Section 10, Section 27, Annex A, Annex E, Annex F

Addenda to Mantis item SV-50

Description

Some parties have expressed concern over a migration strategy for DPI users of the SV 3.1a standard. This proposal attempts to address all concerns of those parties. It also contains simplifications and clarifications of the P1800 DPI standard to which those SV 3.1a users will eventually migrate, as agreed upon in the SV-CC meeting of December 8, 2004. Effectively the transmission of packed array data across the interface will be done using canonical representation. A deprecated SV 3.1a-compatible mode of vendor-specific data representation is supported as well.

This proposal completely supersedes the proposals for SV-5, SV-50, SV-205, SV-278, SV-288, and SV-318. All issues addressed in those proposals have been integrated into this body of work. In addition, SV-335 is handled in this proposal.

Proposal

In 10.5, MODIFY Syntax 10-3 as follows:

```
dpi_import_export ::=
   import "DPI" [ dpi_function_import_property ] [ c_identifier = ] dpi_function_proto ;
 | import "DPI" [ dpi_task_import_property ] [ c_identifier = ] dpi_task_proto ;
 | export "DPI" [ c_identifier = ] function function_identifier ;
 | export "DPI" [ c_identifier = ] task task_identifier ;
```

```
dpi_import_export ::=
   import dpi_spec_string [ dpi_function_import_property ] [ c_identifier = ]
dpi_function_proto ;
 | import dpi_spec_string [ dpi_task_import_property ] [ c_identifier = ] dpi_task_proto ;
 | export dpi_spec_string [ c_identifier = ] function function_identifier ;
 | export dpi_spec_string [ c_identifier = ] task task_identifier ;

dpi_spec_string ::= “DPI” | “DPI-3.1a”
```

In Annex A.2.6, MODIFY the BNF as follows:

```
dpi_import_export ::=
   import "DPI" [ dpi_function_import_property ] [ c_identifier = ] dpi_function_proto ;
 | import "DPI" [ dpi_task_import_property ] [ c_identifier = ] dpi_task_proto ;
 | export "DPI" [ c_identifier = ] function function_identifier ;
 | export "DPI" [ c_identifier = ] task task_identifier ;
```

```
dpi_import_export ::=
   import dpi_spec_string [ dpi_function_import_property ] [ c_identifier = ]
dpi_function_proto ;
 | import dpi_spec_string [ dpi_task_import_property ] [ c_identifier = ] dpi_task_proto ;
 | export dpi_spec_string [ c_identifier = ] function function_identifier ;
 | export dpi_spec_string [ c_identifier = ] task task_identifier ;

dpi_spec_string ::= “DPI” | “DPI-3.1a”
```

In 27.3, MODIFY the text as follows:

Multiple export declarations are allowed with the same *c_identifier*, explicit or implicit, as long as they are in different scopes and have the same type signature (as defined in Section 27.4.4 for imported tasks and functions). Multiple export declarations with the same *c_identifier* in the same scope are forbidden.

It is possible to use the deprecated "DPI-3.1a" version string syntax in an import or export declaration. This syntax indicates that the Accelera SystemVerilog 3.1a 2-state and 4-state packed array argument passing convention is to be used (See Annex E.12). In such cases, all declarations using the same *c_identifier* shall be declared with the same DPI version string syntax.

In Section 27.4.4, MODIFY Syntax 27-1 as follows:

```
dpi_import_export ::=
   import "DPI" [ dpi_function_import_property ] [ c_identifier = ] dpi_function_proto ;
 | import "DPI" [ dpi_task_import_property ] [ c_identifier = ] dpi_task_proto ;
 | export "DPI" [ c_identifier = ] function function_identifier ;
 | export "DPI" [ c_identifier = ] task task_identifier ;
```

```
dpi_import_export ::=
   import dpi_spec_string [ dpi_function_import_property ] [ c_identifier = ]
dpi_function_proto ;
 | import dpi_spec_string [ dpi_task_import_property ] [ c_identifier = ] dpi_task_proto ;
 | export dpi_spec_string [ c_identifier = ] function function_identifier ;
 | export dpi_spec_string [ c_identifier = ] task task_identifier ;

dpi_spec_string ::= "DPI" | "DPI-3.1a"
```

In Section 27.4.4, MODIFY the text as follows:

Note that an import declaration is equivalent to defining a task or function of that name in the SystemVerilog scope in which the import declaration occurs, and thus multiple imports of the same task or function name into the same scope are forbidden. Note that this declaration scope is particularly important in the case of imported context tasks or functions, see Section 27.4.3; for non-context imported tasks or functions the declaration scope has no other implications other than defining the visibility of the task or function.

The *dpi_spec_string* can take values "DPI" and "DPI-3.1a". "DPI-3.1a" is used to indicate that the deprecated Accellera SystemVerilog 3.1a packed array argument passing semantics are to be used. In these semantics arguments are passed in actual simulator representation format rather than in canonical format, as is the case with "DPI". See Annex E.12.

c_identifier provides the linkage name for this task or function in the foreign language. If not provided, this defaults to the same identifier as the SystemVerilog task or function name. In either case, this linkage name must conform to C identifier syntax. An error shall occur if the *c_identifier*, either directly or indirectly, does not conform to these rules.

For any given *c_identifier* (whether explicitly defined with *c_identifier*=, or automatically determined from the task or function name), all declarations, regardless of scope, must have exactly the same type signature. The signature includes the return type and the number, order, direction and types of each and every argument. Type includes dimensions and bounds of any arrays or array dimensions. Signature also includes the **pure**/**context** qualifiers that can be associated with an extern definition, and it includes the value of the *dpi_spec_string*.

Note that multiple declarations of the same imported or exported task or function in different scopes can vary argument names and default values, provided the type compatibility constraints are met.

