Annex A
DPI C-layer

A.1 Overview

The SystemVerilog Direct Programming Interface (DPI) allows direct inter-language function calls between SystemVerilog and any foreign programming language with a C function call protocol and linking model:

— Functions implemented in C and given import declarations in SystemVerilog can be called from SystemVerilog; such functions are referred to as *imported functions*.

— Functions implemented in SystemVerilog and specified in export declarations can be called from C; such functions are referred to as *exported functions*.

The SystemVerilog DPI supports only SystemVerilog data types, which are the sole data types that can cross the boundary between SystemVerilog and a foreign language in either direction. On the other hand, the data types used in C code shall be C types; hence, the duality of types.

A value that is passed through the Direct Programming Interface is specified in SystemVerilog code as a value of SystemVerilog type, while the same value shall be specified in C code as a value of C type. Therefore, a pair of matching type definitions is required to pass a value through DPI: the SystemVerilog definition and the C definition.

It is the user's responsibility to provide these matching definitions. A tool (such as a SystemVerilog compiler) can facilitate this by generating C type definitions for the SystemVerilog definitions used in DPI for imported and exported functions.

Some SystemVerilog types are directly compatible with C types; defining a matching C type for them is straightforward. There are, however, SystemVerilog-specific types, namely packed types (arrays, structures, and unions), 2-state or 4-state, which have no natural correspondence in C. DPI does not require any particular representation of such types and does not impose any restrictions on SystemVerilog implementations. This allows implementors to choose the layout and representation of packed types that best suits their simulation performance.

While not specifying the actual representation of packed types, this C-layer interface defines a canonical representation of packed 2-state and 4-state arrays. This canonical representation is actually based on legacy Verilog Programming Language Interface's (PLI's) `avalue`/`bvalue` representation of 4-state vectors. Library functions provide the translation between the representation used in a simulator and the canonical representation of packed arrays. There are also functions for bit selects and limited part selects for packed arrays, which do not require the use of the canonical representation.

Formal arguments in SystemVerilog can be specified as open arrays solely in import declarations; exported SystemVerilog functions can not 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 in SystemVerilog by using empty square brackets (`[]`)). This corresponds to a relaxation of the DPI argument-matching rules (section 1.5.1). An actual argument shall match the corresponding formal argument regardless of the range(s) for its corresponding dimension(s), which facilitates writing generalized C code that can handle SystemVerilog arrays of different sizes.

The C-layer of DPI basically uses normalized ranges. *Normalized ranges* mean `[n-1:0]` indexing for the packed part (packed arrays are restricted to one dimension) and `[0:n-1]` indexing for a dimension in the unpacked part of an array. Normalized ranges are used for the canonical representation of packed arrays in C and for System Verilog arrays passed as actual arguments to C, with the exception of actual arguments for open arrays. The elements of an open array can be accessed in C by using the same range of indices as defined in System Verilog for the actual argument for that open array and the same indexing as in SystemVerilog.

Function arguments are generally passed by some form of reference, or by value. All formal arguments, except open arrays, are passed by direct reference or value, and, therefore, are directly accessible in C code. Only small values of SystemVerilog input arguments (see section A.7.7) are passed by value. Formal arguments

declared in SystemVerilog as open arrays are passed by a handle (type `svOpenArrayHandle`) and are accessible via library functions. Array-querying functions are provided for open arrays.

Depending on the data types used for imported or exported functions, either binary level or C-source level compatibility is granted. Binary level is granted for all data types that do not mix SystemVerilog packed and unpacked types and for open arrays which can have both packed and unpacked parts. If a data type that mixes SystemVerilog packed and unpacked types is used, then the C code needs to be re-compiled using the implementation-dependent definitions provided by the vendor.

The C-layer of the Direct Programming Interface provides two include files. The main include file, `svdpi.h`, is implementation-independent and defines the canonical representation, all basic types, and all interface functions. The second include file, `svdpi_src.h`, defines only the actual representation of packed arrays and, hence, its contents are implementation-dependent. Applications that do not need to include this file are binary-level compatible.

A.2 Naming conventions

All names introduced by this interface shall conform to the following conventions.

— All names defined in this interface are prefixed with `sv or SV_`.

— Function and type names start with `sv`, followed by initially capitalized words with no separators, e.g., `svBitPackedArrRef`.

— Names of symbolic constants start with `sv_`, e.g., `sv_x`.

— Names of macro definitions start with `SV_`, followed by all upper-case words separated by a underscore (_), e.g., `SV_CANONICAL_SIZE`.

A.3 Portability

Depending on the data types used for ~~the~~ imported or exported functions, the C code can be binary-level or source-level compatible. Applications that do not use SystemVerilog packed types are always binary compatible. Applications that don't mix SystemVerilog packed and unpacked types in the same data type can be written to guarantee binary compatibility. Open arrays with both packed and unpacked parts are also binary compatible.

The values of SystemVerilog packed types can be accessed via interface functions using the canonical representation of 2-state and 4-state packed arrays, or directly through pointers using the implementation representation. The former mode assures binary level compatibility; the latter one allows for tool-specific, performance-oriented tuning of an application, though it also requires recompiling with the implementation-dependent definitions provided by the vendor and shipped with the simulator.

A.3.1 Binary compatibility

Binary compatibility means an application compiled for a given platform shall work with every SystemVerilog simulator on that platform.

A.3.2 Source-level compatibility

Source-level compatibility means an application needs to be re-compiled for each SystemVerilog simulator and implementation-specific definitions shall be required for the compilation.

A.4 Include files

The C-layer of the Direct Programming Interface defines two include files corresponding to these two levels of compatibility: `svdpi.h` and `svdpi_src.h`.

Binary compatibility of an application depends on the data types of the values passed through the interface. If all corresponding type definitions can be written in C without the need to include the `svdpi_src.h` file, then an application is binary compatible. If the `svdpi_src.h` file is required, then the application is not binary compatible and needs to be recompiled for each simulator of choice.

Applications that pass solely C-compatible data types or standalone packed arrays (both 2-state and 4-state) require only the `svdpi.h` file and, therefore, are binary compatible with all simulators. Applications that use complex data types which are constructed of both SystemVerilog packed arrays and C-compatible types also require the `svdpi_src.h` file and, therefore, are not binary compatible with all simulators. They are source-level compatible, however. If an application is tuned for a particular vendor-specific representation of packed arrays and therefore needs vendor specific include files, then such an application is not source-level compatible.

A.4.1 `svdpi.h` include file

Applications which use the Direct Programming Interface with C code usually need this main include file. The include file `svdpi.h` defines the types for canonical representation of 2-state (`bit`) and 4-state (`logic`) values and passing references to SystemVerilog data objects. The file also provides function headers and defines a number of helper macros and constants.

This document fully defines the `svdpi.h` file. The content of `svdpi.h` does not depend on any particular implementation or platform; all simulators shall use the same file. For more details on `svdpi.h`, see section A.9.1.

Applications which only use `svdpi.h` shall be binary-compatible with all SystemVerilog simulators.

A.4.2 `svdpi_src.h` include file

This is an auxiliary include file. `svdpi_src.h` defines data structures for implementation-specific representation of 2-state and 4-state SystemVerilog packed arrays. The interface specifies the contents of this file, i.e., what symbols are defined. The actual definitions of those symbols, however, are implementation-specific and shall be provided by vendors.

