Section 30
SystemVerilog Data Read API

This chapter extends the SystemVerilog VPI with read facilities so that the Verilog Procedural Interface (VPI) acts as an Application Programming Interface (API) for data access and tool interaction irrespective of whether the data is in memory or a persistent form such as a database, and also irrespective of the tool the user is interacting with.

30.1 Motivation

SystemVerilog is both a design and verification language consequently its VPI has a wealth of design and verification data access mechanisms. This makes the VPI an ideal vehicle for tool integration in order to replace arcane, inefficient, and error-prone file-based data exchanges with a new mechanism for tool to tool, and user to tool interface. Moreover, a single access API eases the interoperability investments for vendors and users alike. Reducing interoperability barriers allows vendors to focus on tool implementation. Users, on the other hand, will be able to create integrated design flows from a multitude of best-in-class offerings spanning the realms of design and verification such as simulators, debuggers, formal, coverage or test bench tools.

30.1.1 Requirements

SystemVerilog adds several design and verification constructs including:

— C data types such as `int`, `struct`, `union`, and `enum`.

— Advanced built-in data types such as `string`.

— User defined data types and corresponding methods.

— Data types and facilities that enhance the creation and functionality of testbenches.

The access API shall be implemented by all tools as a minimal set for a standard means for user-tool or tool-tool interaction that involves SystemVerilog object data querying (reading). In other words, there is no need for a simulator to be running for this API to be in effect; it is a set of API routines that can be used for any interaction for example between a user and a waveform tool to *read* the data stored in its database. This usage flow is shown in the figure below.

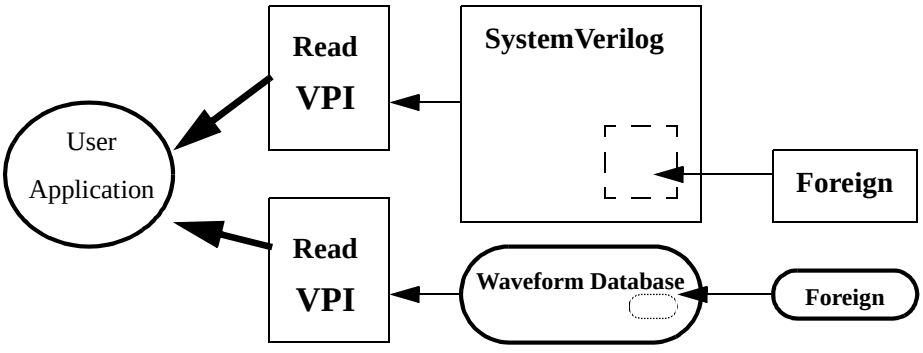


Figure 30-1 — Data read VPI usage model

Our focus in the API is the user view of access. While the API does provide varied facilities to give the user the ability to effectively architect his or her application, it does not address the tool level efficiency concerns such as time-based incremental load of the data, and/or predicting or learning the user access. It is left up to implementors to make this as easy and seamless as possible on the user. To make this easy on tools, the API provides an initialization routine where the user specifies access type and design scope. The user should be pri-

marily concerned with the API specified here, and efficiency issues are dealt with behind the scenes.

30.1.2 Naming conventions

All elements added by this interface shall conform to the 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., vpiName.

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

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

30.2 Extensions to VPI enumerations

These extensions shall be appended to the contents of the `vpi_user.h` file, described in IEEE Std. 1364-2001, Annex G. The numbers in the range **800 - 899** are reserved for the read data access portion of the VPI.

30.2.1 Object types

All objects in VPI have a `vpiType`. This API adds a new object type for data traversal, and two other objects types for object collection and traverse object collection.

```
/* vpiHandle type for the data traverse object */
#define vpiTrvsObj              800 /* use in vpi_handle()       */
#define vpiCollection           810 /* Collection of VPI handles */
#define vpiObjCollection        811 /* Collection of traversable
                                       design objs               */
#define vpiTrvsCollection       812 /* Collection of vpiTrvsObj's*/
```

The other object types that this API references, for example to get a value at a specific time, are all the valid types in the VPI that can be used as arguments in the VPI routines for logic and strength value processing such as `vpi_get_value(<object_handle>, <value_pointer>)`. These types include:

— Constants

— Nets and net arrays

— Regs and reg arrays

— Variables

— Memory

— Parameters

— Primitives

— Assertions

In other words, any limitation in `vpiType` of `vpi_get_value()` will also be reflected in this data access API.

30.2.2 Object properties

This section lists the object property VPI calls.

30.2.2.1 Static info

```
/* Check */
/* use in vpi_get() */
#define vpiIsLoaded                     820 /* is loaded          */
#define vpiHasDataVC                    821 /* has at least one VC
                                           at some point in time
                                           in the database     */
#define vpiHasVC                        822 /* has VC at specific
                                           time               */
#define vpiHasNoValue                   823 /* has no value at
                                           specific time      */
#define vpiInExtension                  824 /* in the extension   */

/* Access */
#define vpiAccessLimitedInteractive     830 /* interactive     */
#define vpiAccessInteractive            831 /* interactive: history */
#define vpiAccessPostProcess            832 /* database        */
/* Member of a collection */
#define vpiMember                       840 /* use in vpi_iterate() */
/* Iteration on instances for loaded */
#define vpiDataLoaded                   850 /* use in vpi_iterate() */
```

30.2.2.2 Dynamic info

30.2.2.2.1 Control constants

```
/* Control Traverse: use in vpi_goto() for a vpiTrvsObj type */
#define vpiMinTime                      860    /* min time      */
#define vpiMaxTime                      861    /* max time      */
#define vpiPrevVC                       862
#define vpiNextVC                       863
#define vpiTime                         864    /* time jump     */
```

These properties can also be used in `vpi_trvs_get_time()` to enhance the access efficiency. The routine `vpi_trvs_get_time()` is similar to `vpi_get_time()` with the additional ability to get the min, max, current, previous VC, and next VC times of the traverse handle; not just the current place it points (as is the case for `vpi_get_time()`). These same `vpiType`s can then be used for access or for moving the traverse handle where the context (get or go to) can distinguish the intent.

30.2.3 System callbacks

The access API adds no new system callbacks. The reader routines (methods) can be called whenever the user application has control and wishes to access data.

30.3 VPI object type additions

30.3.1 Traverse object

To access the value changes of an object over time, the notion of a Value Change (VC) traverse handle is added. A value change traverse object is used to traverse and access value changes not just for the current value (as calling `vpi_get_time()` or `vpi_get_value()` on the object would) but for any point in time: past, present, or future. To create a value change traverse handle the routine `vpi_handle()` is called with a `vpiTrvsObj vpiType`:

```
vpiHandle object_handle; /* design object */
vpiHandle trvsHndl = vpi_handle(/*vpiType*/vpiTrvsObj,
                                /*vpiHandle*/ object_handle);
```

A traverse object exists from the time it is created until its handle is released. It is the application's responsibility to keep a handle to the created traverse object, and to release it when it is no longer needed.

30.3.2 VPI Collection

In order to read data efficiently, we may need to specify a group of objects for example when traversing data we may wish to specify a list of objects that we want to mark as targets of data traversal. To do this grouping we need the notion of a *collection*. A collection represents a user-defined collection of VPI handles. The collection is an ordered list of VPI handles. The `vpiType` of a collection handle can be `vpiCollection`, `vpiObjCollection`, or `vpiTrvsCollection`:

— A collection of type `vpiCollection` is a general collection of VPI handles of objects of any type.

— The collection object of type `vpiObjCollection` represents a collection of VPI *traversable* objects in the design.

— A `vpiTrvsCollection` is a collection of traverse objects of type `vpiTrvsObj`.

Our usage here in the read API is of either:

— Collections of traversable design objects: Used for example in `vpi_handle()` to create traverse handles for the collection. A collection of traversable design objects is of type `vpiObjCollection` (the elements can be any object type in the design except traverse objects of type `vpiTrvsObj`).

