1.1 DirectC C Layer

1.1.1 Overview

This sub-section describes the C layer of DirectC Interface and applies to calls from either direction, both for C functions called from System Verilog code and for exported System Verilog functions called from C code.

The System Verilog DirectC Interface supports only System Verilog data types and they are the sole data types that may cross the boundary between System Verilog and a foreign language in either direction (i.e. when a foreign function is called from System Verilog code or when exported System Verilog function is called from a foreign code).

On the other hand, the data types used in C code must be C types. Hence the duality of types. A value that is passed through the DirectC Interface is specified in System Verilog code as a value of System Verilog type, while the same value must 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: System Verilog definition and C definition. It is the user's responsibility to provide such matching definitions. A tool (System Verilog compiler) may facilitate this by generating C type definitions for System Verilog definitions used in DirectC interface for external and exported functions.

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

While not specifying the actual representation of packed types, the C layer of DirectC defines the canonical representation of packed 2-state and 4-state arrays. This canonical representation is actually based on 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. There are also functions for bit selects and limited part selects for packed arrays, that do not require the use of the canonical representation.

Generally, normalized ranges are assumed for accessing System Verilog arrays. Normalized ranges mean `[n-1:0]` indexing for the packed part (packed arrays are restricted to 1 dimension), and `[0:n-1]` indexing for a dimension in the unpacked part of an array. The open arrays, however, are accessed using the original ranges as defined for the actual arguments in System Verilog and the same indexing as it would have been used in System Verilog.

Formal arguments in System Verilog may be specified as open arrays solely in the external declarations; exported System Verilog functions may not have formal arguments specified as open arrays. The term "open array" refers to the fact that a range of one or more dimensions of a formal argument is unspecified (what is denoted in System Verilog by `[]`). This is solely a relaxation of argument matching rules. An actual argument will match the formal one regardless of its range(s) for the corresponding dimension(s). This allows to write in C a more general code that may handle System Verilog arrays of different sizes.

The function arguments are generally passed by some form of a reference, with the exception of small values of System Verilog input arguments, which are passed by value.

All types of formal arguments but open arrays are passed either by direct reference or by value and therefore are directly accessible in C code. Passing such arguments incurs virtually no overhead.

Formal arguments declared in System Verilog as open arrays are passed by a handle, see the 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.

The binary level compatibility is granted for all data types that do not mix System Verilog packed and unpacked types. Open arrays with both packed and unpacked part are, however, the allowed exception.

If a data type that mixes System Verilog packed and unpacked types is used, then C code must be re-compiled using the implementation dependent definitions provided by the vendor.

The C layer of 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 is implementation dependent. The applications that do not need include this file are binary level compatible.

1.1.2 Naming convention

All names defined in this interface are prefixed with "sv".

Function and type names start with "sv" followed by Capitalized words with no separators, e.g. svBitPackedArrRef.

Names of symbolic constants start with "sv_", e.q. sv_x.

Names of macro definitions start with "sv_" followed by all capitalized words separated by "-", e.g. sv_CANONICAL_SIZE.

1.2 Portability

Depending on the data types used for the external (or exported) functions, C code may be either binary level or source level compatible.

The applications that do not use System Verilog packed types are always binary compatible.

Applications that don’t mix System Verilog packed and unpacked types in the same data type may always be written in a way that will guarantee the binary compatibility. Open arrays with both packed and unpacked part don’t breach the binary compatibility.

The values of System Verilog packed types may be accessed either solely via interface functions using the canonical representation of 2-state and 4-state packed arrays, or they also may be accessed directly thru pointers using the implementation representation. The former mode will assure binary level compatibility. The later one will allow for tool-specific performance oriented tuning of an application though it will require recompilation with the implementation dependent definitions provided by the vendor and shipped with the simulator.

1.2.1 Binary compatibility

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

1.2.2 Source level compatibility

Source level compatibility means that an application will have to be re-compiled for each System Verilog simulator and that the implementation specific definitions will be required for the compilation.

1.2.3 Include files

The C layer of System Verilog DirectC Interface defines two include files corresponding to the two levels of compatibility: "svc.h" and "svc_src.h".

