Section 29
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 file, and also irrespective of the tool the user is interacting with.

29.1 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.

29.1.1 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.
- Test bench data types and facilities.

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 file database. This usage flow is shown in the figure below.

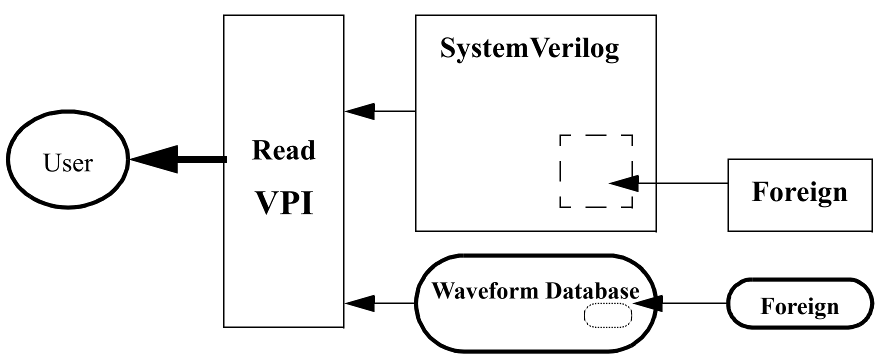

Figure 29-1 — Data read VPI usage model

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

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

29.2 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.

29.2.1 Object type properties

All objects have a `vpiType` property. 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             801 /* Collection of design objs */
#define vpiTrvsCollection         802 /* 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(vpiType,<object_handle>)`. 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(vpiType,<object_handle>)` will also be reflected in this data access API

29.2.2 Object properties

This section lists the object property VPI calls.

29.2.2.1 Static info

```
/* Check type */
#define vpiDataLoaded                       803 /* use in vpi_get() */
```



```
#define vpiTrvsHasVC                         804 /* use in vpi_get() */

/* Access type */
#define vpiAccessLimitedInteractive          805 /* interactive      */
#define vpiAccessInteractive                 806 /* interactive: history */
#define vpiAccessPostProcess                 807 /* data file        */
```

29.2.2.2 Dynamic info

29.2.2.2.1 Control constants

```
/* Control Traverse: use in vpi_control() */
#define vpiTrvsMinTime                       808    /* min time      */
#define vpiTrvsMaxTime                       809    /* max time      */
#define vpiTrvsPrevVC                        810
#define vpiTrvsNextVC                        811
#define vpiTrvsTime                          812    /* time jump     */
```

29.2.2.2.2 Get properties

The following properties are intended to enhance the access efficiency. The function can be alternatively obtained indirectly through a combination of `vpi_control()` call to go to the min, max, or specific time, or without calling `vpi_control()` use the place the handle is already pointing at (if valid), and a `vpi_get_time()` call. No new properties are added here, the same `vpiTypes` can be used where the context (get or go to) can distinguish the intent.

```
/* Get: Use in vpi_trvs_get_time() */
/*
#define vpiTrvsMinTime                       808    // min time
#define vpiTrvsMaxTime                       809    // max time
#define vpiTrvsTime                          812    // traverse handle time
*/
```

29.2.3 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.

29.3 VPI object type additions

29.3.1 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);
```

29.3.2 Collection object

In order to read data efficiently, we may need to specify a group of objects for example when loading (or traversing) data we may wish to specify a list of objects that we want to mark as targets of data load (or traversal).

Such a grouping we refer to as a *collection*. We add the notion of a collection of objects to VPI.

The collection object of type `vpiCollection` represents a user-defined collection of VPI objects in the design; these cannot be traverse objects of type `vpiTrvsObj`. The collection contains a set of 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_scan()` is used to iterate on the elements of the collection.

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

Our usage here in the read API is of either:

- Design object collections: Used for example in `vpi_read_load()` to load data for all the objects. A collection of objects if of type `vpiCollection` in general (the elements can be any object type in the design except traverse objects of type `vpiTrvsObj`).
- Data traverse objects: Used for example in `vpi_control()` to move the traverse handles of all the objects in the collection. A collection of traverse objects is a `vpiTrvsCollection`.

A collection object is created with `vpi_create()`. The first call gives `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(vpiCollection, NULL, NULL);
/* Add design object designObj into it    */
designCollection = vpi_create(vpiCollection, 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);
```

A collection objects 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.

29.3.2.1 Operations on collections

We define a method for loading data of objects in a collection: `vpi_read_load()`. This operation loads all the objects in the collection. It is equivalent to performing a `vpi_read_load()` on every single handle of the object elements in the collection.