Applications that require the `svdpi_src.h` file are only source-level compatible, i.e., they need to be compiled with the version of `svdpi_src.h` provided for a particular implementation of SystemVerilog. If, however, an application makes use of the details of the implementation-specific representation of packed arrays and thus it requires vendor specific include files, then such an application is not source-level compatible.

A.5 Semantic constraints

Note that the constraints expressed here merely restate those expressed in section 1.4.1

Formal and actual arguments of both imported functions and exported functions are bound by the principle "What You Specify Is What You Get." This principle is binding both for the caller and for the callee, in C code and in SystemVerilog code. For the callee, it guarantees the actual arguments are as specified for the formal ones. For the caller, it means the function call arguments shall conform with the types of the formal arguments, which might require type-coercion on the caller side.

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 (see section A.6.2).

In SystemVerilog code, the compiler can change the formal arguments of a native SystemVerilog function and modify its code accordingly, because of optimizations, compiler pragmas, or command line switches. The situation is different for imported functions. A SystemVerilog compiler can not modify the C code, perform any coercions, or make any changes whatsoever to the formal arguments of an imported function.

A SystemVerilog compiler will provide any necessary coercions for the actual arguments of every imported function call. For example, a SystemVerilog compiler might truncate or extend bits of a packed array if the widths of the actual and formal arguments are different. Similarly, a C compiler can provide coercion for C types based on the relationship of the arguments in the exported function's C prototype (formals) and the exported function's C call site (actuals). However, a C compiler can not provide such coercion for SystemVerilog types.

Thus, in each case of an interlanguage function call, either C to SystemVerilog or SystemVerilog to C, the compilers expect but cannot enforce that the types on either side are compatible. It is therefore the user's responsibility to ensure that the imported/exported function types exactly match the types of the corresponding functions in the foreign language.

A.5.1 Types of formal arguments

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 imported declaration; they shall have ranges of packed or unpacked arrays exactly as specified in the imported declaration. Only the SystemVerilog declaration site of the imported function is relevant for such formal arguments.

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 the type information is specified at the import declaration. See also section A.6.1.

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.

A.5.2 `input` arguments

Formal arguments specified in SystemVerilog as `input` must not be modified by the foreign language code. See also section 1.4.1.2

A.5.3 `output` arguments

The initial values of formal arguments specified in SystemVerilog as `output` are undetermined and implementation-dependent. See also section 1.4.1.2

A.5.4 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 C code to SystemVerilog code.

A.5.5 `context` and non-`context` functions

Also refer to section 1.4.3

Some DPI imported functions or other interface functions called from them require that the context of their call be known. It takes special instrumentation of their call instances to provide such context; for example, a 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` in its SystemVerilog import declaration.

All DPI export 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 in the simulator's elaborated database. Thus, there is no such thing as a non-context export function.

For the sake of simulation performance, a non-context 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 then its actual arguments. Only the actual arguments can be affected (read or written) by its call. Therefore, a call of non-`context` imported 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, nor by calling an embedded export function. Therefore, a call to a `context` function is a barrier for SystemVerilog compiler optimizations.

Only the calls of `context` imported functions are properly instrumented and cause conservative optimiza-

tions; 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.

Special DPI utility functions exist that allow imported functions to retrieve and operate on their context. For example, the C implementation of an imported function may use `svGetScope()` to retrieve an svScope corresponding to the instance scope of its corresponding SystemVerilog import declaration. See section A.8 for more details.

A.5.6 `pure` functions

See also 1.4.2

Only non-void functions with no `output` or `inout` arguments can be specified as `pure`. Functions specified as `pure` in their corresponding SystemVerilog import 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 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.

A.5.7 Memory management

See also section 1.4.1.4

The memory spaces owned and allocated by C code and SystemVerilog code are disjoined. Each side is responsible for its own allocated memory. Specifically, C code shall not free the memory allocated by SystemVerilog code (or the SystemVerilog compiler) nor expect SystemVerilog code to free the memory allocated by C code (or the C compiler). This does not exclude scenarios in which C 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 C 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 C code, even when the standard functions `malloc` and `free` are called directly from SystemVerilog code.

A.6 Data types

This section defines the data types of the C-layer of the Direct Programming Interface.

A.6.1 Limitations

Packed arrays can have an arbitrary number of dimensions; though they are eventually always equivalent to a one-dimensional packed array and treated as such. If the packed part of an array in the type of a formal argument in SystemVerilog is specified as multi-dimensional, the SystemVerilog compiler linearizes it. Although the original ranges are generally preserved for open arrays, if the actual argument has a multi-dimensional packed part of the array, it will be normalized into an equivalent one-dimensional packed array.

NOTE—The actual argument can have both packed and unpacked parts of an array; either can be multidimensional.

A.6.2 Duality of types: SystemVerilog types vs. C types

A value that crosses the Direct Programming Interface is specified in SystemVerilog code as a value of SystemVerilog type, while the same value shall be specified in C code as a value of C type. Therefore, each data type that is passed through the Direct Programming Interface requires two matching type definitions: the SystemVerilog definition and C definition.

The user needs to provide such matching definitions. Specifically, for each SystemVerilog type used in the import declarations or export declarations in SystemVerilog code, the user shall provide the equivalent type definition in C reflecting the argument passing mode for the particular type of SystemVerilog value and the direction (`input`, `output`, or `inout`) of the formal SystemVerilog argument. For values passed by reference, a generic pointer `void *` can be used (conveniently `typedefed` in `svdpi.h` or `svdpi_src.h`) without knowing the actual representation of the value.

A.6.3 Data representation

DPI imposes the following additional restrictions on the representation of SystemVerilog data types.

— SystemVerilog types that are not packed and that do not contained packed elements have C compatible representation.

— Basic integer and real data types are represented as defined in section A.6.4.

— Enumeration types are represented as the types associated with them. Enumerated names are not available on C side of interface.

— Representation of packed types is implementation-dependent.

— Unpacked arrays embedded in a structure have C compatible layout regardless of the type of elements. Similarly, standalone arrays passed as actuals to a sized formal argument have C compatible representation.

— Standalone array passed as an actual to an open array formal

  — if the element type is scalar or packed then the representation is implementation dependent

  — otherwise the representation is C compatible. Therefore an element of an array shall have the same representation as an individual value of the same type. Hence, an array's elements can be accessed from C code via normal C array indexing similarly to doing so for individual values.

— The natural order of elements for each dimension in the layout of an unpacked array shall be used, i.e., elements with lower indices go first. For SystemVerilog range `[L:R]`, the element with SystemVerilog index `min(L,R)` has the C index 0 and the element with SystemVerilog index `max(L,R)` has the C index `abs(L-R)`.

NOTE—This does not actually impose any restrictions on how unpacked arrays are implemented; it only says an array that does not satisfy this condition shall not be passed as an actual argument for a formal argument which is a sized array; it can be passed, however, for a formal argument which is 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; this seems to be a reasonable trade-off.

A.6.4 Basic types

Table A1 on page 36 defines the mapping between the basic SystemVerilog data types and the corresponding C types.
Revise this xref w/ Stu; also check/revise variable settings, etc.

Table A1—Mapping data types

SystemVerilog type	C type
`byte`	`char`
`shortint`	`short int`

Table A1—Mapping data types (continued)

SystemVerilog type	C type
`int`	`int`
`longint`	`long long`
`real`	`double`
`shortreal`	`float`
`handle`	`void*`
`string`	`char*`

The representation of SystemVerilog-specific data types like packed `bit` and `logic` arrays is implementation-dependent and generally transparent to the user. Nevertheless, for the sake of performance, applications can be tuned for a specific implementation and make use of the actual representation used by that implementation; such applications shall not be binary compatible, however.

