SystemVerilog 3.1
C/C++ Interface
API Reference Manual
DRAFT

Version 0.8

March 27, 2003

Notices

The information contained in this manual represents the definition of the SystemVerilog 3.1 C/C++ API as reviewed and released by Accellera in March 2003.

Accellera reserves the right to make changes to the SystemVerilog 3.1 C/C++ API and this manual in subsequent revisions and makes no warranties whatsoever with respect to the completeness, accuracy, or applicability of the information in this manual, when used for production design and/or development.

Accellera does not endorse any particular simulator or other CAE tool that is based on the SystemVerilog 3.1 C/C++ API.

Suggestions for improvements to the SystemVerilog 3.1 C/C++ API and/or to this manual are welcome. They should be sent to the SystemVerilog 3.1 C/C++ API email reflector

sv-cc@server.eda.org

or to the address below.

The current Working Group's website address is

www.eda.org/sv-cc

Information about Accellera and membership enrollment can be obtained by inquiring at the address below.

Published as: SCE-MI Reference Manual
Version 0.8, March 27, 2003.

Published by: Accellera
1370 Trancas Street, #163
Napa, CA 94558
Phone: (707) 251-9977
Fax: (707) 251-9877

Printed in the United States of America.

The following individuals contributed to the creation, editing, and review of SystemVerilog 3.1 C/C++ API.

Fill-in this list

Joe Daniels		Technical Editor
Vassilios Gerousis	Infineon	
John Stickley	Mentor Graphics	

Revision history:

Revise this

Version 0.3, 1st draft	02/26/03
Version 0.5	03/20/2003

Table of Contents

Section 1 Direct Programming Interface (DPI)

This chapter highlights the Direct Programming Interface and provides a detailed description of the SysteVerilog layer of the interface. The C layer is defined in Annex A.

1.1 Overview

DPI is an interface between SystemVerilog and a foreign programming language. It consists of two separate layers: the SystemVerilog layer and a foreign language layer. Both sides of DPI are fully isolated. Which programming language is actually used as the foreign language is transparent and irrelevant for the SystemVerilog side of this interface. Neither SystemVerilog compiler nor the foreign language compiler is required to analyze the source code in the other's language. Different programming languages can be used and supported with the same intact SystemVerilog layer. For now, however, SystemVerilog 3.1 defines a foreign language layer only for the C programming language. See Annex A for more details.

The motivation for this interface is two-fold. The methodological requirement is that the interface should allow a heterogeneous system to be built (a design or a testbench) in which some components may be written in a language (or more languages) other than SystemVerilog, hereinafter called the foreign language. On the other hand, there is also a practical need for an easy and efficient way to connect existing code, usually written in C or C++, without the knowledge and the overhead of PLI or VPI.

DPI follows the principle of a black box: the specification and the implementation of a component is clearly separated and the actual implementation is transparent to the rest of the system. Therefore, the actual programming language of the implementation is also transparent, though this standard defines only C linkage semantics. The separation between SystemVerilog code and the foreign language is based on using functions as the natural encapsulation unit in SystemVerilog. By and large, any function can be treated as a black box and implemented either in SystemVerilog or in the foreign language in a transparent way, without changing its calls.

1.1.1 Functions

DPI allows direct inter-language function calls between the languages on either side of the interface. Specifically, functions implemented in a foreign language can be called from SystemVerilog; such functions are referred to as *imported functions*. SystemVerilog functions that are to be called from a foreign code shall be specified in `export` declarations (see section 1.6 for more details). DPI allows for passing SystemVerilog data between the two domains through function arguments and results. There is no intrinsic overhead in this interface.

All functions used in DPI are assumed to complete their execution instantly and consume `0` (zero) simulation time, just as normal SystemVerilog functions. DPI provides no means of synchronization other than by data exchange and explicit transfer of control.

Every imported function needs to be declared. A declaration of an imported function is referred to as an *import declaration*. Import declarations are very similar to SystemVerilog function declarations. Import declarations may occur anywhere where SystemVerilog function definitions are permitted. An import declaration is considered to be a definition of a SystemVerilog function with a foreign language implementation. The same foreign function can be used to implement multiple SystemVerilog functions (this can be a useful way of providing differing default argument values for the same basic function), but a given SystemVerilog name can only be defined once per scope. Imported functions can have zero or more formal `input`, `output`, and `inout` arguments, and they can return a result or be defined as `void` functions.

DPI is based entirely upon SystemVerilog constructs. The usage of imported functions is identical as for native SystemVerilog functions. With few exceptions imported functions and native functions are mutually exchangeable. Calls ~~sites~~ of imported functions are indistinguishable from calls of SystemVerilog functions. This facilitates ease-of-use and minimizes the learning curve.

1.1.2 Data types

SystemVerilog data types are the sole data types that can cross the boundary between SystemVerilog and a foreign language in either direction (i.e., when an imported function is called from SystemVerilog code or an exported SystemVerilog function is called from a foreign code). It is not possible to import the data types or directly use the type syntax from another language. With some restrictions and with some notational extensions, most SystemVerilog data types are allowed in ~~the~~ DPI ~~interface~~. Function result types are restricted to small values, however (see section 1.4.5).

Formal arguments of an imported function can be specified as open arrays. A formal argument is an *open array* when a range of one or more of its dimensions, packed or unpacked, is unspecified (denoted by using empty square brackets (`[]`)). This is solely a relaxation of the argument-matching rules. An actual argument shall match the formal one regardless of the range(s) for its corresponding dimension(s), which facilitates writing generalized code that can handle SystemVerilog arrays of different sizes. See section 1.4.6.1.

1.1.2.1 Data representation

DPI does not add any constraints on how SystemVerilog-specific data types are actually implemented. Optimal representation can be platform dependent. The layout of 2- or 4-state packed structures and arrays is implementation- and platform-dependent.

The implementation (representation and layout) of 4-state values, structures, and arrays is irrelevant for SystemVerilog semantics, and can only impact the foreign side of the interface.

1.2 Two layers of the DPI

DPI consists of two separate layers: the SystemVerilog layer and a foreign language layer. The SystemVerilog layer does not depend on which programming language is actually used as the foreign language. Although different programming languages can be supported and used with the intact SystemVerilog layer, SystemVerilog 3.1 defines a foreign language layer only for the C programming language. Nevertheless, SystemVerilog code shall look identical and its semantics shall be unchanged for any foreign language layer.

1.2.1 DPI SystemVerilog ~~layer~~

The SystemVerilog side of DPI does not depend on the foreign programming language. In particular, the actual function call protocol and argument passing mechanisms used in the foreign language are transparent and irrelevant to SystemVerilog. SystemVerilog code shall look identical regardless of what code the foreign side of the interface is using. The semantics of the SystemVerilog side of the interface is independent from the foreign side of the interface.

This chapter does not constitute a complete interface specification. It only describes the functionality, semantics and syntax of the SystemVerilog ~~layer~~ of the interface. The other half of the interface, the foreign language layer, defines the actual argument passing mechanism and the methods to access (read/write) formal arguments from the foreign code. See Annex A for more details.

1.2.2 DPI foreign language layer

The foreign language layer of the interface (which is transparent to SystemVerilog) shall specify how actual arguments are passed, how they can be accessed from the foreign code, how SystemVerilog-specific data types (such as `logic` and `packed`) are represented, and how to translate them to and from some predefined C-like types.

The data types allowed for formal arguments and results of imported functions or exported functions are generally SystemVerilog types (with some restrictions and with notational extensions for open arrays). The user is responsible for specifying in their foreign code the native types equivalent to the SystemVerilog types used in imported declarations or `export` declarations. ~~EDA~~ Software tools, like a SystemVerilog compiler, can facilitate the mapping of SystemVerilog types onto foreign native types by generating the appropriate function headers.

The SystemVerilog compiler or simulator shall generate and/or use the function call protocol and argument passing mechanisms required for the intended foreign language layer. The same SystemVerilog code (compiled accordingly) shall be usable with different foreign language layers, regardless of the data access method assumed in a specific layer. Annex A defines DPI foreign language layer for the C programming language.

1.3 Global name space of imported and exported functions

Every function imported to SystemVerilog must eventually resolve to a global symbol. Similarly, every function exported from SystemVerilog defines a global symbol. Thus the functions imported to and exported from SystemVerilog have their own global name space, different from `$root` name space. Global names of imported and exported functions must be unique (no overloading is allowed) and shall follow C conventions for naming; specifically, such names must start with a letter or underscore, and may be followed by alphanumeric characters or undersores. Exported and imported functions, however, may be declared with local SystemVerilog names. Import and export declarations allow users to specify a global name for a function in addition to its declared name. If a global name is not explicitly given, it will be the same as the SystemVerilog function name. Example:

```
    export “DPI” foo_plus = function \foo+ ; // “foo+” exported as “foo_plus”
    export “DPI” function foo; // “foo” exported under its own name
    import “DPI” init_1 = function void \init[1] (); // “init_1” is a global name
```

The same global function may be referred to in multiple import declarations in different scopes or/and with different SystemVerilog names, see section 1.4.4.

Multiple export declarations are allowed with the same cname, explicit or implicit, as long as they are in different scopes and have the same type signature (as defined in section 1.4.4 for imported functions). Multiple export declarations with the same cname in the same scope are forbidden.

1.4 Imported functions

The usage of imported functions is similar as for native SystemVerilog functions.

1.4.1 Required properties of imported functions - semantic constraints

This section defines the semantic constraints imposed on imported functions. Some semantic restrictions are shared by all imported functions. Other restrictions depend on whether the special properties `pure` (see section 1.4.2) or `context` (see section 1.4.3) are specified for an imported function. A SystemVerilog compiler is not able to verify that those restrictions are observed and if those restrictions are not satisfied, the effects of such imported function calls can be unpredictable.

1.4.1.1 Instant completion

Imported functions shall ~~contain no timing control whatsoever, directly or indirectly. imported functions shall be non-blocking; they shall~~ complete their execution instantly and consume zero-simulation time, ~~i.e., no simulation time passes during the execution of imported function.~~ similarly to native functions.

1.4.1.2 `input` and `output` arguments

Imported functions can have `input` and `output` arguments. The formal `input` arguments shall not be modified. If such arguments are changed within a function, the changes shall not be visible outside the function; the actual arguments shall not be changed.

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

1.4.1.3 Special properties `pure` and `context`

Special properties can be specified for an imported function: as `pure` or as `context` (see also section 1.4.2 or section 1.4.3).