In 27.6, MODIFY Syntax 27-2 as follows:

```
dpi_import_export ::=                                   // from Annex A.2.6
| export ~~"DPI"~~ dpi_spec_string [ c_identifier = ] function function_identifier ;

dpi_spec_string ::= “DPI” | “DPI-3.1a”
```

In E.1, REPLACE:

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

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

WITH

Some SystemVerilog types are directly compatible with C types; defining a matching C type for them is straightforward. There are, however, SystemVerilog-specific types, namely packed types (arrays, structures, and unions), 2-state or 4-state, which have no natural correspondence in C. DPI defines a canonical representation of 4-state types which is exactly the same as the representation used by the Verilog Programming Interface’s (VPI’s) avalue/bvalue representation of 4-state vectors. DPI defines a 2-state representation model which is consistent with the VPI 4-state model. DPI defines library functions to assist users in working with the canonical data representation.

The DPI C interface includes deprecated functions and definitions related to implementation-specific representation of packed array arguments. These functions are enabled by using the “DPI-3.1a” specification string in import and export declarations (Section 27.4). Refer to E.12 for details on the deprecated functionality.

In E.1, REPLACE:

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

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

WITH

The C layer of DPI defines a portable binary interface. Once DPI C code is compiled into object code, the resulting object code shall work without recompilation in any compliant SystemVerilog implementation.

One normative include file, svdpi.h, is provided as part of the DPI C layer. This file defines all basic types, the canonical 2-state and 4-state data representation, and all interface functions.

In E.2, MODIFY as follows

E.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`~~`svLogicVecVal`.
— Names of symbolic constants start with `sv_`, e.g., `sv_x`.
— Names of macro definitions start with `SV_`, followed by all upper-case words separated by a underscore (_), e.g., ~~`SV_CANONICAL_SIZE`~~ `SV_GET_UNSIGNED_BITS.`

In E.3, REPLACE:

E.3 Portability

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

~~The values of SystemVerilog packed types can be accessed via interface tasks or 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.~~

~~E.3.1 Binary compatibility~~
~~Binary compatibility means an application compiled for a given platform shall work with every SystemVerilog simulator on that platform.~~

~~E.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.~~

WITH

DPI applications are always portable at the binary level. When compiled on a given platform, DPI object code shall work with every SystemVerilog simulator on that platform.

In E.4, MODIFY the text as follows.

~~E.4 Include files~~

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

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

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

E.4 svdpi.h include file

~~E.4.1 svdpi.h include file~~

The C layer of the Direct Programming Interface defines include file svdpi.h.

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

This ~~document~~ section defines the svdpi.h file. The content of svdpi.h does not depend on any particular implementation ~~or platform~~; all simulators shall use the same file. For more details on svdpi.h, see Annex E.9.1 and Annex F. ~~Applications which only use svdpi.h shall be binary-compatible with all SystemVerilog simulators.~~

This file may also contain the deprecated functions and data representations described in Section E.12. Section E.12 also describes the deprecated header svdpi_src.h, which defines the implementation dependent representation of packed values.

~~E.4.2 svdpi_src.h include file~~

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

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

In Section E.6.2, DELETE the following red text:

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

In Section E.63, MODIFY the text as follows:

E.6.3 Data representation

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

— SystemVerilog types that are not packed and that do not contain~~ed~~ packed elements have C compatible representation.
— Basic integer and real data types are represented as defined in Annex E.6.4.
— Enumeration types are represented as the types associated with them. Enumerated names are not available on the C side of the interface.
~~— Representation of packed types is implementation-dependent.~~
— Packed types are represented using the canonical format defined in section E.6.7.
— Unpacked arrays embedded in a structure or union have C compatible layout regardless of the type of elements. Similarly, standalone arrays passed as actuals to a sized formal argument have C compatible representation.
— Standalone array passed as an actual to an open array formal
  — if the element type is a 2- or 4-state scalar or packed then the representation is ~~implementation dependent~~ in canonical form.
  — otherwise the representation is C compatible. Therefore an element of an array shall have the same representation as an individual value of the same type. Hence, an array's elements can be accessed from C code via normal C array indexing similarly to doing so for individual values.
— The natural order of elements for each dimension in the layout of an unpacked array shall be used, i.e., elements with lower indices go first. For SystemVerilog range [L:R], the element with SystemVerilog index `min(L,R)` has the C index 0 and the element with SystemVerilog index `max(L,R)` has the C index `abs(L-R)`.

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

In Section E.6.4, MODIFY the text as follows:

Note that input mode arguments of type **`byte unsigned`** and **`shortint unsigned`** are not equivalent to **`bit`**`[7:0]` or **`bit`**`[15:0]`, respectively, since the former are passed as C types **`unsigned char`** and **`unsigned short`** and the latter are both passed as C **`unsigned int`** (i.e., `svBitVec`~~`32`~~`Val`). A similar lack of equivalence applies to passing such parameters by reference for output and inout modes.

~~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.~~

In Section E.6.7, MODIFY the text as follows:

E.6.7 Canonical representation of packed arrays

The Direct Programming Interface defines the canonical representation of packed 2-state (type ~~`svBitVec32`~~`svBitVecVal`) and 4-state arrays (type ~~`svLogicVec32`~~`svLogicVecVal`). ~~This canonical representation is derived from 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~~. `svLogicVecVal` is fully equivalent to type `s_vpi_vecval`, which is used to represent 4-state logic in VPI.

A packed array is represented as an array of one or more elements (of type ~~`svBitVec32`~~ `svBitVecVal` for 2-state values and ~~`svLogicVec32`~~ `svLogicVecVal` 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 can 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 E-2 defines the encoding used for a packed `logic` array represented as `svLogicVec32`.~~

~~**Table E-2: Encoding of bits in `svLogicVec32`**~~
~~**a b Value**~~
~~0 0 0~~
~~0 1 z~~
~~1 0 1~~
~~1 1 x~~

In Section E.7.1, DELETE the red text as follows:

E.7.1 Overview

Imported and exported function arguments are generally passed by some form of a reference, with the exception of small values of SystemVerilog input arguments (see Annex E.11.7), which are passed by value. Similarly, the function result, which is restricted to small values, is passed by value, i.e., directly returned.

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

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

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

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

In Section E.7.4, MODIFY as follows:

E.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. In the case of packed data, a reference to a canonical data object is passed. The actual argument is usually allocated by a caller. The caller can also pass a reference to an object already allocated somewhere else, for example, its own formal argument passed by reference.