A.6.5 Normalized ranges

Packed arrays are treated as one-dimensional; the unpacked part of an array can have an arbitrary number of dimensions. *Normalized ranges* mean `[n-1:0]` indexing for the packed part and `[0:n-1]` indexing for a dimension of the unpacked part of an array. Normalized ranges are used for accessing all arguments but open arrays. The canonical representation of packed arrays also uses normalized ranges.

A.6.6 Mapping between SystemVerilog ranges and normalized ranges

The SystemVerilog ranges for a formal argument specified as an open array are those of the actual argument for a particular call. Open arrays are accessible, however, by using their original ranges and the same indexing as in the SystemVerilog code.

For all other types of arguments, i.e., all arguments but open arrays, the SystemVerilog ranges are defined in the corresponding SystemVerilog import or export declaration. Normalized ranges are used for accessing such arguments in C code. The mapping between SystemVerilog ranges and normalized ranges is defined as follows.

1) If a packed part of an array has more than one dimension, it is linearized as specified by the equivalence of packed types (see section ??).

2) A packed array of range `[L:R]` is normalized as `[abs(L-R):0]`; its most significant bit has a normalized index `abs(L-R)` and its least significant bit has a normalized index 0.

3) The natural order of elements for each dimension in the layout of an unpacked array shall be used, i.e., elements with lower indices go first. For SystemVerilog range `[L:R]`, the element with SystemVerilog index `min(L,R)` has the C index 0 and the element with SystemVerilog index `max(L,R)` has the C index `abs(L-R)`.

NOTE—The above range mapping from SystemVerilog to C applies to calls made in both directions, i.e., SystemVerilog-calls to C and C-calls to SystemVerilog.

For example, if `logic [2:3][1:3][2:0] b [1:10] [31:0]` is used in SystemVerilog, it needs to be defined in C as if it were declared in SystemVerilog in the following normalized form: `logic [17:0] b [0:9] [0:31]`.

A.6.7 Canonical representation of packed arrays

The Direct Programming Interface defines the canonical representation of packed 2-state (type `svBitVec32`) and 4-state arrays (type `svLogicVec32`). This canonical representation is derived from on the Verilog leg-

acy PLI's `avalue/bvalue` representation of 4-state vectors. Library functions provide the translation between the representation used in a simulator and the canonical representation of packed arrays.

A packed array is represented as an array of one or more elements (of type `svBitVec32` for 2-state values and `svLogicVec32` for 4-state values), each element representing a group of 32 bits.The first element of an array contains the 32 least-significant bits, next element contains the 32 more-significant bits, and so on. The last element may contain a number of unused bits. The contents of these unused bits is undetermined and the user is responsible for the masking or the sign extension (depending on the sign) for the unused bits.

Table A2 on page 38 defines the encoding used for a packed `logic` array represented as `svLogicVec32`.

Table A2—Encoding of bits in `svLogicVec32`

c	d	Value
0	0	0
0	1	1
1	0	z
1	1	x

A.7 Argument passing modes

This section defines the ways to pass arguments in the C-layer of the Direct Programming Interface.

A.7.1 Overview

Imported and exported function arguments are generally passed by some form of a reference, with the exception of small values of SystemVerilog input arguments (see section A.7.7), which are passed by value. Similarly, the function result, which is restricted to small values, is passed by value, i.e., directly returned.

~~The~~ Actual arguments passed by reference typically are passed without changing their representation from the one used by a simulator. There is no inherent copying of arguments (other than any copying resulting from coercing).

~~The~~ Access to packed arrays via ~~the~~ canonical representation involves copying arguments and does incur some overhead, however. Alternatively, for the sake of performance the application can be tuned for a particular tool and access the packed arrays directly through pointers using implementation representation, which could compromise binary and/or source compatibility. Data can be, however, moved around (copied, stored, retrieved) without using canonical representation while preserving binary or source level compatibility at the same time. This is possible by using pointers and size of data and when the detailed knowledge of the data representation is not required.

NOTE—This provides some degree of flexibility and allows the user to control the trade-off of performance vs. portability.

Formal arguments, except open arrays, are passed by direct reference or value, and, therefore, are directly accessible in C code. Formal arguments declared in SystemVerilog as open arrays are passed by a handle (type `svOpenArrayHandle`) and are accessible via library functions.

A.7.2 Calling SystemVerilog functions from C

There is no difference in argument passing between calls from SystemVerilog to C and calls from C to SystemVerilog. Functions exported from SystemVerilog can not have open arrays as arguments. Apart from this restriction, the same types of formal arguments can be declared in SystemVerilog for exported functions and imported functions. A function exported from SystemVerilog shall have the same function header in C as would an imported function with the same function result type and same formal argument list. In the case of arguments passed by reference, an actual argument to SystemVerilog function called from C shall be allocated using the same layout of data as SystemVerilog uses for that type of argument; the caller is responsible for the

allocation. It can be done while preserving ~~the~~ binary compatibility, see section A.7.5 and section A.11.11.

A.7.3 Argument passing by value

Only small values of formal input arguments (see section A.7.7) are passed by value. Function results are also directly passed by value. The user needs to provide the C-type equivalent to the SystemVerilog type of a formal argument if an argument is passed by value.

A.7.4 Argument passing by reference

For arguments passed by reference, their original simulator-defined representation shall be used and a reference (a pointer) to the actual data object is passed. The actual argument is usually allocated by a caller. The caller can also pass ~~over~~ a reference to an object already allocated somewhere else, for example, its own formal argument passed by reference.

If an argument of type `T` is passed by reference, the formal argument shall be of ~~the~~ type `T*`. However, packed arrays can also be passed using generic pointers `void*` (`typedef`ed accordingly to `svBitPackedArrRef` or `svLogicPackedArrRef`).

A.7.5 Allocating actual arguments for SystemVerilog-specific types

This is relevant only for calling exported SystemVerilog functions from C code. The caller is responsible for allocating any actual arguments that are passed by reference.

Static allocation requires ~~the~~ knowledge of the relevant data type. If such a type involves SystemVerilog packed arrays, their actual representation needs to be known to C code; thus, the file `svdpi_src.h` needs to be included, which makes the C code implementation-dependent and not binary compatible.

Sometimes ~~the~~ binary compatibility can be achieved by using dynamic allocation functions. The functions `svSizeOfLogicPackedArr()` and `svSizeOfBitPackedArr()` provide the size of the actual representation of a packed array, which can be used for the dynamic allocation of an actual argument without compromising the portability (see section A.11.11). Such a technique does not work if a packed array is a part of another type.

A.7.6 Argument passing by handle - open arrays

Arguments specified as open (unsized) arrays are always passed by a handle, regardless of direction of the SystemVerilog formal argument, and are accessible via library functions. The actual implementation of a handle is simulator-specific and transparent to the user. A handle is represented by the generic pointer `void *` (`typedef`ed to `svOpenArrayHandle`). Arguments passed by handle shall always have a `const` qualifier, because the user shall not modify the contents of a handle.

A.7.7 `input` arguments

`input` arguments of imported functions implemented in C shall always have a `const` qualifier.