A function whose result depends solely on the values of its input arguments and with no side effects may be specified as `pure`. This will usually allow for more optimizations and thus may result in improved simulation performance. Section section 1.4.2 details the rules that must be obeyed by `pure` functions.

An imported function that is intended to call exported functions or to access SystemVerilog data objects other then its actual arguments (e.g. via VPI or PLI calls) must be specified as `context`. Calls of `context` functions are specially instrumented and may impair SystemVerilog compiler optimizations; therefore simulation performance may decrease if the `context` property is ~~used~~ specified when not necessary. A function not specified as `context` shall not read or write any data objects from SystemVerilog other then its actual arguments. For functions not specified as `context`, the effects of calling PLI, VPI, or exported SystemVerilog functions can be unpredictable and can lead to unexpected behavior; such calls can even crash. Section section 1.4.3 details the restrictions that must be obeyed by non-`context` functions.

1.4.1.4 Memory management

The memory spaces owned and allocated by the foreign code and SystemVerilog code are disjoined. Each side is responsible for its own allocated memory. Specifically, an imported function shall not free the memory allocated by SystemVerilog code (or the SystemVerilog compiler) nor expect SystemVerilog code to free the memory allocated by the foreign code (or the foreign compiler). This does not exclude scenarios where foreign code allocates a block of memory, then passes a handle (i.e., a pointer) to that block to SystemVerilog code, which in turn calls an imported function (e.g. C standard function `free`) which directly or indirectly frees that block.

NOTE—In this last scenario, a block of memory is allocated and freed in the foreign code, even when the standard functions `malloc` and `free` are called directly from SystemVerilog code.

1.4.2 Pure functions

A `pure` function call can be safely eliminated if its result is not needed or if the previous result for the same values of input arguments is available somehow and can be reused without needing to recalculate. Only non-void functions with no `output` or `inout` arguments can be specified as `pure`. Functions specified as `pure` shall have no side effects whatsoever; their results need to depend solely on the values of their input arguments. Calls to such functions can be removed by SystemVerilog compiler optimizations or replaced with the values previously computed for the same values of the input arguments.

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

— perform any file operations

— read or write anything in the broadest possible meaning, includes i/o, environment variables, objects from the operating system or from the program or other processes, shared memory, sockets, etc.

— access any persistent data, like global or static variables.

If a `pure` function does not obey the above restrictions, SystemVerilog compiler optimizations can lead to unexpected behavior, due to eliminated calls or incorrect results being used.

1.4.3 Context functions

Some DPI imported functions require that the context of their call is known. It takes special instrumentation of their call instances to provide such context; for example, an internal variable referring to the "current instance" might need to be set. To avoid any unnecessary overhead, imported function calls in SystemVerilog code are not instrumented unless the imported function is specified as `context`.

All DPI exported functions require that the context of their call is known. This occurs since SystemVerilog function declarations always occur in instantiable scopes, hence allowing a multiplicity of unique function instances.

For the sake of simulation performance, an imported function call shall not block SystemVerilog compiler optimizations. An imported function not specified as `context` shall not access any data objects from SystemVerilog other than its actual arguments. Only the actual arguments can be affected (read or written) by its call.

Therefore, a call o[illegible]-`context` function is not a barrier for optimizations. A `context` imported function, however, can acces[illegible]read or write) any SystemVerilog data objects by calling PLI/VPI, or by calling an export function. Therefore, a call to a `context` function is a barrier for SystemVerilog compiler optimizations.

Only calls of `context` imported functions are properly instrumented and cause conservative optimizations; therefore, only those functions can safely call all functions from other APIs, including PLI and VPI functions or exported SystemVerilog functions. For imported functions not specified as `context`, the effects of calling PLI, VPI, or SystemVerilog functions can be unpredictable and such calls can crash if the callee requires a context that has not been properly set. However note that declaring an import context function does not automatically make any other simulator interface automatically available. For VPI access (or any other interface access) to be possible, the appropriate implementation defined mechanism must still be used to enable these interface(s). Note also that DPI calls do not automatically create or provide any handles or any special environment that may be needed by those other interfaces. It is the user's responsibility to create, manage or otherwise manipulate the required handles/environment(s) needed by the other interfaces.

Con[illegible]functions are always implicitly supplied a scope representing the fully qualified instance name within whic[illegible]e import declaration was present. This scope defines which exported SystemVerilog functions may be called directly from the imported function; only functions defined and exported from the same scope as the import can be called directly. To call any other exported SystemVerilog functions, the imported function will first have to modify its current scope, in essence performing the foreign language equivalent of a SystemVerilog hierarchical function call.

Special DPI utility functions exist that allow e[illegible]nal functions to retrieve and operate on their scope. See Annex A for more details.

1.4.4 Import declarations

Also cross-reference to section 10.6, import and export functions

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

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

Syntax:

import_dpi_decl ::= ***import*** *"DPI" [**pure**|**context**] [cname=] <named_function_proto>;*

where named_function_proto is as defined in section A.2.6 of SV 3.1 BNF

/* EDITOR: UPDATE ABOVE CROSS-REFERENCE AS NECESSARY */

An import declaration specifies the function name, function result type, and types and directions of formal arguments. It can also provide optional default values for formal arguments. Formal argument names are optional unless argument passing by name is needed. An import declaration can also specify an optional function property: `context` or `pure`.

Note that an import declaration is equivalent to defining a function of that name in the SystemVerilog scope in which the import declaration occurs, and thus multiple imports of the same function name into the same scope are forbidden. Note that this declaration scope is particularly important in the case of imported context functions, see section 1.4.3; for non-context imported functions the declaration scope has no other implications other than defining the visibility of the function.

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

For any given `cname` (whether explicitly defined with cname=, or automatically determined from the function name), all declarations, regardless of scope, **must** have exactly the same type signature. Thesignature 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 may be associated with an extern definition.

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

A formal argument name is required to separate the packed and the unpacked dimensions of an array.

The qualifier `ref` can not be used in external declarations. The actual implementation of argument passing depends solely on the foreign language layer and its implementation and shall be transparent to the SystemVerilog side of the interface.

The following are examples of external declarations.

```
import “DPI” function void myInit();
// from standard math library
import “DPI” pure function real sin(real);
// from standard C library: memory management
import “DPI” function handle malloc(int size); // standard C function
import “DPI” function void free(handle ptr); // standard C function
// abstract data structure: queue
import “DPI” function handle newQueue(input string name_of_queue);
// Note the following import uses the same foreign function for
// implementation as the prior import, but has different SystemVerilog name
// and provides a default value for the argument.
import “DPI” newQueue=function handle newAnonQueue(input string s=NULL);
import “DPI” function handle newElem(bit [15:0]);
import “DPI” function void enqueue(handle queue, handle elem);
import “DPI” function handle dequeue(handle queue);
// miscellanea
import “DPI” function bit [15:0] getStimulus();
import “DPI” context function void processTransaction(handle elem,
                     output logic [64:1] arr [0:63]);
```

1.4.5 Function result

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

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

— packed `bit` arrays up to 32 bits and all types that are eventually equivalent to packed `bit` arrays up to 32 bits.

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

1.4.6 Types of formal arguments

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

— Enumerated data types are not supported directly. Instead, an enumerated data type is interpreted as the type associated with that enumerated type.

— SystemVerilog does not specify the actual memory representation of packed structures or any arrays, packed or unpacked. Unpacked structures have an implementation-dependent packing, normally matching the C compiler.

— The actual memory representation of SystemVerilog data types is transparent for SystemVerilog semantics and irrelevant for SystemVerilog code. It can be relevant for the foreign language code on the other side of the interface, however; a particular representation of the SystemVerilog data types can be assumed. This shall not restrict the types of formal arguments of imported functions, with the exception of unpacked arrays. SystemVerilog implementation can restrict which SystemVerilog unpacked arrays are passed as actual arguments for a formal argument which is a sized array, although they can be always passed for an unsized (i.e., open) array. Therefore, the correctness of an actual argument might be implementation-dependent. Nevertheless, an open array provides an implementation-independent solution.

1.4.6.1 Open arrays

The size of the packed dimension, the unpacked dimension, or both dimensions can remain unspecified; such cases are referred to as *open arrays* (or unsized arrays). Open arrays allow the use of generic code to handle different sizes.

Formal arguments of imported functions can be specified as open arrays. (Exported SystemVerilog functions cannot have formal arguments specified as open arrays.) A formal argument is an *open array* when a range of one or more of its dimensions is unspecified (denoted by using square brackets (`[]`)). This is solely a relaxation of the argument-matching rules. An actual argument shall match the formal one regardless of the range(s) for its corresponding dimension(s), which facilitates writing generalized code that can handle SystemVerilog arrays of different sizes.

Although the packed part of an array can have an arbitrary number of dimensions, in the case of open arrays only a single dimension is allowed for the packed part. This is not very restrictive, however, since any packed type is eventually equivalent to one-dimensional packed ass) 6.3ne9-1.9(r 59(;(as)-5.)-5Thh)-5.e.1(p)6.3nuh o2.1(to)-5(n)-2.4(t

1.5 Calling imported functions

The usage of imported functions is identical as for native SystemVerilog functions. The usage and syntax for calling imported functions is identical as for native SystemVerilog functions. Specifically, arguments with default values can be omitted from the call; arguments can be passed by name, if all fomal arguments are named.

1.5.1 Argument passing

Arguments passing for imported functions is ruled by *the WYSIWYG* principle: *What You Specify Is What You Get*, see Section 1.5.1.1. The evaluation order of formal arguments follows general SystemVerilog rules.

Argument compatibility and coercion rules are the same as for native SystemVerilog functions. If a coercion is needed, a temporary variable is created and passed as the actual argument. For `input` and `inout` arguments, the temporary variable is initialized with the value of actual argument with the appropriate coercion; for `output` or `inout` arguments, the value of the temporary variable is assigned to the actual argument with the appropriate conversion. The assignments between a temporary and the actual argument follow general SystemVerilog rules for assignments and automatic coercion.

On the SystemVerilog side of the interface, the values of actual arguments for formal input arguments of imported functions shall not be affected by the callee; the initial values of formal output arguments of imported functions are unspecified (and can be implementation-dependent), and the necessary coercions, if any, are applied as for assignments. imported functions shall not modify the values of their `input` arguments.

For the SystemVerilog side of the interface, the semantics of arguments passing is as if `input` arguments are passed by *copy-in*, `output` arguments are passed by *copy-out*, and `inout` arguments were passed by *copy-in*, *copy-out*. The terms *copy-in* and *copy-out* do not impose the actual implementation, they refer only to "hypothetical assignment".