If an argument of type `T` is passed by reference, the formal argument shall be of type `T*`. ~~However, packed arrays can also be passed using generic pointers `void*` (`typedef`ed accordingly to `svBitPackedArrRef` or `svLogicPackedArrRef`).~~ Packed arrays are passed using a pointer to the appropriate canonical type definition, either svLogicVecVal* or svBitVecVal*.

In Section E.7.5, MODIFY as follows:

E.7.5 Allocating actual arguments for SystemVerilog-specific types

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

Static allocation requires knowledge of the relevant data type. If such a type involves SystemVerilog packed arrays, ~~their actual representation needs to be known to C code; thus, the file `svdpi_src.h` needs to be included, which makes the C code implementation-dependent and not binary compatible.~~ corresponding C arrays of canonical data types (either svLogicVecVal or svBitVecVal) must be allocated and initialized before being passed by reference to the exported SystemVerilog task or function.

~~Sometimes 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 Annex E.11.11). Such a technique does not work if a packed array is a part of another type.~~

Modify Section #.9 as follows:

E.9 Include files

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

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

E.9.1 ~~Binary compatibility i~~Include file `svdpi.h`

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

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

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

```
/* canonical representation */
#define sv_0 0
#define sv_1 1
#define sv_z 2 /* representation of 4-st scalar z */
#define sv_x 3 /* representation of 4-st scalar x */
/* common type for 'bit' and 'logic' scalars. */
typedef unsigned char svScalar;
typedef svScalar svBit; /* scalar */
typedef svScalar svLogic; /* scalar */
```

E.9.1.2 Canonical representation of packed arrays

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

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

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

/* Since the contents of the unused bits is undetermined, the following macros
can 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))))

/*
 * DPI representation of packed arrays.
 * 2-state and 4-state vectors, exactly the same as PLI's avalue/bvalue.
 */
#ifndef VPI_VECVAL
#define VPI_VECVAL
typedef struct vpi_vecval {
    uint32_t a;
    uint32_t b;
} s_vpi_vecval, *p_vpi_vecval;
#endif

/* (a chunk of) packed logic array */
typedef s_vpi_vecval svLogicVecVal;

/* (a chunk of) packed bit array */
typedef uint32_t svBitVecVal;

/* Number of chunks required to represent the given width packed array */
#define SV_PACKED_DATA_NELEMS(WIDTH) (((WIDTH) + 31) >> 5)

/*
 * Since the contents of the unused bits is undetermined,
 * the following macros can 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))) ? ((VALUE) | ~SV_MASK(N)) : ((VALUE) & SV_MASK(N))))
```

E.9.1.3 Implementation-dependent representation

The svDpiVersion() function returns a string indicating which DPI standard is supported by the simulator, and in particular which canonical value representation is being provided. Simulators implementing the current standard, i.e. the VPI based canonical value, must return the string "P1800-2005". Simulators implementing to the prior SystemVerilog 3.1 standards, and thus using the svLogicVec32 value representation, should return the string "SV3.1a".

```
/* Returns either version string “P1800-2005” or “SV3.1a” */
const char* svDpiVersion();

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

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

/* reference to a standalone packed array */
```

```
typedef void* svBitPackedArrRef;
typedef void* svLogicPackedArrRef;

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

int svSizeOfLogicPackedArr(int width);
```

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

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

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

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

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

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

E.9.2 Source-level compatibility include file svdpi_src.h

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

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

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

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

E.9.3 Example 2 — binary compatible Simple packed array application

SystemVerilog:

```
   typedef struct {int ax; int by;} pair;
   import "DPI" function void foo(input int i1, pair i2, output logic [63:0] o3);

   export "DPI" function exported_sv_func;

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

C:

```
   #include "svdpi.h"
   typedef struct {int ax; int by;} pair;
   extern void exported_sv_func(int, int *); /* imported from SystemVerilog */
   void foo(const int i1, const pair *i2, svLogicPackedArrRefsvLogicVecVal* o3)
   {
```

```
      svLogicVec32 arr[SV_CANONICAL_SIZE]; /* 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);

      o3[0].a = i2->x;
      o3[0].b = 0;
      o3[1].a = i2->y;
      o3[1].b = 0;

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

E.9.4 Example 3— source-level compatible aApplication with complex mix of types

SystemVerilog:

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

   export "DPI" function exported_sv_func;

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

C:

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

   typedef struct {
      int a;
      sv_BIT_PACKED_ARRAY(6*8, b) [64]; /* implementation specific representation
*/
      svBitVecVal b[64][SV_PACKED_DATA_NELEMS(6*8)];
      int c;
   } triple;
   /* Note that 'b' is defined as for 'bit [6*8-1:0] b [63:0]' */

   extern void exported_sv_func(int, svLogicPackedArrRefsvLogicVecVal*); /*
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 TO EDITOR: There were indentation errors in the original function "foo()". Please make sure to precisely adhere to the indentation used in the new version of the function "foo" below.]

```
   void foo(const triple *t)
   {
      int i;
      svBitVecVal   aB;
      svLogicVecVal aL[SV_PACKED_DATA_NELEMS(64)];

      /* aB holds results of part-select taken from packed bit array 'b' in
struct triple. */
      /* aL holds the packed logic array filled in by the export function. */

      printf("%d %d\n", t->a, t->c);
      for (i = 0; i < 64; i++) {
         /* Read least significant byte of each word of b into aB, then
process... */
         svGetPartSelBit(&aB, t->b[i], 0, 8);
         ...
      }
      ...
      /* Call SystemVerilog */
      exported_sv_func(2, aL); /* Export function writes data into output arg
"aL" */
      ...
   }
```

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

MODIFY Section 10 as follows:

~~E.10.3 Access to packed arrays via canonical representation~~

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

~~E.10.3.1 Bit selects~~

~~This subsection defines the bit selects portion of the svdpi.h file (see Annex E.9.1 for more details).~~

```
/* Packed arrays are assumed to be indexed n-1:0, where 0 is the index of least
significant bit */
/* functions for bit select */
/* s=source, i=bit-index */
svBit svGetSelectBit(const svBitPackedArrRef s, int i);
svLogic svGetSelectLogic(const svLogicPackedArrRef s, int i);

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

~~E.10.3.2 Part selects~~

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

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

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

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

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

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

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

E.10.3 Utility functions for working with the canonical representation

Packed arrays are accessible via canonical representation. This C-layer interface provides utility functions for working with bit selects and limited (up to 32-bit) part selects in the canonical representation.

