Section 1
Direct Programming Interface (DPI)

This chapter highlights the Direct Programming Interface and provides a detailed description of the SysteVerilog layer of the interface. The C layer is defined in Annex A.

1.1 Overview

Direct Programming Interface (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 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 3.1 defines a foreign language layer only for the C programming language. See Annex A 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 may 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 is 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, though 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.

1.1.1 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 section 1.6 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.

All functions used in DPI are assumed to complete their execution instantly and consume `0` (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 function needs to be declared. A declaration of an imported function is referred to as an *import declaration*. Import declarations are very similar to SystemVerilog function declarations. Import declarations may occur anywhere where SystemVerilog function definitions are permitted. An import declaration is considered to be a definition of a SystemVerilog function with a foreign language implementation. The same foreign function can be used to implement multiple SystemVerilog functions (this can be a useful way of providing differing default argument values for the same basic function), but a given SystemVerilog name can only be defined once per scope. Imported functions can have zero or more formal `input`, `output`, and `inout` arguments, and they can return a result or be defined as `void` functions.

DPI is based entirely upon SystemVerilog constructs. The usage of imported functions is identical as for native SystemVerilog functions. With few exceptions imported functions and native functions are mutually exchangeable. Calls ~~sites~~ of imported functions are indistinguishable from calls of SystemVerilog functions. This facilitates ease-of-use and minimizes the learning curve.

1.1.2 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. 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~~, most SystemVerilog data types are allowed in DPI.~~, see section 1.4.6. Function result types are restricted to small values, however (see section 1.4.5).

Formal arguments of an imported function can be specified as open arrays. A formal argument is an *open array* when a range of one or more of its dimensions, packed or unpacked, is unspecified (denoted by using empty 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. See section 1.4.6.1.

1.1.2.1 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.

1.2 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 3.1 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 will require, of course, that SystemVerilog implementation shall use the appropriate function call protocol, argument passing and linking mechanisms. This shall be, however, transparent to SystemVerilog users. SystemVerilog 3.1 requires only that its implementation shall support C protocols and linkage.

1.2.1 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 chapter 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 A for more details.

1.2.2 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 to translate them 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). The user is

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 A defines DPI foreign language layer for the C programming language.

1.3 Global name space of imported and exported functions

Every function imported to SystemVerilog must eventually resolve to a global symbol. Similarly, every function exported from SystemVerilog defines a global symbol. Thus the functions imported to and exported from SystemVerilog have their own global name space of linkage names, different from `$root` name space. Global names of imported and exported 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 may be followed by alphanumeric characters or undersores. Exported and imported functions, however, may be declared with local SystemVerilog names. Import and export declarations allow users to specify a global name (linkage 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 a form of escaped identifier. The leading backslash character (\) and the trailing white space will be stripped off. If a global name is not explicitly given, it will be the same as the SystemVerilog function name. Example:

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

The same global function may be referred to in multiple import declarations in different scopes or/and with different SystemVerilog names, see section 1.4.4.

Multiple export declarations are allowed with the same cname, explicit or implicit, as long as they are in different scopes and have the same type signature (as defined in section 1.4.4 for imported functions). Multiple export declarations with the same cname in the same scope are forbidden.

1.4 Imported functions

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

1.4.1 Required properties of imported functions - semantic constraints

This section defines the semantic constraints imposed on imported functions. Some semantic restrictions are shared by all imported functions. Other restrictions depend on whether the special properties `pure` (see section 1.4.2) or `context` (see section 1.4.3) are specified for an imported 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 function calls can be unpredictable.

1.4.1.1 Instant completion

Imported functions shall ~~contain no timing control whatsoever, directly or indirectly. imported functions shall be non-blocking; they shall~~ complete their execution instantly and consume zero-simulation time, ~~i.e., no simulation time passes during the execution of imported function.~~ similarly to native functions.

1.4.1.2 `input` and `output` arguments

Imported functions can have `input` and `output` 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~~.

1.4.1.3 Special properties `pure` and `context`

Special properties can be specified for an imported function: as `pure` or as `context` (see also section 1.4.2 or section 1.4.3).

A function whose result depends solely on the values of its input arguments and with no side effects may be specified as `pure`. This will usually allow for more optimizations and thus may result in improved simulation performance. Section section 1.4.2 details the rules that must be obeyed by `pure` functions.