The actual implementation of argument passing is transparent to the SystemVerilog side of the interface. In particular, it is transparent to SystemVerilog whether an argument is actually passed *by value* or *by reference*. The actual argument passing mechanism is defined in the foreign language layer. See Annex A for more details.

1.5.1.1 "What You Specify Is What You Get" principle

The principle "What You Specify Is What You Get" guarantees the types of formal arguments of imported functions — an actual argument is guaranteed to be of the type specified for the formal argument, with the exception of open arrays (for which unspecified ranges are statically unknown). Formal arguments, other than open arrays, are fully defined by import declaration; they shall have ranges of packed or unpacked arrays exactly as specified in the import declaration. Only the declaration site of the imported function is relevant for such formal arguments.

Another way to state this is that no compiler (either C or SystemVerilog) can make argument coercions between a caller's declared formals and the callee's declared formals. This is because the callee's formal arguments are declared in a different language than the caller's formal arguments; hence there is no visible relationship between the two sets of formals. Users are expected to understand all argument relationships and provide properly matched types on both sides of the interface.

Formal arguments defined as open arrays have the size and ranges of the corresponding actual arguments, i.e., have the ranges of packed or unpacked arrays exactly as that of the actual argument. The unsized ranges of open arrays are determined at a call site; the rest of type information is specified at the import declaration.

So, if a formal argument is declared as `bit [15:8] b []`, then it is the import declaration which specifies the formal argument is an unpacked array of packed bit array with bounds `15` to `8`, while the actual argument used at a particular call site defines the bounds for the unpacked part for that call.

1.5.2 Value changes for output and inout arguments

The SystemVerilog simulator is responsible for handling value changes for `output` and `inout` arguments. Such changes shall be detected and handled after control returns from imported functions to SystemVerilog code.

For `output` and `inout` arguments, the value propagation (i.e., value change events) happens as if an actual argument was assigned a formal argument immediately after control returns from imported functions. If there is more than one argument, the order of such assignments and the related value change propagation follows general SystemVerilog rules.

1.6 Exported functions

DPI allows calling SystemVerilog functions from another language. However, such functions must adhere to the same restrictions on argument types and results as are imposed on imported functions. It is an error to export a unctio at does not satisfy such constraints.

SystemVerilog functions that can be called from a f gn code need to be specified in `export` declarations. Export declarations are allowed to occur only in th cope in which the function being exported is defined. Only one export declaration per function is allowed in a given scope.

Note that class member functions may not be exported, but all other SystemVerilog functions may be exported.

Similar to `import` declarations, `export` declarations can define an optional `cname` to be used in the foreign language when calling an exported function.

Syntax:

export_dpi_decl ::= ***export*** *“DPI” [cname=]* **function** *fname* ***;***

`cname` is optional here, it defaults to *`fname`*. For rules describing cname, see section 1.3. Note that all export functions are always *context* functions. No two functions in the same SystemVerilog scope may be exported with the same explicit or implicit cname. The export declaration and the definition of the corresponding SystemVerilog function can occur in any order. Only one export declaration is permitted per SystemVerilog function

Section 2
SystemVerilog Assertion API

This chapter defines the Assertion Application Programming Interface (API) in SystemVerilog 3.1/draft 2.

2.1 Requirements

SystemVerilog 3.1/draft 2 provides assertion capabilities to enable:

— a user’s C code to react to assertion events,

2.2.2 Object properties

This section lists the object property VPI calls. The VPI reserved range for these call is `700 - 729`.

```
/* Directives as properties */

#define vpiSequenceAssertion  701
#define vpiAssertAssertion    702
#define vpiAssumeAssertion    703
#define vpiRestrictAssertion  704
#define vpiCoverAssertion     705
#define vpiCheckAssertion     705 /* inlined behavior assertion */
#define vpiOtherDirectiveAssertion 706 /* placeholder for other assertion
directive */
```

2.2.3 Callbacks

This section lists the system callbacks. The VPI reserved range for these call is `700 - 719`.

1) Assertion

```
#define cbAssertionStart         700
#define cbAssertionSuccess       701
#define cbAssertionFailure       702
#define cbAssertionStepSuccess   703
#define cbAssertionStepFailure   704
#define cbAssertionDisable       705
#define cbAssertionEnable        706
#define cbAssertionReset         707
#define cbAssertionKill          708
```

2) “Assertion system”

```
#define cbAssertionSysInitialized709
#define cbAssertionSysStart      710
#define cbAssertionSysStop       711
#define cbAssertionSysEnd        712
#define cbAssertionSysReset      713
```

2.2.4 Control constants

This section lists the system control constant callbacks. The VPI reserved range for these call is `730 - 759`.

1) Assertion

```
#define vpiAssertionDisable      730
#define vpiAssertionEnable       731
#define vpiAssertionReset        732
#define vpiAssertionKill         733
#define vpiAssertionEnableStep   734
#define vpiAssertionDisableStep  735
```

2) Assertion stepping

```
#define vpiAssertionClockSteps   736
```

3) “Assertion system”

```
#define vpiAssertionSysStart     737
#define vpiAssertionSysStop      738
#define vpiAssertionSysEnd       739
```

```
#define vpiAssertionSysReset     740
```

2.3 Static information

This section defines how to obtain assertion handles and other static assertion information.

2.3.1 Obtaining assertion handles

SystemVerilog 3.1/draft 2 extends the VPI module iterator model (i.e., the instance) to encompass assertions, as shown in Figure 2-1—. **Revise this xref w/ Stu; also check/revise variable settings, etc.**

The following steps highlight how to obtain the assertion handles for named assertions.

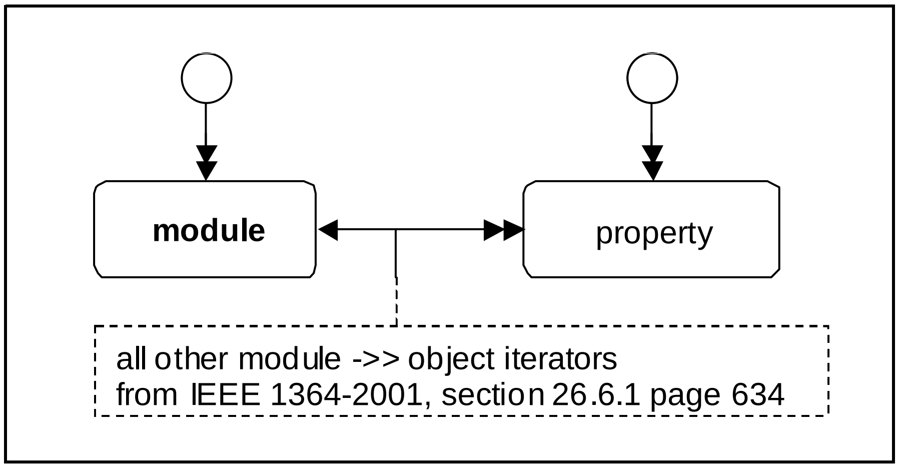

module
property
all other module ->> object iterators
from IEEE 1364-2001, section 26.6.1 page 634

Figure 2-1—Encompassing assertions

1) Iterate all assertions in the design: use a `NULL` reference handle (ref) to `vpi_iterate()`, e.g.,

```
itr = vpi_iterate(vpiAssertion, NULL);
while (assertion = vpi_scan(itr)) {
   /* process assertion */
}
```

2) Iterate all assertions in an instance: pass the appropriate instance handle as a reference handle to `vpi_iterate()`, e.g.,

```
itr = vpi_iterate(vpiAssertion, instanceHandle);
while (assertion = vpi_scan(itr)) {
   /* process assertion */
}
```

3) Obtain the assertion by name: extend `vpi_handle_by_name` to also search for assertion names in the appropriate scope(s), e.g.,

```
vpiHandle = vpi_handle_by_name(assertName, scope)
```

4) /* room for expanding iteration later, filtering based on "object property" e.g.

```
itr = vpi_iterate_property(vpiAssertion, /* property_here: e.g.
vpiCheckAssertion*/, NULL);
while (assertion = vpi_scan(itr)) {
/* process assertion *
}
```

NOTES

1—As with all VPI handles, assertion handles are handles to a specific instance of a specific assertion.

2—These iterators return both assertions and immediate non-temporal checks.

3—Unnamed assertions cannot be found by name.

2.3.2 Obtaining static assertion information

The following information about an assertion is considered to be "static".

— Assertion name

— Instance in which the assertion occurs

— Module definition containing the assertion

— Assertion directive

1) *assert*

2) *check*

3) *assume*

4) *cover*

5) *sequence*

6) Any assertion updates from the *SV-AC*.

— Assertion source information: the file, line, and column where the assertion is defined.

— Assertion clocking domain/expression2

2.3.2.1 Using `vpi_get_assertion_info`

Static information can be obtained directly from an assertion handle by using `vpi_get_assertion_info`, as shown below.

```
typedef struct t_vpi_source_info {
      PLI_BYTE* *fileName;
PLI_INT32 startLine;
PLI_INT32 startColumn;
PLI_INT32 endLine;
PLI_INT32 endColumn;
} s_vpi_source_info, *p_vpi_source_info;
typedef struct t_vpi_assertion_info {
      PLI_BYTE8 *name;     /* name of assertion */
      vpiHandle instance; /* instance containing assertion */
      PLI_BYTE8 modname;   /* name of module/interface containing
               assertion */

      vpiHandle clock; /* clocking expression */
      PLI_INT32 directive; /* vpiAssume, ... */
            s_vpi_source_info sourceInfo;
   s_vpi_assertion_info, *p_vpi_assertion_info;
int vpi_get_assertion_info (assert_handle, p_vpi_assertion_info);
```

This call obtains all the static information associated with an assertion.

The inputs are a valid handle to an assertion and a pointer to an existing `s_vpi_assertion_info` data structure. On success, the function returns `TRUE` and the `s_vpi_assertion_info` data structure is filled in as appropriate. On failure, the function returns `FALSE` and the contents of the assertion data structure are unpredictable.

Assertions can occur in modules and interfaces: assertions defined in modules (by using VPI) shall have instance information; assertions in interfaces shall have a `NULL` instance handle. In either case, `modname` is the definition name.

NOTES

1—The assertion clock is an event expression supplied as the clocking expression to the assertion declaration, i.e., this is a handle to an arbitrary Verilog event expression.