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. If the specified range of a part select is not fully contained within the normalized range of an array, the behavior is undetermined.

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

/* s=source, i=bit-index */
svBit svGetBitselBit(const svBitVecVal* s, int i);
svLogic svGetBitselLogic(const svLogicVecVal* s, int i);

/* d=destination, i=bit-index, s=scalar */
void svPutBitselBit(svBitVecVal* d, int i, svBit s);
void svPutBitselLogic(svLogicVecVal* d, int i, svLogic s);

/*
 * Part select utility functions.
 *
```

```
 * A narrow (<=32 bits) part select is extracted from the
 * source representation and written into the destination word.
 *
 * Normalized ranges and indexing [n-1:0] are used for both arrays.
 *
 * s=source, d=destination, i=starting bit index, w=width
 * like for variable part selects; limitations: w <= 32
 */
void svGetPartselBit(svBitVecVal* d, const svBitVecVal* s, int i, int w);
void svGetPartselLogic(svLogicVecVal* d, const svLogicVecVal* s, int i, int w);

void svPutPartselBit(svBitVecVal* d, const svBitVecVal s, int i, int w);
void svPutPartselLogic(svLogicVecVal* d, const svLogicVecVal s, int i, int w);
```

MODIFY Section 11.3 as follows:

E.11.3 Access functions

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

There are library functions available for copying data between open array handles and canonical form buffers provided by the C programmer. Likewise there are functions to obtain the actual address of SystemVerilog data objects or of an individual element of an unpacked array.

MODIFY Section 11.5 as follows:

E.11.5 Access to elements via canonical representation

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

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

```
/* functions for translation between simulator and canonical representations*/
/* s=source, d=destination */
/* actual <-- canonical */
void svPutBitArrElemVec32 (const svOpenArrayHandle d, const
~~svBitVec32~~svBitVecVal* s,
      int indx1, ...);
void svPutBitArrElem1Vec32(const svOpenArrayHandle d, const
~~svBitVec32~~svBitVecVal* s,
      int indx1);
void svPutBitArrElem2Vec32(const svOpenArrayHandle d, const
~~svBitVec32~~svBitVecVal* s,
      int indx1, int indx2);
void svPutBitArrElem3Vec32(const svOpenArrayHandle d, const
~~svBitVec32~~svBitVecVal* s,
      int indx1, int indx2, int indx3);
void svPutLogicArrElemVec32 (const svOpenArrayHandle d, const
~~svLogicVec32~~svLogicVecVal* s,
      int indx1, ...);
void svPutLogicArrElem1Vec32(const svOpenArrayHandle d, const
~~svLogicVec32~~svLogicVecVal* s,
      int indx1);
void svPutLogicArrElem2Vec32(const svOpenArrayHandle d, const
~~svLogicVec32~~svLogicVecVal* s,
      int indx1, int indx2);
```

```
void svPutLogicArrElem3Vec32(const svOpenArrayHandle d, const
svLogicVec32svLogicVecVal* s,
      int indx1, int indx2, int indx3);

/* canonical <-- actual */
void svGetBitArrElemVec32 (svBitVec32svBitVecVal* d, const svOpenArrayHandle s,
      int indx1, ...);
void svGetBitArrElem1Vec32(svBitVec32svBitVecVal* d, const svOpenArrayHandle s,
      int indx1);
void svGetBitArrElem2Vec32(svBitVec32svBitVecVal* d, const svOpenArrayHandle s,
      int indx1, int indx2);
void svGetBitArrElem3Vec32(svBitVec32svBitVecVal* d, const svOpenArrayHandle s,
      int indx1, int indx2, int indx3);
void svGetLogicArrElemVec32 (svLogicVec32svLogicVecVal* d, const
svOpenArrayHandle s,
      int indx1, ...);
void svGetLogicArrElem1Vec32(svLogicVec32svLogicVecVal* d, const
svOpenArrayHandle s,
      int indx1);
void svGetLogicArrElem2Vec32(svLogicVec32svLogicVecVal* d, const
svOpenArrayHandle s,
      int indx1, int indx2);
void svGetLogicArrElem3Vec32(svLogicVec32svLogicVecVal* d, const
svOpenArrayHandle s,
      int indx1, int indx2, int indx3);
```

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

MODIFY Section 11.10 as follows:

E.11.10 Example 6 — access to packed arrays

SystemVerilog:

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

C:

```
   #include "svdpi.h"

   /* one 128-bit packed vector */
   void foo(const svLogicPackedArrRef packed_vec_128_bit)
   {
      svLogicVec32 arr[SV_CANONICAL_SIZE(128)]; /* canonical representation */
      svGetLogicVec32(arr, packed_vec_128_bit, 128);
      ...
   }

   /* open array of 128-bit packed vectors */
   void boo(const svOpenArrayHandle h)
   {
      int i;
      svLogicVec32 arr[SV_CANONICAL_SIZE(128)]; /* canonical representation */
      for (i = svLow(h, 1); i <= svHigh(h, 1); i++) {
         svLogicPackedArrRef ptr = (svLogicPackedArrRef)svGetArrElemPtr1(h, i);
         /* user need not know the vendor representation! */
         svGetLogicVec32(arr, ptr, 128);
         ...
      }
      ...
   }
   /* Copy out one 128-bit packed vector */
   void foo(const svLogicVecVal* packed_vec_128_bit)
   {
      svLogicVecVal arr[SV_PACKED_DATA_NELEMS(128)]; /* canonical representation
*/
```

