Section 1
DirectC interface

This chapter highlights the DirectC interface and provides a detailed description of the SystemVerilog-layer of the interface. The C-layer is defined in Annex A.

1.1 Introduction

The DirectC interface is an interface between SystemVerilog and a foreign programming language. It consists of two separated layers: the SystemVerilog-layer and a foreign language layer. Both sides of the DirectC interface 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 of the two tools (the SystemVerilog compiler or 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 to build a heterogeneous system (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 the existing code, usually written in C or C++, without the knowledge and the overhead of PLI or VPI. In the simplest typical scenario, a SystemVerilog design and/or a testbench is used in the non-Verilog environment (file system, etc.), which is usually programmed in C or accessible via C.

The DirectC interface 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 is also transparent. 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

The DirectC interface allows direct function calls from the other language on either side of the interface. Specifically, functions implemented in a foreign language can be called from SystemVerilog; such functions are referred to as *external functions*. SystemVerilog functions that are to be called from a foreign code shall be specified in `export` declarations (see section 1.7 for more details). The DirectC interface 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 the DirectC interface are assumed to complete their execution instantly and take `0` (zero) simulation time. **[remove:**In particular, the external functions shall be non-blocking (execute in zero-simulation time) and contain no timing control.**]** DirectC provides no means of synchronization other than by data exchange and explicit transfer of control.

Every external function needs to be declared. A declaration of an external function is referred to as an *external declaration*. External declarations are very similar to SystemVerilog function prototypes. External declarations shall occur only at the top level, i.e., in the `$root` scope. Multiple declarations of the same external function are allowed as long as they are equivalent. External functions can have zero or more formal `input`, `output`, and `inout` arguments, and they can return a result or be defined as `void` functions.

The DirectC interface is entirely based upon SystemVerilog constructs. The usage of external functions is identical as for native SystemVerilog functions. With few exceptions the external functions and the native functions are mutually exchangeable. Calls of external functions are indistinguishable from calls of SystemVerilog functions. This facilitates an 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 a foreign 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. With with some restrictions and with some notational extensions, most SystemVerilog data types are allowed in the DirectC interface. Function result types are restricted to small values, however (see section 1.4.3).

Formal arguments of an external 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 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 a more general code that can handle SystemVerilog arrays of different sizes. See section 1.4.6.

1.2 Two layers of the DirectC interface

The DirectC interface 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.

1.2.1 DirectC SystemVerilog-layer

The SystemVerilog side of the DirectC interface 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 C code. See Annex A for more details.

1.2.2 DirectC 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 translated them to and from some predefined C-like types.

The data types allowed for the formal arguments and results of the external 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 external declarations or `export` declarations. EDA 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.

1.3 Required properties of external functions

This section, 1.3, is aimed at foreign language programmers: what are they allowed to do in their code, what criteria must be met by their legacy code to be hooked-up to SV, etc. In other words, the emphasis should be on what is allowed and what is not allowed.

This section defines the semantic constraints imposed on external functions. Although no specific programming language is assumed, the external functions have some semantic restrictions. A SystemVerilog compiler is not able to verify that those restrictions are observed and if those restrictions are not satisfied, the effects of external function call can be unpredictable.

1.3.1 0-time [previously:Non-blocking] functions

External functions shall contain no timing control whatsoever, directly or indirectly. External functions shall be non-blocking; they shall complete their execution instantly and take zero-simulation time, i.e., no simulation time passes during the execution of external function.

1.3.2 `input` and `output` arguments

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

1.3.3 Special properties `pure` and `context`

Special properties can be specified for an external function: as `pure` or as `context` (see also section 1.6.3). External functions specified as `pure` 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 not to directly or indirectly (i.e., by calling other functions):

— perform any file operations

— read or write anything**revise per changes to C-layer??

— 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.

Some PLI and VPI functions require that the context of their call is known. It takes a special instrumentation of their call to provide such context; for example, some variables referring to the “current instance” or “current task” need to be set. To avoid any unnecessary overhead, external function calls in SystemVerilog code are not instrumented unless the external function is specified as `context`.

For the sake of simulation performance, an external function call shall not block SystemVerilog compiler optimizations. The SystemVerilog compiler needs to be able to determine which SystemVerilog data objects might be accessed (read or written) by a particular external function call. An external function not specified as `context` shall not access any data objects from SystemVerilog other then its actual arguments. Only the actual arguments can be affected (read or written) by its call. (This rule does not prohibit the access to non-SystemVerilog data, like global C variables.) Therefore, a call of non-`context` function does not block the optimizations.