2—A single call returns all the information for efficiency reasons.

2.3.2.2 Extending `vpi_get()` and `vpi_get_str()`

In addition to `vpi_get_assertion_info`, the following existing VPI functions are also extended:

```
vpi_get(), vpi_get_str()
```

`vpi_get()` can be used to query the following VPI properties from a handle to an assertion.

`vpiAssertionDirective`
returns one of `vpiAssertProperty` or `vpiCheckProperty`.

`vpiLineNo`
returns the line number where the assertion is declared.

`vpi_get_str()` can be used to obtain the following VPI properties from an assertion handle.

`vpiFileName`
returns the filename of the source file where the assertion was declared.

`vpiName`
returns the name of the assertion.

`vpiFullName`
returns the fully qualified name of the assertion.

2.4 Dynamic information

This section defines how to place assertion system and assertion callbacks.

2.4.1 Placing assertion “system” callbacks

Use `vpi_register_cb()`, setting the `cb_rtn` element to the function to be invoked and the reason element of the `s_cb_data` structure to one of the following values, to place an assertion system callback.

`cbAssertionSysInitialized`
occurs after the system has initialized. No assertion-specific actions can be performed until this callback completes. The assertion system can initialize before `cbStartOfSimulation` does or afterwards.

`cbAssertionSysStart`
the assertion system has become active and starts processing assertion attempts. This always occur after `cbAssertionSysInitialized`. By default, the assertion system is “started” on simulation startup, but the user can delay this by using assertion system control actions.

`cbAssertionSysStop`
the assertion system has been temporarily suspended. While stopped no assertion attempts are processed and no assertion-related callbacks occur. The assertion system can be stopped and resumed an arbitrary number of times during a single simulation run.

`cbAssertionSysEnd`
occurs when all assertions have completed and no new attempts will start. Once this callback occurs no more assertion-related callbacks shall occur and assertion-related actions shall have no further effect. This typically occurs after the end of simulation.

`cbAssertionSysReset`
occurs when the assertion system is reset, e.g., due to a system control action.

The callback routine invoked follows the normal VPI callback prototype and is passed an `s_cb_data` containing the callback reason and any user data provided to the `vpi_register_cb()` call.

2.4.2 Placing assertions callbacks

Use `vpi_register_assertion_cb()` to place an assertion callback; the prototype is:

```
vpiHandle vpi_register_assertion_cb(
   vpiHandle,  /* handle to assertion */
   PLI_INT32 event,/* event for which callbacks needed */
   PLI_INT32 (*cb_rtn)(       /* callback function */
       PLI_INT32 event,
       vpiHandle assertion,
       p_vpi_attempt_info info,
       PLI_BYTE8 *userData),
   PLI_BYTE8 *user_data/* user data to be supplied to cb */
);
typedef struct t_vpi_assertion_step_info {
   PLI_INT32 matched_expression_count;
   vpiHandle *matched_exprs;  /* array of expressions */
   p_vpi_source_info *exprs_source_info;  /* array of source info */
   PLI_INT32 stateFrom, stateTo;/* identify transition */
} s_vpi_assertion_step_info, *p_vpi_assertion_step_info;
typedef struct t_vpi_attempt_info {
   union {
      vpiHandle failExpr;
      p_vpi_assertion_step_info step;
   } detail;
   s_vpi_time attemptTime,
} s_vpi_attempt_info, *p_vpi_attempt_info;
```

where `event` is any of the following.

`cbAssertionStart`
an assertion attempt has started. For most assertions one attempt starts each and every clock tick.

`cbAssertionSuccess`
when an assertion attempt reaches a success state.

`cbAssertionFailure`
when an assertion attempt fails to reach a success state.

`cbAssertionStepSucess`
the progress of one “thread” along an attempt. By default, step callbacks are not enabled on any assertions; they are enabled on a per-assertion/per-attempt basis, rather than on a per-assertion basis.

`cbAssertionStepFailure`
failure to progress along one “thread” along an attempt. By default, step callbacks are not enabled on any assertions; they are enabled on a per-assertion/per-attempt basis, rather than on a per-assertion basis.

`cbAssertionDisable`
whenever the assertion is disabled (e.g., as a result of a control action).

`cbAssertionEnable`
whenever the assertion is enabled.

`cbAssertionReset`
whenever the assertion is reset.

`cbAssertionKill`
when an attempt is killed (e.g., as a result of a control action).

These callbacks are specific to a given assertion; placing such a callback on one assertion does not cause the callback to trigger on an event occurring on a different assertion. If the callback is successfully placed, a handle to the callback is returned. This handle can be used to remove the callback via `vpi_remove_cb()`. If there were errors on placing the callback, a `NULL` handle is returned. As with all other calls, invoking this function with invalid arguments has unpredictable effects.

Once the callback is placed, the user-supplied function shall be called each time the specified event occurs on the given assertion. The callback shall continue to be called whenever the event occurs until the callback is removed.

The callback function shall be supplied the following arguments:

1) the event that caused the callback
2) the handle for the assertion
3) a pointer to an attempt information structure
4) a reference to the user data supplied when the callback was placed.

The *attempt information structure* contains details relevant to the specific event that occurred.

— On disable, enable, reset and kill events, the `info` field is absent (a `NULL` pointer is given as the value of `info`).

— On start and success events, only the *attempt time* field is valid.

— On a failure event, the *attempt time* and `detail.failExpr` are valid.

— On a step callback, the *attempt time* and `detail.step` elements are valid.

On a step callback, the `detail` describes the set of expressions matched in satisfying a step along the assertion, along with the corresponding source references. In addition, the `step` also identifies the source and destination “states” needed to uniquely identify the path being taken through the assertion. *State ids* are just integers, with `0` identifying the origin state, `1` identifying an accepting state, and any other number representing some intermediate point in the assertion. It is possible for the number of expressions in a step to be `0` (zero), which represents an unconditional transition. In the case of a failing transition, the information provided is just as that for a successful one, but the last expression in the array represents the expression where the transition failed.

NOTES

1—In a failing transition, there shall always be at least one element in the expression array.

2—Placing a step callback results in the same callback function being invoked for both success and failure steps.

2.5 Control functions

This section defines how to obtain assertion system control and assertion control information.

2.5.1 Assertion system control

Use `vpi_control()`, with one of the following operators and no other arguments, to obtain assertion system control information.

Usage example: `vpi_control(vpiAssertionSysReset)`

`vpiAssertionSysReset`
discards all attempts in progress for all assertions and restore the entire assertion system to its initial state.

Usage example: `vpi_control(vpiAssertionSysStop)`

`vpiAssertionSysStop`
considers all attempts in progress as unterminated and disable any further assertions from being started.

Usage example: `vpi_control(vpiAssertionSysStart)`

`vpiAssertionSysStart`
restarts the assertion system after it was stopped (e.g., due to `vpiAssertionSysStop`). Once started, attempts shall resume on all assertions.

Usage example: `vpi_control(vpiAssertionSysEnd)`

`vpiAssertionSysEnd`
discard all attempts in progress and disable any further assertions from starting.

2.5.2 Assertion control

Use `vpi_control()`, with one of the following operators, to obtain assertion control information.

— For the following operators, the second argument shall be a valid assertion handle.

Usage example: `vpi_control(vpiAssertionReset, `*`assertionHandle`*`)`

`vpiAssertionReset`
discards all current attempts in progress for this assertion and resets this assertion to its initial state.

Usage example: `vpi_control(vpiAssertionDisable, `*`assertionHandle`*`)`

`vpiAssertionDisable`
disables the starting of any new attempts for this assertion. This has no effect on any existing attempts. or if the assertion already disabled. By default, all assertions are enabled.

Usage example: `vpi_control(vpiAssertionEnable, `*`assertionHandle`*`)`

`vpiAssertionEnable`
enables starting new attempts for this assertion. This has no effect if assertion already enabled or on any existing attempts.

— For the following operators, the second argument shall be a valid assertion handle and the third argument shall be an attempt start-time (as a pointer to a correctly initialized `s_vpi_time` structure).

Usage example: `vpi_control(vpiAssertionKill, `*`assertionHandle`*`, attempt)`

`vpiAssertionKill`
discards the given attempts, but leaves the assertion enabled and does not reset any state used by this assertion (e.g., `past()` sampling).

Usage example: `vpi_control(vpiAssertionDisableStep, assertionHandle, attempt)`

`vpiAssertionDisableStep`
disables step callbacks for this assertion. This has no effect if stepping not enabled or it is already disabled.

— For the following operator, the second argument shall be a valid assertion handle, the third argument shall be an attempt start-time (as a pointer to a correctly initialized `s_vpi_time` structure) and the fourth argument shall be a “step control” constant.

Usage example:
```
vpi_control(vpiAssertionEnableStep, assertionHandle, attempt,
            vpiAssertionClockSteps)
```

`vpiAssertionEnableStep`
enables step callbacks to occur for this assertion attempt. By default, stepping is disabled for all assertions. This call has no effect if stepping is already enabled for this assertion and attempt, other than possibly changing the stepping mode for the attempt if the attempt has not occurred yet. The stepping mode of any particular attempt cannot be modified after the assertion attempt in question has started.

NOTE—In this release, the only step control constant available is `vpiAssertionClockSteps`, indicating callbacks on a per assertion/clock-tick basis. The assertion clock is the event expression supplied as the clocking expression to the assertion declaration. The assertion shall “advance” whenever this event occurs and, when stepping is enabled, such events shall also cause step callbacks to occur.

Section 3
SystemVerilog Coverage API

3.1 Requirements

This chapter defines the Coverage Application Programming Interface (API) in SystemVerilog 3.1/draft 4.

3.1.1 SystemVerilog API

The following criteria are used within this API.

1) This API shall be similar for all coverages
   There are a wide number of coverage types available, with possibly different sets offered by different vendors. Maintaining a common interface across all the different types enhances portability and ease of use.

2) At a minimum, the following types of coverage shall be supported:

   a) statement coverage

   b) toggle coverage

   c) fsm coverage

      i) fsm states

      ii) fsm transitions

   d) assertion coverage

3) Coverage APIs shall be extensible in a transparent manner, i.e., adding a new coverage type shall not break any existing coverage usage.

4) This API shall provide means to obtain coverage information from specific sub-hierarchies of the design without requiring the user to enumerate all instances in those hierarchies.

3.1.2 Naming conventions

All elements added by this interface shall conform to the Verilog Procedural Interface (VPI) interface naming conventions.

— All names are prefixed by `vpi`.