— Collections of data traverse objects: Used for example in `vpi_goto()` to move the traverse handles of all the objects in the collection (all are of type `vpiTrvsObj`). A collection of traverse objects is a `vpiTrvsCollection`.

The collection contains a set of member VPI objects and can take on an arbitrary size. The collection may be created at any time and existing objects can be added to it. The purpose of a collection is to group design objects and permit operating on each element with a single operation applied to the whole collection group. `vpi_iterate(vpiMember, <collection_handle>)` is used to create a member iterator. `vpi_scan()` can then be used to scan the iterator for the elements of the collection.

A collection object is created with `vpi_create()`. The first call provides `NULL` handles to the collection object and the object to be added. Following calls, which can be performed at any time, provide the collection handle and a handle to the object for addition. The return argument is a handle to the collection object.

For example:

```
vpiHandle designCollection;
vpiHandle designObj;
vpiHandle trvsCollection;
vpiHandle trvsObj;
/* Create a collection of design objects  */
designCollection = vpi_create(vpiObjCollection, NULL, NULL);
/* Add design object designObj into it    */
designCollection = vpi_create(vpiObjCollection, designCollection, designObj);

/* Create a collection of traverse objects*/
trvsCollection = vpi_create(vpiTrvsCollection, NULL, NULL);
/* Add traverse object trvsObj into it     */
trvsCollection = vpi_create(vpiTrvsCollection, trvsCollection, trvsObj);
```

Sometimes it is necessary to filter a collection and extract a set of handles which meet, or do not meet, a specific criterion for a given collection. The function `vpi_filter()` can be used for this purpose in the form

of:

```
vpiHandle colFilterHdl = vpi_filter((vpiHandle) colHdl, (PLI_INT32) filterType, (PLI_INT32) flag);
```

The first argument of `vpi_filter()`, *colHdl*, shall be the collection on which to apply the filter operation. The second argument, *filterType* can be any `vpiType` or VPI Boolean property. This argument is the criterion used for filtering the collection members. The third argument, *flag*, is a Boolean value. If set to `TRUE`, `vpi_filter()` shall return a collection of handles which match the criterion indicated by *filterType*, if set to `FALSE`, `vpi_filter()` shall return a collection of handles which do not match the criterion indicated by *filterType*. The original collection passed as a first argument remains unchanged.

A collection object exists from the time it is created until its handle is released. It is the application's responsibility to keep a handle to the created collection, and to release it when it is no longer needed.

30.3.2.1 Operations on collections

A traverse collection can be obtained (i.e. created) from a design collection using `vpi_handle()`. The call would take on the form of:

```
vpiHandle objCollection;
/* Obtain a traverse collection from the object collection */
vpi_handle(vpiTrvsCollection, objCollection);
```

The usage of this capability is discussed in Section 30.7.7.

We define another optional method, used in the case the user wishes to directly control the data load, for loading data of objects in a collection: `vpi_load()`. This operation loads all the objects in the collection. It is equivalent to performing a `vpi_load()` on every single handle of the object elements in the collection.

We also define a traversal method on collections of traverse handles i.e. collections of type `vpiTrvsCollection`. The method is `vpi_goto()`.

30.4 Object model diagrams

A traverse object of type `vpiTrvsObj` is related to its parent object; it is a means to access the value data of said object. An object can have several traverse objects each pointing and moving in a different way along the value data horizon. This is shown graphically in the model diagram below. The ***traversable*** class is a representational grouping consisting of any object that:

— Has a name

— Can take on a value accessible with `vpi_get_value()`, the value must be variable over time (i.e. necessitates creation of a traverse object to access the value over time).

The class includes nets, net arrays, regs, reg arrays, variables, memory, primitive, primitive arrays, concurrent

assertions, and parameters. It also includes part selects of all the design object types that can have part selects.

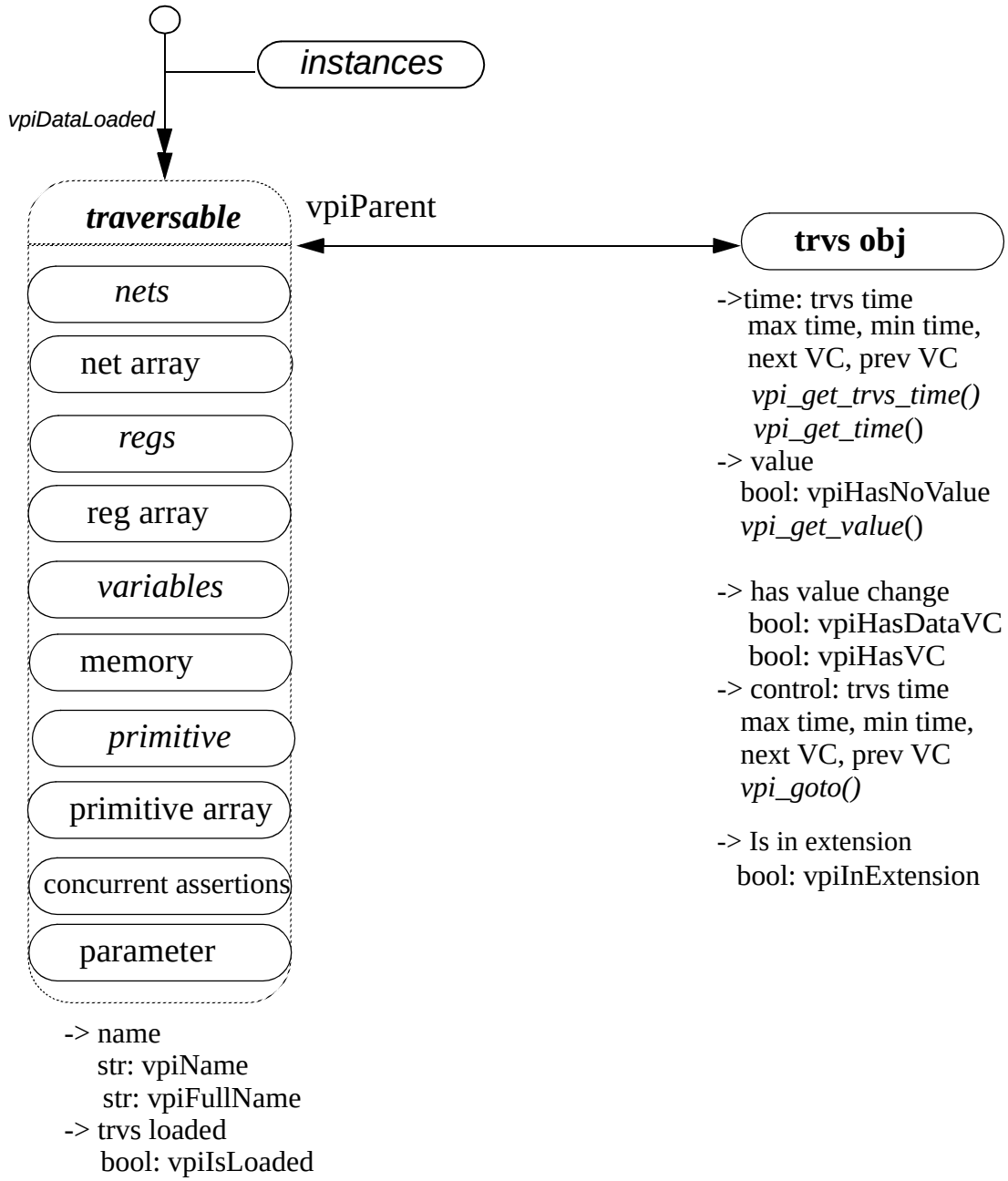
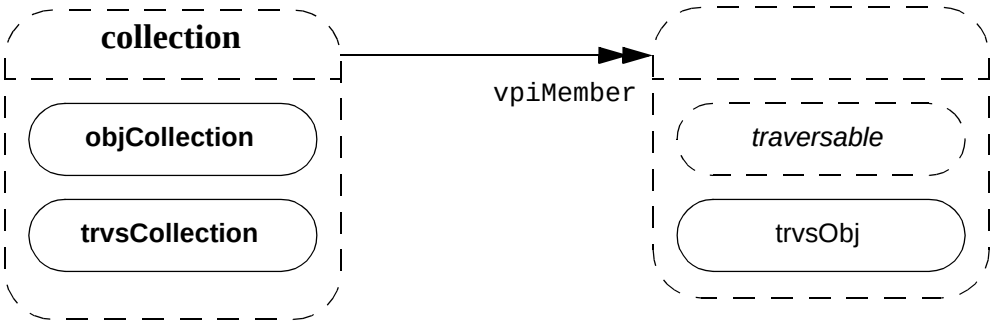


Figure 30-2 — Model diagram of traverse object

A collection object of type `vpiObjCollection` groups together a set of design objects Obj (of any type). A traverse collection object of type `vpiTrvsCollection` groups together a set of traverse objects trvsObj of type `vpiTrvsObj`.



