **int**, **longint**, **real**, **shortreal**, **chandle**, and **string**

— Scalar values of type **bit** and **logic**

The same restrictions apply for the result types of exported functions.

34.4.6 Types of formal arguments

A rich subset of SystemVerilog data types is allowed for formal arguments of import and export tasks or functions. Generally, C-compatible types, packed types, and user-defined types built of types from these two categories can be used for formal arguments of DPI tasks or functions. The set of permitted types is defined inductively.

The following SystemVerilog types are the only permitted types for formal arguments of import and export tasks or functions:

- **`void`**, **`byte`**, **`shortint`**, **`int`**, **`longint`**, **`real`**, **`shortreal`**, **`chandle`**, and **`string`**
- Scalar values of type **`bit`** and **`logic`**
- Packed arrays, structs, and unions composed of types **`bit`** and **`logic`**. Every packed type is eventually equivalent to a packed one-dimensional array. On the foreign language side of the DPI, all packed types are perceived as packed one-dimensional arrays regardless of their declaration in the SystemVerilog code.
- Enumeration types interpreted as the type associated with that enumeration
- Types constructed from the supported types with the help of the constructs:
  - **`struct`**
  - **`union`** (packed forms only)
  - Unpacked array
  - **`typedef`**

Mantis 1322

The following caveats apply for the types permitted in DPI:

- Enumerated data types are not supported directly. Instead, an enumerated data type is interpreted as the type associated with that enumerated type.
- SystemVerilog does not specify the actual memory representation of packed structures or any arrays, packed or unpacked. Unpacked structures have an implementation-dependent packing, normally matching the C compiler.
- In exported DPI tasks or functions, it is erroneous to declare formal arguments of dynamic array types.
- The actual memory representation of SystemVerilog data types is transparent for SystemVerilog semantics and irrelevant for SystemVerilog code. It can be relevant for the foreign language code on the other side of the interface, however; a particular representation of the SystemVerilog data types can be assumed. This shall not restrict the types of formal arguments of imported tasks or functions, with the exception of unpacked arrays. SystemVerilog implementation can restrict which SystemVerilog unpacked arrays are passed as actual arguments for a formal argument that is a sized array, although they can be always passed for an unsized (i.e., open) array. Therefore, the correctness of an actual argument might be implementation dependent. Nevertheless, an open array provides an implementation-independent solution.

34.4.6.1 Open arrays

The size of the packed dimension, the unpacked dimension, or both dimensions can remain unspecified; such cases are referred to as *open arrays* (or *unsized arrays*). Open arrays allow the use of generic code to handle different sizes.

Formal arguments of imported functions can be specified as open arrays. (Exported SystemVerilog functions cannot have formal arguments specified as open arrays.) A formal argument is an open array when a range of one or more of its dimensions is unspecified (denoted by using square brackets, `[]`). This is solely a relaxation of the argument-matching rules. An actual argument shall match the formal one regardless of the

range(s) for its corresponding dimension(s), which facilitates writing generalized code that can handle SystemVerilog arrays of different sizes.

Mantis 1395

Although the packed part of an array can have an arbitrary number of sized dimensions, an unsized dimension must be the sole packed dimension of a formal argument. This is not very restrictive, because any packed type is essentially equivalent to a one-dimensional packed array. The number of unpacked dimensions is not restricted.

If a formal argument has an unsized, packed dimension, it will match any collection of actual argument packed dimensions. Formal argument unpacked dimensions are matched on a dimension by dimension basis (see 7.5) with each unsized formal dimension matching a corresponding actual dimension of any size.

Examples of types of formal arguments (empty square brackets `[]` denote open array):

```
logic
bit [8:1]
bit[]
bit [7:0] array8x10 [1:10] // array8x10 is a formal arg name
logic [31:0] array32xN []  // array32xN is a formal arg name
logic [] arrayNx3 [3:1]    // arrayNx3 is a formal arg name
bit [] arrayNxN []         // arrayNxN is a formal arg name
```

Example of complete import declarations:

Mantis 758

```
import "DPI-C" function void foo(input logic [127:0]);
import "DPI-C" function void boo(logic [127:0] i []); //open array of 128-bit
```

Example of the use of open arrays for different sizes of actual arguments:

```
typedef struct {int i; ... } MyType;

import "DPI-C" function void foo(input MyType i [][]);
      /* 2-dimensional unsized unpacked array of MyType */

MyType a_10x5 [11:20][6:2];
MyType a_64x8 [64:1][-1:-8];

foo(a_10x5);
foo(a_64x8);
```

34.5 Calling imported functions

The usage of imported functions is identical to the usage of native SystemVerilog functions. Hence the usage and syntax for calling imported functions is identical to the usage and syntax of native SystemVerilog functions. Specifically, arguments with default values can be omitted from the call; arguments can be bound by name if all formal arguments are named.