`input` arguments, with the exception of open arrays, are passed by value or by reference, depending on the size. ‘Small’ values of formal input arguments are passed by value. The following data types are considered *small*:

— `byte`, `shortint`, `int`, `longint`, `real`, `shortreal`

— `handle`, `string`

— `bit` (i.e., 2-state) packed arrays up to 32 ~~64 **[previously 32]**~~ bit (canonical representation shall be used, like for a function result).

~~**[There is a problem here: ‘int’ is the same as svBitVec32, long long is not the same as svBitVec32[2], so how to return a value in the canonical representation as a function result, if this value is between 33 and 64 bits?]**~~

`input` arguments of other types are passed by reference.

If an `input` argument is a packed `bit` array passed by value, its value shall be represented using the canonical representation `svBitVec32`. If the size is smaller than 32 bits, the most significant bits are unused and their contents are undetermined. The user is responsible for the masking or the sign extension, depending on the sign, for the unused bits.

A.7.8 `inout` and `output` arguments

`inout` and `output` arguments, with the exception of open arrays, are always passed by reference. The same rules about unused bits apply as in section A.7.7

A.7.9 Function result

Types of a function result are restricted to the following SystemVerilog data types (see Table A1 on page 36 for the corresponding C type):

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

— packed `bit` arrays up to 32 bits.

If the function result type is a packed `bit` array, the returned value shall be represented using the canonical representation `svBitVec32`. If a packed `bit` array is smaller than 32 bits, the most significant bits are unused and their contents are undetermined.

A.8 Context functions

Some DPI imported functions require that the context of their call is known. For example, those calls may be associated with instances of C models that have a one-to-one correspondence with instances of SystemVerilog modules that are making the calls. Alternatively, a DPI imported function may need to access or modify simulator data structures using PLI or VPI calls, or by making a call back into SystemVerilog via an export function. Context knowledge is required for such calls to function properly. It may take special instrumentation of their call to provide such context.

To avoid any unnecessary overhead, imported function calls in SystemVerilog code are not instrumented unless the imported function is specified as `context` in its SystemVerilog import declaration. A small set of DPI utility functions is available to assist programmers when working with context functions (See section A.8.3). If those utility functions are used with any non-context function, a system error will result.

A.8.1 Overview of DPI and VPI context

Both DPI functions and VPI/PLI functions may need to understand their context. However, the meaning of the term is different for the two categories of functions.

DPI imported functions are essentially proxies for native SystemVerilog functions. Native SystemVerilog functions always operate in the scope of their declaration site. For example, a native SystemVerilog function `f()` may be declared in a module `m` which is instantiated as `top.i1_m`. The `top.i1_m` instance of `f()` may be called via hierarchical reference from code in a distant design region. Function `f()` is said to execute in the *context* (aka. instantiated scope) of `top.i1_m`, since it has unqualified visibility only for variables local to that specific instance of `m`. Function `f()` does not have unqualified visibility for any variables in the calling code's scope.

DPI imported functions follow the same model as native SystemVerilog functions. They execute in the context of their surrounding declarative scope, rather than the context of their call sites. This type of context is termed *DPI context*.

This is in contrast to VPI and PLI functions. Such functions execute in a context associated with their call sites. The VPI/PLI programming model relies on C code's ability to retrieve a context handle associated with the associated system task's call site, and then work with the context handle to glean information about arguments, items in the call site's surrounding declarative scope, etc. This type of context is termed *VPI context*.

Note that all DPI export functions require that the context of their call is known. This occurs since SystemVerilog function declarations always occur in instantiable scopes, hence giving rise to a multiplicity of associated

function instances in the simulator's database. Thus, there is no such thing as a non-context export function. All export function calls must have their execution scope specified in advance by use of a context-setting API function.

A.8.2 Context of imported and export functions

DPI imported and export functions may be declared anywhere a normal SystemVerilog function may be declared. Specifically, this means that they can be declared in `module`, `program`, `interface`, or `generate` declarative scope.

A context imported function executes in the context of the instantiated scope surrounding its declaration. This means that such functions can see other variables in that scope without qualification. As explained in section A.8.1, this should not be confused with the context of the function's call site, which may actually be anywhere in the SystemVerilog design hierarchy. The context of an imported or exported function corresponds to the fully qualified name of the function, minus the function name itself.

~~Note that context is transitive through imported and export context functions. That is, if an imported function is running in a certain context, and if it in turn calls an exported function, the exported function will inherit the context from the imported function.~~ Note that context is transitive through imported and export context functions declared in the same scope. That is, if an imported function is running in a certain context, and it in turn calls an exported function that is available in the same context, the exported function can be called without any use of `svSetScope`(). For example, consider a SystemVerilog call to a native function f(), which in turn calls a native function g(). Now replace the native function f() with an equivalent imported context C function, f'(). The system will behave identically regardless if f() or f'() is in the call chain above g(). g() has the proper execution context in both cases.

A.8.3 Working with DPI context functions in C code

DPI defines a small set of functions to help programmers work with DPI context functions. The term scope is used in the function names for consistency with other SystemVerilog terminology. The terms *scope* and *context* are equivalent for DPI functions.

There are functions that allow the user to retrieve and manipulate the current operational scope. It is an error to use these functions with any C code that is not executing under a call to a DPI context imported function.

There are also functions that provide users with the power to set data specific to C models into the SystemVerilog simulator for later retrieval. These are the "put" and "get" user data functions, which are similar to facilities provided in VPI and PLI.

The put and get user data functions are flexible and allow for a number of use models. Users may wish to share user data across multiple context imported functions defined in the same SV scope. Users may wish to have unique data storage on a per function basis. Shared or unique data storage is controllable by a user-defined key.

To achieve shared data storage, a related set of context imported functions should all use the same userKey. To achieve unique data storage, a context import function should use a unique key. Note that it is a requirement on the user that such a key be truly unique from all other keys that could possibly be used by C code. This includes completely unknown C code that could be running in the same simulation. It is suggested that taking addresses of static C symbols (such as a function pointer, or address of some static C data) always be done for user key generation. Generating keys based on arbitrary integers is not a safe practice.

~~Note that it is never possible to share user data storage across different contexts. For example, if a Verilog module 'm' declares a context imported function 'f', and 'm' is instantiated more than once in the SystemVerilog design, then 'f' will execute under different values of svScope. No such executing instances of 'f' can share user data with each other, at least not using the system-provided user data storage area accessible via svPutUserData().~~