-> creation, addition
vpi_create()
-> filtering
vpi_filter()
-> control: trvs time
max time, min time,
next VC, prev VC
vpi_goto()

-> load, unload
vpi_load()
vpi_unload()

Figure 30-3 — Model diagram of collection

30.5 Usage extensions to VPI routines

Several VPI routines, that have existed before SystemVerilog, have been extended in usage with the addition of new object types and/or properties. While the extensions are fairly obvious, they are emphasized here again to turn the reader’s attention to the extended usage.

Table 30-1: Usage extensions to Verilog 2001 VPI routines

To	Use	New Usage
Get tool's reader version	`vpi_get_vlog_info()`	Reader version return
Create an iterator for the loaded objects (using `vpi_iterate(vpiData-Loaded, <instance>)`). Create an iterator for (object or traverse) collections using `vpi_iterate(vpiMember, <collection>)`.	`vpi_iterate()`	Add iteration types `vpiData-Loaded` and `vpiMember`. Extended with collection handle to create a collection member element iterator.
Obtain a traverse (collection) handle from an object (collection) handle	`vpi_handle()`	Add new types `vpiTrvsObj` and `vpiTrvsCollection`. Extended with collection handle (of traversable objects) to create a traverse collection from an object collection.
Obtain a property.	`vpi_get()`	Extended with the new check properties: `vpiIsLoaded` , `vpiHas-DataVC`, `vpiHasVC`, `vpiHasNoValue`, and `vpiIsEx-tension`.
Get a value.	`vpi_get_value()`	Use traverse handle as argument to get value where handle points.
Get time traverse handle points at.	`vpi_get_time()`	Use traverse handle as argument to get time where handle points.
Free traverse handle Free (traverse) collection handle.	`vpi_free_object()`	Use traverse handle as argument Use (traverse) collection handle as argument.

30.6 VPI routines added in SystemVerilog

This section lists all the VPI routines added in SystemVerilog.

Table 30-2: VPI routines added in SystemVerilog

To	Use
Create a new handle: used to - create an object (traverse) collection - Add a (traverse) object to an existing collection.	`vpi_create()`
Initialize read interface by loading the appropriate reader extension library (simulator, waveform, or other tool). All VPI routines defined by the reader extension library shall be called by indirection through the returned pointer; only built-in VPI routines can be called directly.	`vpi_load_extension()`
Close database and perform any tool cleanup (if opened in `vpiAccessPostProcess` or `vpiAccessInteractive` mode).	`vpi_close()`
Initialize load access.	`vpi_load_init()`
Load data (for a single design object or a collection) onto memory if the users wishes to exercise this level of data load control.	`vpi_load()`
Unload data (for a single design object or a collection) from memory if the user wishes to exercise this level of data load control.	`vpi_unload()`
Get the traverse handle min, max, current, previous VC, or next VC time.	`vpi_trvs_get_time()`
Move traverse (collection) to min, max, or specific time. Return a new traverse (collection) handle containing all the objects that have a VC at that time.	`vpi_goto()`
Filter a collection and extract a set of handles which meet, or do not meet, a specific criterion for a given collection.	`vpi_filter()`

30.7 Reading data

Reading data is performed in 3 steps:

1) A design object must be *selected* for traverse access from a database (or from memory).

2) Indicate the intent to access data. This is typically done by a `vpi_load_init()` call as a hint from the user to the tool on which areas of the design are going to be accessed. The tool will then load the data in an invisible fashion to the user (for example, either right after the call, or at traverse handle creation, or usage). Alternatively, if the user wishes he can (also) choose to add a specific `vpi_load()` call (this can be done at any point in time) to load, or force the load of, a specific object or collection of objects. This can be done either instead of, or in addition to, the objects in the scope or collection specified in `vpi_load_init()`). `vpi_unload()` can be used by the user to force the tool to unload specific objects. It should be noted that traverse handle creation will fail for unloaded objects or collections.

3) Once an object is selected, and marked for load, a traverse object handle can be created and used to traverse the design objects' stored data.

4) At this point the object is available for reading. The traverse object permits the data value traversal and access.

30.7.1 VPI read initialization and load access initialization

Selecting an object is done in 3 steps:

1) The first step is to initialize the read access with a call to `vpi_load_extension()` to load the reader extension and set:

   a) Name of the reader library to be used specified as a character string. This is either a full pathname to this library or the single filename (without path information) of this library, assuming a vendor specific way of defining the location of such a library. The latter method is more portable and therefore recommended. Neither the full pathname, nor the single filename shall include an extension, the name of the library must be unique and the appropriate extension for the actual platform should be provided by the application loading this library More details are in Section 30.9.

   b) Name of the database holding the stored data or flush database in case of `vpiAccessPostProcess` or `vpiAccessInteractive` respectively; a `NULL` can be used in case of `vpiAccessLimitedInteractive`. This is the logical name of a database, not the name of a file in the file system. It is implementation dependent whether there is any relationship to an actual on-disk object and the provided name. See *access mode* below for more details on the access modes.

   c) Access mode: The following VPI properties set the mode of access

   — `vpiAccessLimitedInteractive`: Means that the access will be done for the data stored in the tool memory (e.g. simulator), the history (or future) that the tool stores is implementation dependent. If the tool does not store the requested info then the querying routines shall return a fail. The database name argument to `vpi_load_extension()` in this mode will be ignored (even if not `NULL`).

   — `vpiAccessInteractive`: Means that the access will be done interactively. The tool will then use the database specified as a “flush” area for its data. This mode is very similar to the `vpiAccess`LimitedInteractive with the additional requirement that all the past history (before current time) shall be stored (for the specified scope/collection, see the *access scope/collection* description of `vpi_load_init()`.

   — `vpiAccessPostProcess`: Means that the access will be done through the specified database. All data queries shall return the data stored in the specified database. Data history depends on what is stored in the database, and can be nothing (i.e. no data).

`vpi_load_extension()` can be called multiple times for different reader interface libraries (coming from different tools), database specification, and/or read access. A call with `vpiAccessInteractive` means that the user is querying the data stored inside the simulator database and uses the VPI routines supported by the simulator. A call with `vpiAccessPostProcess` means that the user is accessing the data stored in the database and uses the VPI services provided by the waveform tool. The application, if accessing several databases and/or using multiple read API libraries, can use the routine `vpi_get(vpiInExtension, <vpiHandle>)` to check whether a handle belongs to that database. The call is performed as follows:

```
reader_extension_ptr->vpi_get(vpiInExtension, <vpiHandle>);
```

where `reader_extension_ptr` is the reader library pointer returned by the call to `vpi_load_extension()`. `TRUE` is returned if the passed handle belongs to that extension, and `FALSE` otherwise. If the application uses the built-in library (i.e. the one provided by the tool it is running under), there is no need to use indirection to call the VPI routines; they can be called directly. An initial call must however be made to set the access mode, specify the database, and check for error indicated by a `NULL` return.

In case of `vpiAccessPostProcess` or `vpiAccessInteractive` mode `vpi_close()` shall be called to allow the tool to close the opened database and perform any cleanup. Handles obtained before the call

to vpi_close() are no longer valid after this call.

