Section 29
SystemVerilog Data Read and Write API

This chapter extends the SystemVerilog VPI with read and write 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

The data access API permits access to SystemVerilog data. 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 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), or storage of such data (writing). 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 or to *write* data so that the tool (or any other tool in its class) can store the data.

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. 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 and write 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 the file object of the *writer*.

```
/* vpiHandle type for the data write file object */
#define vpiDataWriteFileType      800 // use in vpi_data_write_open()
```

The other object types that this API (*reader* or *writer*) references, for example to get a value at a specific time for, 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

```
/* Create, Load, Check */
#define vpiDataReadLoadList           801 // load list
#define vpiDataReadIsLoaded           802 // use in vpi_get()
#define vpiDataReadTrvsHndl           803 // use in vpi_handle()
#define vpiDataReadTrvsHasVC          804 // use in vpi_get()

/* Access type */
#define vpiDataReadAccess             805 // tool memory
#define vpiDataReadAccessInteractive 806 // interactive
#define vpiDataReadAccessPostProcess 807 // data file
```

29.2.2.2 Dynamic info

29.2.2.2.1 Control constants

```
/* Control Traverse: use in vpi_control() */
#define vpiDataReadTrvsMinTime        808    // min time
#define vpiDataReadTrvsMaxTime        809    // max time
#define vpiDataReadTrvsGotoPrevVC     810
```



```
#define vpiDataReadTrvsGotoNextVC    811

/* Jump: use in vpi_data_read_jump() */
#define vpiDataReadTrvsTime          812    // traverse handle time
```

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 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 goto) can distinguish the intent.

```
/* Get: Use in vpi_data_read_get_time() */
//#define vpiDataReadTrvsMinTime     808    // min time
//#define vpiDataReadTrvsMaxTime     809    // max time
//#define vpiDataReadTrvsTime        812    // traverse handle time
```

29.2.3 System callbacks

This section lists the system callbacks. The reader /writer routines (methods) can be called whenever the user application task has control and wishes to access data. Primarily the callback is for the writer to know when it has to write a value: The reason is **cbValueChange**.

29.3 Usage extensions to VPI routines

Several VPI routines have been extended in usage with the addition of new object properties. In effect, this is already covered with the addition of the new properties above, we just emphasize this again here 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 `vpiDataReadIs-Loaded`
Obtain a traverse handle	`vpi_handle()`	Add a new property `vpiDataR-eadTrvsHndl`
Scan the objects	`vpi_scan()`	Add a new object to iterate on of type `vpiDataReadLoadList` to get its elements
Obtain a property	`vpi_get()`	Extended with the new properties
Get a value	`vpi_get_value()`	Use traverse handle as argument to get value where handle points
Get time	`vpi_get_time()`	Use traverse handle as argument to get time where handle points
Free traverse handle	`vpi_free_object()`	Use traverse handle as argument Use object list handle as argument
Move traverse handle	`vpi_control()`	Use traverse properties

29.4 New additions to VPI routines

This section lists all the new VPI routine additions.

Table 29-2: New Reader VPI routines

To	Use
Get read interface version	`vpi_data_read_get_version()`
Initialize read interface	`vpi_data_read_init()`
Load data onto memory	`vpi_data_read_load()`
Unload data from memory	`vpi_data_read_unload()`
Create object load list	`vpi_data_read_createloadlist()`
Add to object load list	`vpi_data_read_addtoloadlist()`
Check if object is in load list	`vpi_data_read_isinloadlist()`
Reset object load list	`vpi_data_read_resetloadlist()`
Jump to a specific time	`vpi_data_read_jump()`
Get the traverse handle time	`vpi_data_read_get_time()`

Table 29-3: New Writer VPI routines

To	Use
Get write interface version	`vpi_data_write_get_version()`
Open file, set version, create write object	`vpi_data_write_open()`
Begin tree creation	`vpi_data_write_begintree()`
Set the scale unit	`vpi_data_write_setscaleunit()`
Create a scope (and set as current)	`vpi_data_write_createscope()`
Move up out of current scope	`vpi_data_write_createupscope()`
Create a var in scope	`vpi_data_write_createvar()`
Close the tree creation	`vpi_data_write_endtree()`
Create a time where a value change occurs	`vpi_data_write_createtime()`
Create a value change	`vpi_data_write_createvalue()`
Close (and free) write object (and file)	`vpi_data_write_close()`

29.5 Data reader

29.5.1 Object selection for loading

Selecting an object is done in 3 steps:

1) The first step is to initialize the read access by setting:

a) Access type: The vpi properties set the type of access

i) `vpiDataReadAccess`: 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 a return a fail. The file argument to `vpi_data_read_init()` in this mode will be ignored (even if not `NULL`).

ii) `vpiDataReadAccessInteractive`: 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 `vpiDataReadAccess` with the additional requirement that all the past history (before current time) shall be stored (for the specified scope, see the *Access Scope* description below).

iii) `vpiDataReadAccessPostProcess`: 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).

d) 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 list* described below. It can be used either in a complementary or in an exclusive fashion to *access list*. `NULL` is to be passed to the list when *access scope* is used in an exclusive fashion.

e) Access list: The specified list stores 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 list* is used in an exclusive fashion.