34.5.1 Argument passing

Argument passing for imported functions is ruled by the WYSIWYG principle: What You Specify Is What You Get (see 34.5.1.1). The evaluation order of formal arguments follows general SystemVerilog rules.

Argument compatibility and coercion rules are the same as for native SystemVerilog functions. If a coercion is needed, a temporary variable is created and passed as the actual argument. For **`input`** and **`inout`** arguments, the temporary variable is initialized with the value of the actual argument with the appropriate coer-

cion. For **output** or **inout** arguments, the value of the temporary variable is assigned to the actual argument with the appropriate conversion. The assignments between a temporary and the actual argument follow general SystemVerilog rules for assignments and automatic coercion.

On the SystemVerilog side of the interface, the values of actual arguments for formal input arguments of imported functions shall not be affected by the callee. The initial values of formal output arguments of imported functions are unspecified (and can be implementation dependent), and the necessary coercions, if any, are applied as for assignments. Imported functions shall not modify the values of their input arguments.

For the SystemVerilog side of the interface, the semantics of arguments passing is as if **input** arguments are passed by *copy-in*, **output** arguments are passed by *copy-out*, and **inout** arguments were passed by *copy-in, copy-out*. The terms *copy-in* and *copy-out* do not impose the actual implementation; they refer only to "hypothetical assignment".

The actual implementation of argument passing is transparent to the SystemVerilog side of the interface. In particular, it is transparent to SystemVerilog whether an argument is actually passed by value or by reference. The actual argument passing mechanism is defined in the foreign language layer. See Annex I for more details.

34.5.1.1 WYSIWYG principle

The WYSIWYG principle guarantees the types of formal arguments of imported functions: an actual argument is guaranteed to be of the type specified for the formal argument, with the exception of open arrays (for which unspecified ranges are statically unknown). Formal arguments, other than open arrays, are fully defined by import declaration; they shall have ranges of packed or unpacked arrays exactly as specified in the import declaration. Only the declaration site of the imported function is relevant for such formal arguments.

Another way to state this is that no compiler (either C or SystemVerilog) can make argument coercions between a caller's declared formal and the callee's declared formals. This is because the callee's formal arguments are declared in a different language from the caller's formal arguments; hence there is no visible relationship between the two sets of formals. Users are expected to understand all argument relationships and provide properly matched types on both sides of the interface.

The unsized dimensions of open array formal arguments have the size of the corresponding actual argument dimensions. A formal's unsized, unpacked dimensions take on the ranges of the corresponding actual dimension. A solitary, unsized, packed dimension assumes the linearized, normalized range of the actual's packed dimensions (see I.6.6). The unsized ranges of open arrays are determined at a call site; the rest of the type information is specified at the import declaration. Mantis 1393

Therefore, if a formal argument is declared as **bit** `[15:8] b []`, then the import declaration specifies that the formal argument is an unpacked array of packed bit array with bounds 15 to 8, while the actual argument used at a particular call site defines the bounds for the unpacked part for that call.

It is sometimes permissible to pass a dynamic array as an actual argument to an imported DPI task or function. The rules for passing dynamic array actual arguments to imported DPI tasks and functions are identical to the rules for native SystemVerilog tasks and functions. Refer to 7.5 for details on such use of dynamic arrays.

34.5.2 Value changes for output and inout arguments

The SystemVerilog simulator is responsible for handling value changes for **output** and **inout** arguments. Such changes shall be detected and handled after control returns from imported functions to SystemVerilog code.

For **output** and **inout** arguments, the value propagation (i.e., value change events) happens as if an actual argument was assigned a formal argument immediately after control returns from imported functions. If there is more than one argument, the order of such assignments and the related value change propagation follows general SystemVerilog rules.

34.6 Exported functions

DPI allows calling SystemVerilog functions from another language. However, such functions must adhere to the same restrictions on argument types and results as imposed on imported functions. It is an error to export a function that does not satisfy such constraints.

SystemVerilog functions that can be called from foreign code need to be specified in **export** declarations. Export declarations are allowed to occur only in the scope in which the function being exported is defined. Only one export declaration per function is allowed in a given scope.

One important restriction exists. Class member functions cannot be exported, but all other SystemVerilog functions can be exported.

Similar to import declarations, **export** declarations can define an optional *c_identifier* to be used in the foreign language when calling an exported function.

