All text in the baseline P1800/D5 is unmodified in color choices. It contains editor's changes in dark green. All changes are dark blue. My editorial comments are in gold.

35. Programming language interface (PLI/SVPI) overview

35.1 General

This clause describes:

- The definition and history of PLI and SVPI
- User-defined system tasks and functions
- SVPI sizetf, compiletf and calltf routines
- The PLI mechanism
- Access to SystemVerilog and simulation objects
- List of SVPI routines by functional category

35.2 PLI purpose and history

The *Programming Language Interface* (PLI) is a procedural interface that allows foreign language functions to access the internal data structures of a SystemVerilog simulation. The *SystemVerilog Procedural Interface* (~~VPI~~SVPI) is part of the PLI. SVPI provides a library of C-language functions and a mechanism for associating foreign language functions with SystemVerilog user-defined system task and function names.

The PLI provides a means for SystemVerilog users to dynamically access and modify data in an instantiated SystemVerilog data structure. An instantiated SystemVerilog data structure is the result of compiling and elaborating SystemVerilog source descriptions and generating the hierarchy modeled by module instances, primitive instances, and other SystemVerilog constructs that represent scope. The PLI procedural interface provides a library of C language functions that can directly access data within an instantiated SystemVerilog data structure.

A few of the many possible applications for the PLI procedural interface are as follows:

- C language delay calculators for SystemVerilog model libraries that can dynamically scan the data structure of a SystemVerilog tool and then dynamically modify the delays of each instance of models from the library
- C language applications that dynamically read test vectors or other data from a file and pass the data into a SystemVerilog tool
- Custom graphical waveform and debugging environments for SystemVerilog software products
- Source code decompilers that can generate SystemVerilog source code from the compiled data structure of a SystemVerilog tool
- Simulation models written in the C language and dynamically linked into SystemVerilog simulations
- Interfaces to actual hardware, such as a hardware modeler, that dynamically interact with simulations

There are ~~three~~four primary generations of the SystemVerilog PLI:

a) *Task/function* routines, called *TF* routines, made up the first generation of the PLI. These routines, most of which started with the characters **tf_**, were primarily used for operations involving user-defined system task/function arguments, along with utility functions, such as setting up call-back mechanisms and writing data to output devices. The TF routines were sometimes referred to as *utility* routines

b) *Access* routines, called *ACC* routines, formed the second generation of the PLI. These routines, which all started with the characters **acc_**, provided an object-oriented access directly into a SystemVerilog structural description. ACC routines were used to access and modify information, such as delay values and logic values, on a wide variety of objects that exist in a SystemVerilog description. There was some overlap in functionality between ACC routines and TF routines.

c) *Verilog procedural interface* routines, called *VPI* routines, are the third generation of the PLI. These routines, most of which start with the characters **vpi_**, provide an object-oriented access for both SystemVerilog structural and behavioral objects. The VPI routines are a superset of the functionality of the TF routines and ACC routines.

d) *SystemVerilog procedural interface* routines, called *SVPI* routines, are the fourth generation of the PLI. These routines and the information model behind them are an extension of the VPI routines and information model to encompass the core SystemVerilog language, assertions, and coverage. For backward compatibility, these routines retain the prefix vpi in the data model and in the names of the constants and functions that make up the procedural interface. The acronym VPI has been deprecated.

NOTE—IEEE Std 1364-2005 deprecated the task/function (TF) and access (ACC) routines These deprecated routines are not included in this standard. See Clause 21 through Clause 25, Annex E, and Annex F of IEEE Std 1364-2001 for the deprecated text.

This clause, along with Clause 37, Annex L and Annex N, describe the ~~VPI~~SVPI procedural interface standard and interface mechanisms.

AUTHOR'S NOTE there are necessary changes to the rest of this clause, pending approval of the approach to changing terminology --John Shields

35.3 User-defined system task/function names

A user-defined system task/function name is the name that will be used within a SystemVerilog source file to invoke specific PLI applications. The name shall adhere to the following rules:

— The first character of the name shall be the dollar sign ($).
— The remaining characters shall be letters, digits, the underscore character (_), or the dollar sign ($).
— Uppercase and lowercase letters shall be considered to be unique—the name is case sensitive.
— The name can be any size, and all characters are significant.