An imported function that is intended to call exported functions or to access SystemVerilog data objects other then its actual arguments (e.g. via VPI or PLI calls) must be specified as `context`. Calls of `context` functions are specially instrumented and may impair SystemVerilog compiler optimizations; therefore simulation performance may decrease if the `context` property is ~~used~~ specified when not necessary. A function not specified as `context` shall not read or write any data objects from SystemVerilog other then its actual arguments. For functions not specified as `context`, the effects of calling PLI, VPI, or exported SystemVerilog functions can be unpredictable and can lead to unexpected behavior; such calls can even crash. Section section 1.4.3 details the restrictions that must be obeyed by non-`context` functions.

1.4.1.4 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, 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`) which 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.

1.4.2 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 non-void 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 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 not to directly or indirectly (i.e., by calling other functions):

— perform any file operations

— read or write anything in the broadest possible meaning, includes i/o, 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.

1.4.3 Context functions

Some DPI imported functions require that the context of their call is 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 function calls in SystemVerilog code are

not instrumented unless the imported function is specified as `context`.

All DPI exported functions require that the context of their call is known. This occurs since SystemVerilog function declarations always occur in instantiable scopes, hence allowing a multiplicity of unique function instances.

For the sake of simulation performance, an imported function call shall not block SystemVerilog compiler optimizations. An imported 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 non-`context` function is not a barrier for optimizations. A `context` imported function, however, can access (read or write) any SystemVerilog data objects by calling PLI/VPI, or by calling an export function. Therefore, a call to a `context` function is a barrier for SystemVerilog compiler optimizations.

Only calls of `context` imported functions are properly instrumented and cause conservative optimizations; therefore, only those functions can safely call all functions from other APIs, including PLI and VPI functions or exported SystemVerilog functions. For imported functions not specified as `context`, the effects of calling PLI, VPI, or SystemVerilog functions can be unpredictable and such calls can crash if the callee requires a context that has not been properly set. However note that declaring an import context 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). Note also that DPI calls do not automatically create or provide any handles or any special environment that may be needed by those other interfaces. It is the user's responsibility to create, manage or otherwise manipulate the required handles/environment(s) needed by the other interfaces.

Context imported 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 functions may be called directly from the imported function; only functions defined and exported from the same scope as the import can be called directly. To call any other exported SystemVerilog functions, the imported function will first have to modify its current scope, in essence performing the foreign language equivalent of a SystemVerilog hierarchical function call.

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

1.4.4 Import declarations

Also cross-reference to section 10.6, import and export functions

Each imported function shall be declared. Such declarations are referred to as *import declarations*. The syntax of an import declaration is similar to the syntax of SystemVerilog function prototypes.

Imported functions are similar to SystemVerilog functions. Imported functions can have zero or more formal `input`, `output`, and `inout` arguments. Imported functions can return a result or be defined as `void` functions.

Syntax:

import_dpi_decl ::= ***import*** *"DPI" [****pure****|****context****] [cname=] <named_function_proto>;*

where named_function_proto is as defined in section A.2.6 of SV 3.1 BNF

/* EDITOR: UPDATE ABOVE CROSS-REFERENCE AS NECESSARY */

An import declaration specifies the 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 passing by name is needed. An import declaration can also specify an optional function property: `context` or `pure`.

Note that an import declaration is equivalent to defining a function of that name in the SystemVerilog scope in

which the import declaration occurs, and thus multiple imports of the same function name into the same scope are forbidden. Note that this declaration scope is particularly important in the case of imported context functions, see section 1.4.3; for non-context imported functions the declaration scope has no other implications other than defining the visibility of the function.

An `import` declaration can define an optional global name (`cname`). If not provided, `cname` defaults to the same identifier as the SystemVerilog function name. In either case, this global name, explicit or implicit, provides the linkage name for this function in the foreign language. For rules describing `cname`, see section section 1.3. An error will occur if the `cname`, either explicit or implicit, does not conform to these rules.

For any given global name (whether explicitly defined with cname=, or automatically determined from the 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. Type includes dimensions and bounds of any arrays or array dimensions. Signature also includes the pure/context qualifiers that may be associated with an extern definition.

Note that multiple declarations of the same imported or exported function in different scopes may vary argument names and default values, 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` can not 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.

```
import “DPI” function void myInit();
// from standard math library
import “DPI” pure function real sin(real);
// from standard C library: memory management
import “DPI” function handle malloc(int size); // standard C function
import “DPI” function void free(handle ptr); // standard C function
// abstract data structure: queue
import “DPI” function handle 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” newQueue=function handle newAnonQueue(input string s=NULL);
import “DPI” function handle newElem(bit [15:0]);
import “DPI” function void enqueue(handle queue, handle elem);
import “DPI” function handle dequeue(handle queue);
// miscellanea
import “DPI” function bit [15:0] getStimulus();
import “DPI” context function void processTransaction(handle elem,
                    output logic [64:1] arr [0:63]);
```

1.4.5 Function 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`, `handle`, and `string`

— packed `bit` arrays up to 32 bits and all types that are eventually equivalent to packed `bit` arrays up to 32 bits.