A `context` function, however, can access (read or write) any SystemVerilog data objects by calling PLI/VPI; therefore, a call to a `context` function is a barrier for SystemVerilog compiler optimizations. Only the calls of `context` functions are properly instrumented and cause conservative optimizations; only those functions can safely call all functions from other APIs, including PLI and VPI functions or exported SystemVerilog functions.

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, due to incorrect optimizations; such calls can even crash. **revise per changes to C-layer??

1.3.4 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 external 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 a external function that directly (if it is the standard function `free`) 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 External declarations

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

An external declaration specifies the function name, function result type, and types and directions of formal arguments. It can also provide optional names and default values for formal arguments. Formal argument names can be used for argument passing by name, otherwise they serve as comments and are meaningless. An external declaration can also specify an optional function property: `context` or `pure`.

A declared external function eventually resolves to a global symbol of the same name defined outside the SystemVerilog design. Therefore, external function names need to be unique; no overloading is allowed for an external function.

The names of external functions shall follow C conventions for naming. They need to be legal global names. Additionally, an external function name shall not start with `the $` character (to avoid clashes with system tasks or functions).

External declarations can occur only in the `$root` scope. External declarations can occur anywhere in the `$root` scope, i.e., outside of modules, interfaces, functions, and tasks. An external declaration of an external function need not precede an invocation of that external function.

The scope of an external declaration is the whole design. A name of external function shall not clash with any name declared in the `$root` scope. Regular name resolving rules apply to external functions. Therefore, local names declared inside a module (or interface) shall obscure external function names; qualified names like `$root.xf_name` can be used to resolve a name to a name declared in the `$root` scope.

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

Function declarations default to the formal direction `input` if no direction has been specified. Once a direction is given, subsequent formals default to the same direction. Default data types are not allowed; data type need to be explicitly given.

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 external 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. Specifically, an actual argument can be passed *by value* or *by reference* depending on the direction and the data type of the formal argument.

The following are examples of external declarations.

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

1.4.1 Multiple declarations

An external function can have multiple declarations (i.e., declarations with the same function name) as long as they are all equivalent. This allows building a design from several components that use the same external function; each component containing its own external declaration of that function.

1.4.2 Equivalence of external declarations

Two declarations are equivalent if they specify the same function result type, the same number of formal arguments, and for each formal argument respectively, the same direction and equivalent types. SystemVerilog types equivalence rules shall be applied. If a default value is specified for a formal argument, the same default value shall be specified in both declarations. If the function property `context` or `pure` is specified, both declarations shall specify the same property.

The optional names of formal arguments are irrelevant, with the exception of functions where argument passing by name is used (see section 1.5.2). Two declarations can use different names for the same formal argument or have that argument unnamed, and do so independently for each declaration.

An external function is termed *unambiguously defined* if for every formal argument of that function either all external declarations of that function define the same name or none of them define a name of that formal argument. This is required only for functions which are called with arguments passed by name.

1.4.3 Function result

Function result types are restricted to small values. The following SystemVerilog data types are allowed for `external` function results:

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

— packed `bit` arrays up to 64 **[previously 32]** bits and all types that are eventually equivalent to packed `bit` arrays up to 64 bits.

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

1.4.4 Types of formal arguments

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

— 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 external 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.

1.4.5 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). These arrays allow the use of generic code to handle different sizes.

Formal arguments of external 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 a more general 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 external declarations:

```
extern void foo(input logic [127:0]);
extern void boo(input 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;

extern 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);
```

1.4.6 Restrictions on data representation

The DirectC interface 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.

1.4.7 Syntax of external declaration

The syntax of an external declaration is based on SystemVerilog syntax for functions, with an ANSII style port list, cf. *LRM Section 10*.

Add actual syntax here

1.5 External function calls

The usage of external functions is identical as for native SystemVerilog functions. The usage and syntax for calling external functions is identical as for native SystemVerilog functions.

1.5.1 Default values of formal arguments

External declarations can specify a default value for each formal argument with the same restrictions as for native SystemVerilog functions. All external declarations of a function shall specify the same default values (see section 1.4.2). When an external function is called, arguments with default values can be omitted from the call and the compiler shall insert their corresponding values as for native SystemVerilog functions.