35.3.1 Defining system task/function names

User-defined system task/function names are defined using a system task/function callback registry, which is part of the PLI mechanism. Registering system tasks and functions is described in 35.9.1.

35.3.2 Overriding built-in system task/function names

Clause 19 and Clause 20 define a number of built-in system tasks and functions that are part of the SystemVerilog language. In addition, SystemVerilog tools can include other built-in system tasks and functions specific to the tool. These built-in system task/function names begin with the dollar sign ($) just as user-defined system task/function names.

If a user-provided PLI application is associated with the same name as a built-in system task/function (using the PLI mechanism), the user-provided C application shall override the built-in system task/function, replacing its functionality with that of the user-provided C application. For example, a user could write a random number generator as a PLI application and then associate the application with the name **`$random`**, thereby overriding the built-in **`$random`** function with the user's application.

SystemVerilog timing checks, such as **$setup**, are not system tasks and cannot be overridden.

The built-in system functions **$signed** and **$unsigned** can be overridden. These system functions are unique in the SystemVerilog in that the return width is based on the width of their argument. If overridden, the PLI version shall have the same return width for all instances of the system function. The PLI return width is defined by the PLI *sizetf* routine.

35.4 User-defined system task/function arguments

When a user-defined system task/function is used in a SystemVerilog source file, it can have arguments that can be used by the PLI applications associated with the system task/function. In the following example, the user-defined system task `$get_vector` has two arguments:

```
$get_vector("test_vector.pat", input_bus);
```

The arguments to a system task/function are referred to as *task/function arguments* (often abbreviated as *tfargs*). These arguments are not the same as C language arguments. When the PLI applications associated with a user-defined system task/function are called, the task/function arguments are not passed to the PLI application. Instead, a number of PLI routines are provided that allow the PLI applications to read and write to the task/function arguments. See Clause 37 for information on specific routines that work with task/function arguments.

35.5 User-defined system task/function types

The type of a user-defined system task/function determines how a PLI application is called from the SystemVerilog source code. The types are as follows:

— A user *task* can be used in the same places a SystemVerilog task can be used (see 13.3). A user-defined system task can read and modify the arguments of the task, but does not return any value.

— A user *function* can be used in the same places a SystemVerilog function can be used (see 13.5). A user-defined system function can read and modify the arguments of the function, and it returns a value. The bit width of a vector shall be determined by a user-supplied *sizetf* application (see 35.8.1).

35.6 User-supplied PLI applications

User-supplied PLI applications are C language functions that utilize the library of PLI C functions to access and interact dynamically with SystemVerilog software implementations as the SystemVerilog source code is executed.

These PLI applications are not independent C programs. They are C functions that are linked into a tool and become part of the tool. This allows the PLI application to be called when the user-defined system task/function **$** name is compiled or executed in the SystemVerilog source code (see 35.8).

35.7 PLI include files

The libraries of PLI functions are defined in C include files, which is a normative part of this standard. This file also defines constants, structures, and other data used by the library of PLI routines and the interface mechanisms. The file is `vpi_user.h` (listed in Annex L) and `sv_vpi_user.h` (listed in Annex N). PLI applications that use the VPI routines shall include these files.

35.8 VPI sizetf, compiletf and calltf routines

VPI-based system tasks have *sizetf*, *compiletf*, and *calltf* routines, which perform specific actions for the task/function. The *sizetf*, *compiletf*, and *calltf* routines are called during specific periods during processing. The purpose of each of these routines is explained in 35.8.1 through 35.8.4.

35.8.1 sizetf VPI application routine

A *sizetf* VPI application routine can be used in conjunction with user-defined system functions. A function shall return a value, and tools that execute the system function need to determine how many bits wide that return value shall be. When *sizetf* shall be called is described in 35.10.2 and 37.35.1. Each *sizetf* routine shall be called at most once. It shall be called if its associated system function appears in the design. The value returned by the *sizetf* routine shall be the number of bits that the *calltf* routine shall provide as the return value for the system function. If no *sizetf* routine is specified, a user-defined system function shall return 32 bits. The *sizetf* routine shall not be called for user-defined system tasks or for functions whose *sysfunctype* is set to **vpiRealFunc**.