A user wanting to share a data area across multiple contexts must do so by allocating the common data area then storing the pointer to it individually for each of the contexts in question via multpile calls to `svPutUserData`(). This is because, although a common user key can be used, the data must be associated with the individual scopes (denoted by `svScope`) of those contexts.


```
/* Functions for working with DPI context functions */

/* Retrieve the active instance scope currently associated with the executing
 * imported function.
 * Unless a prior call to svSetScope has occured, this is the scope of the
 * function's declaration site, not call site.
 * The return value is undefined if this function is invoked from a non-context
 * imported function.
 */
svScope svGetScope();

/* Set context for subsequent export function execution.
 * This function must be called before calling an export function, unless
 * the export function is called while executing an extern function. In that
 * case the export function will inherit the scope of the surrounding extern
 * function. This is known as the "default scope".
 * The return is the previous active scope (as per svGetScope)
 */
svScope svSetScope(const svScope scope);

/* Gets the fully qualified name of a scope handle */
const char* svGetNameFromScope(const svScope);

/* Retrieve svScope to instance scope of an arbitrary function declaration.
 * (Will be either module, program, interface, or generate scope)
 * The return value will be NULL for unrecognized scope names.
 */
svScope svGetScopeFromName(const char* scopeName);

/* Store an arbitrary user data pointer for later retrieval by svGetUserData()
 * The userKey is generated by the user. It must be guaranteed by the user to
 * be unique from all other userKey's for all unique data storage requirements
 * It is recommended that the address of static functions or variables in the
 * user's C code be used as the userKey.
 * It is illegal to pass in NULL values for either the scope or userData
 * arguments. It is also an error to call svPutUserData() with an invalid
 * svScope. This function returns -1 for all error cases, 0 upon success. It is
 * suggested that userData values of 0 (NULL) not be used as otherwise it will
 * be impossible to discern error status returns when calling svGetUserData()
 */
int svPutUserData(const svScope scope, void *userKey, void* userData);

/* Retrieve an arbitrary user data pointer that was previously
 * stored by a call to svPutUserData().  See the comment above
 * svPutUserData() for an explanation of userKey, as well as
 * restrictions on NULL and illegal svScope and userKey values.
 * This function returns NULL for all error cases and non-NULL
 * user data pointer upon success.
 * This function also returns NULL in the event that a prior call
 * to svPutUserData() was never made.
 */
void* svGetUserData(const svScope scope, void* userKey);

/* Returns the file and line number in the SV code from which the extern call
 * was made. If this information available, returns TRUE and updates fileName
 * and lineNumber to the appropriate values. Behavior is unpredictable if
 * fileName or lineNumber are not appropriate pointers. If this information is
 * not available return FALSE and contents of fileName and lineNumber not
```

```
     * modified. Whether this information is available or not is implementation
     * specific. Note that the string provided (if any) is owned by the SV
     * implementation and is valid only until the next call to any SV function.
     * Applications must not modify this string or free it
     */
    int svGetCallerInfo(char **fileName, int *lineNumber);
```

A.8.4 Example 1 — Using DPI context functions

```
SV Side:
      // Declare an imported context sensitive C function with cname “MyCFunc”
      import “DPI” context MyCFunc = function integer MapID(int portID);

C Side:
         // Define the function and model class on the C++ side:
         class MyCModel {
         private:
            int locallyMapped(int portID); // Does something interesting...
         public:
            // Constructor
            MyCModel(const char* instancePath) {
               svScope svScope = svGetScopeByName(instancePath);

               // Associate “this” with the corresponding SystemVerilog scope
               // for fast retrieval during runtime.
               svPutUserData(svScope, (void*) MyCFunc, this);
            }

         friend int MyCFunc(int portID);
      };

      // Implementation of imported context function callable in SV
      int MyCFunc(int portID) {
         // Retrieve SV instance scope (i.e. this function’s context).
         svScope = svGetScope();

         // Retrieve and make use of user data stored in SV scope
         MyCModel* me = (MyCModel*)svGetUserData(svScope, (void*) MyCFunc);
         return me->locallyMapped(portID);
      }
```

A.8.5 Relationship between DPI and VPI/PLI interfaces

DPI allows C code to run in the context of a SystemVerilog simulation, thus it is natural for users to consider using VPI/PLI C code from within imported functions.

There is no specific relationship defined between DPI and the existing Verilog programming interfaces (VPI and PLI). Programmers must make no assumptions about how DPI and the other interfaces interact. In particular, note that a vpiHandle is not equivalent to an svOpenArrayHandle, and the two must not be interchanged and passed between functions defined in two different interface standards.

If a user wants to call VPI or PLI functions from within an imported function, the imported function must be flagged with the context qualifier.

Not all VPI or PLI functionality is available from within DPI context imported functions. For example, a SystemVerilog imported function is not a system task, and thus making the following call from within an imported function would result in an error:

```
      /* Get handle to system task call site in preparation for argument scan */
```



```
vpiHandle myHandle = vpi_handle(vpiSysTfCall, NULL);
```

Similarly, receiving `misctf` callbacks and other activities associated with system tasks are not supported inside DPI imported functions. Users should use VPI or PLI if they wish to accomplish such actions.

However, the following kind of code is guaranteed to work from within DPI context imported functions:

```
/* Prepare to scan all top level modules */
vpiHandle myHandle = vpi_iterate(vpiModule, NULL);
```

A.9 Include files

The C-layer of the Direct Programming Interface defines two include files. The main include file, `svdpi.h`, is implementation-independent and defines the canonical representation, all basic types, and all interface functions. The second include file, `svdpi_src.h`, defines only the actual representation of packed arrays and, hence, is implementation-dependent. Both files are shown in Annex B.

Applications which do not need to include `svdpi_src.h` are binary-level compatible.

A.9.1 Binary compatibility include file `svdpi.h`

Applications which use the Direct Programming Interface with C code usually need this main include file. The include file `svdpi.h` defines the types for canonical representation of 2-state (`bit`) and 4-state (`logic`) values and passing references to SystemVerilog data objects, provides function headers, and defines a number of helper macros and constants.

This document fully defines the `svdpi.h` file. The content of `svdpi.h` does not depend on any particular implementation or platform; all simulators shall use the same file. The following subsections (and section A.10.3.1) detail the contents of the `svdpi.h` file.

A.9.1.1 Scalars of type `bit` and `logic`

```
/* canonical representation */

#define sv_0   0
#define sv_1   1
#define sv_z   2  /* representation of 4-st scalar z */
#define sv_x   3  /* representation of 4-st scalar x */

/* common type for 'bit' and 'logic' scalars. */
typedef unsigned char svScalar;

typedef svScalar svBit;    /* scalar */
typedef svScalar svLogic;  /* scalar */
```

A.9.1.2 Canonical representation of packed arrays

```
/* 2-state and 4-state vectors, modelled upon PLI's avalue/bvalue */
#define SV_CANONICAL_SIZE(WIDTH) (((WIDTH)+31)>>5)

typedef unsigned int
        svBitVec32;/* (a chunk of) packed bit array */

typedef struct { unsigned int c; unsigned int d;};
        svLogicVec32; /* (a chunk of) packed logic array */

/* Since the contents of the unused bits is undetermined, the following macros
may be handy */
#define SV_MASK(N) (~(-1<<(N)))

#define SV_GET_UNSIGNED_BITS(VALUE,N)\
   ((N)==32?(VALUE):((VALUE)&SV_MASK(N)))
```

```
#define SV_GET_SIGNED_BITS(VALUE,N)\
   ((N)==32?(VALUE):\
    (((VALUE)&(1<<((N)1)))?((VALUE)|~SV_MASK(N)):((VALUE)&SV_MASK(N))))
```

A.9.1.3 Implementation-dependent representation

```
/* a handle to a scope (an instance of a module or an interface)*/
typedef void* svScope;

/* a handle to a generic object (actually, unsized array) */
typedef void* svOpenArrayHandle;

/* reference to a standalone packed array */
typedef void* svBitPackedArrRef;
typedef void* svLogicPackedArrRef;

/* total size in bytes of the simulator's representation of a packed array */
/* width in bits */
int svSizeOfBitPackedArr(int width);
int svSizeOfLogicPackedArr(int width);
```

A.9.1.4 Translation between the actual representation and the canonical representation

```
/* functions for translation between the representation actually used by
   simulator and the canonical representation */

/* s=source, d=destination, w=width */

/* actual <-- canonical  */
void svPutBitVec32   (svBitPackedArrRef   d, const svBitVec32*   s, int w);
void svPutLogicVec32 (svLogicPackedArrRef d, const svLogicVec32* s, int w);

/* canonical <-- actual  */
void svGetBitVec32   (svBitVec32*   d, const svBitPackedArrRef   s, int w);
void svGetLogicVec32 (svLogicVec32* d, const svLogicPackedArrRef s, int w);
```

The above functions copy the whole array in either direction. The user is responsible for providing the correct width and for allocating an array in the canonical representation. The contents of the unused bits is undetermined.