— All *type names* shall start with `vpi`, followed by initially capitalized words with no separators, e.g., `vpiCoverageStmt`.

— All callback names shall start with `cb`, followed by initially capitalized words with no separators, e.g., `cbAssertionStart`.

— All *function names* shall start with `vpi_`, followed by all lowercase words separated by underscores (_), e.g., `vpi_control()`.

3.1.3 Nomenclature

The following terms are used in this standard.

Statement coverage — whether a statement has been executed or not, where *statement* is anything defined as a statement in the LRM. *Covered* means it executed at least once. Some implementations also permit querying the execution count. The granularity of statement coverage can be per-statement or per-statement block (however defined).

FSM coverage — the number of states in a finite state machine (FSM) that this simulation reached. This standard does not require FSM automatic extraction, but a standard mechanism to force specific extraction is available via pragmas.

Toggle coverage — for each bit of every signal (wire and register), whether that bit has both a `0` value and a `1` value. *Full coverage* means both are seen; otherwise, some implementations can query for *partial coverage*. Some implementations also permit querying the toggle count of each bit.

Assertion coverage — for each assertion, whether it has had at least one success. Implementations permit querying for further details, such as attempt counts, success counts, failure counts and failure coverage.

These terms define the “primitives” for each coverage type. Over instances or blocks, the coverage number is merely the sum of all contained primitives in that instance or block.

3.2 SystemVerilog real-time coverage access

This section ...

3.2.1 Predefined coverage constants in SystemVerilog

The following predefine `‘define`s represent basic real-time coverage capabilities accessible directly from SystemVerilog.

— Coverage control

```
‘define SV_COV_START       0
‘define SV_COV_STOP        1
‘define SV_COV_RESET       2
‘define SV_COV_QUERY       3
```

— Scope definition (hierarchy traversal/accumulation type)

```
‘define SV_COV_MODULE      10
‘define SV_COV_HIER        11
```

— Coverage type identification

```
‘define SV_COV_ASSERTION   20
‘define SV_COV_FSM_STATE   21
‘define SV_COV_STATEMENT   22
‘define SV_COV_TOGGLE      23
```

— Status results

```
‘define SV_COV_OVERFLOW    -2
‘define SV_COV_ERROR       -1
‘define SV_COV_NOCOV       0
‘define SV_COV_OK          1
‘define SV_COV_PARTIAL     2
```

3.2.2 Built-in coverage access system functions

This section ...

3.2.2.1 $coverage_control

```
$coverage_control(control_constant,
                  coverage_type,
                  scope_def,
                  modules_or_instance)
```

This function enables, disables, resets or queries the availability of coverage information for the specified portion of the hierarchy. The return value is a `‘defined` name, with the value indicating the success of the action.

`‘SV_COV_OK`
the request is successful. When querying, if starting, stopping, or resetting this means the desired effect occurred, coverage is available. A successful reset clears all coverage (i.e., using a ...`get() == 0` after a successful …`reset()`).

`‘SV_COV_ERROR`
the call failed with no action, typically due to errors in the arguments, such as a non-existing module or instance specifications.

`‘SV_COV_NOCOV`
coverage is not available for the requested portion of the hierarchy.

`‘SV_COV_PARTIAL`
coverage is only partially available in the requested portion of the hierarchy (i.e., some instances have the requested coverage information, some don’t).

Starting, stopping, or resetting coverage multiple times in succession for the same instance(s) has no further effect if coverage has already been started, stopped, or reset for that/those instance(s).

The hierarchy(ies) being controlled or queried are specified as follows.

`‘SV_MODULE_COV, “unique module def name”`
provides coverage of all instances of the given module (the unique module name is a string), excluding any child instances in the instances of the given module. The module definition name can use special notation to describe nested module definitions.

`‘SV_COV_HIER, “module name”`
provides coverage of all instances of the given module, including all the hierarchy below.

`‘SV_MODULE_COV, instance_name`
provides coverage of the one named instance. The instance is specified as a normal Verilog hierarchical path.

`‘SV_COV_HIER, instance_name`
provides coverage of the named instance, plus all the hierarchy below it.

All the permutations are summarized in Table 3-1 on page 23.
Revise this xref w/ Stu; also check/revise variable settings, etc.

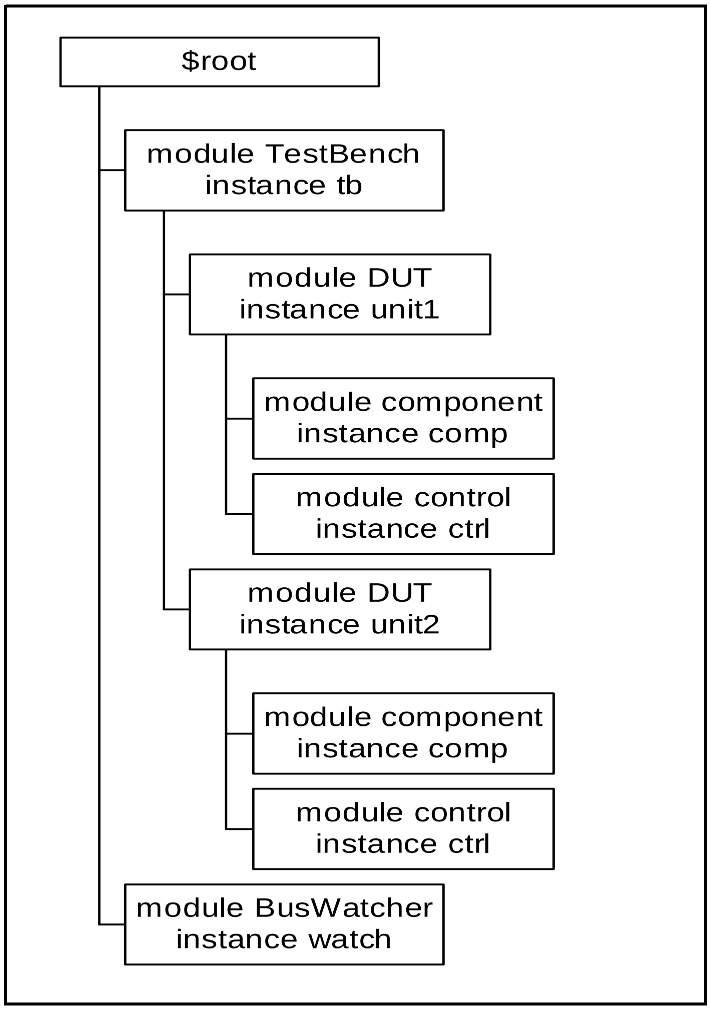
Table 3-1: Instance coverage permutations

Control/query	“Definition name”	instance.name
`‘SV_COV_MODULE`	The sum of coverage for all instances of the named module, excluding any hierarchy below those instances.	Coverage for just the named instance, excluding any hierarchy in instances below that instance.

Table 3-1: Instance coverage permutations (continued)

Control/query	"Definition name"	instance.name
`'SV_COV_HIER`	The sum of coverage for all instances of the named module, including all coverage for all hierarchy below those instances.	Coverage for the named instance and any hierarchy below it.

NOTE—Definition names are represented as strings, whereas instance names are referenced by hierarchical paths. A hierarchical path need not include any **.** if the path refers to an instance in the current context (i.e., normal Verilog hierarchical path rules apply).



Example 3-1Hierarchical instance example

If coverage is enabled on all instances shown in Example 3-1, then:

```
$coverage_control('SV_COV_CHECK, 'SV_COV_TOGGLE, 'SV_COV_HIER, $root)
```

checks all instances to verify they have coverage and, in this case, returns `'SV_COV_OK`.

```
$coverage_control('SV_COV_RESET, 'SV_COV_TOGGLE, 'SV_COV_MODULE,
                  "DUT")
```

resets coverage collection on both instances of the DUT, specifically, `$root.tb.unit1` and `$root.tb.unit2`, but leaves coverage unaffected in all other instances.

```
$coverage_control(‘SV_COV_RESET, ‘SV_COV_TOGGLE, ‘SV_COV_MODULE,
                  $root.tb.unit1)
```

resets coverage of only the instance `$root.tb.unit1`, leaving all other instances unaffected.

```
$coverage_control(‘SV_COV_STOP, ‘SV_COV_TOGGLE, ‘SV_COV_HIER,
                  $root.tb.unit1)
```

resets coverage of the instance `$root.tb.unit1` and also reset coverage for all instances below it, specifically `$root.tb.unit1.comp` and `$root.tb.unit1.ctrl`.

```
$coverage_control(`SV_COV_START, `SV_COV_TOGGLE, `SV_COV_HIER, “DUT”)
```

starts coverage on all instances of the module DUT and of all hierarchy(ies) below those instances. In this design, coverage is started for the instances `$root.tb.unit1`, `$root.tb.unit1.comp`, `$root.tb.unit1.ctrl`, `$root.tb.unit2`, `$root.tb.unit2.comp`, and `$root.tb.unit2.ctrl`.

3.2.2.2 $coverage_get_max

```
$coverage_get_max(coverage_type, scope_def, modules_or_instance)
```

This function obtains the value representing 100% coverage for the specified coverage type over the specified portion of the hierarchy. This value shall remain constant across the duration of the simulation.

NOTE—This value is proportional to the design size and structure, so it also needs to be constant through multiple independent simulations and compilations of the same design, assuming any compilation options do not modify the coverage support or design structure.

The return value is an integer, with the following meanings.

`-2 (‘SV_COV_OVERFLOW)`
the value exceeds a number that can be represented as an integer.

`-1 (‘SV_COV_ERROR)`
an error occurred (typically due to using incorrect arguments).

`0 (‘SV_COV_NOCOV)`
no coverage is available for that coverage type on that/those hierarchy(ies).

`+pos_num`
the maximum coverage number (where *`pos_num`* > `0`), which is the sum of all coverable items of that type over the given hierarchy(ies).

The scope of this function is specified as per `$coverage_control` (see section 3.2.2.1).

3.2.2.3 $coverage_get

```
$coverage_get(coverage_type, scope_def, modules_or_instance)
```

This function obtains the current coverage value for the given coverage type over the given portion of the hierarchy. This number can be converted to a coverage percentage by use of the equation:

$$\mathrm{cov}\,erage\% = \frac{\mathrm{cov}\,erage\_get()}{\mathrm{cov}\,erage\_get\_\max()} * 100$$

The return value follows the same pattern as `$coverage_get_max` (see section 3.2.2.2), but with ~~the~~ *`pos_num`* ~~number~~ representing the current coverage level, i.e., the number of the coverable items that have been covered in this/these hierarchy(ies).