35.8.2 compiletf VPI application routine

A *compiletf* VPI application routine shall be called when the user-defined system task/function name is encountered during parsing or compiling the SystemVerilog source code. This routine is typically used to check the correctness of any arguments passed to the user-defined system task/function in the SystemVerilog source code. The *compiletf* routine shall be called one time for each instance of a system task/function in the source description. Providing a *compiletf* routine is optional, but it is recommended that any arguments used with the system task/function be checked for correctness to avoid problems when the *calltf* or other PLI routines read and perform operations on the arguments. When the *compiletf* is called is described in 35.10.2 and 37.35.1.

35.8.3 calltf VPI application routine

A *calltf* VPI application routine shall be called each time the associated user-defined system task/function is executed within the SystemVerilog source code. For example, the following SystemVerilog loop would call the *calltf* routine that is associated with the `$get_vector` user-defined system task name 1024 times:

```
for (i = 1; i <= 1024; i = i + 1)
   @(posedge clk) $get_vector("test_vector.pat", input_bus);
```

In this example, the *calltf* might read a test vector from a file called `test_vector.pat` (the first task/function argument), perhaps manipulate the vector to put it in a proper format for SystemVerilog, and then assign the vector value to the second task/function argument called `input_bus`.

35.8.4 Arguments to sizetf, compiletf, and calltf application routines

The *sizetf*, *compiletf*, and *calltf* routines all take one argument. When the tool calls these routines, it will pass to them the value supplied in the `s_vpi_systf_data` structure's *user_data* field when the user-defined system task/function was registered. See 37.35.

35.9 PLI mechanism

The PLI mechanism provides a means to have PLI applications called for various reasons when the associated system task/function $ name is encountered in the SystemVerilog source description. For example, when a SystemVerilog simulator first compiles the SystemVerilog source description, a specific compiletf PLI routine can be called that performs syntax checking to ensure the user-defined system task/function is

being used correctly. Then, as simulation is executing, a specific calltf PLI routine can be called to perform the operations required by the PLI application. User-defined system tasks and functions, and their associated routines and data, are defined by registering *system task/function callbacks* (see 35.9.1).

The PLI mechanism also enables having specific PLI applications automatically called by the simulator for miscellaneous reasons, such as the end of a simulation time step or a logic value change on a specific signal. This dynamic interaction with simulation is accomplished by registering *simulation callbacks* (see 35.9.2).

35.9.1 Registering user-defined system tasks and functions

User-defined system tasks and functions are created using the routine **vpi_register_systf()** (see 37.35). The registration of system tasks ~~must~~ shall occur prior to elaboration or the resolution of references.

The intended use model would be to place a reference to a routine within the **`vlog_startup_routines[]`** array. This routine would register all user-defined system tasks and functions when it is called.

Through the VPI, an application can perform the following:

- Specify a user-defined system task/function name that can be included in SystemVerilog source descriptions; the user-defined system task/function name shall begin with a dollar sign ($), such as `$get_vector`.
- Provide one or more PLI C applications to be called by a tool (such as a logic simulator).
- Define which PLI C applications are to be called—and when the applications should be called—when the user-defined system task/function name is encountered in the SystemVerilog source description.
- Define whether the PLI applications should be treated as *function*s (which return a value) or *task*s (analogous to subroutines in other programming languages).
- Define a data argument to be passed to the PLI applications each time they are called.

35.9.2 Registering simulation callbacks

Dynamic tool interaction shall be accomplished with a registered callback mechanism. VPI callbacks allow an application to request that a SystemVerilog tool, such as a logic simulator, call a user-defined application when a specific activity occurs. For example, the application can request that the application routine `my_monitor()` be called when a particular net changes value or that `my_cleanup()` be called when the tool execution has completed.

The VPI simulation callback facility shall provide the application with the means to interact dynamically with a tool, detecting the occurrence of value changes, advancement of time, end of simulation, etc. This feature allows integration with other simulation systems, specialized timing checks, complex debugging features, etc.