1.5.2 Argument passing by position and by name

SystemVerilog allows arguments to external functions to be passed by name or by position. Arguments can be passed by name only for functions that are unambiguously defined (see section 1.4.2).

1.6 Argument passing

This section, 1.6, is aimed at SV programmers. “I’m planning to call an external function. When shall I specify it as pure or context?” Here the emphasis should be on something different. (Note that if every function is specified as context, we are on the safe side, only the performance is poor.) The situation is like this: if I specify a function as pure, I’ll enable more optimizations. When is this safe? When I specify a function as context, I’ll disable optimizations. When should I do this?

Arguments passing for external functions is ruled by *WYSIWYG* principle: *What You Specify Is What You Get*, see section 1.6.1. The evaluation order of formal arguments follows general SystemVerilog rules. Directions and types of formal arguments of external functions are never coerced, regardless of the actual argument.

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 external functions shall not be affected by the callee; the initial values of formal `output` arguments of external functions are unspecified (and can be implementation-dependent), and the necessary coercions, if any, are applied as for assignments. External 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.

1.6.1 “What You Specify Is What You Get” principle

The principle “What You Specify Is What You Get” guarantees the types of formal arguments of external 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 external declaration; they shall have ranges of packed or unpacked arrays exactly as specified in the external declaration. Only the declaration site of the external function is relevant for such formal arguments.

The formal arguments defined as open arrays have the size and ranges of the actual argument, 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 external declaration.

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

1.6.2 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 the control returns from external 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 external functions. If there is more than one argument, the order of such assignments and the related value change propagation follows general SystemVerilog rules.

1.6.3 Properties/qualifiers `pure` and `context`

The usage of the DirectC interface shall not prohibit compiler optimizations by introducing unpredictability in accessing and/or modifying SystemVerilog data. The ability to access Verilog data from an external function, directly or indirectly (for example, by calling an exported function), shall be easily visible to the compiler. To facilitate this, special properties can be specified for an external function: as `pure` or as `context`.

1.6.3.1 `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 **revise per changes to C-layer??

— 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.

1.6.3.2 `context` functions

Some PLI and VPI functions require that the context of their call is known. It takes a special instrumentation of their call to provide such context; for example, some variables referring to the “current instance” or “current task” need to be set. To avoid any unnecessary overhead, external function calls in SystemVerilog code are not instrumented unless the external function is specified as `context` in its SystemVerilog external declaration.

For the sake of simulation performance, an external function call shall not block SystemVerilog compiler optimizations. An external function not specified as `context` shall not access any data objects from SystemVerilog other then its actual arguments. Only the actual arguments can be affected (read or written) by its call. Therefore, a call of non-`context` function is not a barrier for optimizations.

A `context` function, however, can access (read or write) any SystemVerilog data objects by calling PLI/VPI; therefore, a call to a `context` function is a barrier for SystemVerilog compiler optimizations. Only the calls of `context` functions are properly instrumented and cause conservative optimizations; only those functions can safely call all functions from other APIs, including PLI and VPI functions or exported SystemVerilog functions. For 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.

An external function, unless declared as `context`, can access only those SystemVerilog data objects that are explicitly passed as actual arguments to its call (it shall not access data objects that have been not passed explicitly as arguments of the particular call). An external function declared as `context` can safely access other SystemVerilog data objects (e.g., via VPI or PLI calls). **revise per changes to C-layer??

NOTE—This might also require turning on PLI/ACC capabilities for the respective modules.

1.7 Exported functions

The DirectC interface allows calling SystemVerilog functions from another language, however, only those functions which satisfy the same restrictions for formal arguments and function result as `external` functions can be used.

SystemVerilog functions that can be called from a foreign code need to be specified in `export` declarations. `export` declarations can occur only within the `$root` scope, i.e., outside of modules, interfaces, functions and tasks.

An `export` declaration shall specify a unique instance of a module or an interface for the exported function, unless the function is a global one, i.e. declared in the `$root` scope. An `export` declaration can define an optional name to be used in the foreign language for calling the exported function.

Syntax:

```
    export_decl ::= export [cname] [modulename::] fname ;
```

`cname` is optional here, it defaults to `fname` (for example, for functions defined in the `$root` scope).

Example:

[these examples have to be replaced with new ones, they don't comply with the solution agreed upon]

```
export $root.exported_sv_func; // [the syntax needs to be checked!]
export global_func; // must be defined in $roots
export foo1 top.m1.foo; // same function from two different instances
export foo2 top.m2.foo; // exported under two different names
```