Multiple databases, possibly in different access modes (for example a simulator database opened in vpiAccessInteractive and a database opened in vpiAccessPostProcess, or two different databases opened in vpiAccessPostProcess) can be accessed at the same time. Section 30.9 shows an example of how to access multiple databases from multiple read interfaces simultaneously.

2) Next step is to specify the elements that will be accessed. This is accomplished by calling vpi_load_init() and specifying a scope and/or an item collection. At least one of the two (scope or collection) needs to be specified. If both are specified then the union of all the object elements forms the entire set of objects the user may access.

   — Access scope: The specified scope handle, and nesting mode govern the scope that access returns. Data queries outside this scope (and its sub-scopes as governed by the nesting mode) shall return a fail in the access routines unless the object belongs to *access collection* described below. It can be used either in a complementary or in an exclusive fashion to *access collection*. NULL is to be passed to the collection when *access scope* is used in an exclusive fashion.

   — Access collection: The specified collection stores the traverse object handles to be loaded. It can be used either in a complementary or in an exclusive fashion to *access scope*. NULL is to be passed to the scope when *access collection* is used in an exclusive fashion.

vpi_load_init() enables access to the objects stored in the database and can be called multiple times. The load access specification of a call remains valid until the next call is executed. This routine serves to initialize the tool load access and provides an entry point for the tool to perform data access optimizations.

30.7.2 Object selection for traverse access

In order to select an object for access, we must first obtain the object handle. This can be done using the VPI routines (that are supported in the tool being used) for traversing the HDL hierarchy and obtaining an object handle based on the type of object relationship to the starting handle.

Any tool that implements this read API (e.g. waveform tool) shall implement at least a basic subset of the design navigation VPI routines that shall include vpi_handle_by_name() to permit the user to get a vpiHandle from an object name. It is left up to tool implementation to support additional design navigation relationships. Therefore, if the application wishes to access similar elements from one database to another, it shall use the *name* of the object, and then call vpi_handle_by_name(), to get the object handle from the relevant database. This level of indirection is always safe to do when switching the database query context, and shall be guaranteed to work.

It should be noted that an object's vpiHandle depends on the access mode specified in vpi_load_extension() and the database accessed (identified by the returned extension pointer, see Section 30.9). A handle obtained through a post process access mode (vpiAccessPostProcess) from a waveform tool for example is not interchangeable *in general* with a handle obtained through interactive access mode (vpiAccessLimitedInteractive or vpiAccessInteractive) from a simulator. Also handles obtained through post process access mode of different databases are not interchangeable. This is because objects, their data, and relationships in a stored database could be quite different from those in the simulation model, and those in other databases.

30.7.3 Optionally loading objects

As mentioned earlier vpi_load_init() allows the tool implementing the reader to load objects in a fashion that is invisible to the user. Optionally, if the user chooses to do their own loading at some point in time, then once the object handle is obtained they can use the VPI data load routine vpi_load() with the object's vpiHandle to load the data for the specific object onto memory. Alternatively, for efficiency considerations, vpi_load() can be called with a design object collection handle of type vpiObjCollection. The collection must have already been created by either using vpi_create() and the (additional) selected object handles added to the load collection using vpi_create() with the created collection list passed as argument. The object(s) data is not accessible as of yet to the user's read queries; a traverse handle must still be cre-

ated. This is presented in Section 30.7.4.

Note that loading the object means loading the object from a database into memory, or marking it for active use if it is already in the memory hierarchy. Object loading is the portion that tool implementors need to look at for efficiency considerations. Reading the data of an object, if loaded in memory, is a simple consequence of the load initialization (`vpi_load_init()`) and/or `vpi_load()` optionally called by the user. The API does not specify here any memory hierarchy or caching strategy that governs the access (load or read) speed. It is left up to tool implementation to choose the appropriate scheme. It is recommended that this happens in a fashion invisible to the user without requiring additional routine calls.

The API here provides the tool with the chance to prepare itself for data load and access with the `vpi_load_init()`. With this call, the tool can examine what objects the user wishes to access before the actual read access is made. The API also provides the user the ability to force loads and unloads but it is recommended to leave this to the tool unless there is a need for the user application to influence this aspect.

30.7.3.1 Iterating the design for the loaded objects

The user shall be allowed to optionally iterate for the loaded objects in a specific instantiation scope using `vpi_iterate()`. This shall be accomplished by calling `vpi_iterate()` with the appropriate reference handle, and using the property `vpiDataLoaded`. This is shown below.

a) Iterate all data read loaded objects in the design: use a `NULL` reference handle (`ref_h`) to `vpi_iterate()`, e.g.,

```
itr = vpi_iterate(vpiDataLoaded, /* ref_h */ NULL);
while (loadedObj = vpi_scan(itr)) {
/* process loadedObj */
}
```

b) Iterate all data read loaded objects in an instance: pass the appropriate instance handle as a reference handle to `vpi_iterate()`, e.g.,

```
itr = vpi_iterate(vpiDataLoaded, /* ref_h */ instanceHandle);
while (loadedObj = vpi_scan(itr)) {
/* process loadedObj */
}
```

30.7.3.2 Iterating the object collection for its member objects

The user shall be allowed to iterate for the design objects in a design collection using `vpi_iterate()` and `vpi_scan()`. This shall be accomplished by creating an iterator for the members of the collection and then use `vpi_scan()` on the iterator handle e.g.

```
vpiHandle var_handle;      /* some object             */
vpiHandle varCollection;   /* object collection       */
vpiHandle Var;             /* object handle           */
vpiHandle itr;             /* iterator handle         */
/* Create object collection                           */
varCollection = vpi_create(vpiObjCollection, NULL, NULL);
/* Add elements to the object collection              */
varCollection    =    vpi_create(vpiObjCollection,    varCollection,
var_handle);

/* Iterating a collection for its elements */
itr = vpi_iterate(vpiMember, varCollection);   /* create iterator  */
while (Var = vpi_scan(itr)) {                  /* scan iterator    */
```



```
/* process Var */
}
```

30.7.4 Reading an object

The sections above have outlined:

— How to select an object for access, in other words, marking this object as a target for access. This is where the design navigation VPI is used.

— How to call `vpi_load_init()` as a hint on the areas to be accessed, and/or optionally load an object into memory after obtaining a handle and then either loading objects individually or as a group using the object collection.

— How to optionally iterate the design scope and the object collection to find the loaded objects if needed.

In this section reading data is discussed. Reading an object data means obtaining its value changes. VPI, before this extension, had allowed a user to query a value at a specific point in time--namely the current time, and its access does not require the extra step of giving a load hint or actually loading the object data. We add that step here because we extend VPI with a temporal access component: The user can ask about all the values in time (regardless of whether that value is available to a particular tool, or found in memory or a database, the mechanism is provided) since accessing this value horizon involves a larger memory expense, and possibly a considerable access time. Let's see now how to access and traverse this value timeline of an object.

To access the value changes of an object over time we use a traverse object as introduced earlier in Section 30.3.1. Several VPI routines are also added to traverse the value changes (using this new handle) back and forth. This mechanism is very different from the "iteration" notion of VPI that returns objects related to a given object, the traversal here can walk or jump back and forth on the value change timeline of an object. To create a value change traverse handle the routine `vpi_handle()` must be called in the following manner:

```
vpiHandle trvsHndl = vpi_handle(vpiTrvsObj, object_handle);
```

Note that the user (or tool) application can create more than one value change traverse handle for the same object, thus providing different views of the value changes. Each value change traverse handle shall have a means to have an internal index, which is used to point to its "current" time and value change of the place it points. In fact, the value change traversal can be done by increasing or decreasing this internal index. What this index is, and how its function is performed is left up to tools' implementation; we only use it as a concept for explanation here.

Once created the traverse handle can point anywhere along the timeline; its initial location is left for tool implementation. However, if the traverse object has no value changes the handle shall point to the minimum time (of the trace), so that calls to `vpi_get_time()` can return a valid time. It is up to the user to call an initial `vpi_goto()` to move to the desired initial pointing location.