The reasons for which callbacks shall be provided can be separated into four categories:

- *Simulation event* (e.g., a value change on a net or a behavioral statement execution)
- *Simulation time* (e.g., the end of a time queue or after certain amount of time)
- *Simulator action or feature* (e.g., the end of compile, end of simulation, restart, or enter interactive mode)
- *User-defined system task/function execution*

VPI simulation callbacks shall be registered by the application with the function **vpi_register_cb()** (see 37.34). This routine indicates the specific reason for the callback, the application routine to be called, and what system and *user_data* shall be passed to the callback application when the callback occurs. A facility is

also provided to call the callback functions when a SystemVerilog tool is first invoked. A primary use of this facility shall be for registration of user-defined system tasks and functions.

35.10 VPI access to SystemVerilog objects and simulation objects

Accessible SystemVerilog objects and simulation objects and their relationships and properties are described using *data model diagrams*. These diagrams are presented in Clause 36. The data model diagrams indicate the routines and constants that are required to access and manipulate objects within an application environment. An associated set of routines to access these objects is defined in Clause 37.

VPI also includes a set of utility routines for functions such as handle comparison, file handling, and redirected printing, which are described in Table 35-9 (in 35.11).

VPI routines provide access to objects in an *instantiated* SystemVerilog design. An instantiated design is one where each instance of an object is uniquely accessible. For instance, if a module `m` contains wire `w` and is instantiated twice as `m1` and `m2`, then `m1.w` and `m2.w` are two distinct objects, each with its own set of related objects and properties.

VPI is designed as a *simulation* interface, with access to both SystemVerilog objects and specific simulation objects. This simulation interface is different from a hierarchical language interface, which would provide access to source code information, but would not provide information about simulation objects.

35.10.1 Error handling

To determine whether an error occurred, the routine **vpi_chk_error()** (see 37.2) shall be provided. The **vpi_chk_ error()** routine shall return a nonzero value if an error occurred in the previously called VPI routine. Callbacks can be set up for when an error occurs as well. The **vpi_chk_error()** routine can provide detailed information about the error.

35.10.2 Function availability

Certain features of VPI ~~must~~ shall occur early in the execution of a tool. In order to allow this process to occur in an orderly manner, some functionality ~~must~~ shall be restricted in these early stages. Specifically, when the routines within the **`vlog_startup_routines[ ]`** array are executed, there is very little functionality available. Only two routines can be called at this time:

- **vpi_register_systf()** (see 37.35)
- **vpi_register_cb()** (see 37.34)

In addition, the **vpi_register_cb()** routine can only be called for the following reasons:

- **cbEndOfCompile**
- **cbStartOfSimulation**
- **cbEndOfSimulation**
- **cbUnresolvedSystf**
- **cbError**
- **cbPLIError**

See 37.35 for a further explanation of the use of the **`vlog_startup_routines[ ]`** array.

The next earliest phase is when the *sizetf* routines are called for the user-defined system functions. At this phase, no additional access is permitted. After the *sizetf* routines are called, the routines registered for reason

cbEndOfCompile are called. At this point, and continuing until the tool has finished execution, all functionality is available.

35.10.3 Traversing expressions

The VPI routines provide access to any expression that can be written in the source code. Dealing with these expressions can be complex because very complex expressions can be written in the source code. Expressions with multiple operands will result in a handle of type **vpiOperation**. To determine how many operands, access the property **vpiOpType**. This operation will be evaluated after its subexpressions. Therefore, it has the least precedence in the expression.

An example of a routine that traverses an entire complex expression is listed below:

```
void traverseExpr(vpiHandle expr)
{
   vpiHandle subExprI, subExprH;

   switch (vpi_get(vpiExpr,expr))
   {
         case vpiOperation:
            subExprI = vpi_iterate(vpiOperand, expr);
            if (subExprI)
               while (subExprH = vpi_scan(subExprI))
                  traverseExpr(subExprH);
            /* else it is of op type vpiNullOp */
            break;
         default:
            /* Do whatever to the leaf object. */
            break;
      }
}
```

35.11 List of VPI routines by functional category

The VPI routines can be divided into groups based on primary functionality:

- Simulation-related callbacks
- System task/function callbacks
- Traversing SystemVerilog hierarchy
- Accessing properties of objects
- Accessing objects from properties
- Delay processing
- Logic and strength value processing
- Simulation time processing
- Miscellaneous utilities

Table 35-1 through Table 35-9 list the VPI routines by major category. Clause 37 defines each of the VPI routines, listed in alphabetical order.