---

dpi_import_export ::= *// from A.2.6*
| **export** dpi_spec_string [c_identifier **=**] **function** function_identifier **;**
dpi_spec_string ::= **"DPI-C"** | **"DPI"**

Syntax 34-2—DPI export declaration syntax (excerpt from Annex A)

The *c_identifier* is optional here. It defaults to *function_identifier*. For rules describing *c_identifier*, see 34.3. No two functions in the same SystemVerilog scope can be exported with the same explicit or implicit *c_identifier*. The export declaration and the definition of the corresponding SystemVerilog function can occur in any order. Only one export declaration is permitted per SystemVerilog function, and all export functions are always context functions.

34.7 Exported tasks

SystemVerilog allows tasks to be called from a foreign language, similar to functions. Such tasks are termed *exported tasks*.

All aspects of exported functions described above in 34.6 apply to exported tasks. This includes legal declaration scopes as well as usage of the optional *c_identifier*.

It is never legal to call an exported task from within an imported function. This semantics is identical to native SystemVerilog semantics, in which it is illegal for a function to perform a task enable.

It is legal for an imported task to call an exported task only if the imported task is declared with the **context** property. See 34.4.3 for more details.

One difference between exported tasks and exported functions is that SystemVerilog tasks do not have return value types. The return value of an exported task is an **int** value that indicates if a disable is active or not on the current execution thread.

Similarly, imported tasks return an **int** value that is used to indicate that the imported task has acknowledged a disable. See 34.8 for more detail on disables in DPI.

34.8 Disabling DPI tasks and functions

It is possible for a **disable** statement to disable a block that is currently executing a mixed language call chain. When a DPI import task or function is disabled, the C code is required to follow a simple disable protocol. The protocol gives the C code the opportunity to perform any necessary resource cleanup, such as closing open file handles, closing open VPI handles, or freeing heap memory.

An imported task or function is said to be in the disabled state when a **disable** statement somewhere in the design targets either it or a parent for disabling. An imported task or function can only enter the disabled state immediately after the return of a call to an exported task or function. An important aspect of the protocol is that disabled import tasks and functions must programmatically acknowledge that they have been disabled. A task or function can determine that it is in the disabled state by calling the API function `svIsDisabledState()`.

The protocol is composed of the following items:

a) When an exported task returns due to a disable, it must return a value of 1. Otherwise, it must return 0.

b) When an imported task returns due to a disable, it must return a value of 1. Otherwise, it must return 0.

c) Before an imported function returns due to a disable, it must call the API function `svAckDisabledState()`.

d) Once an imported task or function enters the disabled state, it is illegal for the current function invocation to make any further calls to exported tasks or functions.

Item b, item c, and item d are mandatory behavior for imported DPI tasks and functions. It is the responsibility of the DPI programmer to correctly implement the behavior.

Item a is guaranteed by SystemVerilog simulators. In addition, simulators must implement checks to ensure that item b, item c, and item d are correctly followed by imported tasks and functions. If any protocol item is not correctly followed, a fatal simulation error is issued.

If an exported task itself is the target of a disable, its parent imported task is not considered to be in the disabled state when the exported task returns. In such cases, the exported task shall return value 0, and calls to `svIsDisabledState()` shall return 0 as well.

When a DPI imported task or function returns due to a disable, the values of its **output** and **inout** parameters are undefined. Similarly, function return values are undefined when an imported function returns due to a disable. C programmers can return values from disabled functions, and C programmers can write values into the locations of **output** and **inout** parameters of imported tasks or functions. However, SystemVerilog simulators are not obligated to propagate any such values to the calling SystemVerilog code if a disable is in effect.

34.9 Import and export functions

subclause from 1800-2005 12.5

The syntax for the import and export of functions is as follows:

QUESTION: This subclause seems to be mostly, if not entirely, redundant with the rest of this Clause. Can this subclause be deleted? If not, can the redundant text be replaced with cross references?

```
dpi_import_export ::=                                          // from A.2.6
      import dpi_spec_string [ dpi_function_import_property ] [ c_identifier = ] dpi_function_proto ;
    | import dpi_spec_string [ dpi_task_import_property ] [ c_identifier = ] dpi_task_proto ;
    | export dpi_spec_string [ c_identifier = ] function function_identifier ;
    | export dpi_spec_string [ c_identifier = ] task task_identifier ;
dpi_spec_string ::= "DPI-C" | "DPI"
dpi_function_import_property ::= context | pure
dpi_function_proto[8,9] ::= function_prototype
function_prototype ::= function function_data_type function_identifier ( [ tf_port_list ] )
```

8) dpi_function_proto return types are restricted to small values, per 34.4.5.

9) Formals of dpi_function_proto and dpi_task_proto cannot use pass by reference mode and class types cannot be passed at all; for the complete set of restrictions see 34.4.6.