30.7.4.1 Traversing value changes of objects

After getting a traverse `vpiHandle`, the application can do a forward or backward walk or jump traversal by using `vpi_goto()` on a `vpiTrvsObj` object type with the new traverse properties.

Here is a sample code segment for the complete process from handle creation to traversal.

```
p_vpi_extension reader_p;  /* Pointer to VPI reader extension structure */
vpiHandle instanceHandle;  /* Some scope object is inside              */
vpiHandle var_handle;      /* Object handle                            */
vpiHandle vc_trvs_hdl;     /* Traverse handle                          */
vpiHandle itr;
p_vpi_value value_p;       /* Value storage                            */
p_vpi_time time_p;         /* Time storage                             */
PLI_INT32 code;            /* return code                              */
...
/* Initialize the read interface: Access data from memory              */
```



```
/* NOTE: Use built-in VPI (e.g. that of simulator application is running
   under)                                                                */
reader_p = vpi_load_extension(NULL, NULL, vpiAccessLimitedInteractive);

if (reader_p == NULL) ... ; /* Not successful */

/* Initialize the load: Access data from simulator) memory, for scope
instanceHandle and its subscopes */
/* NOTE: Call marks access for all the objects in the scope */
vpi_load_init(NULL, instanceHandle, 0);


itr = vpi_iterate(vpiVariables, instanceHandle);
while (var_handle = vpi_scan(itr)) {
/* Demo how to force the load, this part can be skipped in general */
   if (vpi_get(vpiIsLoaded, var_handle) == 0) { /* not loaded*/
      /* Load data: object-based load, one by one */
      if (!vpi_load(var_handle)); /* Data not found !         */
         break;
   }
/*-- End of Demo how to force the load, this part can be skipped in general */
   /* Create a traverse handle for read queries */
   vc_trvs_hdl = vpi_handle(vpiTrvsObj, var_handle);
   /* Go to minimum time */
   vc_trvs_hdl = vpi_goto(vpiMinTime, vc_trvs_hdl, NULL, NULL);
   /* Get info at the min time */
   vpi_get_time(vc_trvs_hdl, time_p); /* Minimum time */
   vpi_printf(...);
   vpi_get_value(vc_trvs_hdl, value_p); /* Value */
   vpi_printf(...);
   if (vpi_get(vpiHasDataVC, vc_trvs_hdl)) { /* Have any VCs ? */
      for (;;) { /* All the elements in time */
         vc_trvs_hdl = vpi_goto(vpiNextVC, vc_trvs_hdl, NULL, &code);
         if (!code) {
            /* failure (e.g. already at MaxTime or no more VCs) */
            break; /* cannot go further */
         }
         /* Get Max */
         /* vpi_trvs_get_time(vpiMaxTime, vc_trvs_hdl, time_p); */
         vpi_get_time(vc_trvs_hdl, time_p);     /* Time of VC  */
         vpi_get_value(vc_trvs_hdl, value_p);   /* VC data     */
      }
   }
}
/* free handles */
vpi_free_object(...);
```

The code segment above declares an interactive access scheme, where only a limited history of values is provided by the tool (e.g. simulator). It then creates a Value Change (VC) traverse handle associated with an object whose handle is represented by `var_handle` but only after `vpi_load_init()` is called. It then creates a traverse handle, `vc_trvs_hdl`. With this traverse handle, it first calls `vpi_goto()` to move to the minimum time where the value has changed. It moves the handle (internal index) to that time by calling `vpi_goto`() with a `vpiMinTime` argument. It then repeatedly calls `vpi_goto()` with a `vpiNextVC` to move the internal index forward repeatedly until there is no value change left. `vpi_get_time()` gets the actual time where this VC is, and data is obtained by `vpi_get_value()`.

The traverse and collection handles can be freed when they are no longer needed using `vpi_free_object()`.

30.7.4.2 Jump Behavior

Jump behavior refers to the behavior of `vpi_goto()` with a `vpiTime` property, `vpiTrvsObj` type, and a jump time argument. The user specifies a time to which he or she would like the traverse handle to jump, but the specified time may or not have value changes. In that case, the traverse handle shall point to the latest VC equal to or less than the time requested.

In the example below, the whole simulation run is from time `10` to time `65`, and a variable has value changes at time `10`, `15` and `50`. If we create a value change traverse handle associated with this variable and try to jump to a different time, the result will be determined as follows:

— Jump to `12`; traverse handle return time is `10`.

— Jump to `15`; traverse handle return time is `15`.

— Jump to `65`; traverse handle return time is `50`.

— Jump to `30`; traverse handle return time is `15`.

— Jump to 0; traverse handle return time is `10`.

— Jump to `50`; traverse handle return time is `50`.

If the jump time has a value change, then the internal index of the traverse handle will point to that time. Therefore, the return time is exactly the same as the jump time.

If the jump time does not have a value change, and if the jump time is not less than the minimum time of the whole trace[2] run, then the return time is aligned backward. If the jump time is less than the minimum time, then the return time will be the minimum time. In case the object has *hold value semantics* between the VCs such as static variables, then the return of `vpi_goto()` (with a specified time argument to jump to) is a new handle pointing to that time to indicate *success*. In case the time is greater than the trace maximum time, or we have an automatic object or an assertion or any other object that does not hold its value between the VCs then the return code should indicate *failure* (and the backward time alignment is still performed). In other words the time returned by the traverse object shall never exceed the trace maximum; the maximum point in the trace is not marked as a VC unless there is truly a value change at that point in time (see the example in this sub-section).

30.7.4.3 Dump off regions

When accessing a database, it is likely that there are gaps along the value time-line where possibly the data recording (e.g. dumping from simulator) was turned off. In this case the starting point of that interval shall be marked as a VC if the object had a stored value before that time. `vpi_goto()`, whether used to jump to that time or using next VC or previous VC traversal from a point before or after respectively, shall stop at that VC. Calling `vpi_get_value()` on the traverse object pointing to that VC shall have no effect on the value argument passed; the time argument will be filled with the time at that VC. `vpi_get()` can be called in the form: `vpi_get(vpiHasNoValue, <traverse handle>)` to return `TRUE` if the traverse handle has no value (i.e. pointing to the start of a dump off region) and `FALSE` otherwise.

There is, of course, another VC (from no recorded value to an actual recorded value) at the end of the dump off interval, if the end exits i.e. there is additional dumping performed and data for this object exits before the end of the trace. There are no VCs in between the two marking the beginning and end (if they exist); a move to the next VC from the start point leads to the end point.

30.7.5 Sample code using object (and traverse) collections

```
p_vpi_extension reader;    /* Pointer to reader VPI library */
vpiHandle scope;           /* Some scope we are looking at  */
vpiHandle var_handle;      /* Object handle                 */
```