Question: These tables were not updated as part of the merge. Are there additional routines to add? Would a better place for these tables be the beginning of clause 37?

Table 35-1—VPI routines for simulation-related callbacks

To	Use
Register a simulation-related callback	**vpi_register_cb()**
Remove a simulation-related callback	**vpi_remove_cb()**
Get information about a simulation-related callback	**vpi_get_cb_info()**

Table 35-2—VPI routines for system task/function callbacks

To	Use
Register a system task/function callback	**vpi_register_systf()**
Get information about a system task/function callback	**vpi_get_systf_info()**

Table 35-3—VPI routines for traversing SystemVerilog hierarchy

To	Use
Obtain a handle for an object with a one-to-one relationship	**vpi_handle()**
Obtain handles for objects in a one-to-many relationship	**vpi_iterate()** **vpi_scan()**
Obtain a handle for an object in a many-to-one relationship	**vpi_handle_multi()**

Table 35-4—VPI routines for accessing properties of objects

To	Use
Get the value of objects with types of `int` or `bool`	**vpi_get()**
Get the value of objects with types of string	**vpi_get_str()**

Table 35-5—VPI routines for accessing objects from properties

To	Use
Obtain a handle for a named object	**vpi_handle_by_name()**
Obtain a handle for an indexed object	**vpi_handle_by_index()**
Obtain a handle to a word or bit in an array	**vpi_handle_by_multi_index()**

Table 35-6—VPI routines for delay processing

To	Use
Retrieve delays or timing limits of an object	**vpi_get_delays()**
Write delays or timing limits to an object	**vpi_put_delays()**

Table 35-7—VPI routines for logic and strength value processing

To	Use
Retrieve logic value or strength value of an object	**vpi_get_value()**
Write logic value or strength value to an object	**vpi_put_value()**

Table 35-8—VPI routines for simulation time processing

To	Use
Find the current simulation time or the scheduled time of future events	**vpi_get_time()**

Table 35-9—VPI routines for miscellaneous utilities

To	Use
Write to the output channel of the tool that invoked the PLI application and the current log file	**vpi_printf()**
Write to the output channel of the tool that invoked the PLI application and the current log file using varargs	**vpi_vprintf()**
Flush data from the current simulator output buffers	**vpi_flush()**
Open a file for writing	**vpi_mcd_open()**
Close one or more files	**vpi_mcd_close()**
Write to one or more files	**vpi_mcd_printf()**
Write to one or more open files using varargs	**vpi_mcd_vprintf()**
Flush data from a given *mcd* output buffer	**vpi_mcd_flush()**
Retrieve the name of an open file	**vpi_mcd_name()**
Retrieve data about tool invocation options	**vpi_get_vlog_info()**
See whether two handles refer to the same object	**vpi_compare_objects()**
Obtain error status and error information about the previous call to a VPI routine	**vpi_chk_error()**
Free memory allocated by VPI routines	**vpi_free_object()**
Add application-allocated storage to application saved data	**vpi_put_data()**
Retrieve application-allocated storage from application saved data	**vpi_get_data()**
Store user data in VPI work area	**vpi_put_userdata()**
Retrieve user data from VPI work area	**vpi_get_userdata()**
Control simulation execution (e.g., stop, finish)	**vpi_sim_control()**

35.12 VPI backwards compatibility features and limitations

Mantis 1385 (entire sub-clause)

The VPI data model has evolved over many previous versions in order to keep up with corresponding features of the Verilog language. Substantial efforts have been made to maintain backwards-compatibility with

prior versions whenever possible. However, some critical incompatible changes were needed that could not be avoided. This ~~section~~ subclause identifies those incompatibilities and provides a way for older affected applications to continue to run in newer VPI environments, with some important restrictions.

35.12.1 VPI Incompatibilities with other standard versions

The following table summarizes the VPI incompatibilities with prior IEEE standard versions.