The scope of this function is specified as per `$coverage_control` (see section 3.2.2.1).

The return value is an integer, with the following meanings.

`-2 (‘SV_COV_OVERFLOW)`
the value exceeds a number that can be represented as an integer.

`-1 (‘SV_COV_ERROR)`
an error occurred (typically due to using incorrect arguments).

`0 (‘SV_COV_NOCOV)`
no coverage is available for that coverage type on that/those hierarchy(ies).

`+`*`pos_num`*
the maximum coverage number (where *`pos_num`* `> 0`), which is the sum of all coverable items of that type over the given hierarchy(ies).

3.2.2.4 $coverage_merge

```
$coverage_merge(coverage_type, “name”)
```

This function loads and merges coverage data for the specified coverage into the simulator. *`name`* is an arbitrary string used by the tool, in an *implementation-specific* way, to locate the appropriate coverage database, i.e., tools are allowed to store coverage files any place they want with any extension they want *as long* as the user can retrieve the information by asking for a specific saved name from that coverage database. If *`name`* does not exist or does not correspond to a coverage database from the same design, an error shall occur. If an error occurs during loading, the coverage numbers generated by this simulation might not be meaningful.

The return values from this function are:

`‘SV_COV_OK`
the coverage data has been found and merged.

`‘SV_COV_NOCOV`
the coverage data has been found, but did not contain the coverage type requested.

`‘SV_COV_ERROR`
the coverage data was not found or it did not correspond to this design, or another error.

3.2.2.5 $coverage_save

```
$coverage_save(coverage_type, “name”)
```

This function saves the current state of coverage to the tool’s coverage database and associates it with the file named *`name`*. This file name shall not contain any directory specification or extensions. Data saved to the database shall be retrieved later by using `$coverage_merge` and supplying the same name. Saving coverage shall not have any effect on the state of coverage in this simulation.

The return values from this function are:

`‘SV_COV_OK`
the coverage data was successfully saved.

`‘SV_COV_NOCOV`
no such coverage is available in this design (nothing was saved).

`‘SV_COV_ERROR`
some error occurred during the save. If an error occurs, the tool shall automatically remove the coverage database entry for *`name`* to preserve the coverage database integrity. It is *not* an error to overwrite a previously existing *`name`*.

NOTES

1—The coverage database format is implementation-dependent.

2—Mapping of names to actual directories/files is implementation-dependent. There is no requirement that a coverage name map to any specific set of files or directories.

3.3 FSM recognition

Coverage tools need to have automatic recognition of many of the common FSM coding idioms in Verilog/ SystemVerilog. This standard does not attempt to describe or require any specific automatic FSM recognition mechanisms. However, the standard does prescribe a means by which non-automatic FSM extraction occurs. The presence of any of these standard FSM description additions shall override the tool's default extraction mechanism.

Identification of an FSM consists of identifying the following items:

1) the state register (or expression)

2) the next state register (this is optional)

3) the legal states.

3.3.1 Specifying the signal that holds the current state

**This section reads a bit like a user's guide; convert this into an annex??

Use the following pragma to identify the vector signal that holds the current state of the FSM:

```
/* tool state_vector signal_name */
```

let's define these terms (in the next draft)

where `tool` and `state_vector` are required keywords. This pragma needs to be specified inside the module definition where the signal is declared.

Another pragma is also required, to specify an enumeration name for the FSM. This enumeration name is also specified for the next state and any possible states, associating them with each other as part of the same FSM. There are two ways to do this:

— Use the same pragma:

```
/* tool state_vector signal_name enum enumeration_name */
```

— Use a separate pragma in the signal's declaration:

```
/* tool state_vector signal_name */
reg [7:0] /* tool enum enumeration_name */ signal_name;
```

In either case, `enum` is a required keyword; if using a separate pragma, `tool` is also a required keyword and the pragma needs to be specified immediately after the bit-range of the signal.

3.3.2 Specifying the part-select that holds the current state

A part-select of a vector signal can be used to hold the current state of the FSM. When `cmView` displays or reports FSM coverage data, it names the FSM after the signal that holds the current state. If a part-select holds the current state in the user's FSM, the user needs to also specify a name for the FSM that `cmView` can use. The FSM name is not the same as the enumeration name.

Specify the part-select by using the following pragma:

```
/* tool state_vector signal_name[n:n] FSM_name enum enumeration_name */
```

3.3.3 Specifying the concatenation that holds the current state

Like specifying a part-select, a concatenation of signals can be specified to hold the current state (when including an FSM name and an enumeration name):

```
/* tool state_vector {signal_name , signal_name, ...} FSM_name enum
   enumeration_name */
```

The concatenation is composed of all the signals specified. Bit-selects or part-selects of signals cannot be used in the concatenation.

3.3.4 Specifying the signal that holds the next state

The signal that holds the next state of the FSM can also be specified with the pragma that specifies the enumeration name:

```
reg [7:0] /* tool enum enumeration_name */
signal_name
```

This pragma can be omitted if, and only if, the FSM does not have a signal for the next state.

3.3.5 Specifying the current and next state signals in the same declaration

The tool assumes the first signal following the pragma holds the current state and the next signal holds the next state when a pragma is used for specifying the enumeration name in a declaration of multiple signals, e.g.,

```
/* tool state_vector cs */
reg [1:0] /* tool enum myFSM */ cs, ns, nonstate;
```

In this example, the tool assumes signal `cs` holds the current state and signal `ns` holds the next state. It assumes nothing about signal `nonstate`.

3.3.6 Specifying the possible states of the FSM

The possible states of the FSM can also be specified with a pragma that includes the enumeration name:

```
parameter /* tool enum enumeration_name */
S0 = 0,
s1 = 1,
s2 = 2,
s3 = 3;
```

Put this pragma immediately after the keyword `parameter`, unless a bit-width for the parameters is used, in which case, specify the pragma immediately after the bit-width:

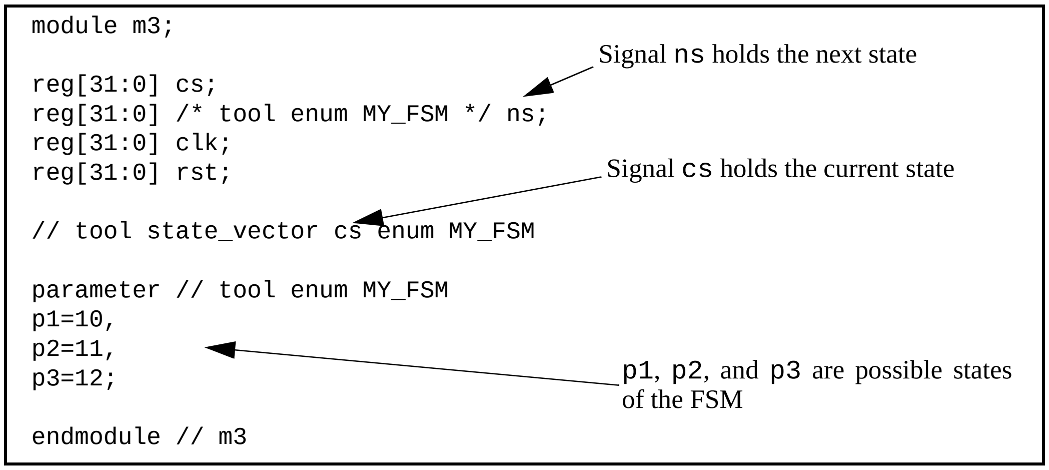
```
parameter [1:0] /* tool enum enumeration_name */
S0 = 0,
s1 = 1,
s2 = 2,
s3 = 3;
```

3.3.7 Pragmas in one-line comments

These pragmas work in both block comments, between the `/*` and `*/` character strings, and one-line comments, following the `//` character string, e.g.,

```
parameter [1:0] // tool enum enumeration_name
S0 = 0,
s1 = 1,
s2 = 2,
s3 = 3;
```

3.3.8 Example



```
module m3;

reg[31:0] cs;
reg[31:0] /* tool enum MY_FSM */ ns;
reg[31:0] clk;
reg[31:0] rst;

// tool state_vector cs enum MY_FSM

parameter // tool enum MY_FSM
p1=10,
p2=11,
p3=12;

endmodule // m3
```

Signal ns holds the next state
Signal cs holds the current state
p1, p2, and p3 are possible states of the FSM

Example 3-2FSM specified with pragmas

3.4 VPI coverage extensions

This section ...

3.4.1 VPI entity/relation diagrams related to coverage

This section ...

3.4.2 Extensions to VPI enumerations

— Coverage control

```
#define vpiCoverageStart
#define vpiCoverageStop
#define vpiCoverageReset
#define vpiCoverageCheck
#define vpiCoverageMerge
#define vpiCoverageSave
```

— VPI properties

1) Coverage type properties

```
#define vpiAssertCoverage
#define vpiFsmStateCoverage
#define vpiStatementCoverage
#define vpiToggleCoverage
```

2) Coverage status properties

```
#define vpiCovered
#define vpiCoverMax
#define vpiCoveredCount
```

3) Assertion-specific coverage status properties

```
#define vpiAssertAttemptCovered
#define vpiAssertSuccessCovered
#define vpiAssertFailureCovered
```

4) FSM-specific methods

```
#define vpiFsmStates
#define vpiFsmStateExpression
```

— FSM handle types (vpi types)

```
#define vpiFsm
#define vpiFsmHandle
```

3.4.3 Obtaining coverage information

All **what?? use `vpi_get()` along with the appropriate properties and object handles.

coverage type, *instance*
the number of covered items in the given instance.

`vpiCovered`, *handle*
the number of items of the handle type is covered. This is only applicable to: statement handles, signal (wire/reg) handles, assertion handles, and FSM handles.

`vpiCoveredCount`, *handle*
the number of times each item of the handle type is covered. This is only easily interpretable when *handle* points to a unique coverable item (otherwise this is the sum of counts of all contained items).

`vpiCoveredMax`, *handle*
the total possible coverable items in the given handle. Handle types limited as per above. `vpiCovered-Max` is only really useful when *handle* is a handle to an object potentially containing more than one coverable item.

— Use `vpi_iterate(vpiFsm, `*`instance-handle`*`)` to get the iterator to all FSMs in an instance.

— Use `vpi_handle(vpiFsmStateExpression, `*`fsm-handle`*`)` to get the handle to the signal or expression encoding the FSM state.