[2] The word trace can be replaced by “simulation”; we use trace here for generality since a dump file can be generated by several tools.


```
vpiHandle some_net;       /* Handle of some net           */
vpiHandle some_reg;       /* Handle of some reg           */
vpiHandle vc_trvs_hdl1;   /* Traverse handle              */
vpiHandle vc_trvs_hdl2;   /* Traverse handle              */
vpiHandle itr;            /* Iterator                     */
vpiHandle objCollection;  /* Object collection            */
vpiHandle trvsCollection; /* Traverse collection          */

PLI_BYTE8 *data = “my_database”;/* database               */
p_vpi_time time_p;              /* time                   */
PLI_INT32 code;           /* Return code                  */

/* Initialize the read interface: Post process mode, read from a database */
/* NOTE: Uses “toolX” library                                             */
reader_p = vpi_load_extension(“toolX”, data, vpiAccessPostProcess);

if (reader_p == NULL) ... ; /* Not successful */

/* Get the scope using its name */
scope = reader_p->vpi_handle_by_name(“top.m1.s1”, NULL);
/* Create object collection */
objCollection = reader_p->vpi_create(vpiObjCollection, NULL, NULL);

/* Add data to collection: All the nets in scope */
/* ASSUMPTION: (waveform) tool “toolX” supports this navigation
   relationship */
itr = reader_p->vpi_iterate(vpiNet, scope);
while (var_handle = reader_p->vpi_scan(itr)) {
   objCollection = reader_p->vpi_create(vpiObjCollection, objCollection,
   var_handle);
}
/* Add data to collection: All the regs in scope */
/* ASSUMPTION: (waveform) tool supports this navigation relationship */
itr = reader_p->vpi_iterate(vpiReg, scope);
while (var_handle = reader_p->vpi_scan(itr)) {
   objCollection = reader_p->vpi_create(vpiObjCollection, objCollection,
   var_handle);
}

/* Initialize the load: focus only on the signals in the object collection:
objCollection */
reader_p->vpi_load_init(objCollection, NULL, 0);

/* Demo scanning the object collection */
itr = reader_p->vpi_iterate(vpiMember, objCollection);
while (var_handle = reader_p->vpi_scan(itr)) {
   ...
}


/* Application code here */
some_net = ...;
time_p = ...;
some_reg = ...;
....
vc_trvs_hdl1 = reader_p->vpi_handle(vpiTrvsObj, some_net);
vc_trvs_hdl2 = reader_p->vpi_handle(vpiTrvsObj, some_reg);
vc_trvs_hdl1 = reader_p->vpi_goto(vpiTime, vc_trvs_hdl1, time_p, &code);
vc_trvs_hdl2 = reader_p->vpi_goto(vpiTime, vc_trvs_hdl2, time_p, &code);
```



```
/* Data querying and processing here */
....

/* free handles*/
reader_p->vpi_free_object(...);

/* close database */
reader_p->vpi_close(0, data);
```

The code segment above initializes the read interface for post process read access from database `data`. It then creates an object collection `objCollection` then adds to it all the objects in `scope` of type `vpiNet` and `vpiReg` (assuming this type of navigation is allowed in the tool). Load access is initialized and set to the objects listed in `objCollection`. `objCollection` can be iterated using `vpi_iterate()` to create the iterator and then using `vpi_scan()` to scan it assuming here that the waveform tool provides this navigation. The application code is then free to obtain traverse handles for the objects, and perform its querying and data processing as it desires.

The code segment below shows a simple code segment that mimics the function of a $dumpvars call to access data of all the regs in a specific scope and its subscopes and process the data.

```
p_vpi_extension reader_p;  /* Reader library pointer       */
vpiHandle big_scope;       /* Some scope we are looking at  */
vpiHandle obj_handle;      /* Object handle                 */
vpiHandle obj_trvs_hdl;    /* Traverse handle               */
vpiHandle signal_iterator; /* Iterator for signals          */
p_vpi_time time_p;         /* time                          */

/* Initialize the read interface: Access data from simulator          */
/* NOTE: Use built-in VPI (e.g. that of simulator application is running
   under                                                              */
reader_p = vpi_load_extension(NULL, NULL, vpiAccessLimitedInteractive);

if (reader_p == NULL) ... ; /* Not successful */

/* Initialize the load access: data from (simulator) memory, for scope
   big_scope and its subscopes */
/* NOTE: Call marks load access */
vpi_load_init(NULL, big_scope, 0);

/* Application code here */
/* Obtain handle for all the regs in scope */
signal_iterator = vpi_iterate(vpiReg, big_scope);

/* Data querying and processing here */
while ( (obj_handle = vpi_scan(signal_iterator)) != NULL ) {
   assert(vpi_get(vpiType, obj_handle) == vpiReg);
   /* Create a traverse handle for read queries */
   obj_trvs_hdl = vpi_handle(vpiTrvsObj, obj_handle);
   time_p = ...; /* some time */
   obj_trvs_hdl = vpi_goto(vpiTime, obj_trvs_hdl, time_p, &code);
   /* Get info at time */
   vpi_get_value(obj_trvs_hdl, value_p); /* Value */
   vpi_printf("....");
}
/* free handles*/
vpi_free_object(...);
```


30.7.6 Object-based traversal

Object based traversal can be performed by creating a traverse handle for the object and then moving it back and forth to the next or previous Value Change (VC) or by performing jumps in time. A traverse object handle for any object in the design can be obtained by calling `vpi_handle()` with a `vpiTrvsObj` type, and an object `vpiHandle`. This is the method described in Section 30.7.4, and used in all the code examples thus far.

Using this method, the traversal would be object-based because the individual object traverse handles are created, and then the application can query the (value, time) pairs for each VC. This method works well when the design is being navigated and there is a need to access the (stored) data of any individual object.

30.7.7 Time-ordered traversal

Alternatively, we may wish to do a time-ordered traversal i.e. a time-based examination of values of several objects. We can do this by using a collection. We first create a *traverse collection* of type `vpiTrvsCollection` of the objects we are interested in traversing from the design object collection of type `vpiObjCollection` using `vpi_handle()` with a `vpiTrvsCollection` type and collection handle argument. We can then call `vpi_goto()` on the traverse collection to move to next or previous or do jump in time for the collection as a whole. A move to next (previous) VC means move to the next (previous) *earliest* VC among the objects in the collection; any traverse handle that does not have any VC is ignored; on return its new handle points to the same place as its old. A jump to a specific time aligns the new returned handles of all the objects in the collection (as if we had done this object by object, but here it is done in one-shot for all elements).

We can choose to loop in time by incrementing the time, and doing a jump to those time increments. This is shown in the following code snippet.

```
vpiHandle objCollection = ...;
vpiHandle trvsCollection;
p_vpi_time time_p;
PLI_INT32 code;

/* Obtain (create) traverse collection from object collection */
trvsCollection = vpi_handle(vpiTrvsCollection, objCollection);
/* Loop in time: increments of 100 units */
for (i = 0; i < 1000; i = i + 100) {
   time_p = ...;
   /* Go to point in time */
   trvsCollection = vpi_goto(vpiTime, trvsCollection, time_p, &code);
   ...
}
```

Alternatively, we may wish to get a new collection returned of all the objects that have a value change at the given time we moved the traverse collection to. In this case `vpi_filter()` follows the call to `vpi_goto()`. The latter returns a new collection with all the new traverse objects, whether they have a VC or not. `vpi_filter()` allows us to filter the members that have a VC at that time. This is shown in the code snippet that follows.

```
...
vpiHandle rettrvsCollection; /* Collection for all the objects      */
vpiHandle vctrvsCollection; /* collection for the objects with VC   */
vpiHandle itr;              /* collection member iterator           */
...
/* Go to earliest next VC in the collection */
for (;;) { /* for all collection VCs in time */
   rettrvsCollection = vpi_goto(vpiNextVC, trvsCollection, NULL, &code);
   if (!code) {
      /* failure (e.g. already at MaxTime or no more VCs) */
      break; /* cannot go further */
   }
```



```
      vctrvsCollection = vpi_filter(rettrvsCollection, vpiHasVC, 1);
      /* create iterator then scan the VC collection */
      itr = vpi_iterate(vpiMember, vctrvsCollection);
      while (vc_trvs1_hdl = vpi_scan(itr)) {
         /* Element has a VC */
         vpi_get_value(vc_trvs1_hdl, value_p);  /* VC data */
         /* Do something at this VC point */
         ...
      }
      ...
   }
```

30.8 Optionally unloading the data

The implementation tool should handle unloading the unused data in a fashion invisible to the user. Managing the data caching and memory hierarchy is left to tool implementation but it should be noted that failure to unload may affect the tool performance and capacity.

The user can optionally choose to call `vpi_unload()` to unload the data from (active) memory if the user application is done with accessing the data.

Calling `vpi_unload()` before releasing (freeing) traverse (collection) handles that are manipulating the data using `vpi_free_object()` is not recommended practice by users; the behavior of traversal using *existing* handles is not defined here. It is left up to tool implementation to decide how best to handle this. Tools shall, however, prevent creation of new traverse handles, after the call to `vpi_unload()`, by returning the appropriate fail codes in the respective creation routines.