Although the put/get functionality provided for `bit` and `logic` packed arrays is sufficient, yet basic, it requires unnecessary copying of the whole packed array when perhaps only some bits are needed. For the sake of convenience and improved performance, bit selects and limited (up to 32 bits) part selects are also supported, see section A.10.3.1 and section A.10.3.2.

A.9.2 Source-level compatibility include file `svdpi_src.h`

Only two symbols are defined: the macros that allow declaring variables to represent the SystemVerilog packed arrays of type `bit` or `logic`.

```
#define SV_BIT_PACKED_ARRAY(WIDTH,NAME) ...
#define SV_LOGIC_PACKED_ARRAY(WIDTH,NAME) ...
```

The actual definitions are implementation-specific. For example, a SystemVerilog simulator might define the later macro as follows.

```
#define SV_LOGIC_PACKED_ARRAY(WIDTH,NAME) \
                              svLogicVec32 NAME [ SV_CANONICAL_SIZE(WIDTH) ]
```

A.9.3 Example 2 — binary compatible application

SystemVerilog:

```
typedef struct {int a; int b;} pair;
import “DPI” function void foo(input int i1, pair i2, output logic [63:0] o3);

export “DPI” function exported_sv_func;

function void exported_sv_func(input int i, output int o [0:7]);
   begin ...end
endfunction
```

C:

```
#include "svdpi.h"

typedef struct {int a; int b;} pair;

extern void exported_sv_func(int, int *); /* imported from SystemVerilog */

void foo(const int i1, const pair *i2, svLogicPackedArrRef o3)
{
   svLogicVec32 arr[SV_CANONICAL_SIZE(64)]; /* 2 chunks needed */
   int tab[8];

   printf("%d\n", i1);
   arr[1].c = i2->a;
   arr[1].d = 0;
   arr[2].c = i2->b;
   arr[2].d = 0;
   svPutLogicVec32 (o3, arr, 64);

   /* call SystemVerilog */
   exported_sv_func(i1, tab); /* tab passed by reference */
   ...
}
```

A.9.4 Example 3— source-level compatible application

SystemVerilog:

```
typedef struct {int a; bit [6:1][1:8] b [65:2]; int c;} triple;
   // troublesome mix of C types and packed arrays
import “DPI” function void foo(input triple i);

export “DPI” function exported_sv_func;

function void exported_sv_func(input int i, output logic [63:0] o);
   begin ... end
endfunction
```

C:

```
#include "svdpi.h"
#include "svdpi_src.h"

typedef struct {
        int a;
        sv_BIT_PACKED_ARRAY(6*8, b) [64]; /* implementation specific
                                             representation */
```

```
        int c;
        } triple;

/* Note that 'b' is defined as for 'bit [6*8-1:0] b [63:0]' */

extern void exported_sv_func(int, svLogicPackedArrRef); /* imported from
                                                          SystemVerilog */

void foo(const triple *i)
{
   int j;
   /* canonical representation */
   svBitVec32  arr[SV_CANONICAL_SIZE(6*8)]; /* 6*8 packed bits */
   svLogicVec32 aL[SV_CANONICAL_SIZE(64)];

   /* implementation specific representation */
   SV_LOGIC_PACKED_ARRAY(64, my_tab);

   printf("%d %d\n", i->a, i->c);
   for (j=0; j<64; j++) {
      svGetBitVec32(arr, (svBitPackedArrRef)&(i->b[j]), 6*8);
   ...
}
...
/* call SystemVerilog */
exported_sv_func(2, (svLogicPackedArrRef)&my_tab); /* by reference */
svGetLogicVec32(aL, (svLogicPackedArrRef)&my_tab, 64);   ... }
```

NOTE—a, b, and c are directly accessed as fields in a structure. In the case of b, which represents unpacked array of packed arrays, the individual element is accessed via the library function `svGetBitVec32()`, by passing its address to the function.

A.10 Arrays

Normalized ranges are used for accessing SystemVerilog arrays, with the exception of formal arguments specified as open arrays.

A.10.1 Multidimensional arrays

Packed arrays shall be one-dimensional. Unpacked arrays can have an arbitrary number of dimensions.

A.10.2 Direct access to unpacked arrays

Unpacked arrays, with the exception of formal arguments specified as open arrays, shall have the same layout as used by a C compiler; they are accessed using C indexing (see section A.6.6).

A.10.3 Access to packed arrays via canonical representation

Packed arrays are accessible via canonical representation; this C-layer interface provides functions for moving data between implementation representation and canonical representation (any necessary conversion is performed on-the-fly (see section A.9.1.3)), and for bit selects and limited (up to 32-bit) part selects. (Bit selects do not involve any canonical representation.)

A.10.3.1 Bit selects

This subsection defines the bit selects portion of the `svdpi.h` file (see section A.9.1 for more details).

```
/* Packed arrays are assumed to be indexed n-1:0,
   where 0 is the index of least significant bit */
```



```
/* functions for bit select */

/* s=source, i=bit-index */
svBit svGetSelectBit(const svBitPackedArrRef s, int i);
svLogic svGetSelectLogic(const svLogicPackedArrRef s, int i);

/* d=destination, i=bit-index, s=scalar */
void svPutSelectBit(svBitPackedArrRef d, int i, svBit s);
void svPutSelectLogic(svLogicPackedArrRef d, int i, svLogic s);
```

A.10.3.2 Part selects

Limited (up to 32-bit) part selects are supported. A *part select* is a slice of a packed array of types `bit` or `logic`. Array slices are not supported for unpacked arrays. Additionally, 64-bit wide part select can be read as a single value of type `unsigned long long`.

Functions for part selects only allow access (read/write) to a narrow subrange of up to 32 bits. A canonical representation shall be used for such narrow vectors. If the specified range of part select is not fully contained within the normalized range of an array, the behaviour is indetermined. .

For the convenience, bit type part selects are returned as a function result. In addition to a general function for narrow part selects (<= 32-bits), there are two specialized functions for 32 and 64 bits.

```
/*
 * functions for part select
 *
 * a narrow (<=32 bits) part select is copied between
 * the implementation representation and a single chunk of
 * canonical representation
 * Normalized ranges and indexing [n-1:0] are used for both arrays:
 * the array in the implementation representation and the canonical array.
 *
 * s=source, d=destination, i=starting bit index, w=width
 * like for variable part selects; limitations: w <= 32
 */
```

NOTE—For the sake of symmetry, a canonical representation (i.e., an array) is used both for `bit` and `logic`, although a simpler `int` can be used for `bit`-part selects (<= 32-bits):

```
/* canonical <-- actual */
void svGetPartSelectBit(svBitVec32* d, const svBitPackedArrRef s, int i,
                        int w);
svBitVec32 svGetBits(const svBitPackedArrRef s, int i, int w);
svBitVec32 svGet32Bits(const svBitPackedArrRef s, int i); // 32-bits
unsigned long long svGet64Bits(const svBitPackedArrRef s, int i); // 64-bits
void svGetPartSelectLogic(svLogicVec32* d, const svLogicPackedArrRef s, int i,
                          int w);

/* actual <-- canonical */
void svPutPartSelectBit(svBitPackedArrRef d, const svBitVec32 s, int i,
                        int w);
void svPutPartSelectLogic(svLogicPackedArrRef d, const svLogicVec32 s, int i,
                        int w);
```

A.11 Open arrays

Formal arguments specified as open arrays allows passing actual arguments of different sizes (i.e., different range and/or different number of elements), which facilitates writing more general C code that can handle SystemVerilog arrays of different sizes. The elements of an open array can be accessed in C by using the same

range of indices and the same indexing as in SystemVerilog. Plus, inquiries about the dimensions and the original boundaries of SystemVerilog actual arguments are supported for open arrays.