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 29-4: 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()`

29.5.2 Loading objects

Once the object handle is obtained then we can use the VPI data load routine `vpi_data_read_load()` with the object's `vpiHandle` to load the data for the specific object onto memory. Alternatively, for efficiency considerations, `vpi_data_read_load()` can be called with a list handle. The list must have already been created using `vpi_data_read_createloadlist()` and the selected object handles added to the load list using `vpi_data_read_addtoloadlist()`. The object(s) shall then be accessible to the user's read queries.

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_data_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.

29.5.3 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 `vpiDataReadIsLoaded`. 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(vpiDataReadIsLoaded, /* 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(vpiDataReadIsLoaded, /* ref_h */ instanceHandle);
while (loadedObj = vpi_scan(itr)) {
/* process loadedObj */
}
```

29.5.4 Iterating the load list for the loaded objects

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

```
vpiHandle var_handle; // some object
loadlist = vpi_data_read_createloadlist();     // create load list
vpi_data_read_addtolist(loadlist, var_handle); // add elements to it
while (loadedObj = vpi_scan(loadlist)) {       // scan the list
/* process loadedObj */
}
```

29.5.5 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 list.

— How to iterate the design scope and the load list to find the loaded objects.

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, the notion of a Value Change (VC) traverse handle is added. 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(vpiDataReadTrvsHndl, 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.

29.5.5.1 Traversing value changes

After getting a traverse `vpiHandle`, the application can do a forward, backward walk traversal by using `vpi_control()` with the new traverse properties. Forward and backward jumping can be performed with the `vpi_data_read_jump()` VPI routine. 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 loads all the objects in the scope
vpi_data_read_init(vpiDataReadAccess, NULL, NULL, instanceHandle, 0);

itr = vpi_iterate(vpiVariables, instanceHandle);
while (var_handle = vpi_scan(itr)) {
    if (vpi_get(vpiDataReadIsLoaded, var_handle) == 0) { // not loaded
        if (!vpi_data_read_load(var_handle)); // Data not found !
            break;
    }
    vc_trvs_hdl = vpi_handle(vpiDataReadTrvsHndl, var_handle);
    // Goto minimum time
```

```
        vpi_control(vpiDataReadMinTime, vc_trvs_hdl);
        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(vpiDataReadTrvsHasVC, vc_trvs_hdl))
        for (;;) { // scan all the elements in time
            if (vpi_control(vpiDataReadGotoNextVC, vc_trvs_hdl) == 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
        }
    }
    // free handles
    vpi_free_object(...);
```

The code segment creates a Value Change (VC) traverse handle associated with an object, whose handle is represented by `var_handle`, and creates a traverse handle, `vc_trvs_hdl`. With this traverse handle, it first calls `vpi_control()` to get the minimum time where the value has changed, then it moves the handle (internal index) to that time by calling with a `vpiDataReadMinTime`; and, finally, it calls `vpi_control()` with a `vpiDataReadGotoNextVC` 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 handle can be freed when it is no longer needed using `vpi_free_object()`.

29.5.5.2 Jump Behavior

Jump behavior refers to the behavior of `vpi_data_read_jump()`. 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. then the traverse handle shall point to the latest VC equal to or less than the time requested. In the figure 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.

---

[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.

In case the object has *hold value semantics* between the VCs such as static variables, then the return code of `vpi_data_read_jump()` 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).