30.9 Reading data from multiple databases and/or different read library providers

The VPI routine `vpi_load_extension()` is used to load VPI extensions. Such extensions include reader libraries from such tools as waveform viewers. `vpi_load_extension()` shall return a pointer to a function pointer structure with the following definition.

```
typedef struct {
 void *user_data;     /* Attach user data here if needed         */

 /* Below this point user application MUST NOT modify any values */
 size_t struct_size;  /* Must be set to sizeof(s_vpi_extension)  */
 long struct_version; /* Set to 1 for SystemVerilog 3.1a         */
 PLI_BYTE8 *extension_version;
 PLI_BYTE8 *extension_name;

/* One function pointer for each of the defined VPI routines:
 - Each function pointer has to have the correct prototype
*/

 ...
 PLI_INT32 (*vpi_chk_error)(error_info_p);
 ...
 PLI_INT32 (*vpi_vprintf)(PLI_BYTE8 *format, ...);
 ...

} s_vpi_extension, *p_vpi_extension;
```

Subsequent versions of the `s_vpi_extension` structure shall only *extend* it by adding members at the *end*; previously existing entries must not be changed, removed, or re-ordered in order to preserve backward compatability. The `struct_size` entry allows users to perform basic sanity checks (e.g. before type casting),

and the `struct_version` permits keeping track and checking the version of the `s_vpi_extension` structure. The structure also has a `user_data` field to give users a way to attach data to a particular load of an extension if they wish to do so.

The structure shall have an entry for *every* VPI routine. If a particular extension does not support a specific VPI routine, then it shall still have an entry (with the correct prototype), and a dummy body that shall always have a return (consistent with the VPI prototype) to signify failure (i.e. `NULL` or `FALSE` as the case may be). The routine call must also raise the appropriate VPI error, which can be checked by `vpi_chk_error()`, and/or automatically generate an error message in a manner consistent with the specific VPI routine.

The order of the VPI routines must be exactly that of the standard. If tool providers want to add their own implementation extensions, those extensions must only have the effect of making the `s_vpi_extension` structure *larger* and any non-standard content must occur *after* all the standard fields. This permits applications to use the pointer to the extended structure as if it was a `p_vpi_extension` pointer, yet still allow the applications to go beyond and access or call tool-specific fields or routines in the extended structure. For example, a tool extended `s_vpi_extension` could be:

```
typedef struct {
   /* inline a copy of s_vpi_extension              */
   /* begin                                         */
   void *user_data;
   ...
   /* end                                           */
   /* “toolZ” extension with one additional routine  */
   int (*toolZfunc)(int);
} s_toolZ_extension, *p_toolZ_extension;
```

An example of use of the above extended structure is as follows:

```
p_vpi_extension h;
p_toolZ_extension hZ;

h = vpi_load_extension(“toolZ”, <args>);
if ( h && (h->struct_size >= ...)
       && !(strcmp(h->extension_version, “...”)
       && !strcmp(h->extension_name, “toolZ”) ) {
   hZ = (p_toolZ_extension) h;
   /* Can now use hZ to access all the VPI routines, including toolZ’s
      ‘toolZfunc’ */
   ...
}
```

The SystemVerilog tool the user application is running under is responsible for loading the appropriate extension, i.e. the reader API library in the case of the read API. The extension name is used for this purpose, following a specific policy, for example, this extension name can be the name of the library to be loaded. Once the reader API library is loaded all VPI function calls that wish to use the implementation in the library shall be performed using the returned `p_vpi_extension` pointer as an indirection to call the function pointers specified in s_vpi_extension or the extended vendor specific structure as described above. Note that, as stated earlier, in the case the application is using the built-in routine implementation (i.e. the ones provided by the tool (e.g. simulator) it is running under) then the de-reference through the pointer is not necessary.

Multiple databases can be opened for read *simultaneously* by the application. After a `vpi_load_extension()` call, a top scope handle can be created for that database to be used later to derive any other handles for objects in that database. An example of multiple database access is shown below. In the example, scope1 and scope2 are the top scope handles used to point into database1 and database2 respectively and perform the processing (comparing data in the two databases for example).

```
p_vpi_extension reader_pX; /* Pointer to reader libraryfunction struct  */
p_vpi_extension reader_pY; /* Pointer to reader libraryfunction struct  */
```



```
vpiHandle scope1, scope2;  /* Some scope we are looking at             */
vpiHandle var_handle;      /* Object handle                            */
vpiHandle some_net;        /* Handle of some net                       */
vpiHandle some_reg;        /* Handle of some reg                       */
vpiHandle vc_trvs_hdl1;    /* Traverse handle                          */
vpiHandle vc_trvs_hdl2;    /* Traverse handle                          */
vpiHandle itr;             /* Iterator                                 */
vpiHandle objCollection1, objCollection2;/* Object collection    */
vpiHandle trvsCollection1, trvsCollection2;/* Traverse collection*/
p_vpi_time time_p;              /* time                     */

PLI_BYTE8 *data1 = “database1”;
PLI_BYTE8 *data2 = “database2”;

/* Initialize the read interface: Post process mode, read from a database */
/* NOTE: Use library from “toolX”                                        */
reader_pX = vpi_load_extension(“toolX”, data1, vpiAccessPostProcess);
/* Get the scope using its name */
/* NOTE: scope handle comes from database: data1                        */
scope1 = reader_pX->vpi_handle_by_name(“top.m1.s1”, NULL);

/* Initialize the read interface: Post process mode, read from a database */
/* NOTE: Use library from “toolY”                                        */
reader_pY = vpi_load_extension(“toolY”, data2, vpiAccessPostProcess);
/* Get the scope using its name */
/* NOTE: scope handle comes from database: data2                        */
scope2 = reader_pY->vpi_handle_by_name(“top.m1.s1”, NULL);

/* Create object collections */
objCollection1 = reader_pX->vpi_create(vpiObjCollection, NULL, NULL);
objCollection2 = reader_pY->vpi_create(vpiObjCollection, NULL, NULL);

/* Add data to collection1: All the nets in scope1,
   data comes from database1 */
/* ASSUMPTION: (waveform) tool supports this navigation relationship */
itr = reader_pX->vpi_iterate(vpiNet, scope1);
while (var_handle = reader_pX->vpi_scan(itr)) {
   objCollection1 = reader_pX->vpi_create(vpiObjCollection, objCollection1,
   var_handle);
}

/* Add data to collection2: All the nets in scope2,
   data comes from database2 */
/* ASSUMPTION: (waveform) tool supports this navigation relationship */
itr = reader_pY->vpi_iterate(vpiNet, scope2);
while (var_handle = reader_pY->vpi_scan(itr)) {
   objCollection2 = reader_pY->vpi_create(vpiObjCollection, objCollection2,
   var_handle);
}

/* Initialize the load: focus only on the signals in the object collection:
objCollection */
reader_pX->vpi_load_init(objCollection1, NULL, 0);
reader_pY->vpi_load_init(objCollection2, NULL, 0);

/* Demo: Scan the object collection */
itr = reader_pX->vpi_iterate(vpiMember, objCollection1);
while (var_handle = reader_pX->vpi_scan(itr)) {
   ...
```



```
}
itr = reader_pY->vpi_iterate(vpiMember, objCollection2);
while (var_handle = reader_pY->vpi_scan(itr)) {
   ...
}

/* Application code here: Access Objects from database1 or database2 */
some_net = ...;
time_p = ...;
some_reg = ...;
....
/* Data querying and processing here */
....

/* free handles*/
reader_pX->vpi_free_object(...);
reader_pY->vpi_free_object(...);


/* close databases */
reader_pX->vpi_close(0, data1);
reader_pY->vpi_close(0, data2);
```

30.10 VPI routines added in SystemVerilog

This section describes the additional VPI routines in detail.