NOTE—Both packed and unpacked array dimensions can be unsized.

All formal arguments declared in SystemVerilog as open arrays are passed by handle (type `svOpenArrayHandle`), regardless of the direction of a SystemVerilog formal argument. Such arguments are accessible via interface functions.

A.11.1 Actual ranges

The formal arguments defined as open arrays have the size and ranges of the actual argument, as determined on a per-call basis. The programmer shall always have a choice whether to specify a formal argument as a sized array or as an open (unsized) array.

In the former case, all indices are normalized on the C side (i.e., `0` and up) and the programmer needs to know the size of an array and be capable of determining how the ranges of the actual argument map onto C-style ranges (see section A.6.6).

Tip: programmers may decide to stick to `[n:0]name[0:k]` style ranges in SystemVerilog.

In the later case, i.e., an open array, individual elements of a packed array are accessible via interface functions, which facilitate the SystemVerilog-style of indexing with the original boundaries of the actual argument.

If a formal argument is specified as a sized array, then it shall be passed by reference, with no overhead, and is directly accessible as a normalized array. If a formal argument is specified as an open (unsized) array, then it shall be passed by handle, with some overhead, and is mostly indirectly accessible, again with some overhead, although it retains the original argument boundaries.

NOTE—This provides some degree of flexibility and allows the programmer to control the trade-off of performance vs. convenience.

The following example shows the use of sized vs. unsized arrays in SystemVerilog code.

```
// both unpacked arrays are 64 by 8 elements, packed 16-bit each
logic [15: 0] a_64x8 [63:0][7:0];
logic [31:16] b_64x8 [64:1][-1:-8];

import “DPI” function void foo(input logic [] i [][]);
       // 2-dimensional unsized unpacked array of unsized packed logic

import “DPI” function void boo(input logic [31:16] i [64:1][-1:-8]);
       // 2-dimensional sized unpacked array of sized packed logic

foo(a_64x8);
foo(b_64x8); // C code may use original ranges [31:16][64:1][-1:-8]

boo(b_64x8); // C code must use normalized ranges [15:0][0:63][0:7]
```

A.11.2 Array querying functions

These functions are modelled upon the SystemVerilog array querying functions and use the same semantics (see *SystemVerilog 3.0 LRM 16.3*).

If the dimension is 0, then the query refers to the packed part (which is one-dimensional) of an array, and dimensions `> 0` refer to the unpacked part of an array.

```
/* h= handle to open array, d=dimension */
int svLeft(const svOpenArrayHandle h, int d);
int svRight(const svOpenArrayHandle h, int d);
```



```
int svLow(const svOpenArrayHandle h, int d);
int svHigh(const svOpenArrayHandle h, int d);
int svIncrement(const svOpenArrayHandle h, int d);
int svLength(const svOpenArrayHandle h, int d);
int svDimensions(const svOpenArrayHandle h);
```

A.11.3 Access functions

Similarly to sized arrays, there are functions for copying data between the simulator representation and the canonical representation and to obtain the actual address of SystemVerilog data object or of an individual element of an unpacked array. This information might be useful for simulator-specific tuning of the application.

Depending on the type of an element of an unpacked array, different access methods shall be used when working with elements.

— Packed arrays (`bit` or `logic`) are accessed via copying to or from the canonical representation.

— Scalars (1-bit value of type `bit` or `logic`) are accessed (read or written) directly.

— Other types of values (e.g., structures) are accessed via generic pointers; a library function calculates an address and the user needs to provide the appropriate casting.

— Scalars and packed arrays are accessible via pointers only if the implementation supports this functionality (per array), e.g., one array can be represented in a form that allows such access, while another array might use a compacted representation which renders this functionality unfeasible (both occurring within the same simulator).

SystemVerilog allows arbitrary dimensions and, hence, an arbitrary number of indices. To facilitate this, variable argument list functions shall be used. For the sake of performance, specialized versions of all indexing functions are provided for 1, 2, or 3 indices.

A.11.4 Access to the actual representation

The following functions provide an actual address of the whole array or of its individual elements. These functions shall be used for accessing elements of arrays of types compatible with C. These functions are also useful for vendors, because they provide access to the actual representation for all types of arrays.

If the actual layout of the SystemVerilog array passed as an argument for an open unpacked array is different than the C layout, then it is not possible to access such an array as a whole; therefore, the address and size of such an array shall be undefined (zero (`0`), to be exact). Nonetheless, the addresses of individual elements of an array shall be always supported.

NOTE—No specific representation of an array is assumed here; hence, all functions use a generic pointer `void *`.

```
/* a pointer to the actual representation of the whole array of any type */
/* NULL if not in C layout */
void *svGetArrayPtr(const svOpenArrayHandle);

int svSizeOfArray(const svOpenArrayHandle); /* total size in bytes or 0 if not
in C
                                        layout */

/* Return a pointer to an element of the array
   or NULL if index outside the range or null pointer */

void *svGetArrElemPtr(const svOpenArrayHandle, int indx1, ...);

/* specialized versions for 1-, 2- and 3-dimensional arrays: */
void *svGetArrElemPtr1(const svOpenArrayHandle, int indx1);
void *svGetArrElemPtr2(const svOpenArrayHandle, int indx1, int indx2);
```

```
void *svGetArrElemPtr3(const svOpenArrayHandle, int indx1, int indx2, int
indx3);
```

Access to an individual array element via pointer makes sense only if the representation of such an element is the same as it would be for an individual value of the same type. Representation of array elements of type `scalar` or *packed value* is implementation-dependent; the above functions shall return `NULL` if the representation of the array elements differs from the representation of individual values of the same type.

A.11.5 Access to elements via canonical representation

This group of functions is meant for accessing elements which are packed arrays (`bit` or `logic`).