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 loadCollection;
/* Obtain a traverse collection from the load collection */
vpi_handle(vpiTrvsCollection, loadCollection);
```

The usage of this capability is discussed in Section 29.7.6.

We also define a method on collections of traverse handles i.e. collections of type `vpiTrvsCollection`. This method is `vpi_control()`. Its function is equivalent to applying `vpi_control()` with the same time control arguments to move the traverse handle of every single object in the collection.

29.4 Object model diagram additions

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 many 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 “named valued” class is a representational grouping consisting of any object that:

— Has a name

— Can take on a value accessible with `vpi_get_value(vpiType,<object_handle>)` where `vpiType` is the type of the named object; one that can take on a value. See Section 29.1.1.

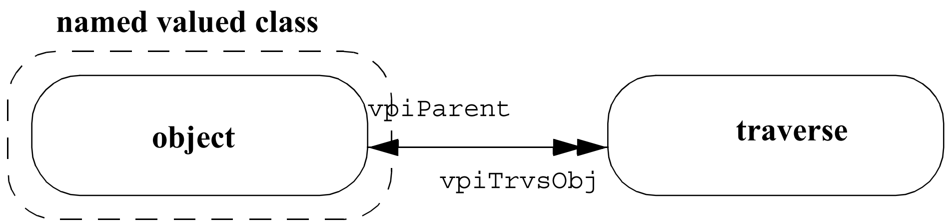

Figure 29-2 — Model diagram of traverse object

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

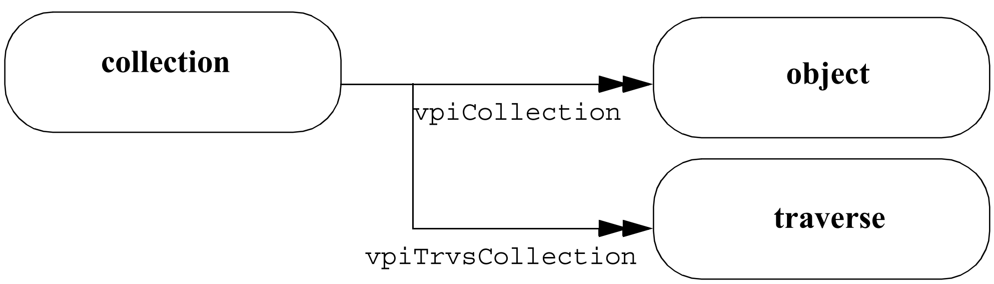

Figure 29-3 — Model diagram of object collection

29.5 Usage extensions to VPI routines

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

Table 29-1: Usage extensions to existing VPI routines

To	Use	New Usage
Iterate on all loaded objects	`vpi_iterate()`	Add property `vpiDataLoaded`
Obtain a traverse (collection) handle from an object (collection) handle	`vpi_handle()`	Add a new property `vpiTrvsObj` and `vpiTrvsCollection`
Scan the (traverse) objects in a collection	`vpi_scan()`	Add a (traverse) object collection (`vpiTrvsCollection`) `vpiCollection` as a handle argument to get its elements
Obtain a property	`vpi_get()`	Extended with the new check properties: `vpiDataLoaded` and `vpiTrvsHasVC`
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
Move traverse (collection) handle to min, max, or specific time	`vpi_control()`	Use traverse (collection) handles/ properties. Extended with a time argument in case of jump to specific time.

29.6 New additions to VPI routines

This section lists all the new VPI routine additions.

Table 29-2: New Reader VPI routines

To	Use
Create a new handle: used to - create an object (traverse) collection for loading - Add a (traverse) object to an existing collection	`vpi_create()`
Get read interface version	`vpi_read_get_version()`
Initialize read interface	`vpi_read_init()`
Load data (for a single design object or a collection) onto memory	`vpi_read_load()`
Unload data (for a single design object or a collection) from memory	`vpi_read_unload()`
Get the traverse handle min, max, or current time where it points.	`vpi_trvs_get_time()`

29.7 Reading data

Reading data is performed in 3 steps:

1) A design object must be *selected* for loading from a database (or from memory) into active memory.

2) Once an object is selected, it can then be *loaded* into memory. This step creates the traverse object handle used to traverse the design objects stored data.

3) At this point the object is available for reading. A traverse object must be created to permit the data value traversal and access.

29.7.1 Object selection for loading

Selecting an object is done in 3 steps:

1) The first step is to initialize the read access with a call to `vpi_read_init()` by setting:

a) Access type: The vpi properties set the type 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 file argument to `vpi_read_init()` 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 data file specified as a “flush” file for its data. This mode is very similar to the `vpiAccess`Limit-edInteractive 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 below).

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

b) Specifying the elements that will be accessed is accomplished by specifying a scope and/or an item collection. At least one of the two 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_read_init()` can be called multiple times. The access specification of a call remains valid until the next call is executed.

2) The next step entails obtaining the object handle. This can be done using any of the VPI routines for traversing the HDL hierarchy and obtaining an object handle based on the type of object relationship to the starting handle. These routines are listed in the following table.

Table 29-3: VPI routines for obtaining handle from hierarchy or property

To	Use
Obtain a handle for an object with a one-to-one relationship	`vpi_handle()`
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()`
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()`

29.7.2 Loading objects

Once the object handle is obtained then we can use the VPI data load routine `vpi_read_load()` with the object's `vpiHandle` to load the data for the specific object onto memory. Alternatively, for efficiency considerations, `vpi_read_load()` can be called with a design object collection handle of type `vpiCollection`. The collection must have already been created using `vpi_create()` and the 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 created. This is presented in Section 29.7.3.

Note that loading the object means loading the object from a file 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. 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. The API does provide the tool with the chance to prepare itself with the `vpi_read_init()`. With this call, the tool can examine what type of access, and what signals the user wishes to access before the actual load and then read access is made.

29.7.2.1 Iterating the design for the loaded objects

The user shall be allowed to 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 */
}
```