`vpi_load_extension()`
Synopsis: Load specified VPI extension. The general form of this function allows for later extensions. For the reader-specific form, initialize the reader with access mode, and specify the database if used.
Syntax: `vpi_load_extension(PLI_BYTE8 *extension_name, ...)` in its general form
`vpi_load_extension( PLI_BYTE8 *extension_name, PLI_BYTE8 *name, vpiType mode, ...)` for the reader extension
Returns: `PLI_INT32`, 1 for success, 0 for fail.
Arguments:
PLI_BYTE8 `*extension_name`: Extension name of the extension library to be loaded. In the case of the reader, this is the reader VPI library (with the supported navigation VPI routines).
`...`: Contains all the additional arguments. For the reader extension these are:
`PLI_BYTE8 *name`: Database.
`vpiType mode`:
`vpiAccessLimitedInteractive`: Access data in tool memory, with limited history. The tool shall at least have the current time value, no history is required.
`vpiAccessInteractive`: Access data interactively. Tool shall keep value history up to the current time.
`vpiAccessPostProcess`: Access data stored in specified database.
`...`: Additional arguments if required by specific reader extensions.
Related routines: None.

30.10.1 VPI reader routines

`vpi_close()`
Synopsis: Close the database if open.
Syntax: `vpi_close(PLI_INT32 tool, vpiType prop, PLI_BYTE8* name)`

Returns: `PLI_INT32`, 1 for success, 0 for fail.
Arguments:
`PLI_INT32 tool`: `0` for the reader.
`vpiType prop`:
`vpiAccessPostProcess`: Access data stored in specified database.
`vpiAccessInteractive`: Access data interactively, database is the flush area. Tool shall keep value history up to the current time.
`PLI_BYTE8* name`: Name of the database. This can be the logical name of a database or the actual name of the data file depending on the tool implementation.
Related routines: None.

`vpi_load_init()`
Synopsis: Initialize the load access to scope and/or collection of objects.
Syntax: `vpi_load_init(vpiHandle objCollection, vpiHandle scope, PLI_INT32 level)`
Returns: `PLI_INT32`, 1 for success, 0 for fail.
Arguments:
`vpiHandle objCollection:` Object collection of type `vpiObjCollection`, a collection of design objects.
`vpiHandle scope`: Scope of the load.
`PLI_INT32 level`: If 0 then enables read access to scope and all its subscopes, 1 means just the scope.
Related routines: None.

`vpi_trvs_get_time()`
Synopsis: Retrieve the time of the object or collection of objects traverse handle.
Syntax: `vpi_trvs_get_time(vpiType prop, vpiHandle obj, p_vpi_time time_p)`
Returns: `PLI_INT32`, 1 for success, 0 for fail.
Arguments:
`vpiType prop`:
`vpiMinTime`: Gets the minimum time of traverse object or traverse collection. Returns failure if traverse object or collection has no value changes and `time_p` is not modified.
`vpiMaxTime`: Gets the maximum time of traverse object or traverse collection. Returns failure if traverse object or collection has no value changes and `time_p` is not modified.
`vpiTime`: Gets the time where traverse handle points. Returns failure if traverse object or collection has no value changes and `time_p` is not modified. In the case of a collection, it returns success (and `time_p` is updated) only when all the traverse objects in the collection are pointing to the same time, otherwise returns failure and `time_p` is not modified.
`vpiNextVC`: Gets the time where traverse handle points next. Returns failure if traverse object or collection has no next VC and `time_p` is not modified. In the case of a collection, it returns success when any traverse object in the collection has a next VC, `time_p` is updated with the smallest next VC time.
`vpiPrevVC`: Gets the time where traverse handle previously points. Returns failure if traverse object or collection has no previous VC and `time_p` is not modified. In the case of a collection, it returns success when any traverse object in the collection has a previous VC, `time_p` is updated with the largest previous VC time.
`vpiHandle obj`: Handle to a traverse object of type `vpiTrvsObj` or a traverse collection of type `vpiTrvsCollection.`
`p_vpi_time time_p`: Pointer to a structure containing the returned time information.
Related routines: `vpi_get_time()`. Difference is that `vpi_trvs_get_time()` is more general in that it allows an additional `vpiType` argument to get the min/max/prev/next current time of handle. `vpi_get_time()` can only get the current time of traverse handle.

vpi_load()
Synopsis: Load the data of the given object into memory for data access and traversal if object is an object handle; load the whole collection (i.e. set of objects) if passed handle is an object collection of type vpiObjCollection.
Syntax: vpi_load(vpiHandle h)
Returns: PLI_INT32, 1 for success of loading (all) object(s) (in collection), 0 for fail of loading (any) object (in collection).
Arguments:
vpiHandle h: Handle to a design object (of any valid type) or object collection of type vpiObjCollection.
Related routines: None

vpi_unload()
Synopsis: Unload the given object data from (active) memory if object is an object handle, unload the whole collection if passed object is a collection of type vpiObjCollection. See Section 30.8 for a description of data unloading.
Syntax: vpi_unload(vpiHandle h)
Returns: PLI_INT32, 1 for success, 0 for fail.
Arguments:
vpiHandle h: Handle to an object or collection (of type vpiObjCollection).
Related routines: None.

vpi_create()
Synopsis: Create or add to an object or traverse collection.
Syntax: vpi_create(vpiType prop, vpiHandle h, vpiHandle obj)
Returns: vpiHandle of type vpiObjCollection for success, NULL for fail.
Arguments:
vpiType prop:
vpiObjCollection: Create (or add to) object (vpiObjCollection) or traverse (vpiTrvsCollection) collection.
vpiHandle h: Handle to a (object) traverse collection of type (vpiObjCollection)
vpiTrvsCollection, NULL for first call (creation)
vpiHandle obj: Handle of object to add, NULL if for first time creation of collection.
Related routines: None.

vpi_goto()
Synopsis: Try to move to min, max or specified time. A new traverse (collection) handle is returned pointing to the specified time. If the traverse handle (members of collection) has a VC at that time then the returned handle (members of returned collection) is updated to point to the specified time, otherwise it is not updated. If the passed handle has no VC (for collection this means no VC for any object) a fail is indicated, otherwise a success is indicated. In case of a jump to a specified time, and there is no value change at the specified time, then the value change traverse index of the returned (new) handle (member of returned collection) is aligned based on the jump behavior defined in Section 30.7.4.2, and its time will be updated based on the aligned traverse point. The time argument passed is only relevant in case of a jump to a time (otherwise ignored). It is updated if there is a VC (for collection this means a VC for any object) to the new time, otherwise the value is not updated.
Syntax: vpi_goto(vpiType prop, vpiHandle obj, p_vpi_time time_p, PLI_INT32 *ret_code)
Returns: vpiHandle of type vpitrvsObj (vpiObjCollection).
Arguments:
vpiType prop:
vpiMinTime: Goto the minimum time of traverse collection handle.
vpiMaxTime: Goto the maximum time of traverse collection handle.
vpiTime: Jump to the time specified in time_p.

`vpiHandle obj`: Handle to a traverse object (collection) of type `vpitrvsObj` (`vpitrvsCollection`)
`p_vpi_time time_p`: Pointer to a structure containing time information. Used only if `prop` is of type `vpiTime`, otherwise it is ignored.
`PLI_INT32 *ret_code:` Pointer to a return code indicator. It is 1 for success and 0 for fail.

Related routines: None.

`vpi_filter()`

Synopsis: Filter a general collection, a traversable object collection, or traverse collection according to a specific criterion. Return a collection of the handles that meet the criterion. Original collection is not changed.

Syntax: `vpi_filter(vpiHandle h, PLI_INT32 ft, PLI_INT32 flag)`

Returns: `vpiHandle` of type `vpiObjCollection` for success, `NULL` for fail.

Arguments:

`vpiHandle h`: Handle to a collection of type `vpiCollection`, `vpiObjCollection` or `vpiTrvsCollection`
`PLI_INT32 ft`: Filter criterion, any vpiType or a VPI Boolean property.
`PLI_INT32 flag`: Flag to indicate whether to match criterion (if set to `TRUE`), or not (if set to `FALSE`).

Related routines: None.