```
      memcpy(arr, packed_vec_128_bit, sizeof(arr));
      ...
   }

   /* Copy out each word of an open array of 128-bit packed vectors */
   void boo(const svOpenArrayHandle h)
   {
      int i;
      svLogicVecVal arr[SV_PACKED_DATA_NELEMS(128)]; /* canonical representation
*/
      for (i = svLow(h, 1); i <= svHigh(h, 1); i++) {
         const svLogicVecVal* ptr = (svLogicVecVal*)svGetArrElemPtr1(h, i);
         memcpy(arr, ptr, sizeof(arr));
         ...
      }
      ...
   }
```

DELETE Section 11.11 (It will re-appear in the new section 11.12 below)

~~E.11.11 Example 7 — binary compatible calls of exported functions~~

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

~~SystemVerilog:~~

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

~~C:~~

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

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

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

ADD Section E.12

E.12 SystemVerilog 3.1a-compatible access to packed data (deprecated functionality)

The functionality described in this section is deprecated and need not be implemented by a P1800-2005 compliant simulator. The functionality provides backwards compatibility with SystemVerilog 3.1a regarding the semantics of packed array arguments. This section will describe the 3.1a semantics.

The main difference between 3.1a and P1800 semantics is that in 3.1a, packed data arguments are passed by opaque handle types svLogicPackedArrRef and svBitPackedArrRef. An implementation need not do any conversion or marshalling of data into the canonical format. The C programmer is provided a set of utility

functions that copies data between actual vendor format and canonical format. Other utilities are provided that put and get bit and part selects from actual vendor representation.

E.12.1 Determining the compatibility level of an implementation

Function svDpiVersion() is provided to allow the determination of an implementation's support for the SystemVerilog standard. In simulators that only support the 3.1a standard, users must make use of the opaque handle types for all 2-state and 4-state arguments. See Section E.9.1.3.

When using a P1800-2005 implementation, it is possible for users to make use of 3.1a-compatible semantics on a per-function basis. Import and export declarations annotated with the "DPI-3.1a" syntax shall yield the 3.1a argument passing semantics on the C side of the interface. Import and export declarations annotated with the "DPI" syntax shall yield the 1800 argument passing semantics. See Sections 27.3 and 27.4.4.

The svdpi.h file may contain definitions and function prototypes for use with 3.1a-compliant packed data access. P1800-2005 implementations are not obligated to provide these definitions and prototypes in the include file.

If a P1800-2005 implementation does not support the functionality in this section, it is possible that the DPI C code may not successfully bind to the implementation.

E.12.2 svdpi.h definitions for 3.1a-style packed data processing

The following definitions are used to define 3.1a-style canonical access to packed data.

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

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

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

The following definitions describe implementation-dependent packed data representation.

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

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

int svSizeOfLogicPackedArr(int width);
```

The following functions provide translation between actual vendor representation and canonical representation. The 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 are undetermined.

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

```
/* 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 following functions provide support for bit select processing on actual vendor data representation.

```
/* 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);
```

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. Canonical representation shall be used for such narrow vectors. If the specified range of a part select is not fully contained within the normalized range of an array, the behavior is undetermined.

```
/*
 * 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
 *
 * In part select operations, the data is copied to or from the
 * canonical representation part ('chunk') designated by range [w-1:0]
 * and the implementation representation part designated by range [w+i-1:i].
 */

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

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

E.12.3 Source-level compatibility include file `svdpi_src.h`

Only two symbols are defined: the macros that allow declaring variables to represent the SystemVerilog packed arrays of type `bit` or `logic`. Applications that do not need this file to compile are deemed binary-

compatible. That is, the DPI C code does not need to be recompiled to run on different simulators. Applications that make use of svdpi_src.h must be recompiled for each simulator on which they are to be run.

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

The actual definitions are implementation-specific, but must not define an array type (see definition in ISO 9899-2001 clause 6.2.5). For example, a SystemVerilog simulator might define the latter macro as follows:

```
#define SV_LOGIC_PACKED_ARRAY(WIDTH,NAME) \
     struct { svLogicVec32 __unnamed [SV_CANONICAL_SIZE(WIDTH)]; } NAME
```

E.12.4 Example 7 – Deprecated SystemVerilog 3.1a binary compatible application

SystemVerilog:

```
   typedef struct {int x; int y;} pair;
   import "DPI-3.1a" function void foo(input int i1, pair i2, output logic [63:0]
o3);

   export "DPI-3.1a" function exported_sv_func;

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

C:

```
   #include "svdpi.h"
   typedef struct {int x; int y;} 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[0].c = i2->x;
      arr[0].d = 0;
      arr[1].c = i2->y;
      arr[1].d = 0;
      svPutLogicVec32(o3, arr, 64);

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

E.12.5 Example 8 – Deprecated SystemVerilog 3.1a source compatible application

SystemVerilog:

```
   typedef struct {int a; bit [6:1][1:8] b [65:2]; int c;} triple;
   // troublesome mix of C types and packed arrays
   import "DPI-3.1a" function void foo(input triple t);

   export "DPI-3.1a" function exported_sv_func;

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

C:

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

```
   typedef struct {
      int a;
      SV_BIT_PACKED_ARRAY(6*8, b) [64]; /* implementation specific representation
*/
      int c;
   } triple;
   /* Note that 'b' is defined as for 'bit [6*8-1:0] b [63:0]' */

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

   void foo(const triple *t)
   {
      int j;
      /* canonical representation */
      svBitVec32   aB[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", t->a, t->c);
      for (i = 0; i < 64; i++) {
         svGetBitVec32(aB, (svBitPackedArrRef)&(t->b[i]), 6*8);
         ...
      }
      ...
      /* call SystemVerilog */
      exported_sv_func(2, (svLogicPackedArrRef)&my_tab); /* by reference */
      svGetLogicVec32(aL, (svLogicPackedArrRef)&my_tab, 64);
      ...
   }
```

E.12.6 Example 9 – Deprecated SystemVerilog 3.1a binary compatible calls of export functions

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

SystemVerilog:

```
   export "DPI-3.1a" function myfunc;
   ...
   function void myfunc (output logic [31:0] r); ...
   ...
```

C:

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

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

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

MODIFY Annex F as follows:

Annex F
Include file~~s~~ svdpi.h

This annex shows the contents of the `svdpi.h` ~~and svdpi_src.h~~ include file~~s~~. This is a normative include file that must be provided by all SystemVerilog simulators. However, there is deprecated functionality at the bottom of the file that need not be provided. This functionality is clearly delimited by comments in the file.

Implementations shall ensure that types uint8_t and uint32_t are defined, but the exact method of doing so is not prescribed by the standard.. The section in the include file shown below is a suggested way of defining uint8_t and uint32_t for a wide variety of SystemVerilog platforms.

~~F.1 Binary-level compatibility include file svdpi.h~~