Table 35-10—Summary of VPI incompatibilities across standard versions

Incompatibility	1364			1800	
See detailed descriptions below	1995	2001	2005	2005	2008
1) **vpiMemory** exists as an object	Y	D	N	N	N
2) **vpiMemoryWord** exists as an object	Y	D	N	N	N
3) **vpiIntegerVar** and **vpiTimeVar** can be arrays	Y	Y	Y	N	N
4) **vpiRealVar** can be an array	N	Y	Y	N	N
5) **vpiVariables** iterations include **vpiReg** and **vpiRegArray**	N	N	N	Y	Y
6) **vpiReg** iterations on **vpiRegArray** include other objects	N	N	N	Y	Y
7) **vpiRegArray** iterations include variable arrays	N	N	N	Y	Y

Table Key:

— Y = Behavior, function or object present in that version
— D = Behavior, function or object deprecated (present, but use discouraged) in that version
— N = Behavior, function or object not applicable or no longer present in that version

For the above table and details below, the types **vpiReg** and **vpiRegArray** are the same as **vpiLogicVar** and **vpiArrayVar**, respectively, as shown in the 1800 VPI data model (see 36.16 detail 19 36.16, detail 19).

Incompatibility Details:

1) **vpiMemory** exists as an object:

Unpacked unidimensional **`reg`** arrays were exclusively characterized as **vpiMemory** objects in 1364-1995, and later deprecated in 1364-2001. This object type was replaced by **vpiRegArray** in1364-2005, leaving **vpiMemory** allowed as only a one-to-many transition for 1364-2005 and 1800 standard versions (see 36.18). 1364-2001 allowed either **vpiMemory** or **vpiRegArray** types to represent unpacked unidimensional arrays of **vpiReg** objects.

2) **vpiMemoryWord** exists as an object:

Elements of unpacked unidimensional **`reg`** arrays were exclusively characterized as **vpiMemoryWord** objects in 1364-1995, and later deprecated in 1364-2001. This object type was replaced by **vpiReg** in 1364-2005, leaving **vpiMemoryWord** allowed only as an iterator for 1364-2005 and 1800 standard versions (see 36.18). 1364-2001 allowed either **vpiMemoryWord** or **vpiReg** types to represent elements of unpacked unidimensional arrays of **vpiReg** objects.

3) **vpiIntegerVar** and **vpiTimeVar** can be arrays

vpiIntegerVar and **vpiTimeVar** objects could represent unpacked arrays instead of simple variables in all 1364 standards. In 1800 standard versions, these array types are always represented as **vpiRegArray** objects, and **vpiIntegerVar** and **vpiTimeVar** objects are always non-array variables (see 36.16).

4) **vpiRealVar** can be an array

This object type was allowed to represent an unpacked array of such variables in 1364-2001 and 1364-2005 standards (**vpiRealVar** arrays were not yet allowed in 1364-1995). In 1800 standard versions, these are now exclusively represented as **vpiRegArray** objects (see 36.16).

5) **vpiVariables** iterations include **vpiReg** and **vpiRegArray**

In all 1364 standards, **vpiReg** and **vpiRegArray** objects were excluded from **vpiVariables** iterations, and only accessed instead by iterations on **vpiReg** (from a scope or **vpiRegArray**), or **vpiRegArray** (from a scope), respectively. In 1800 standards, they are both included in **vpiVariables** iterations (see 36.16).

6) **vpiReg** iterations on **vpiRegArray** include other objects

This is a consequence of **vpiRegArray** objects being used to represent unpacked arrays of non-**vpiReg** elements in 1800 standards (see 36.16). **vpiReg** iterations on these array objects can retrieve array elements that are of type **vpiIntegerVar** or **vpiTimeVar** for example, which is not expected in standards 1364- 2001 and 1364-2005.

7) **vpiRegArray** iterations include variable array objects

This is another consequence of **vpiRegArray** objects being used to represent unpacked arrays of non-**vpiReg** elements in 1800 standards (see 36.16). In 1364-2001 and 1364-2005 standards, **vpiRegArray** iterations only included arrays of **vpiReg** objects, but, in 1800 standards, this iteration includes arrays of **vpiIntegerVar**, **vpiTimeVar**, and **vpiRealVar**.