29.5.6 Sample code using the load list

```
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 loadlist;     // Load list

char *datafile = ...;   // data file
p_vpi_time time_p;      // time

...
// Create load list
loadlist = vpi_data_read_createloadlist();
assert(vpi_get(vpiDataReadLoadList, loadlist) == 1); // load list type

// Add data to list: All the ports in scope
itr = vpi_iterate(vpiPort, scope);
while (var_handle = vpi_scan(itr)) {
   vpi_data_read_addtolist(loadlist, var_handle);
}
// Add data to list: All the regs in scope
itr = vpi_iterate(vpiReg, scope);
while (var_handle = vpi_scan(itr)) {
   vpi_data_read_addtolist(loadlist, var_handle);
}

// Initialize the read interface: Post process mode, read from a file,
// and focus only on the signals in the load list: loadlist
vpi_data_read_init(vpiDataReadAccessPostProcess,  datafile,  loadlist,
NULL, 0);

// Silly check to demo scanning the list, and checking if in list
while (var_handle = vpi_scan(loadlist)) {
   assert(vpi_data_read_isinloadlist(loadlist, var_handle));
}

// Load the data in one shot using load list
vpi_data_read_load(loadlist);

// Application code here
some_port = ...;
time_p = ...;
some_reg = ...;
....
vc_trvs_hdl1 = vpi_handle(vpiDataReadTrvsHndl, some_port);
vc_trvs_hdl2 = vpi_handle(vpiDataReadTrvsHndl, some_reg);
vpi_data_read_jump(vpiDataReadTrvsTime, some_port, p_vpi_time time_p);
```

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

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

The code segment above creates an object load list `loadlist` 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 `loadlist`. A sanity check is done by scanning the `loadlist` using `vpi_scan()` and testing that all the elements are in the list. The selected objects are then loaded in one shot using `vpi_data_read_load()` with loadlist 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.

29.5.7 Reader VPI routine definitions

This section describes the additional VPI routines in detail.

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

`vpi_data_read_init()`
Synopsis: Initialize the reader with access type and access scope.
Syntax: `vpi_data_read_init(vpiType prop, char* filename, vpiHandle loadlist, vpiHandle scope, PLI_INT32 level)`
Returns: `PLI_INT32`, 1 for success, 0 for fail
Arguments:
- `vpiType prop`:
  - `vpiDataReadAccess`: Access data in tool memory
  - `vpiDataReadAccessInteractive`: Access data interactively (tool keeps value history)
  - `vpiDataReadAccessPostProcess`: Access data stored in specified data file
- `char* filename`: Data file
- `vpiHandle loadlist:` Load list
- `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_data_read_get_time()`
Synopsis: Retrieve the time of the traverse handle.
Syntax: `vpi_data_read_get_time(vpiType prop, vpiHandle obj, p_vpi_time time_p)`
Returns: void
Arguments:
- `vpiType prop`:
  - `vpiDataReadTrvsMinTime`: Gets the minimum time of traverse handle
  - `vpiDataReadTrvsMaxTime`: Gets the maximum time of traverse handle
  - `vpiDataReadTrvsTime`: Gets the time where traverse handle points
- `vpiHandle obj`: Handle to an object
- `p_vpi_time time_p`: Pointer to a structure containing time information

Related routines: `vpi_get_time()`. Difference is that `vpi_data_read_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 handle.

vpi_data_read_jump()

Synopsis: Try to move value change traverse index to time, if there is no value change at time, then the value change traverse index is aligned based on the jump behavior defined earlier, and the time will be updated based on the aligned traverse index. For details on the success or fail return, refer to the jump behavior section. 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.

Syntax: `vpi_data_read_jump(vpiType prop, vpiHandle obj, p_vpi_time time_p)`

Returns: `PLI_INT32`, 1 for success, 0 for fail

Arguments:

- `vpiType prop`:
  - `vpiDataReadTrvsMinTime`: Goto the minimum time of traverse handle
  - `vpiDataReadTrvsMaxTime`: Goto the maximum time of traverse handle
  - `vpiDataReadTrvsTime`: Goto the time specified in `time_p`
- `vpiHandle obj`: Handle to an object
- `p_vpi_time time_p`: Pointer to a structure containing time information

Related routines: None

vpi_data_read_load()

Synopsis: Load the given object into memory for data access and traversal if object is an object handle, load the whole list if object is a load list of type `vpiDataReadLoadList`.

Syntax: `vpi_data_read_load(vpiHandle h)`

Returns: `PLI_INT32`, 1 for success, 0 for fail

Arguments:

- `vpiHandle h`: Handle to an object or list

Related routines: None

vpi_data_read_unload()

Synopsis: Unload the given object from memory if object is an object handle, load the whole list if object is a load list of type `vpiDataReadLoadList`.

Syntax: `vpi_data_read_unload(vpiHandle h)`

Returns: `PLI_INT32`, 1 for success, 0 for fail

Arguments:

- `vpiHandle h`: Handle to an object or list

Related routines: None

vpi_data_read_createloadlist()

Synopsis: Create a load list.