```
#ifndef INCLUDED_SVDPI
#define INCLUDED_SVDPI

#ifdef __cplusplus
extern "C" {
#endif

/* Ensure that uint8_t and uint32_t are defined on all OS platforms. */
#if defined (_MSC_VER)
typedef unsigned __int32 uint32_t;
typedef unsigned __int8 uint8_t;
#elif defined(__MINGW32__)
#include <stdint.h>
#elif defined(__linux)
#include <inttypes.h>
#else
#include <sys/types.h>
#endif

/* 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 uint8_t svScalar;
typedef svScalar svBit; /* scalar */
typedef svScalar svLogic; /* scalar */

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

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

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

/* Since the contents of the unused bits is undetermined, the following macros
can
   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))))


/*
 * DPI representation of packed arrays.
```

```
 * 2-state and 4-state vectors, exactly the same as PLI's avalue/bvalue.
 */
#ifndef VPI_VECVAL
#define VPI_VECVAL
typedef struct vpi_vecval {
    uint32_t a;
    uint32_t b;
} s_vpi_vecval, *p_vpi_vecval;
#endif

/* (a chunk of) packed logic array */
typedef s_vpi_vecval svLogicVecVal;

/* (a chunk of) packed bit array */
typedef uint32_t svBitVecVal;

/* Number of chunks required to represent the given width packed array */
#define SV_PACKED_DATA_NELEMS(WIDTH) (((WIDTH) + 31) >> 5)

/*
 * Since the contents of the unused bits is undetermined,
 * the following macros can 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))) ? ((VALUE) | ~SV_MASK(N)) : ((VALUE) & SV_MASK(N))))


/* implementation-dependent representation */

/*
 * Implementation-dependent representation.
 */
/*
 * Return implementation version information string ("P1800-2005" or "SV3.1a").
 */
const char* svDpiVersion();

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

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

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

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

/* Translation between the actual representation and 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);
```

```
/* 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);

/*
 * 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
 *
 * In part select operations, the data is copied to or from the
 * canonical representation part ('chunk') designated by range [w-1:0]
 * and the implementation representation part designated by range [w+i-1:i].
 */
/* canonical <-- actual */
void svGetPartSelectBit(svBitVec32* d, const svBitPackedArrRef s, int i,
     int w);
svBitVec32 svGetBits(const svBitPackedArrRef s, int i, int w);
svBitVec32 svGet32Bits(const svBitPackedArrRef s, int i); // 32-bits
unsigned long long svGet64Bits(const svBitPackedArrRef s, int i); // 64-bits
void svGetPartSelectLogic(svLogicVec32* d, const svLogicPackedArrRef s, int i,
     int w);
/* actual <-- canonical */
void svPutPartSelectBit(svBitPackedArrRef d, const svBitVec32 s, int i, int w);
void svPutPartSelectLogic(svLogicPackedArrRef d, const svLogicVec32 s, int i,
     int w);

/*
 * Bit select utility functions.
 *
 * Packed arrays are assumed to be indexed n-1:0,
 * where 0 is the index of least significant bit
 */

/* s=source, i=bit-index */
svBit svGetBitselBit(const svBitVecVal* s, int i);
svLogic svGetBitselLogic(const svLogicVecVal* s, int i);

/* d=destination, i=bit-index, s=scalar */
void svPutBitselBit(svBitVecVal* d, int i, svBit s);
void svPutBitselLogic(svLogicVecVal* d, int i, svLogic s);

/*
 * Part select utility functions.
 *
 * A narrow (<=32 bits) part select is extracted from the
 * source representation and written into the destination word.
 *
 * Normalized ranges and indexing [n-1:0] are used for both arrays.
 *
 * s=source, d=destination, i=starting bit index, w=width
 * like for variable part selects; limitations: w <= 32
 */
void svGetPartselBit(svBitVecVal* d, const svBitVecVal* s, int i, int w);
void svGetPartselLogic(svLogicVecVal* d, const svLogicVecVal* s, int i, int w);
```

```
void svPutPartselBit(svBitVecVal* d, const svBitVecVal s, int i, int w);
void svPutPartselLogic(svLogicVecVal* d, const svLogicVecVal s, int i, int w);


/* Array querying functions */
/* These functions are modelled upon the SystemVerilog array querying functions
and use the same semantics*/
/* If the dimension is 0, then the query refers to the packed part (which is
onedimensional)
of an array, and dimensions > 0 refer to the unpacked part of an
array.*/
/*
 * Open array querying functions
 * These functions are modeled upon the SystemVerilog array
 * querying functions and use the same semantics.
 *
 * If the dimension is 0, then the query refers to the
 * packed part of an array (which is one dimensional).
 * Dimensions > 0 refer to the unpacked part of an array.
 */
/* h= handle to open array, d=dimension */
int svLeft(const svOpenArrayHandle h, int d);
int svRight(const svOpenArrayHandle h, int d);
int svLow(const svOpenArrayHandle h, int d);
int svHigh(const svOpenArrayHandle h, int d);
int svIncrement(const svOpenArrayHandle h, int d);
int svLength(const svOpenArrayHandle h, int d);
int svDimensions(const svOpenArrayHandle h);

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

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

/*
 * Return a pointer to an element of the array
 * or NULL if index outside the range or null pointer
 */
void *svGetArrElemPtr(const svOpenArrayHandle, int indx1, ...);

/* specialized versions for 1-, 2- and 3-dimensional arrays: */
void *svGetArrElemPtr1(const svOpenArrayHandle, int indx1);
void *svGetArrElemPtr2(const svOpenArrayHandle, int indx1, int indx2);
void *svGetArrElemPtr3(const svOpenArrayHandle, int indx1, int indx2, int indx3);