35.12.2 VPI Mechanisms to deal with incompatibilities

In order to ease the transition to the latest VPI standard for older applications, capability shall be provided to emulate the incompatible VPI behaviors where they conflict with the current standard. This allows older VPI applications dependent on these behaviors to be run unmodified, as long as they are applied only to designs (or portions of designs) with which they are compatible. This capability is intended only as an interim measure to allow extra time for applications to be upgraded; it does not provide general emulation of older behaviors for newer design constructs. For example, it does not allow 1364 applications to run on portions of designs requiring 1800-level simulation capability.

As described in 35.12.2.1 and 35.12.2.2 below, two mechanisms to support this shall be provided, which can be used in combination.

35.12.2.1 Mechanism 1: Compile-based binding to a compatibility mode

This mechanism requires recompilation of the VPI application source code, and is based on defining a compiler symbol that binds a particular application to a particular compatibility mode. To use this scheme, one of the following compiler symbols ~~must~~ shall be defined prior to compilation of any of the standard VPI include files in the application source code- either using a “#define” in the source code itself (setting it to the numeric constant “1”), or defined on the C-compiler command-line:

```
VPI_COMPATIBILITY_VERSION_1364v1995
VPI_COMPATIBILITY_VERSION_1364v2001
VPI_COMPATIBILITY_VERSION_1364v2005
```

```
VPI_COMPATIBILITY_VERSION_1800v2005
VPI_COMPATIBILITY_VERSION_1800v2008
```

No more than one of these symbols shall be defined for a given application, and it ~~must~~ shall be consistently defined for all of its source code that can access any portion of VPI, including callback functions. This ensures that all design information is handled in the same way for a given mode across the entire application. A compilation error will occur during the processsing of vpi_user.h if more than one of the above symbols is defined.

Example:

VPI source code file with a compatibility mode selected:

```
/* VPI application mytask */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define VPI_COMPATIBILITY_VERSION_1364v2001 1
#include "vpi_user.h"
#include "sv_vpi_user.h"
#include "my_appl_header.h"
...
...
```

Alternatively, the same mode selection could be performed by defining the following option on the C-compiler command line:

```
-DVPI_COMPATIBILITY_VERSION_1364v2001
```

When a mode is selected by one of the means above, C-preprocessor constructs in "vpi_user.h" cause the following VPI functions to be redefined to mode-specific versions:

```
vpi_compare_objects
vpi_control
vpi_get
vpi_get_str
vpi_get_value
vpi_handle
vpi_handle_by_index
vpi_handle_by_multi_index
vpi_handle_by_name
vpi_handle_multi
vpi_iterate
vpi_put_value
vpi_register_cb
vpi_scan
```

For example, defining the mode symbol 'VPI_COMPATIBILITY_VERSION_1364v2001' as shown above will cause 'vpi_handle' to be redefined as:

```
vpi_handle_1364v2001
```

This retargets all calls to 'vpi_handle' in the recompiled application to this mode-specific variant, achieving mode-compatible behavior. See "vpi_compatibility.h" (Annex A) for the complete set of definitions.

35.12.2.2 Mechanism 2: Selection of default VPI compatibility mode run by host simulator

A means to set the default VPI compatibility mode shall be made available by the simulation provider. This shall determine the compatibility mode VPI behavior for all applications not using the compile-based scheme detailed in mechanism #1. Although VPI applications choosing this mechanism can be run without modification or recompilation, only one such default mode shall be selectable for a given simulation run. Additional applications requiring different modes in the same run-time simulation environment ~~must~~ shall use the compile-based mechanism to do so.

35.12.3 Limitations of VPI compatibility mechanisms

When a VPI application uses the compatibility mode mechanism, the application user and application provider should ensure that the design or design partition to which the application is applied is consistent with the mode, and does not include constructs that are only supported in other modes. If the design contains unsupported constructs, the behavior of the VPI implementation is undefined. The extent of checking for consistency between constructs and mode is left to the discretion of the VPI implementation.

In general, VPI users and application developers are strongly encouraged to update their applications to the latest VPI version as soon as possible. The compatibility mode feature should be used only as a temporary solution until such upgrades can be completed or become available. It should be expected that older modes will be phased out as new versions of the standard become available.