— Use `vpi_iterate(vpiFsmStates, `*`fsm-handle`*`)` to get the iterator to all states of an FSM.

— Use `vpi_get_value(fsm_state_handle, `*`state-handle`*`)` to get the value of a state.

3.4.4 Controlling coverage

Revise similar to Assertions

`vpi_control()`
has three arguments: *coverage control* (`start`, `stop`, `reset`, `query`), *coverage type*, and the *handle* to the appropriate instance or assertion. Statement, toggle, and FSM coverage are not individually controllable (i.e., they are controllable only at the instance level, not on a per statement/signal/FSM). The semantics and behavior are specified as per the equivalent system function `$coverage_control` `(`see section 3.2.2.1).

`vpi_control()`
has three arguments: *coverage control* (`merge`, `save`), *coverage type*, and *`name`*. This merges coverage into the current simulation. The semantics and behavior are specified as per the equivalent system functions `$coverage_merge` (see section 3.2.2.4) and `$coverage_save` (see section 3.2.2.5).

Annex A
DPI C-layer

A.1 Overview

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

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

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

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

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

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

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

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

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

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

Function arguments are generally passed by some form of reference, or by value. All formal arguments, except open arrays, are passed by direct reference or value, and, therefore, are directly accessible in C code. Only small values of SystemVerilog input arguments (see section A.7.7) are passed by value. Formal arguments declared in SystemVerilog as open arrays are passed by a handle (type `svOpenArrayHandle`) and are accessible via library functions. Array-querying functions are provided for open arrays.

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

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

A.2 Naming conventions

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

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

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

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

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

A.3 Portability

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

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

A.3.1 Binary compatibility

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

A.3.2 Source-level compatibility

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

A.4 Include files

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

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

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

A.4.1 `svdpi.h` include file

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

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

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

A.4.2 `svdpi_src.h` include file

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

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

A.5 Semantic constraints

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

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

Another way to state this is that no compiler (either C or SystemVerilog) can make argument coercions between a caller’s declared formals and the callee’s declared formals. This is because the callee’s formal arguments are declared in a different language than the caller’s formal arguments; hence there is no visible relationship between the two sets of formals. Users are expected to understand all argument relationships and provide properly matched types on both sides of the interface (see section A.6.2).

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

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

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

A.5.1 Types of formal arguments

The principle "What You Specify Is What You Get" guarantees the types of formal arguments of imported functions — an actual argument is guaranteed to be of the type specified for the formal argument, with the exception of open arrays (for which unspecified ranges are statically unknown). Formal arguments, other than open arrays, are fully defined by imported declaration; they shall have ranges of packed or unpacked arrays exactly as specified in the imported declaration. Only the SystemVerilog declaration site of the imported function is relevant for such formal arguments.

Formal arguments defined as open arrays have the size and ranges of the actual argument, i.e., have the ranges of packed or unpacked arrays exactly as that of the actual argument. The unsized ranges of open arrays are determined at a call site; the rest of the type information is specified at the import declaration. See also section A.6.1.

So, if a formal argument is declared as `bit [15:8] b []`, then it is the import declaration which specifies the formal argument is an unpacked array of packed bit array with bounds `15` to `8`, while the actual argument used at a particular call site defines the bounds for the unpacked part for that call.

A.5.2 `input` arguments

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

A.5.3 `output` arguments

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

A.5.4 Value changes for `output` and `inout` arguments

The SystemVerilog simulator is responsible for handling value changes for `output` and `inout` arguments. Such changes shall be detected and handled after ~~the~~ control returns from C code to SystemVerilog code.

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

Also refer to section 1.4.3

Some DPI imported functions or other interface functions called from them require that the context of their call be known. It takes special instrumentation of their call instances to provide such context; for example, a variable referring to the "current instance" might need to be set. To avoid any unnecessary overhead, imported function calls in SystemVerilog code are not instrumented unless the imported function is specified as `context` in its SystemVerilog import declaration.

All DPI export functions require that the context of their call is known. This occurs since SystemVerilog function declarations always occur in instantiable scopes, hence allowing a multiplicity of unique function instances in the simulator's elaborated database. Thus, there is no such thing as a non-context export function.

For the sake of simulation performance, an non-context imported function call shall not block SystemVerilog compiler optimizations. An imported func not specified as `context` shall not access any data objects from SystemVerilog other then its actual arguments. Only the actual arguments can be affected (read or written) by its call. Therefore, a call of non-`context` imported function is not a barrier for optimizations. A `context` imported function, however, can access (read or write) any SystemVerilog data objects by calling PLI/VPI, nor by calling an embedded export function. Therefore, a call to a `context` function is a barrier for SystemVerilog compiler optimizations.

Only the calls of `context` imported functions are properly instrumented and cause conservative optimizations; therefore, only those functions can safely call all functions from other APIs, including PLI and VPI functions or exported SystemVerilog functions. For imported functions not specified as `context`, the effects of calling PLI, VPI, or SystemVerilog functions can be unpredictable and such calls can crash if the callee requires a context that has not been properly set.

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

A.5.6 `pure` functions

See also 1.4.2

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

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

— perform any file operations

— read or write anything in the broadest possible meaning, includes i/o, environment variables, objects from the operating system or from the program or other processes, shared memory, sockets, etc.

— access any persistent data, like global or static variables.

If a `pure` function does not obey the above restrictions, SystemVerilog compiler optimizations can lead to unexpected behavior, due to eliminated calls or incorrect results being used.

A.5.7 Memory management

See also section 1.4.1.4

The memory spaces owned and allocated by C code and SystemVerilog code are disjoined. Each side is responsible for its own allocated memory. Specifically, C code shall not free the memory allocated by SystemVerilog code (or the SystemVerilog compiler) nor expect SystemVerilog code to free the memory allocated by C code (or the C compiler). This does not exclude scenarios in which C code allocates a block of memory, then passes a handle (i.e., a pointer) to that block to SystemVerilog code, which in turn calls a C function that directly (if it is the standard function `free`) or indirectly frees that block.

NOTE—In this last scenario, a block of memory is allocated and freed in C code, even when the standard functions `malloc` and `free` are called directly from SystemVerilog code.

A.6 Data types

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

A.6.1 Limitations

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

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

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

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

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

A.6.3 Data representation

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

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

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

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

— Representation of packed types is implementation-dependent.

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

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

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

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

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

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

A.6.4 Basic types

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

Table A1—Mapping data types

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

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

A.6.5 Normalized ranges

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

A.6.6 Mapping between SystemVerilog ranges and normalized ranges

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

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

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

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

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

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

For exampl4

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

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

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

Table A2—Encoding of bits in `svLogicVec32`

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

A.7 Argument passing modes

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

A.7.1 Overview

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

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

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

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

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

A.7.2 Calling SystemVerilog functions from C

There is no difference in argument passing between calls from SystemVerilog to C and calls from C to SystemVerilog. Functions exported from SystemVerilog can not have open arrays as arguments. Apart from this restriction, the same types of formal arguments can be declared in SystemVerilog for exported functions and imported functions. A function exported from SystemVerilog shall have the same function header in C as would an imported function with the same function result type and same formal argument list. In the case of arguments passed by reference, an actual argument to SystemVerilog function called from C shall be allocated using the same layout of data as SystemVerilog uses for that type of argument; the caller is responsible for the allocation. It can be done while preserving ~~the~~ binary compatibility, see section A.7.5 and section A.11.11.

A.7.3 Argument passing by value

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

A.7.4 Argument passing by reference

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

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

A.7.5 Allocating actual arguments for SystemVerilog-specific types

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

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

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

A.7.6 Argument passing by `sv_handle` open arrays

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

A.7.7 `input` arguments

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

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

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

— `handle`, `string`

— `bit` (i.e., 2-state) packed arrays up to 64 [previously 32] -bit (canonical representation shall be used, like for a function result).
[There is a problem here: 'int' is the same as svBitVec32, long long is not the same as svBitVec32[2], so how to return a value in the canonical representation as a function result, if this value is between 33 and 64 bits?]

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

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

A.7.8 `inout` and `output` arguments

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

A.7.9 Function result

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

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

— packed `bit` arrays up to 32 bits.

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

A.8 Context functions

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

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

A.8.1 Overview of DPI and VPI context

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

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

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

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

Note that all DPI export functions require that the context of their call is known. This occurs since SystemVerfunction instances in the simulator's database. Thus, there is no such thing as a non-context export function. All export function calls must have their execution scope specified in advance by use of a context-setting API function.

A.8.2 Context of imported and export functions

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

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

Note that context is transitive through imported and export context functions. That is, if an imported function is running in a certain context, and if it in turn calls an exported function, the exported function will inherit the context from the imported function. For example, consider a SystemVerilog call to a native function f(), which in turn calls a native function g(). Now replace the native function f() with an equivalent imported context C function, f'(). The system will behave identically regardless if f() or f'() is in the call chain above g(). g() has the proper execution context in both cases.

A.8.3 Working with DPI context functions in C code

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

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

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

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

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

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

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

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

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

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

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

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

/* Retrieve an arbitrary user data pointer that was previously
 * stored by a call to svPutUserData().  See the comment above
 * svPutUserData() for an explanation of userKey, as well as
 * restrictions on NULL and illegal svScope and userKey values.
 * This function returns NULL for all error cases, 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
 */
  t svGetCallerInfo(char **fileName, int *lineNumber);
```

A.8.4 Example 1 — Using DPI context functions

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

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

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

         friend int MyCFunc(int portID);
      };

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

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

A.8.5 Relationship between DPI and VPI/PLI interfaces

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

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

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

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

```
/* Get handle to system task call site in preparation for argument scan */
vpiHandle myHandle = vpi_handle(vpiSysTfCall, NULL);
```

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

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

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

A.9 Include files

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

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

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

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

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

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

```
/* canonical representation */

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

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

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

A.9.1.2 Canonical representation of packed arrays

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

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

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

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

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

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

A.9.1.3 Implementation-dependent representation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A.9.3 Example 1— binary compatible application

SystemVerilog:

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

export “DPI” function exported_sv_func;

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

C:

```
#include "svdpi.h"

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

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

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

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

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

A.9.4 Example 2— source-level compatible application

SystemVerilog:

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

export “DPI” function exported_sv_func;

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

C:

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

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

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

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

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

```
      svLogicVec32 aL[SV_CANONICAL_SIZE(64)];

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

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

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

A.10 Arrays

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

A.10.1 Multidimensional arrays

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

A.10.2 Direct access to unpacked arrays

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

A.10.3 Access to packed arrays via canonical representation

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

A.10.3.1 Bit selects

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

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

   /* functions for bit select */

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

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

A.10.3.2 Part selects

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

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

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

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

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

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

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

A.11 Open arrays

Formal arguments specified as open arrays allows passing actual arguments of different sizes (i.e., different range and/or different number of elements), which facilitates writing ~~a~~ more general C code that can handle SystemVerilog arrays of different sizes. The elements of an open array can be accessed in C by using the same range of indices and the same indexing as in SystemVerilog. Plus, inquiries about the dimensions and the original boundaries of SystemVerilog actual arguments are supported for open arrays.