Syntax: `vpi_data_read_createloadlist()`

Returns: `vpiHandle` of type `vpiDataReadLoadList` for success, `NULL` for fail

Arguments: None

Related routines: None

vpi_data_read_addtoloadlist()

Synopsis: Add the given object onto the load list.

Syntax: `vpi_data_read_addtoloadlist(vpiHandle list, vpiHandle obj)`

Returns: `PLI_INT32`, 1 for success, 0 for fail

Arguments:

- `vpiHandle list`: Handle to load list
- `vpiHandle obj`: Handle to an object

Related routines: None

vpi_data_read_isinloadlist()

Synopsis: Check if the given object is in the load list.

Syntax: `vpi_data_read_isinloadlist(vpiHandle list, vpiHandle obj)`

Returns: `PLI_INT32`, 1 for success, 0 for fail
Arguments:
`vpiHandle list`: Handle to load list
`vpiHandle obj`: Handle to an object
Related routines: None

`vpi_data_read_resetloadlist()`
Synopsis: Reset a load list.
Syntax: `vpi_data_read_readloadlist(vpiHandle list)`
Returns: 1 for success, `0` for fail
Arguments:
`vpiHandle list`: Handle to load list
Related routines: None

29.6 Data Writer

A dump file may contain two kinds of information: design hierarchies, and value changes. The design hierarchies include the hierarchies between each scope (design entity) and the variables that each scope holds. The value changes are the "waveform" data. Each value change includes a time and a value. In order to write information into the dump file, writer shall support two kinds of creation modes:

— Tree creation mode, and

— Value Change (VC) creation mode.

Under tree creation mode, the application calls VPI writer APIs to create the design hierarchies, variables and their data. Under value change creation mode, the application creates the VC data. A handle for a variable can be used to write out the data. A variable may have more than one name i.e. an "alias." For example, a variable may be called "bus" in scope "top" and "data_bus" in scope "system." But both "bus" and "data_bus" refer to the same variable that has a unique handle. The application can create an alias to a variable by calling variable creation API with the different name but the same handle.

A dump file may contain none, one, or multiple design hierarchies. The design hierarchy is like a multi-tree. A node corresponds to a scope of the design hierarchy. A scope can contain scopes and variables. The capability that a scope can contain scopes recursively builds the design hierarchy. In order to traverse the design hierarchy, we must introduce the "current scope" concept: meaning "which node we are at now" in the writing process. Some of the tree creation APIs can move the "current scope" to another one so that the application can describe and build the design hierarchy. The "current scope" also determines which scope the newly created variables belong to: if a variable is created, then it belongs to the "current scope". The writer is built around a **time-based scheme**: the application stops at a specific time where value changes occurred, then it figures out what variables have value changes at that specific time. Finally, it creates the value changes of those variables at that specific time. The same steps repeat until it reaches the maximum time of the run. This creation scheme works quite naturally with the **cbValueChange** notification approach already built into VPI.

The value change creation is composed of 2 steps:

— Tag time creation

— Value creation at that tag

The time is created by calling `vpi_data_write_createtime()`, while the value is created by calling `vpi_data_write_createvalue()` API. The value change callback is the VPI **cbValueChange** simulation event reason.

29.6.1 Writer VPI routines definitions

`vpi_data_write_get_version()`
Synopsis: Get the writer version.
Syntax: `vpi_data_write_get_version()`

Returns: `char*`, for the version string
Arguments: None
Related routines: None

`vpi_data_write_open()`
Synopsis: Open file with a file name and a version.
Syntax: `vpi_data_write_open(char* fname, char* version)`
Returns: `vpiHandle` of type `vpiDataWriteFileType` (can be checked with `vpi_get()`) if successful, `NULL` otherwise
Arguments:
`char* fname`: Name of file
`char* version`: Version string
Related routines: None

`vpi_data_write_begintree()`
Synopsis: Begin tree creation.
Syntax: `vpi_data_write_begintree(vpiHandle data_write_obj)`
Returns: `PLI_INT32`, 1 for success, 0 for fail
Arguments:
`vpiHandle data_write_obj`: Handle to the write object
Related routines: `vpi_data_write_openfile()`

`vpi_data_write_setscaleunit()`
Synopsis: Set the scale unit.
Syntax: `vpi_data_write_setscaleunit(vpiHandle data_write_obj, char* scaleunit)`
Returns: `PLI_INT32`, 1 for success, 0 for fail
Arguments:
`vpiHandle data_write_obj`: Handle to the write object
`char* scaleunit`: Scale unit string e.g. 10ps.
Related routines: None

