

Annex A

DirectC C-layer

A.1 Overview

The DirectC interface allows direct function calls from the other language on either side of the interface:

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

~~This section describes the C-layer of the DirectC Interface and applies to calls from either direction: C functions called from SystemVerilog code and exported SystemVerilog functions called from C code.~~

The SystemVerilog DirectC Interface 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 (i.e., when a foreign function is called from SystemVerilog code or an exported SystemVerilog function is called from a foreign code). 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 DirectC 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, passing a value through the DirectC Interface takes a pair of matching type definitions: the SystemVerilog definition and C definition.

It is the user's responsibility to provide these matching definitions. A tool (such as the SystemVerilog compiler) can facilitate this by generating C type definitions for the SystemVerilog definitions used in the DirectC interface for external 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. DirectC does not require any particular representation of such types and does not impose any restrictions on SystemVerilog implementation. 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 Verilog legacy 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 the external 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 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 C code that can handle SystemVerilog arrays of different sizes.

The C-layer of the DirectC Interface 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 the 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 a 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 `svHandle`) and are accessible via library functions. Array-querying functions are provided for open arrays.

Depending on the data types used for the external (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 DirectC Interface provides two include files. The main include file, `svc.h`, is implementation-independent and defines the canonical representation, all basic types, and all interface functions. The second include file, `svc_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 dash (-), e.g., `SV_CANONICAL_SIZE`.

A.3 Portability

Depending on the data types used for the external (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 the 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 DirectC Interface defines two include files corresponding to these two levels of compatibility: `svc.h` and `svc_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 an `svc_src.h` file, then an

application is binary compatible. If an `svc_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 an `svc.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 an `svc_src.h` file and, therefore, are not be binary compatible with all simulators. They are source-level compatible, however.

A.4.1 `svc.h` include file

Applications which use the DirectC interface with C code usually need this main include file. The include file `svc.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 `svc.h` file. The content of `svc.h` does not depend on any particular implementation or platform; all simulators shall use the same file. For more details on `svc.h`, see section A.8.1.

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

A.4.2 `svc_src.h` include file

This is an auxiliary include file. `svc_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 the vendors.

Applications that require an `svc_src.h` file are only source-level compatible, i.e., they need to be compiled with the version of `svc_src.h` provided for a particular implementation of SystemVerilog.

A.5 Semantic constraints

~~This section defines the semantic constraints of the C-layer of the DirectC Interface.~~

Formal and actual arguments of both external functions and exported functions are bound by the principle “What You Specify Is What You Get.” This principle is bounding 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.

In SystemVerilog code, the compiler can change the formal arguments of native SystemVerilog function and modify its code accordingly, because of optimizations, compiler pragmas, or command line switches. Similarly, a SystemVerilog compiler shall provide any necessary coercions for the actual arguments of every 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.

The situation is different for external and exported functions. A SystemVerilog compiler can not modify the C code or perform any coercion or make any changes whatsoever to the formal arguments of an external function. A C compiler can provide coercion only for C types, but not for SystemVerilog types; while at the same time, the SystemVerilog code of an exported function expects the types of arguments to be exactly as declared. The caller needs to guarantee this. Thus, if the user passes a 10-bit packed array to SystemVerilog, when a 40-bit array is expected, the error is not be corrected by the C compiler; although, if the same happened in SystemVerilog code, the SystemVerilog compiler would handle it correctly by providing coercion and a temporary variable.

A.5.1 Types of formal arguments

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 (SystemVerilog) of the external 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 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.

A.5.2 input arguments

The formal arguments specified in SystemVerilog as `input` shall not be modified.

A.5.3 output arguments

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

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

Some PLI and VPI functions require that the context of their call is known. It takes a special instrumentation of their call **instances** 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 than 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; therefore, 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.

External functions shall **remove** be non-blocking; they shall **complete** their execution instantly and take zero-simulation time. **remove on Doug's request** External functions, whether specified as `context` or not, shall contain no timing control whatsoever (directly or indirectly) via calling PLI or VPI functions. **complete**

A.5.6 pure functions

Only non-void functions with no `output` or `inout` arguments can be specified as `pure`. Functions specified as `pure` in their corresponding SystemVerilog external declarations shall have no side effects; their results need to depend solely on the values of their input arguments. Calls to such functions can be removed by SystemVerilog compiler optimizations or replaced with the values previously computed for the same values of the input arguments.

Specifically, a `pure` function is assumed 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

The memory spaces owned and allocated by C code and SystemVerilog code are disjointed. 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 DirectC 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 multidimensional packed part of the array, the equivalent one-dimensional packed array shall be normalized.

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 DirectC 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 DirectC 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 external 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 `svc.h` or `svc_src.h`) without knowing the actual representation of the value.