/* Functions for translation between simulator and canonical representations*/
/* These functions copy the whole packed array in either direction. The user is
responsible for allocating an array in the canonical representation. */
/*
 * Functions for copying from simulator storage into user space.
 * These functions copy the whole packed array in either direction.
 * The user is responsible for allocating an array to hold the
 * canonical representation.
 */

/* s=source, d=destination */
/* actual <- canonical */
void svPutBitArrElemVec32(const svOpenArrayHandle d, const svBitVec32* s,
        int indx1, ...);
void svPutBitArrElem1Vec32(const svOpenArrayHandle d, const svBitVec32* s, int
        indx1);
void svPutBitArrElem2Vec32(const svOpenArrayHandle d, const svBitVec32* s, int
        indx1,
        int indx2);
void svPutBitArrElem3Vec32(const svOpenArrayHandle d, const svBitVec32* s,
```

```
        int indx1, int indx2, int indx3);
void svPutLogicArrElemVec32(const svOpenArrayHandle d, const svLogicVec32* s,
        int indx1, ...);
void svPutLogicArrElem1Vec32(const svOpenArrayHandle d, const svLogicVec32* s,
        int indx1);
void svPutLogicArrElem2Vec32(const svOpenArrayHandle d, const svLogicVec32* s,
        int indx1, int indx2);
void svPutLogicArrElem3Vec32(const svOpenArrayHandle d, const svLogicVec32* s,
        int indx1, int indx2, int indx3);
/* From user space into simulator storage */
void svPutBitArrElemVecVal(const svOpenArrayHandle d, const svBitVecVal* s,
        int indx1, ...);
void svPutBitArrElem1VecVal(const svOpenArrayHandle d, const svBitVecVal* s,
        int indx1);
void svPutBitArrElem2VecVal(const svOpenArrayHandle d, const svBitVecVal* s,
        int indx1, int indx2);
void svPutBitArrElem3VecVal(const svOpenArrayHandle d, const svBitVecVal* s,
        int indx1, int indx2, int indx3);
void svPutLogicArrElemVecVal(const svOpenArrayHandle d, const svLogicVecVal* s,
        int indx1, ...);
void svPutLogicArrElem1VecVal(const svOpenArrayHandle d, const svLogicVecVal* s,
        int indx1);
void svPutLogicArrElem2VecVal(const svOpenArrayHandle d, const svLogicVecVal* s,
        int indx1, int indx2);
void svPutLogicArrElem3VecVal(const svOpenArrayHandle d, const svLogicVecVal* s,
        int indx1, int indx2, int indx3);



/* canonical <-- actual */
void svGetBitArrElemVec32(svBitVec32* d, const svOpenArrayHandle s, int indx1,
        ...);
void svGetBitArrElem1Vec32(svBitVec32* d, const svOpenArrayHandle s, int indx1);
void svGetBitArrElem2Vec32(svBitVec32* d, const svOpenArrayHandle s, int indx1,
        int indx2);
void svGetBitArrElem3Vec32(svBitVec32* d, const svOpenArrayHandle s,
        int indx1, int indx2, int indx3);
void svGetLogicArrElemVec32(svLogicVec32* d, const svOpenArrayHandle s, int
        indx1,
        ...);
void svGetLogicArrElem1Vec32(svLogicVec32* d, const svOpenArrayHandle s, int
        indx1);
void svGetLogicArrElem2Vec32(svLogicVec32* d, const svOpenArrayHandle s, int
        indx1,
        int indx2);
void svGetLogicArrElem3Vec32(svLogicVec32* d, const svOpenArrayHandle s,
        int indx1, int indx2, int indx3);
/* From simulator storage into user space */
void svGetBitArrElemVecVal(svBitVecVal* d, const svOpenArrayHandle s,
        int indx1, ...);
void svGetBitArrElem1VecVal(svBitVecVal* d, const svOpenArrayHandle s,
        int indx1);
void svGetBitArrElem2VecVal(svBitVecVal* d, const svOpenArrayHandle s,
        int indx1, int indx2);
void svGetBitArrElem3VecVal(svBitVecVal* d, const svOpenArrayHandle s,
        int indx1, int indx2, int indx3);
void svGetLogicArrElemVecVal(svLogicVecVal* d, const svOpenArrayHandle s,
        int indx1, ...);
void svGetLogicArrElem1VecVal(svLogicVecVal* d, const svOpenArrayHandle s,
        int indx1);
void svGetLogicArrElem2VecVal(svLogicVecVal* d, const svOpenArrayHandle s,
        int indx1, int indx2);
void svGetLogicArrElem3VecVal(svLogicVecVal* d, const svOpenArrayHandle s,
        int indx1, int indx2, int indx3);