29.7.2.2 Iterating the load collection for its element loaded objects

The user shall be allowed to iterate for the loaded objects in the load collection using `vpi_scan()`. This shall be accomplished by using the load collection handle as a reference handle to `vpi_scan()` e.g.

```
vpiHandle var_handle;      /* some object             */
vpiHandle varCollection;   /* object collection       */
vpiHandle loadedVar;       /* Loaded object handle    */
/* Create load object collection                      */
varCollection = vpi_create(vpiCollection, NULL, NULL);
/* Add elements to the object collection              */
varCollection = vpi_create(vpiCollection, varCollection, var_handle);

/* Iterating a collection for its elements */
while (loadedVar = vpi_scan(varCollection)) {  /* scan the list */
/* process loadedVar */
}
```

29.7.3 Reading an object

So far we have outlined:

- How to select an object for loading, in other words, marking this object as a target for access.
- How to load an object into memory by obtaining a handle and then either loading objects individually or as a group using the load collection.
- How to iterate the design scope and the load collection to find the loaded objects if needed.

Reading an object 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 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 file, the mechanism is provided). Since accessing this value horizon involves a larger memory expense, and possibly a considerable access time, we have added also in this Chapter the notion of loading an objects's data for read. 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 introduced earlier in Section 29.3.1. 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 accesses properties of an 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. It is up to the user to call an initial `vpi_control()` to the desired initial pointing location.

29.7.3.1 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_control()` with the new traverse properties.

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

```
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                  */
...
/* Initialize the read interface
   Access data from (say simulator) memory, for scope instanceHandle
   and its subscopes */
/* Call marks access for all the objects in the scope */
vpi_read_init(vpiAccessLimitedInteractive, NULL, NULL, instanceHandle, 0);

itr = vpi_iterate(vpiVariables, instanceHandle);
while (var_handle = vpi_scan(itr)) {
   if (vpi_get(vpiDataLoaded, var_handle) == 0) { /* not loaded*/
      /* Load data: object-based load, one by one */
      if (!vpi_read_load(var_handle)); /* Data not found !     */
         break;
   }
   /* Create a traverse handle for read queries */
   vc_trvs_hdl = vpi_handle(vpiTrvsObj, var_handle);
   /* Go to minimum time */
   vpi_control(vpiTrvsMinTime, vc_trvs_hdl);
   /* 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(vpiTrvsHasVC, vc_trvs_hdl)) { /* Have VCs ? */
      for (;;) { /* scan all the elements in time */
         if (vpi_control(vpiTrvsNextVC, vc_trvs_hdl) == 0) {
            /* already at MaxTime */
            break; /* cannot go further */
```



```
            }
            /* Get Max */
            /* vpi_trvs_get_time(vpiTrvsMaxTime, 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 the object is loaded into memory first. It then creates a traverse handle, `vc_trvs_hdl`. With this traverse handle, it first calls `vpi_control()` to go to the minimum time where the value has changed and moves the handle (internal index) to that time by calling `vpi_control`() with a `vpiTrvsMinTime` argument. It then repeatedly calls `vpi_control()` with a `vpiTrvsNextVC` 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()`.

29.7.3.2 Jump Behavior

Jump behavior refers to the behavior of `vpi_control()` with a `vpiTrvsTime` property 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`0` to time `65`, and a variable has value changes at time `0`, `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 `10`; traverse handle return time is `0`.

— 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 (`-1`); traverse handle return time is `0`.

— 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 code of `vpi_control()` (with a specified time argument to jump to) should 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).

[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.

29.7.4 Sample code using (load and traverse) object collections