There will be no confusion whether passing a particular data type is binary compatible or only source level compatible. The rule is straightforward: if a corresponding type definition can be written in C without the need to include "svc_src.h" file, then the binary compatibility is granted. Everything that requires "svc_src.h" is not binary compatible and needs recompilation for each simulator of choice.

Applications that pass solely C compatible data types or standalone packed arrays (both 2-state and 4-state) will require only "svc.h" and therefore will be binary compatible will all simulators.

Applications that use complex data types that are constructed both of System Verilog packed arrays and C-compatible types, will require also "svc_src.h" file and therefore will not be binary compatible will all simulators. The source level compatibility is, however, granted.

1.2.4 "svc.h" include file

This is the main include file needed by all applications that use DirectC interface with C code. "svc.h" is fully defined by the interface and is implementation independent. It defines the canonical representation of 2-state (**`bit`**) and 4-state (**`logic`**) values, defines the types used for passing references to System Verilog data objects, provides function headers and defines a number of helper macros and constants.

The applications that use only "svc.h" will be binary compatible with all System Verilog simulators.

1.2.5 "svc_src.h" include file

This is the auxiliary include file. "svc_src.h" defines data structures for implementation specific representation of 2-state and 4-state System Verilog packed arrays. The contens of this file, i.e. what symbols are defined is specified by the interface. The actual definitions of those symbols, however, are implementation specific and will be provided by the vendors.

User's applications that require "svc_src.h" file will be only source-level compatible, i.e. they will have to be compiled with the version of "svc_src.h" provided for a particular implementation of System Verilog.

1.3 Semantical Constraints

1.3.1 Types of formal arguments

The principle "What You Specify Is What You Get" guarantees the types of formal arguments of external functions. For open arrays some ranges may remain unspecified.

The formal arguments other than open arrays are fully defined by the external declaration and therefore they are assumed to have ranges of packed or unpacked arrays exactly as specified in the external declaration. Only the declaration site (System Verilog) of the external function is relevant for such formal arguments.

The formal arguments defined as open arrays have the size and ranges of the actual argument, i.e. have the ranges of packed or unpacked arrays exactly as of the actual argument. The actual ranges of an open array are determined per call, hence the call sites of the external function are relevant for the formal arguments specified as open arrays.

To be exact, 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 that specifies that 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 particular call.

1.3.2 Input Arguments

The formal arguments specified in System Verilog as **`input`** must not be modified.

1.3.3 Output Arguments

The initial values of formal arguments specified in System Verilog as **`output`** are undetermined and may be implementation dependent.

1.3.4 Value changes for output and inout arguments

There is neither a need nor the means to notify the caller that a value of a formal argument specified in System Verilog as **`output`** or **`inout`** has change. System Verilog simulator is responsible for handling the value changes for output and inout arguments. Such changes will be detected and handled after the control returns from C code to System Verilog code.

1.3.5 `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 to provide such context; for example, some variables referring to "current instance" or "current task" must be set.

Such context and the involved instrumentation of a function call is, however, usually not needed. In order to avoid an unnecessary overhead, external function calls in System Verilog code are not instrumented unless the external function is specified as **`context`** in its System Verilog external declaration.

Only the calls of **`context`** functions are properly instrumented and therefore only those functions may safely call all functions from other APIs, including PLI and VPI functions, or exported System Verilog functions.

For functions not specified as **`context`** the effects of calling PLI or VPI functions or System Verilog functions may be, depending on a function, unpredictable and such calls may crash if the callee requires the context that has not been set properly.

1.3.6 `Pure` functions

Only non-void functions with no output or inout arguments may be specified as **`pure`**.

The functions specified as **`pure`** in their corresponding System Verilog external declarations will be assumed to have no side effects. Their result is assumed to depend solely on the values of their input arguments. Calls to such functions may be removed by System Verilog compiler optimizations or replaced with the values previously computed for the same values of input arguments.

Specifically, **`pure`** function is assumed not to do directly or indirectly (i.e. by calling other function):

— perform any file operations

— read or write anything

— access any persistent data like global or static variables.

If a **`pure`** function does not obey the above restrictions, then System Verilog compiler optimizations may cause the behaviour different than expected, due to eliminated calls or wrong results used.