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

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

A.11.1 Actual ranges

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

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

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

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

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

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

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

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

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

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

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

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

A.11.2 Array querying functions

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

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

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

A.11.3 Access functions

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

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

- Packed arrays (`bit` or `logic`) are accessed via copying to or from the canonical representation.
- Scalars (1-bit value of type `bit` or `logic`) are accessed (read or written) directly.
- Other types of values (e.g., structures) are accessed via generic pointers; a library function calculates an address and the user needs to provide the appropriate casting.
- Scalars and packed arrays are accessible via pointers only if the implementation supports this functionality (per array), e.g., one array can be represented in a form that allows such access, while another array might use a compacted representation which renders this functionality unfeasible (both occurring within the same simulator).

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

A.11.4 Access to the actual representation

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

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

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

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

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

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

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

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

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

A.11.5 Access to elements via canonical representation

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

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

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

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

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

void svGetLogicArrElemVec32 (svLogicVec32* d, const svOpenArrayHandle _s, int
indx1,
                              ...);
void svGetLogicArrElem1Vec32(svLogicVec32* d, const svOpenArrayHandle _s, int
indx1);
void svGetLogicArrElem2Vec32(svLogicVec32* d, const svOpenArrayHandle _s, int
indx1,
                             int indx2);
void svGetLogicArrElem3Vec32(svLogicVec32* d, const svOpenArrayHandle _s,
                             int indx1, int indx2, int indx3);
```

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

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

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

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

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

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

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

A.11.7 Access to array elements of other types

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

A.11.8 Example 3— two-dimensional open array

SystemVerilog:

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

import "DPI" function void foo(input MyType i [][]); /* 2-dimensional unsized
unpacked array
                                                   unpacked array of MyType */

MyType a_10x5 [11:20][6:2];
MyType a_64x8 [64:1][-1:-8];

foo(a_10x5);
foo(a_64x8);
```

C:

```
#include "svdpi.h"

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

void foo(const svOpenArrayHandle _h)
{
   MyType my_value;
```

```
      int i, j;
      int lo1 = svLow(h, 1);
      int hi1 = svHigh(h, 1);
      int lo2 = svLow(h, 2);
      int hi2 = svHigh(h, 2);

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

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

A.11.9 Example 4— open array

SystemVerilog:

```
   e8(es., )13.3struct...} (M)13.3(yTyp(;)]TJ 0 -2.44 TD(import. )Tj 0.105 g4.92 0 TD [“M)13.3DPI
```

```
            *(MyType *)svGetArrElemPtr1(hout, j++) =
                         *(MyType *)svGetArrElemPtr1(hin, i++);
        }

      }

    }
```

A.11.10 Example 5 — access to packed arrays

SystemVerilog:

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

C:

```
#include "svdpi.h"

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

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

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

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

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

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

A.11.11 Example 6 — binary compatible calls of exported functions

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

SystemVerilog:

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

C:

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

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

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

Annex B
Include files

This annex shows the contents of the svdpi.h and svdpi_src.h include files.

B.1 Binary-level compatibility include file svdpi.h

```
/* 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 */

/* 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))))

/* implementation-dependent representation */
/* 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 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);

/* 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
 */
/* 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 svGetPartSelectBit(svBitVec32* d, const svBitPackedArrRef s, int i,~~
svBitVec32 svGetBits(const svBitPackedArrRef s, int i, int w);
svBitVec32 svGet32Bits(const svBitPackedArrRef s, int i); // 32-bits
long long svGet64Bits(const svBitPackedArrRef s, int i); // 64-bits
void svPutPartSelectLogic(svLogicPackedArrRef d, const svLogicVec32 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
one-dimensional) of an array, and dimensions > 0 refer to the unpacked part of
an array.*/

/* h= handle to open array, d=dimension */
int svLeft(const svOpenArrayHandle h, int d);
int svRight(const svOpenArrayHandle h, int d);
int svLow(const svOpenArrayHandle h, int d);
int svHigh(const svOpenArrayHandle h, int d);
int svIncrement(const svOpenArrayHandle h, int d);
int svLength(const svOpenArrayHandle h, int d);
int svDimensions(const svOpenArrayHandle h);

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

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

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

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

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

/* 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. */
/* 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);

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 occured, 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 will inherit the scope of the surrounding extern
   function. This is known as the “default scope”.
   The return is the previous active scope (as per svGetScope) */
svScope svSetScope(const svScope scope);

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

/* Retrieve svScope to instance scope of an arbitrary function declaration.
   (Will be either module, program, interface, or generate scope) */
svScope svGetScopeFromName(const char* scopeName);

/* Set arbitrary user data pointer into specified instance scope */
  id svPutUserData(const svScope scope, void* userData);

/* Retrieve arbitrary user data from specified instance scope */
void* svGetUserData(const svScope scope);

/ * 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);
```

B.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 will go here */
#define SV_LOGIC_PACKED_ARRAY(WIDTH,NAME)/* actual definition will go here */
```

Code in object code form. Figure C1— depicts the inclusion of object code and its relations to the various steps involved in this integration process. **Revise this xref w/ Stu; also check/revise variable settings, etc.**

.

Figure C1— Inclusion of object code into a SystemVerilog application

Compiled object code can be specified by one of the following two methods:

1) by an entry in a bootstrap file; see section C.3.1 for more details on this file and its content. Its location shall be specified with one instance of the switch `-sv_liblist` *`pathname`*. This switch can be used multiple times to define the usage of multiple bootstrap files.

2) by specifying the file with one instance of the switch `-sv_lib` *`pathname_without_extension`* (i.e., the filename shall be specified without the platform specific extension). The SystemVerilog application is responsible for appending the appropriate extension for the actual platform. This switch can be used multiple times to define multiple libraries holding object code.

Both methods shall be provided and made available concurrently, to permit any mixture of their usage. Every location can be an absolute pathname or a relative pathname, where the value of the switch `-sv_root` is used to identify an appropriate prefix for relative pathnames (see section C.2 for more details on forming pathnames).

All compiled object code need to be loaded in the specification order similarly to the above scheme; first the content of the bootstrap file is processed starting with the first line, then the set of `-sv_lib` switches is processed in order of their occurrence. Any library shall only be loaded once.

C.2.1 Bootstrap file

The object code bootstrap file has the following syntax.

1) The first line contains the string `#!SV_LIBRARIES`.

2) An arbitrary amount of entries follow, one entry per line, where every entry holds exactly one library location. Each entry consists only of the *`pathname_without_extension`* of the object code file to be loaded and can be surrounded by an arbitrary number of blanks; at least one blank shall precede the entry in the line. The value *`pathname_without_extension`* is equivalent to the value of the switch `-sv_lib`.

3) Any amount of comment lines can be interspersed between the entry lines; a comment line starts with the character # after an arbitrary (including zero) amount of blanks and is terminated with a newline.

C.2.2 Examples

1) If the pathname root has been set by the switch `-sv_root` to `/home/user` and the following object files need to be included:

```
/home/user/myclibs/lib1.so
/home/user/myclibs/lib3.so
/home/user/proj1/clibs/lib4.so
/home/user/proj3/clibs/lib2.so
```

then use either of the methods in Example C-1. Both methods are equivalent.

```
#!SV_LIBRARIES
 myclibs/lib1
 myclibs/lib3
 proj1/clibs/lib4
 proj3/clibs/lib2
```

Bootstrap file method

```
...
-sv_lib myclibs/lib1
-sv_lib myclibs/lib3
-sv_lib proj1/clibs/lib4
-sv_lib proj3/clibs/lib2
...
```

Switch list method

Example C-1Using a simple bootstrap file or a switch list

2) If the current working directory is `/home/user`, using the series of switches shown in Example C-2 (left column) result in loading the following files (right column).

```
-sv_lib svLibrary1
-sv_lib svLibrary2
-sv_root /home/project2/shared_code
-sv_lib svLibrary3
-sv_root /home/project3/code
-sv_lib svLibrary4
```

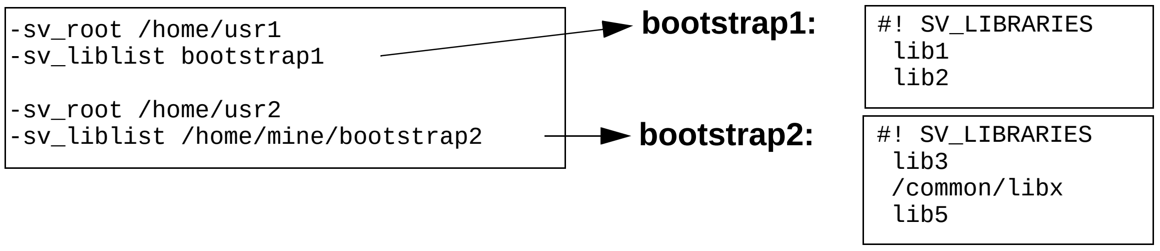
Switches

```
/home/user/svLibrary1.so
/home/user/svLibrary2.so

/home/project2/shared_code/svLibrary3.so

/home/project3/code/svLibrary4.so
```

Files

Example C-2Using a combination of `-sv_lib` and `-sv_root` switches

3) Further, using the set of switches and contents of bootstrap files shown in Example C-3:



```
-sv_root /home/usr1
-sv_liblist bootstrap1

-sv_root /home/usr2
-sv_liblist /home/mine/bootstrap2
```

```
#! SV_LIBRARIES
 lib1
 lib2
```

```
#! SV_LIBRARIES
 lib3
 /common/libx
 lib5
```

Example C-3Mixing `-sv_root` *and bootstrap files*

results in loading the following files:

```
/home/usr1/lib1.ext
/home/usr1/lib2.ext
/home/usr2/lib3.ext
/common/libx.ext
/home/usr2/lib5.ext
```

where *`ext`* stands for the actual extension of the corresponding file.