Syntax 34-3—Import and export syntax (excerpt from Annex A)

In both **import** and **export**, *c_identifier* is the name of the foreign function (import/export), and *function_identifier* is the SystemVerilog name for the same function. If *c_identifier* is not explicitly given, it shall be the same as the SystemVerilog function *function_identifier*. An error shall be generated if, and only if, the *c_identifier* has characters that are not valid in a C function identifier.

Several SystemVerilog functions can be mapped to the same foreign function by supplying the same *c_identifier* for several *fnames*. The corresponding SystemVerilog functions must have identical argument types, as defined in the next paragraph.

For any given *c_identifier*, all declarations, regardless of scope, must have exactly the same function signature. The function signature includes the return type and the number, order, direction, and types of each and every argument. Each type includes dimensions and bounds of any arrays/array dimensions. For **import** declarations, arguments can be open arrays. Open arrays are defined in 34.4.6.1. The signature also includes the **pure**/**context** qualifiers that can be associated with an import definition.

Only one **import** or **export** declaration of a given *function_identifier* shall be permitted in any given scope. More specifically, for an **import**, the import must be the sole declaration of *function_identifier* in the given scope. For an **export**, the function must be declared in the scope where the export occurs, and there must be only one export of that *function_identifier* in that scope.

For exported functions, the exported function must be declared in the same scope that contains the **export** "DPI" declaration. Only SystemVerilog functions can be exported (specifically, this excludes exporting a class method).

QUESTION: Should this be "DPI-C"?

All **import** "DPI" functions declared this way can be invoked by hierarchical reference the same as any normal SystemVerilog function. Declaring a SystemVerilog function to be exported does not change the semantics or behavior of this function from the SystemVerilog perspective (i.e., there is no effect in SystemVerilog usage other than making this exported function also accessible to C callers).

QUESTION: Should this be "DPI-C"?

Only nonvoid functions with no **output** or **inout** arguments can be specified as **pure**. Functions specified as pure in their corresponding SystemVerilog external declarations shall have no side effects; their results

need to depend solely on the values of their input arguments. Calls to such functions can be removed by SystemVerilog compiler optimizations or replaced with the values previously computed for the same values of the input arguments.

Specifically, a pure function is assumed to not directly or indirectly (i.e., by calling other functions) perform the following:

— Perform any file operations

— Read or write anything in the broadest possible meaning, including input/output, environment variables, objects from the operating system, or from the program or other processes, shared memory, sockets, etc.

— Access any persistent data, like global or static variables

If a pure function does not obey the above restrictions, SystemVerilog compiler optimizations can lead to unexpected behavior, due to eliminated calls or incorrect results being used.

An unqualified imported function can have side effects, but cannot read or modify any SystemVerilog signals other than those provided through its arguments. Unqualified imports shall not be permitted to invoke exported SystemVerilog functions.

Imported functions with the **`context`** qualifier can invoke exported SystemVerilog functions and can read or write to SystemVerilog signals other than those passed through their arguments, either through the use of other interfaces or as a side effect of invoking exported SystemVerilog functions. Context functions shall always implicitly be supplied a scope representing the fully qualified instance name within which the **`import`** declaration was present (i.e., an import function always runs in the instance in which the **`import`** declaration occurred). This is the same semantics as SystemVerilog functions, which also run in the scope they are defined, rather than in the scope of the caller.

Import context functions can have side effects and can use other SystemVerilog interfaces (including but not limited to VPI). However, declaring an import context function does not automatically make any other simulator interface available. For VPI access (or any other interface access) to be possible, the appropriate implementation-defined mechanism must still be used to enable these interface(s). Also, SystemVerilog DPI calls do not automatically create or provide any handles or any special environment that might be needed by the other interfaces. It shall be the user's responsibility to create, manage, or otherwise manipulate the required handles/environment(s) needed by the other interfaces. The `svGetScopeName()` and related functions exist to provide a name-based linkage from DPI to other interfaces. Exported functions can only be invoked if the current DPI context refers to an instance in which the named function is defined.

To access functions defined in any other scope, the foreign code shall have to change DPI context appropriately. Attempting to invoke an exported SystemVerilog function from a scope in which it is not directly visible shall result in a run-time error. How such errors are handled shall be implementation dependent. If an imported function needs to invoke an exported function that is not visible from the current scope, it needs to change, via `svSetScope`, the current scope to a scope that does have visibility to the exported function. This is conceptually equivalent to making a hierarchically qualified function call in SystemVerilog. The current SystemVerilog context shall be preserved across a call to an exported function, even if current context has been modified by an application. For noncontext imports, the context is not defined, and attempting to use any functionality depending on context from noncontext imports can lead to unpredictable behavior.