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

1.4.6 Types of formal arguments

~~With some restrictions and with notational extensions, all SystemVerilog data types are allowed for formal arguments of imported functions.~~

Rich subset of SystemVerilog data types is allowed for formal arguments of import and export 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 functions. The set of permitted types is defined inductively.

The following SystemVerilog types are permitted for formal arguments of import and export functions:

— `void`, `byte`, `shortint`, `int`, `longint`, `real`, `shortreal`, `handle`, and `string`

— scalar values of type `bit` and `logic`

— packed one dimensional arrays of type `'bit` and `logic`
Note however, that every packed type, whatever is its structure, is eventually equivalent to a packed one dimensional array. Therefore practically all packed types are supported, although their internal structure (individual fields of structs, multiple dimensions of arrays) will be transparent and irrelevant.

— enumeration types interpreted as the type associated with an enumeration type

— types constructed from the supported types with the help of the constructs:

  — struct

  — unpacked array

  — typedef

— all and only the types listed above are permitted

The following caveat applies 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.

— 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 functions, with the exception of unpacked arrays. SystemVerilog implementation can restrict which SystemVerilog unpacked arrays are passed as actual arguments for a formal argument which 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.

1.4.6.1 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.

Although the packed part of an array can have an arbitrary number of dimensions, in the case of open arrays only a single dimension is allowed for the packed part. This is not very restrictive, however, since any packed type is eventually equivalent to one-dimensional packed array. The number of unpacked dimensions is not restricted.

If a formal argument is specified as an open array with a range of its packed or one or more of its unpacked dimensions unspecified, then the actual argument shall match the formal one — regardless of its dimensions and sizes of its linearized packed or unpacked dimensions corresponding to an unspecified range of the formal argument, respectively.

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

```
logic
bit [8:1]
bit[]
bit [7:0] b8x10 [1:10] // b8x10 is a formal arg name
logic [31:0] l32x [] // l32x is a formal arg name
logic [] lx3 [3:1] // lx3 is a formal arg name
bit [] an_unsized_array [] // an_unsized_array is a formal arg name
```

Example of complete import declarations:

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

The following example shows the use of open arrays for different sizes of actual arguments:

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

import “DPI” 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);
```

1.5 Calling imported functions

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

1.5.1 Argument passing

Argument passing for imported functions is ruled by *the WYSIWYG* principle: *What You Specify Is What You Get*, see section 1.5.1.1. 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 actual argument with the appropriate coercion; 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 A for more details.

1.5.1.1 "What You Specify Is What You Get" principle

The principle "What You Specify Is What You Get" 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 formals and the callee's declared formals. This is because the callee's formal arguments are declared in a different language than 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.

Formal arguments defined as open arrays have the size and ranges of the corresponding actual arguments, i.e., have the ranges of packed or unpacked arrays exactly as that of the actual argument. The unsized ranges of open arrays are determined at a call site; the rest of type information is specified at the import declaration.

So, if a formal argument is declared as `bit [15:8] b []`, then it is the import declaration which specifies 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.

1.5.2 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.

1.6 Exported functions

DPI allows calling SystemVerilog functions from another language. SystemVerilog functions that can be called from foreign code need to be specified in `export` declarations; such functions are referred to as *exported* functions.

Exported functions must adhere to the same restrictions on argument types and results as are imposed on imported functions. It is an error to export a function that does not satisfy such constraints. Class member functions may not be exported~~, but all other SystemVerilog functions may be exported~~.

Export declarations are allowed to occur only in the scope in which the function being exported is defined. 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.

Similarly to `import` declarations, `export` declarations can define an optional global name (`cname`) to be used as a linkage name in the foreign language when calling an exported function. For rules describing cname, see section section 1.3.

No two functions in the same SystemVerilog scope may be exported with the same explicit or implicit global name (`cname`). It is permitted, however, to use the same global name, explicit or implicit, for functions exported from different scopes as long as they have the same type signature (as defined in section 1.4.4 for imported functions).

Syntax:

export_dpi_decl ::= ***export*** *"DPI" [cname=]* ***function*** *fname* ***;***

~~cname is optional here, it defaults to fname. Note that all export functions are always *context* functions.~~