```
vpiHandle scope;           /* Some scope we are looking at  */
vpiHandle var_handle;      /* Object handle                 */
vpiHandle some_port;       /* Handle of some port           */
vpiHandle some_reg;        /* Handle of some reg            */
vpiHandle vc_trvs_hdl1;    /* Traverse handle               */
vpiHandle vc_trvs_hdl2;    /* Traverse handle               */
vpiHandle itr;             /* Iterator                      */
vpiHandle loadCollection;  /* Load collection               */
vpiHandle trvsCollection;  /* Traverse collection           */

char *datafile = ...;/* data file   */
p_vpi_time time_p;   /* time        */
...
/* Create load collection */
loadCollection = vpi_create(vpiCollection, NULL, NULL);

/* Add data to collection: All the ports in scope */
itr = vpi_iterate(vpiPort, scope);
while (var_handle = vpi_scan(itr)) {
   loadCollection = vpi_create(vpiCollection, loadCollection, var_handle);
}
/* Add data to collection: All the regs in scope */
itr = vpi_iterate(vpiReg, scope);
while (var_handle = vpi_scan(itr)) {
   loadCollection = vpi_create(vpiCollection, loadCollection, var_handle);
}
/* Initialize the read interface: Post process mode, read from a file,
   and focus only on the signals in the load collection: loadCollection */
vpi_read_init(vpiAccessPostProcess, datafile, loadCollection, NULL, 0);

/* Demo scanning the load collection */
while (var_handle = vpi_scan(loadCollection)) {
   ...
}

/* Load the data in one shot using load collection */
vpi_read_load(loadCollection);

/* Application code here */
some_port = ...;
time_p = ...;
some_reg = ...;
....
vc_trvs_hdl1 = vpi_handle(vpiTrvsObj, some_port);
vc_trvs_hdl2 = vpi_handle(vpiTrvsObj, some_reg);
vpi_control(vpiTrvsTime, some_port, time_p);
/* Data querying and processing here */
....

/* free handles*/
vpi_free_object(...);
```

The code segment above creates an object load collection`loadCollection` then adds to it all the objects in `scope` of type `vpiPort` and `vpiReg`. It then initializes the read interface for post process read access from file `datafile` with access to the objects listed in `loadCollection`. `loadCollection` can be scanned using `vpi_scan()` if need be. The selected objects are then loaded in one shot using `vpi_read_load()` with `loadCollection` as argument. The application code is then free to obtain traverse handles for the loaded objects, and perform its querying and data processing as it desires.

29.7.5 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 29.7.3, 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 an object.

29.7.6 Time-ordered traversal

Alternatively, we may wish to do a time-ordered traversal i.e. a time-based scan 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 `vpiCollection` using `vpi_handle()` with a `vpiTrvsCollection` type and collection handle argument. We can then call `vpi_control()` 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. A jump to a specific time aligns the 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 loadCollection = ...;
vpiHandle trvsCollection;
p_vpi_time time_p;

/* Obtain (create) traverse collection from load collection */
trvsCollection = vpi_handle(vpiTrvsCollection, loadCollection);
/* Loop in time: increments of 100 units */
for (i = 0; i < 1000; i = i + 100) {
   time_p = ...;
   /* Go to point in time */
   vpi_control(vpiTrvsTime, trvsCollection, time_p);
}
```

Alternatively we may wish to go to the next VC of the traverse collection defined to be the *VC with the smallest time* among the VCs in the traverse object in the collection; again traverse objects with no VCs are ignored. This is shown in the following code snippet.

```
vpiHandle loadCollection = ...;
vpiHandle trvsCollection;
p_vpi_time time_p;

/* Create traverse collection */
trvsCollection = vpi_handle(vpiTrvsCollection, loadCollection);

/* Go to earliest next VC in the collection */
for (;;) { /* scan all the elements in time */
   if (vpi_control(vpiTrvsNextVC, trvsCollection) == 0) {
      /* already at MaxTime */
      break; /* cannot go further */
   }
   vpi_get_time(vc_trvs_hdl, time_p);     /* Time of VC */
   vpi_get_value(vc_trvs_hdl, value_p);   /* VC data */
   /* Do something at this VC point */
```

```
    ...
}
```