1.3.7 Memory management

The memory spaces owned and allocated by C code and System Verilog code are disjoined. Each side is responsible for its own allocated memory. Specifically, C code shall not free the memory allocated by System Verilog code (or the System Verilog compiler) nor expect that System Verilog code will free the memory allocated by C code (or the C compiler). The above 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 System Verilog code, which in turn calls C function which directly (if it is the standard function **`free`**) or indirectly frees that block. Note that in the above scenario a block of memory is allocated and freed in C code, even when standard functions **`malloc`** and **`free`** are called directly from System Verilog code.

1.4 Data Types

1.4.1 Limitations

The packed arrays are assumed to be always one-dimensional. Note that any packed data type in System Verilog is eventually equivalent to a one-dimensional packed array. Hence the above limitation is practically not restrictive. If the packed part of an array in the type of a formal argument in System Verilog is specified as multi-dimensional, then the System Verilog compiler linearizes it.

Although the original ranges are generally preserved for open arrays, if the actual argument has multidimensional packed part of the array, the equivalent one-dimensional packed array will be normalized.

1.4.2 Duality of types: System Verilog types vs. C types

A value that crosses the DirectC Interface is specified in System Verilog code as a value of System Verilog type, while the same value must be specified in C code as a value of C type. Therefore each data type that is passed through DirectC Interface requires two matching type definitions: System Verilog definition and C definition. It is user's responsibility to provide such matching definitions. Specifically, for each System Verilog type used in the external declarations or export declarations in System Verilog code, the user must provide the equivalent type definition in C. To be exact, the actual C definition must reflect the argument passing mode for the particular type of System Verilog value and the direction (input, output, inout) of the formal System Verilog argument. Specifically, for values passed by reference a generic pointer `void *` may be used (typedefed conveniently) without the knowledge of the actual representation of the value.

1.4.3 Assumed Data Representation

DirectC does not impose any additional restrictions on the representation of System Verilog data types.

The following is assumed about the representation of System Verilog data:

a) representation of packed types is implementation dependent

b) basic integer and real data types are represented as defined in System Verilog LRM sub-sections 3.3 and 3.4.2, see also See “Basic Types” below.

Need cross referece to above

c) layout of unpacked structures is same as used by C compiler (System Verilog LRM sub-section 3.7)

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

Note that this is a restriction imposed on the System Verilog side of the interface! Depending on the implementation, a particular array may or may be not accepted as an actual argument for the formal argument which is a sized array (it will be always accepted for open array).

A natural order of elements is assumed for each dimension in the layout of an unpacked array, i.e. elements with lower indices go first. In other words, for System Verilog range `[L:R]`, the element with System Verilog index `min(L,R)` will have C index 0, and the element with System Verilog index `max(L,R)` will have C index `abs(L-R)`.

e) layout of the unsized (aka open) standalone unpacked arrays is implementation dependent with the following restriction: an element of an array must have the same representation as individual a value of the same type, with the exception of scalars (**`bit`** or **`logic`**) and packed arrays as a type of an element. Hence array's elements other than scalars or packed arrays can be accessed via pointers similarly to individual values.

Note that d) actually does not impose any restrictions on how unpacked arrays are implemented; it says only that an array that does not satisfy d) may not be passed as an actual argument for the formal argument which is a sized array; it may be passed, however, for unsized (i.e. open) array. Therefore, the correctness of an actual argument may be implementation dependent. Nevertheless an open array provides implementation independent solution. This seems to be a reasonable trade-off.

1.4.4 Basic Types

The following table defines the mapping between the basic System Verilog data types and the corresponding C types:

Table 1-1:

System Verilog Type	C Type
`char`	`char`
`byte`	`char`
`shortint`	`short int`
`int`	`int`
`longint`	`long long`
`real`	`double`
`shortreal`	`float`
`handle`	`void*`
`string`	`char*`

The representation of System Verilog 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 may be tuned for a specific implementation and make use of the actual representation used by that implementation; such applications will not be binary compatible, however.

1.4.5 Normalized Ranges

The packed arrays are assumed to be always one-dimensional, the unpacked part of an array may 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 the open arrays. The canonical representation of packed arrays also uses normalized ranges.