A.6.3 Data representation

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

- Basic integer and real data types are **represented as defined** in section A.6.4.
- Representation of packed types is implementation-dependent.
- The layout of **the** **unsized** (or open) standalone unpacked arrays is implementation-dependent with the following restriction:
 - an element of an array shall have the same representation as an individual value of the same type, except for scalars (`bit` or `logic`) and packed arrays.

Hence, an array's elements, other than scalars or packed arrays, can be accessed [from C code](#) via pointers similarly to doing so for individual values.

- The layout of unpacked arrays, with the exception of actual arguments passed for formal arguments specified as open arrays, is the same as used by a C compiler; this includes arrays embedded in structures and any standalone arrays (i.e., those not embedded in any structure).

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 $\text{abs}(L - R)$.

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 the formal argument which is a sized array; it can be passed, however, for 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.

[DirectC](#) also supports the following SystemVerilog data types.

- [Basic integer and real data types are represented as defined in SystemVerilog LRM sections 3.3 and 3.4.2; see also section A.6.4.](#)
- [The layout of unpacked structures is same as used by a C compiler \(see SystemVerilog LRM section 3.7\).](#)

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 |
|------------------------|------------------------|
| <code>char</code> | <code>char</code> |
| <code>byte</code> | <code>char</code> |
| <code>shortint</code> | <code>short int</code> |
| <code>int</code> | <code>int</code> |
| <code>longint</code> | <code>long long</code> |
| <code>real</code> | <code>double</code> |
| <code>shortreal</code> | <code>float</code> |
| <code>handle</code> | <code>void*</code> |
| <code>string</code> | <code>char*</code> |

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 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 external 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] [31:0]`.

A.6.7 Canonical representation of packed arrays

The DirectC interface defines the canonical representation of packed 2-state (type `svBitVec32`) and 4-state arrays (type `svLogicVec32`). This canonical representation is actually based on the Verilog legacy 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 37 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 DirectC Interface.

A.7.1 Overview

External 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 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 compatibility.

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 `svHandle`) 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. Otherwise, the same types of formal arguments can be declared in SystemVerilog for exported functions and external functions. A function exported from SystemVerilog shall have the same function header in C as the external 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 with preserving the binary compatibility, see section A.7.5 and section A.10.11.**

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 the 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 be also passed using generic pointers `void*` (typedefed 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 the 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 `svc_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.10.11). Such a technique does not work if a packed array is a part of another type.

A.7.6 Argument passing by `sv_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 *` (typedefed to `sv_handle`). 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 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*:

- `char`, `byte`, `shortint`, `int`, `longint`, `real`, `shortreal`
 - `handle`, `string`
 - `bit` (i.e., 2-state) packed arrays up to 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.

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):

- `char`, `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 Include files

The C-layer of the DirectC Interface defines two include files. The main include file, `svc.h`, is implementation-independent and defines the canonical representation, all basic types, and all interface functions. The second include file, `svc_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 `svc_src.h` are binary-level compatible.

A.8.1 Binary compatibility include file `svc.h`

Applications which use the DirectC interface with C code usually need this main include file. The include file `svc.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 `svc.h` file. The content of `svc.h` does not depend on any particular implementation or platform; all simulators shall use the same file. The following subsections (and section A.9.3.1) detail the contents of the `svc.h` file.

A.8.1.1 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.8.1.2 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; } /* as in VCS */
    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.8.1.3 Implementation-dependent representation

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

/* 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 svSizeOfLogicPackedArr(int width);
int svSizeOfBitPackedArr(int width);
```

A.8.1.4 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 the convenience and improved performance, bit selects and limited (up to 32 bits) part selects are also supported, see section A.9.3.1 and section A.9.3.2.

A.8.2 Source-level compatibility include file `svc_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, [VCS a SystemVerilog simulator](#) might define the later macro as follows.

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

A.8.3 Example 1 — binary compatible application

SystemVerilog:

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

export extern $root.exported_sv_func; // whatever is the syntax ...

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

C:

```
#include "svc.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.8.4 Example 2 — 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
extern void foo(input triple i);

export extern $root.exported_sv_func; // whatever is the syntax ...

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

```

C:

```

#include "svc.h"
#include "svc_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.9 Arrays

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

A.9.1 Multidimensional arrays

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

A.9.2 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.9.3 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.8.1.3)), and for bit selects and limited (up to 32-bit) part selects. (Bit selects do not involve any canonical representation.)

A.9.3.1 Bit selects

This subsection defines the bit selects portion of the `svc.h` file (see section A.8.1 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.9.3.2 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.