`vpi_data_write_createscope()`
Synopsis: Create a (sub)scope (under current scope).
Syntax: `vpi_data_write_createscope(vpiHandle data_write_obj, vpiType scopetype, char* name)`
Returns: `PLI_INT32`, 1 for success, 0 for fail
Arguments:
`vpiHandle data_write_obj`: Handle to the write object
`vpiType scopetype`: Type of the scope (e.g. module/task/...)
`char* name`: Name of scope
Related routines: None

`vpi_data_write_createupscope()`
Synopsis: Make current scope go to its parent scope.
Syntax: `vpi_data_write_createupscope(vpiHandle data_write_obj)`
Returns: `PLI_INT32`, 1 for success, 0 for fail
Arguments:
`vpiHandle data_write_obj`: Handle to the write object
Related routines: None

`vpi_data_write_createvar()`
Synopsis: Create a var in scope.
Syntax: `vpi_data_write_createvar(vpiHandle data_write_obj, vpiHandle obj)`

Returns: `PLI_INT32`, 1 for success, 0 for fail
Arguments:
`vpiHandle data_write_obj`: Handle to the write object
`vpiHandle obj`: Handle of the var (object)
Related routines: None

`vpi_data_write_endtree()`
Synopsis: This routine closes the tree creation (and the data creation portion can follow).
Syntax: `vpi_data_write_endtree(vpiHandle data_write_obj)`
Returns: `PLI_INT32`, 1 for success, 0 for fail
Arguments:
`vpiHandle data_write_obj`: Handle to the write object
Related routines: None

`vpi_data_write_createtime()`
Synopsis: Create a time where a value change occurs.
Syntax: `vpi_data_write_createtime(vpiHandle data_write_obj, p_vpi_time time_p)`
Returns: `PLI_INT32`, 1 for success, 0 for fail
Arguments:
`vpiHandle data_write_obj`: Handle to the write object
`p_vpi_time time_p`: Time point to create
Related routines: None

`vpi_data_write_createvalue()`
Synopsis: Create a value change.
Syntax: `vpi_data_write_createvalue(vpiHandle data_write_obj, p_vpi_value value_p)`
Returns: `PLI_INT32`, 1 for success, 0 for fail
Arguments:
`vpiHandle data_write_obj`: Handle to the write object
`p_vpi_value value_p`: Value to create
Related routines: None

`vpi_data_write_close()`
Synopsis: Close object (and file).
Syntax: `vpi_data_write_close(vpiHandle data_write_obj)`
Returns: `vpiHandle` if successful, NULL otherwise
Arguments:
`vpiHandle data_write_obj`: Handle to the write object
Related routines: None

29.6.2 Write scheme

A writer application needs to first create a tree (say after **cbEndofCompile** call) at a point when it is sure all the scopes have been created. It must also register a (value change) callback with the engine that is computing (e.g. a simulator) so that this write data routine can be called when a value change occurs (in the scope the application is interested in). So, with the tree created, the user (or tool) application can then start writing the data using the writer handle and the object (that has had a value change) handle. By also querying the time at which this value change occurred the application can write all the (time, value) pairs of the variables it is interested in.

The flow then is as follows:

1) Tree creation: Can be done at any point the application has control and wishes to write out its own dump file. Data can come from any tool that implements this VPI interface

   a) Open file and set version: This step returns a `vpiHandle` for the file object (i.e. the write target). This handle is not to be confused with a design object handle: It is a reference to a `vpiDataWriteType` object. This additional type serves to delieanate the usage of a `vpiHandle` for the writer portion from that of the design objects (used to interact with a simulator for example); the handles are not interchangeable. So, regardless of whether the tool or application that supports the writer portion has support for (or even knowledge of) the full VPI objects and methods, it shall use the VPI data structures related to the writer (in the same manner as the reader), in particular the writer API uses:

      i) `vpiHandle`: in lieu of a handle (i.e. pointer).

      ii) `PLI_INT32`: for the return type (i.e. `int`).

      iii) `p_vpi_time`: as the (pointer to) data structure for time representation.

      iv) `p_vpi_value`: as the (pointer to) data structure for value representation.

      v) All the object types and properties of the VPI accessible by `vpi_get_value(vpiType,<object_handle>)`

   f) Set scale unit

   g) Begin tree

   h) Create scope(s)

   i) Create var(s) inside the scopes

   j) End tree

   k) Register the value creation routine with a value change callback service

2) Value creation: Happens when the routine that does this is called

   a) Create a time point

   b) Create a value at this time point

The version matching insures that the writer stamps the file it generates with its version, so that only a reader with a matching version attempts to read such a data file.