1.4.6 Mapping between System Verilog ranges and normalized ranges

The System Verilog 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, using their original ranges and the same indexing as in System Verilog code.

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

First, if a packed part of an array has more than 1 dimension, it is linearized as specified by the equivalence of packed types.

Then a packed array of range `[L:R]` will be normalized as `[abs(L-R):0]`; its most significant bit will have a normalized index `abs(L-R)`, and its least significant bit will have a normalized index 0.

A natural order of elements is assumed for each dimension in the layout of an unpacked array, i.e. elements with lower indices go first. In other words, for System Verilog range `[L:R]`, the element with System Verilog index `min(L,R)` will have C index 0, and the element with System Verilog index `max(L,R)` will have C index `abs(L-R)`.

Note that the above range mapping from System Verilog to C applies to calls made in both directions, i.e. System Verilog-calls-C and C-calls-System Verilog.

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

1.4.7 Canonical representation of packed arrays

DirectC interface defines the canonical representation of packed 2-state and 4-state arrays. See the types **`svBitVec32`**, **`svLogicVec32.`** This canonical representation is actually based on 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, each element representing a group of 32 bits.The first element of an array contains 32 least significant bits, next element contains 32 more significant bits, and so on. The last element may contain a number of unused bits. The contents of the unsused bits is undetermined. The user is responsible for the masking or the sign extension, depending on the signness, for the unused bits.

Table 1-2 defines the encoding used for packed **`logic`** array represented as **`svLogicVec32.`**

Table 1-2: Encoding of bits in `svLogicVec32`

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

1.5 Argument passing modes

1.5.1 Overview

The types of function result are restricted to small values and a result is directly returned.

The function arguments are generally passed by some form of a reference, with the exception of small values of System Verilog input arguments, which are passed by value. Similarly, function result, which is restricted to small values, is passed by value, i.e. directly returned.

The actual arguments passed by reference by and large will be passed as they are, without changing their representation from the one used by a simulator. There will be no inherent copying of arguments (other than resulting from coercing). Therefore there will be no inherent overhead on argument passing because generally no copying or translation between different representations will be required.

The access to packed arrays via the canonical representation will involve, however, arguments copying and hence will incur some overhead. Alternatively, for the sake of performance the application may be tuned for a particular tool and may access the packed arrays directly thru pointers using the implementation representation, what would of course defy the binary compatibility. Note that this provides some degree of flexibility and allows the user to control the trade-off of performance vs. portability.

All types of formal arguments but open arrays are passed either by direct reference or by value and therefore are directly accessible in C code. Passing such arguments incurs virtually no overhead.

Formal arguments declared in System Verilog as open arrays are passed by handle, see the type **`svHandle`**, and are accessible via library functions.

1.5.2 Calling System Verilog functions from C

There is no difference in argument passing between calls from System Verilog to C and calls from C to System Verilog. Note that the functions exported from System Verilog may not have open arrays as arguments. Otherwise the same types of formal arguments may be declared in System Verilog both for the exported functions and for the external functions. A function exported from System Verilog will have the same function header 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 System Verilog function called from C must be allocated using the same layout of data as System Verilog would use for that type of argument; the caller is responsible for the allocation.

1.5.3 Argument passing by value

Only 'small' values of formal input arguments are passed by value. Also function results are directly passed by value. User must provide the C type equivalent to System Verilog type of a formal argument, if an argument is passed by value.

1.5.4 Argument passing by reference

For the arguments passed by reference, their original simulator-defined representation will be used and a reference (a pointer) to the actual data object will be passed.

The actual argument is usually allocated by a caller. The caller may 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, than the formal argument shall be of the type `T*`. However, packed arrays may be also passed using generic pointers `void*` typedefed accordingly to **`svBitPackedArrRef`** or **`svLogicPackedArrRef`**.

1.5.4.1 Allocating actual arguments for System Verilog specific types.

This is relevant only for calling (exported) System Verilog functions from C code. The caller is responsible for allocating the actual arguments that are passed by reference.

Statical allocation requires the knowledge of the relevant data type. If such a type involves System Verilog packed arrays, their actual representation must be known to C code and hence file "svc_src.h" must be included what would deem the C code implementation-dependent and thus not binary compatible.