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

```
svLogic svGetLogicArrElem(const svOpenArrayHandle s, int indx1, ...);
svLogic svGetLogicArrElem1(const svOpenArrayHandle s, int indx1);
svLogic svGetLogicArrElem2(const svOpenArrayHandle s, int indx1, int indx2);
svLogic svGetLogicArrElem3(const svOpenArrayHandle s, int indx1, int indx2, int
        indx3);
void svPutLogicArrElem(const svOpenArrayHandle d, svLogic value, int indx1, ...);
void svPutLogicArrElem1(const svOpenArrayHandle d, svLogic value, int indx1);
void svPutLogicArrElem2(const svOpenArrayHandle d, svLogic value, int indx1,
        int indx2);
void svPutLogicArrElem3(const svOpenArrayHandle d, svLogic value, int indx1, int
        indx2,
        int indx3);
void svPutBitArrElem(const svOpenArrayHandle d, svBit value, int indx1, ...);
void svPutBitArrElem1(const svOpenArrayHandle d, svBit value, int indx1);
void svPutBitArrElem2(const svOpenArrayHandle d, svBit value, int indx1, int
        indx2);
void svPutBitArrElem3(const svOpenArrayHandle d, svBit value, int indx1, int
        indx2,
        int indx3);


/* Functions for working with DPI context functions */

/* Retrieve the active instance scope currently associated with the executing
   imported function.
   Unless a prior call to svSetScope has occurred, this is the scope of the
   function's declaration site, not call site.
   Returns NULL if called from C code that is *not* an imported function. */
/*
 * Retrieve the active instance scope currently associated with the executing
 * imported function.  Unless a prior call to svSetScope has occurred, this
 * is the scope of the function's declaration site, not call site.
 * Returns NULL if called from C code that is *not* an imported function.
 */
svScope svGetScope();

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

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

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

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

/*
```

```
/* Retrieve an arbitrary user data pointer that was previously
 * stored by a call to svPutUserData(). See the comment above
 * svPutUserData() for an explanation of userKey, as well as
 * restrictions on NULL and illegal svScope and userKey values.
 * This function returns NULL for all error cases, 0 upon success.
 * This function also returns NULL in the event that a prior call
 * to svPutUserData() was never made.
 */
void* svGetUserData(const svScope scope, void* userKey);

/*
/* Returns the file and line number in the SV code from which the extern call
 * was made. If this information available, returns TRUE and updates fileName
 * and lineNumber to the appropriate values. Behavior is unpredictable if
 * fileName or lineNumber are not appropriate pointers. If this information is
 * not available return FALSE and contents of fileName and lineNumber not
 * modified. Whether this information is available or not is implementation
 * specific. Note that the string provided (if any) is owned by the SV
 * implementation and is valid only until the next call to any SV function.
 * Applications must not modify this string or free it
 */
int svGetCallerInfo(char **fileName, int *lineNumber);

/*
 * Returns 1 if the current execution thread is in the disabled state.
 * Disable protocol must be adhered to if in the disabled state.
 */
int svIsDisabledState();

/*
 * Imported functions call this API function during disable processing to
 * acknowledge that they are correctly participating in the DPI disable protocol.
 * This function must be called before returning from an imported function that
is
 * in the disabled state.
 */
void svAckDisabledState();



/*
 ************************************************************
 * DEPRECATED PORTION OF FILE STARTS FROM HERE.
 * IEEE-P1800-compliant tools may not provide
 * support for the following functionality.
 ************************************************************
 */

/*
 * Canonical representation of packed arrays
 * 2-state and 4-state vectors, modeled upon PLI's avalue/bvalue
 */
#define SV_CANONICAL_SIZE(WIDTH) (((WIDTH)+31)>>5)
typedef unsigned int svBitVec32;/* (a chunk of) packed bit array */
typedef struct { unsigned int c; unsigned int d;}
svLogicVec32; /* (a chunk of) packed logic array */

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

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

/* Translation between the actual representation 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);

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

/* 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);

/*
 * 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
 */

/* canonical <-- actual */
void svGetPartSelectBit(svBitVec32* d, const svBitPackedArrRef s, int i,
        int w);
svBitVec32 svGetBits(const svBitPackedArrRef s, int i, int w);
svBitVec32 svGet32Bits(const svBitPackedArrRef s, int i); /* 32-bits */

unsigned long long svGet64Bits(const svBitPackedArrRef s, int i);

/* 64-bits */
void svGetPartSelectLogic(svLogicVec32* d, const svLogicPackedArrRef s, int i,
        int w);
/* actual <-- canonical */
void svPutPartSelectBit(svBitPackedArrRef d, const svBitVec32 s, int i, int w);
void svPutPartSelectLogic(svLogicPackedArrRef d, const svLogicVec32 s, int i,
        int w);

/*
 * Functions for open array translation between simulator and canonical
representations.
 * These functions copy the whole packed array in either direction. The user is
 * responsible for allocating an array in the canonical representation.
 */

/* s=source, d=destination */
/* actual <-- canonical */
void svPutBitArrElemVec32(const svOpenArrayHandle d, const svBitVec32* s,
        int indx1, ...);
void svPutBitArrElem1Vec32(const svOpenArrayHandle d, const svBitVec32* s, int
        indx1);
void svPutBitArrElem2Vec32(const svOpenArrayHandle d, const svBitVec32* s, int
        indx1,
        int indx2);
void svPutBitArrElem3Vec32(const svOpenArrayHandle d, const svBitVec32* s,
        int indx1, int indx2, int indx3);
void svPutLogicArrElemVec32(const svOpenArrayHandle d, const svLogicVec32* s,
        int indx1, ...);
void svPutLogicArrElem1Vec32(const svOpenArrayHandle d, const svLogicVec32* s,
```

```
        int indx1);
void svPutLogicArrElem2Vec32(const svOpenArrayHandle d, const svLogicVec32* s,
        int indx1, int indx2);
void svPutLogicArrElem3Vec32(const svOpenArrayHandle d, const svLogicVec32* s,
        int indx1, int indx2, int indx3);

/* canonical <-- actual */
void svGetBitArrElemVec32(svBitVec32* d, const svOpenArrayHandle s, int indx1,
        ...);
void svGetBitArrElem1Vec32(svBitVec32* d, const svOpenArrayHandle s, int indx1);
void svGetBitArrElem2Vec32(svBitVec32* d, const svOpenArrayHandle s, int indx1,
        int indx2);
void svGetBitArrElem3Vec32(svBitVec32* d, const svOpenArrayHandle s,
        int indx1, int indx2, int indx3);
void svGetLogicArrElemVec32(svLogicVec32* d, const svOpenArrayHandle s, int
        indx1,
        ...);
void svGetLogicArrElem1Vec32(svLogicVec32* d, const svOpenArrayHandle s, int
        indx1);
void svGetLogicArrElem2Vec32(svLogicVec32* d, const svOpenArrayHandle s, int
        indx1,
        int indx2);
void svGetLogicArrElem3Vec32(svLogicVec32* d, const svOpenArrayHandle s,
        int indx1, int indx2, int indx3);

/*
 ********************************************************
 * DEPRECATED PORTION OF FILE ENDS HERE.
 ********************************************************
 */

#ifdef __cplusplus
}
#endif

#endif
```

~~F.2 Source-level compatibility include file svdpi_src.h~~

```
/* macros for declaring variables to represent the SystemVerilog */
/* packed arrays of type bit or logic */
/* WIDTH= number of bits,NAME = name of a declared field/variable */
#define SV_BIT_PACKED_ARRAY(WIDTH,NAME)/* actual definition goes here */
#define SV_LOGIC_PACKED_ARRAY(WIDTH,NAME)/* actual definition goes here */
```