29.8 Unloading the data

Once the user application is done with accessing the data it had loaded, it shall call `vpi_read_unload()` to unload the data from (active) memory. Failure to call this unload may affect the tool performance and capacity and its consequences are not addressed here since managing the data caching and memory hierarchy is left to tool implementation.

Calling `vpi_read_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 by returning the appropriate fail codes in the respective creation routines; this situation is similar to attempting to create traverse handles before doing any data loads with `vpi_read_load()`, and shall be treated in a similar fashion.

29.9 Reader VPI routines

29.9.1 Extensions to existing routines

This section describes the extensions to existing VPI routines. Most are obvious and shown in Table 29-1. `vpi_control()` is described here again for clarity.

`vpi_control()`
Synopsis: Try to move value change traverse index to min, max or specified time. If the request is for a next or previous VC and there is none (for collection this means no VC for any object) a fail is returned, otherwise a success is returned. If there is no value change at specified time in a jump, then the value change traverse index is aligned based on the jump behavior defined earlier in Section 29.7.3.2, and the time will be updated based on the aligned traverse point. If there is a value change occurring at the requested time, then the value change traverse index is moved to that tag with success return, otherwise if the object does not have hold semantics a fail is returned. In the case of a collection, a fail is only returned when none of the objects in its group can return a true.
Syntax: `vpi_control(vpiType prop, vpiHandle obj, p_vpi_time time_p)`
Returns: `PLI_INT32`, 1 for success, 0 for fail.
Arguments:
- `vpiType prop`:
  - `vpiTrvsMinTime`: Goto the minimum time of traverse handle.
  - `vpiTrvsMaxTime`: Goto the maximum time of traverse handle.
  - `vpiTrvsTime`: Jump to the time specified in `time_p`.
- `vpiHandle obj`: Handle to a traverse object of type `vpiTrvsObj`.
- `p_vpi_time time_p`: Pointer to a structure containing time information. Used only if `prop` is of type `vpiTrvsTime`, otherwise it is ignored.

Related routines: None.

29.9.2 Additional routines

This section describes the additional VPI routines in detail.

`vpi_read_getversion()`
Synopsis: Get the reader version.
Syntax: `vpi_read_getversion()`
Returns: `char*`, for the version string
Arguments: None
Related routines: None

vpi_read_init()
Synopsis: Initialize the reader with access type and access scope, and/or collection of objects.
Syntax: `vpi_read_init(vpiType prop, char* filename, vpiHandle loadCollection, vpiHandle scope, PLI_INT32 level)`
Returns: `PLI_INT32`, 1 for success, 0 for fail.
Arguments:
- `vpiType prop`:
  - `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 data file.
- `char* filename`: Data file.
- `vpiHandle loadCollection:` Load collection of type `vpiCollection`, a collection of design objects.
- `vpiHandle scope`: Scope of the read.
- `PLI_INT32 level`: If 0 then enables 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: void
Arguments:
- `vpiType prop`:
  - `vpiTrvsMinTime`: Gets the minimum time of traverse handle or traverse collection.
  - `vpiTrvsMaxTime`: Gets the maximum time of traverse handle or traverse collection.
  - `vpiTrvsTime`: Gets the time where traverse handle points. If the object handle is a traverse collection, then the behavior is undefined and left to tool implementation.
- `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/current time of handle. `vpi_get_time()` can only get the current time of traverse handle.

vpi_read_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 a load collection of type `vpiCollection`.
Syntax: `vpi_read_load(vpiHandle h)`
Returns: `PLI_INT32`, 1 for success, 0 for fail.
Arguments:
- `vpiHandle h`: Handle to a design object (of any valid type) or object collection of type `vpiCollection.`

Related routines: None

vpi_read_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 `vpiCollection`. See Section 29.8 for a description of data unloading.

Syntax: `vpi_read_unload(vpiHandle h)`
Returns: `PLI_INT32`, 1 for success, 0 for fail.
Arguments:
`vpiHandle h`: Handle to an object or collection (of type `vpiCollection`).
Related routines: None.

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