Sometimes the binary compatibility may be achieved by using dynamic allocation. Functions **`svSizeOfLogicPackedArr()`** and **`svSizeOfBitPackedArr()`** provide the size of the actual representation of a packed array. The needed size may be used for the dynamic allocation of an actual argument, without compromising the portability. Such a technique will not work if a packed array is a part of another type.

1.5.5 Argument passing by sv_handle - Open Arrays

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

1.5.6 Input arguments

Input arguments should always have **`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 32-bit; canonical representation will be used, similarly for function result

Input arguments of other types are passed by reference.

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

1.5.7 Inout and output arguments

Inout and output arguments, with the exception of open arrays, are always passed by reference.

1.5.8 Function result

Types of function result are restricted to the following System Verilog data types (see 1.4.4. for the corresponding C type):

— basic types **`char`**, **`byte`**, **`shortint`**, **`int`**, **`longint`**, **`real`**, **`shortreal, handle`**, **`string`**

— packed **`bit`** arrays up to 32-bit

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

1.6 Include files

The C layer of 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.

The applications that need not include "svc_src.h" are binary level compatible.

1.6.1 Binary compatibility include file "svc.h"

This is the main include file needed by most of applications that use DirectC interface with C code. "svc.h" is fully defined by the interface.

It defines the canonical representation of 2-state (**`bit`**) and 4-state (**`logic`**) values, and all types used for passing references to System Verilog data objects.

All interface functions are declared in this file.

1.6.1.1 scalars of type bit and logic

This file contains the following definitions:

```
/* 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 */
```

1.6.1.2 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))))
```

1.6.1.3 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);
```

1.6.1.4 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, the bit selects and limited (up to 32 bits) part selects are also supported, see 1.7.4, 1.7.5.

1.6.2 Source level compatibility include file "svc_src.h"

Only two symbols are defined: the macros that allow to declare variables representing System Verilog 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 might define the later macro as follows:

```
#define sv_LOGIC_PACKED_ARRAY(WIDTH,NAME) \
                              svLogicVec32 NAME [ sv_CANONICAL_SIZE(WIDTH) ]
```

1.6.3 Example 1 - binary compatible application

System Verilog:

```
   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 System Verilog */

   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 System Verilog */
      exported_sv_func(i1, tab); /* tab passed by reference */
      ...
   }
```

1.6.4 Example 2 - source level compatible application

System Verilog:

```
   typedef struct {int a; bit [6:1][1:8] b [65:2]; int c;} triple;
      // troublesome mix od 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
                                                              System Verilog */

   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 System Verilog */
```

```
    exported_sv_func(2, (svLogicPackedArrRef)&my_tab); /* by reference */
    svGetLogicVec32(aL, (svLogicPackedArrRef)&my_tab, 64);   ... }
```

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

1.7 Arrays

Generally, normalized ranges are assumed for accessing System Verilog arrays, with the exception of formal arguments specified as open arrays.

1.7.1 Multidimensional arrays

The packed arrays are assumed to be always one-dimensional. Upacked arrays may have arbitrary number of dimensions.

1.7.2 Direct access to unpacked arrays

Unpacked ararys, with the exception of formal arguments specified as open arrays, are assumed to have the same layout as used by C compiler, and they are accessed using C indexing; See "Mapping between System Verilog ranges and normalized ranges" on page 6..

1.7.3 Access to packed arrays via canonical representation

Packed arrays are accessible via canonical representation; interface provides functions for moving data between implementation representation and canonical representation, any necessary convertion is performed on the fly; See "Implementation dependent representation" on page 10.

There are also functions for bit selects and limited (up to 32 bits) part selects.

1.7.4 Bit selects

```
/* 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);
```

1.7.5 Part selects

Limited (up to 32 bits) part selects are supported. Part select denotes here a slice of packed array of types **`bit`** or **`logic`**. Array slices are not supported for unpacked arrays.

Functions for part selects allow access (read/write) to only a narrow subranges of up to 32 bits. A canonical representation will 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
 */
```

Please note that for the sake of symmetry a canonical representation (i.e. an array) is used both for **`bit`** and **`logic`**, though a simpler **`int`** could be used for **`bit`** part selects <= 32-bit.