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.

```
/*
 * 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);
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.10 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 a 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 argument 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 `svHandle`), regardless of the direction of a SystemVerilog formal argument. Such arguments are accessible via interface functions.

A.10.1 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).

Hint: 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];

extern void foo(input logic [] i [][]);
    // 2-dimensional unsized unpacked array of unsized packed logic

extern 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.10.2 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 svHandle h, int d);
int svRight(const svHandle h, int d);
int svLow(const svHandle h, int d);
int svHigh(const svHandle h, int d);
int svIncrement(const svHandle h, int d);
int svLength(const svHandle h, int d);
int svDimensions(const svHandle h);
```

A.10.3 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, [a](#) variable argument list functions shall be used. For the sake of performance, [the](#) specialized versions of all indexing functions are provided for 1, 2, or 3 indices.

A.10.4 Access to the actual representation

The following functions provide an actual address of the whole array or of its individual [elements](#). These func-

tions shall be used for accessing elements of the arrays of types compatible with C. These functions are also useful for the 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 array as a whole; therefore, the address and size of such 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 svHandle);

int svSizeOfArray(const svHandle); /* 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 svHandle, int indx1, ...);

/* specialized versions for 1-, 2- and 3-dimensional arrays: */
void *svGetArrElemPtr1(const svHandle, int indx1);
void *svGetArrElemPtr2(const svHandle, int indx1, int indx2);
void *svGetArrElemPtr3(const svHandle, 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.10.5 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 svHandle d, const svBitVec32* s,
                           int indx1, ...);
void svPutBitArrElem1Vec32(const svHandle d, const svBitVec32* s, int indx1);
void svPutBitArrElem2Vec32(const svHandle d, const svBitVec32* s, int indx1,
                           int indx2);
void svPutBitArrElem3Vec32(const svHandle d, const svBitVec32* s,
                           int indx1, int indx2, int indx3);

void svPutLogicArrElemVec32 (const svHandle d, const svLogicVec32* s,
                             int indx1, ...);
void svPutLogicArrElem1Vec32(const svHandle d, const svLogicVec32* s,
                             int indx1);
void svPutLogicArrElem2Vec32(const svHandle d, const svLogicVec32* s,
                             int indx1, int indx2);
void svPutLogicArrElem3Vec32(const svHandle d, const svLogicVec32* s,
```

```

        int indx1, int indx2, int indx3);

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

void svGetLogicArrElemVec32 (svLogicVec32* d, const svHandle s, int indx1,
                             ...);
void svGetLogicArrElem1Vec32(svLogicVec32* d, const svHandle s, int indx1);
void svGetLogicArrElem2Vec32(svLogicVec32* d, const svHandle s, int indx1,
                             int indx2);
void svGetLogicArrElem3Vec32(svLogicVec32* d, const svHandle 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.10.6 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 svHandle s, int indx1, ...);
svBit    svGetBitArrElem1(const svHandle s, int indx1);
svBit    svGetBitArrElem2(const svHandle s, int indx1, int indx2);
svBit    svGetBitArrElem3(const svHandle s, int indx1, int indx2, int indx3);

svLogic  svGetLogicArrElem (const svHandle s, int indx1, ...);
svLogic  svGetLogicArrElem1(const svHandle s, int indx1);
svLogic  svGetLogicArrElem2(const svHandle s, int indx1, int indx2);
svLogic  svGetLogicArrElem3(const svHandle s, int indx1, int indx2, int indx3);

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

void svPutBitArrElem (const svHandle d, svBit value, int indx1, ...);
void svPutBitArrElem1(const svHandle d, svBit value, int indx1);
void svPutBitArrElem2(const svHandle d, svBit value, int indx1, int indx2);
void svPutBitArrElem3(const svHandle d, svBit value, int indx1, int indx2,
                        int indx3);

```

A.10.7 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.10.8 Example 3 — two-dimensional open array

SystemVerilog:

```
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);

```

C:

```

#include "svc.h"

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

void foo(const svHandle 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.10.9 Example 4 — open array

SystemVerilog:

```

typedef struct { ... } MyType;

extern void foo(input MyType i [], output MyType o []);

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

foo(source, target);

```

C:

```

#include "svc.h"

typedef struct ... } MyType;

void foo(const svHandle hin, const svHandle 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.10.10 Example 5 — access to packed arrays

SystemVerilog:

```

extern void foo(input logic [127:0]);
extern void boo(input logic [127:0] i []); // open array of 128-bit

```

C:

```

#include "svc.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 svHandle 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.10.11 Example 6 — binary compatible calls of exported functions

This example demonstrates the source compatibility include file `svc_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 a.b.c.myfunc;
...
function void myfunc (output logic [31:0] r); ...
...

```

C:

```

#include "svc.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 */
...

```