The following functions copy a single vector from a canonical representation to an element of an open array or other way round. The element of an array is identified by indices, bound by the ranges of the actual argument, i.e., the original SystemVerilog ranges are used for indexing.

```
/* functions for translation between simulator and canonical representations*/
/* s=source, d=destination */
/* actual <-- canonical  */
void svPutBitArrElemVec32 (const svOpenArrayHandle _d, const svBitVec32* s,
                           int indx1, ...);
void svPutBitArrElem1Vec32(const svOpenArrayHandle _d, const svBitVec32* s,
int indx1);
void svPutBitArrElem2Vec32(const svOpenArrayHandle _d, const svBitVec32* s,
int indx1,
                           int indx2);
void svPutBitArrElem3Vec32(const svOpenArrayHandle d, const svBitVec32* s,
                           int indx1, int indx2, int indx3);

void svPutLogicArrElemVec32 (const svOpenArrayHandle _d, const svLogicVec32*
s,
                             int indx1, ...);
void svPutLogicArrElem1Vec32( const svOpenArrayHandle _d, const svLogicVec32*
s,
                              int indx1);
void svPutLogicArrElem2Vec32(const svOpenArrayHandle _d, const svLogicVec32*
s,
                             int indx1, int indx2);
void svPutLogicArrElem3Vec32(const svOpenArrayHandle _d, const svLogicVec32*
s,
                             int indx1, int indx2, int indx3);

/* canonical <-- actual  */
void svGetBitArrElemVec32 (svBitVec32* d, const svOpenArrayHandle _s, int
indx1, ...);
void svGetBitArrElem1Vec32(svBitVec32* d, const svOpenArrayHandle _s, int
indx1);
void svGetBitArrElem2Vec32(svBitVec32* d, const svOpenArrayHandle _s, int
indx1,
                           int indx2);
void svGetBitArrElem3Vec32(svBitVec32* d, const svOpenArrayHandle _s,
                           int indx1, int indx2, int indx3);

void svGetLogicArrElemVec32 (svLogicVec32* d, const svOpenArrayHandle _s, int
indx1,
                             ...);
void svGetLogicArrElem1Vec32(svLogicVec32* d, const svOpenArrayHandle _s, int
indx1);
```



```
void svGetLogicArrElem2Vec32(svLogicVec32* d, const svOpenArrayHandle _s, int
indx1,
                             int indx2);
void svGetLogicArrElem3Vec32(svLogicVec32* d, const svOpenArrayHandle _s,
                             int indx1, int indx2, int indx3);
```

The above functions copy the whole packed array in either direction. The user is responsible for allocating an array in the canonical representation.

A.11.6 Access to scalar elements (`bit` and `logic`)

Another group of functions is needed for scalars (i.e., when an element of an array is a simple scalar, `bit`, or `logic`):

```
svBit   svGetBitArrElem (const svOpenArrayHandle _s, int indx1, ...);
svBit   svGetBitArrElem1(const svOpenArrayHandle _s, int indx1);
svBit   svGetBitArrElem2(const svOpenArrayHandle _s, int indx1, int indx2);
svBit   svGetBitArrElem3(const svOpenArrayHandle _s, int indx1, int indx2, int
indx3);

svLogic svGetLogicArrElem (const svOpenArrayHandle _s, int indx1, ...);
svLogic svGetLogicArrElem1(const svOpenArrayHandle _s, int indx1);
svLogic svGetLogicArrElem2(const svOpenArrayHandle _s, int indx1, int indx2);
svLogic svGetLogicArrElem3(const svOpenArrayHandle _s, int indx1, int indx2,
int indx3);

void svPutLogicArrElem (const svOpenArrayHandle _d, svLogic value, int indx1,
...);
void svPutLogicArrElem1(const svOpenArrayHandle _d, svLogic value, int indx1);
void svPutLogicArrElem2(const svOpenArrayHandle _d, svLogic value, int indx1,
                        int indx2);
void svPutLogicArrElem3(const svOpenArrayHandle _d, svLogic value, int indx1,
int indx2,
                        int indx3);

void svPutBitArrElem (const svOpenArrayHandle _d, svBit value, int indx1,
...);
void svPutBitArrElem1(const svOpenArrayHandle _d, svBit value, int indx1);
void svPutBitArrElem2(const svOpenArrayHandle _d, svBit value, int indx1, int
indx2);
void svPutBitArrElem3(const svOpenArrayHandle _d, svBit value, int indx1, int
indx2,
                      int indx3);
```

A.11.7 Access to array elements of other types

If an array's elements are of a type compatible with C, there is no need to use canonical representation. In such situations, the elements are accessed via pointers, i.e., the actual address of an element shall be computed first and then used to access the desired element.

A.11.8 Example 4— two-dimensional open array

SystemVerilog:

```
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);
```

C:

```
#include "svdpi.h"

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

void foo(const svOpenArrayHandle _h)
{
   MyType my_value;
   int i, j;
   int lo1 = svLow(h, 1);
   int hi1 = svHigh(h, 1);
   int lo2 = svLow(h, 2);
   int hi2 = svHigh(h, 2);

   for (i = lo1; i <= hi1; i++) {
       for (j = lo2; j <= hi2; j++) {

          my_value = *(MyType *)svGetArrElemPtr2(h, i, j);
          ...
          *(MyType *)svGetArrElemPtr2(h, i, j) = my_value;
          ...
          }
      ...
      }
}
```

A.11.9 Example 5— open array

SystemVerilog:

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

import “DPI” function void foo(input MyType i [], output MyType o []);

MyType source [11:20];
MyType target [11:20];

foo(source, target);
```

C:

```
#include "svdpi.h"

typedef struct  ... } MyType;

void foo(const svOpenArrayHandle _hin, const svOpenArrayHandle _hout)
{
   int count = svLength(hin, 1);
   MyType *s = (MyType *)svGetArrayPtr(hin);
   MyType *d = (MyType *)svGetArrayPtr(hout);

   if (s && d) { /* both arrays have C layout */
```



```
   /* an efficient solution using pointer arithmetic */
   while (count--)
         *d++ = *s++;

   /* even more efficient:
      memcpy(d, s, svSizeOfArray(hin));
   */

} else { /* less efficient yet implementation independent */

    int i = svLow(hin, 1);
    int j = svLow(hout, 1);
    while (i <= svHigh(hin, 1)) {
        *(MyType *)svGetArrElemPtr1(hout, j++) =
                    *(MyType *)svGetArrElemPtr1(hin, i++);
    }

  }

}
```

A.11.10 Example 6 — access to packed arrays

SystemVerilog:

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

C:

```
#include "svdpi.h"

/* one 128-bit packed vector */
void foo(const svLogicPackedArrRef packed_vec_128_bit)
{
   svLogicVec32 arr[SV_CANONICAL_SIZE(128)]; /* canonical representation */

   svGetLogicVec32(arr, packed_vec_128_bit, 128);
   ...
}

/* open array of 128-bit packed vectors */
void boo(const svOpenArrayHandle _h)
{
   int i;
   svLogicVec32 arr[SV_CANONICAL_SIZE(128)]; /* canonical representation */

   for (i = svLow(h, 1); i <= svHigh(h, 1); i++) {

       svLogicPackedArrRef ptr = (svLogicPackedArrRef)svGetArrElemPtr1(h, i);
       /* user need not know the vendor representation! */

       svGetLogicVec32(arr, ptr, 128);
       ...
   }
   ...
}
```

A.11.11 Example 7 — binary compatible calls of exported functions

This example demonstrates the source compatibility include file `svdpi_src.h` is not needed if a C function dynamically allocates the data structure for simulator representation of a packed array to be passed to an exported SystemVerilog function.

SystemVerilog:

```
export“DPI” function myfunc;
...
function void myfunc (output logic [31:0] r); ...
...
```

C:

```
#include "svdpi.h"
extern void myfunc (svLogicPackedArrRef r); /* exported from SV */

   /* output logic packed 32-bits */
...
svLogicVec32 my_r[SV_CANONICAL_SIZE(32)];
/* my array, canonical representation */

/* allocate memory for logic packed 32-bits in simulator's representation */
svLogicPackedArrRef r =
   (svLogicPackedArrRef)malloc(svSizeOfLogicPackedArr(32));
myfunc(r);
/* canonical <-- actual */
svGetLogicVec32(my_r, r, 32);
/* will use only the canonical representation from now on */
free(r); /* don't need any more */
...
```