```
/* 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);
```

1.8 Open Arrays

Formal arguments specified as open arrays allow to pass actual arguments of different sizes (different range and/or different number of elements). This facilitates writing in C a more general code that may handle System Verilog arrays of different sizes.

Better yet, the elements of an open array are accessed in C code using the same range of indices and the same indexing as in System Verilog code. Additionally, inquires about the dimensions and the original boundaries of System Verilog actual argument are supported for open arrays. This may make them more attractive for the users and less confusing than sized arrays, which always use normalized ranges and thus require the mapping between the original indices or ranges in System Verilog and the normalized ranges and indices used in C.

All formal arguments declared in System Verilog as open arrays are passed by handle (type **`svHandle`**) regardless of a direction of System Verilog formal argument. Such arguments are accessible via interface functions.

1.8.1 Actual ranges

The formal arguments defined as open arrays have the size and ranges of the actual argument, determined on per call basis.

The programmer will 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 will be normalized on the C side (i.e. 0 and up) and the programmer is assumed to know the size of an array. Programmer is also assumed to be capable of figuring out how the ranges of the actual argument will map onto C-style ranges, See “Mapping between System Verilog ranges and normalized ranges” on page 6.

Hint: programmers may decide to stick to `[n:0]name[0:k]` style ranges in System Verilog.

In the later case, i.e. open array, individual elements of a packed array will be accessible via interface functions, what would facilitate System Verilog-style of indexing with the original boundaries of the actual argument, all this for the price of some overhead.

Note that this provides some degree of flexibility and allows programmer to control the trade-off of performance vs. convenience.

If a formal argument is specified as a sized array, then it will be passed by reference, with no overhead, and will be directly accessible as a normalized array.

If a formal argument is specified as a open (unsized) array, then it will be passed by handle, with some overhead, and will be accessible mostly indirectly, again with some overhead, although with the original boundaries.

The following example shows the use of sized vs. unsized ararys in System Verilog 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]
```

1.8.2 Array querying functions

These functions are modelled upon System Verilog array querying functions, with the same semantics (cf. System Verilog 3.0 LRM 16.3).

Need a cross reference above

If the dimension is 0, then the query refers to the packed part (which is always assumed to be 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);
```

1.8.3 Access functions

Similarly to sized arrays, there are functions for copying data between the simulator representation and the canonical representation.

It will be also possible to get the actual address of System Verilog data object or of an individual element of an unpacked array. This may be useful for the simulator-specific tuning of the application.

Depending on the type of an element of an unpacked array, different access methods must beused:

— 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 should provide   the appropriate casting.

— All types but scalars and packed arrays may be accessed via pointers, as described above. Scalars and packed arrays may or may be not accessible via pointers depending on implementation.

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

1.8.4 Access to the actual representation

The following functions provide an actual address of the whole array or of its individual element. These functions will be used for accessing elements of the arrays of types compatible with C.

These functions will be also usefull for the vendors, because they provide access to the actual representation for all types of arrays.

If the actual layout of the System Verilog array passed as an argument for an open unpacked array is different than C layout, then it will not be posssible to access such array as a whole and therefore the address and size of such array will be undefined (zero, to be exact). Nonetheless the adresses of individual elements of an array will be always supported.

Note that 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's element via pointer makes sense only if the representation of such element is 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; if it differes from the representation of individual values of the same type, the above functions will return null.

1.8.5 Access via canonical representation

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

The following functions will copy a single vector from a canonical representation to an element of an open array or other way round.

Element of an array is identified by indices bound by the ranges of the actual argument. In other words, original System Verilog ranges are used for indexing.

```
/* functions for translation between simulator’s 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.

1.8.6 Access to scalars (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);
```

1.8.7 Access to array elements of other types

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

1.9 Example 3 - open array

System Verilog:

```
    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_bin.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;
              ...
              }
          ...
          }
    }
```

1.10 Example 4 - open array

System Verilog:

```
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_bin.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 poiter arithmetics */
   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++);
    }

  }

}
```

1.11 Example 5 - access to packed arrays

System Verilog:

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

C:

```
#include "svc_bin.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);
       ...
   }
   ...
}
```