

# System Verilog Assertion API

João Geada

## Change Log

Version	Date	Authors	Description
0.1	11/25/2002	Joao	First draft proposal for incorporating Synopsys assertions API donation into SV framework.
0.2	12/03/2002	João	Updated with comments/suggestions for sv-cc face to face meeting on 12/3/2002. Major changes: API changed to an extension of VPI, making all names consistent with naming scheme, additional static information APIs. Also synch-ed up terminology to current state of sv-ac (eg assertion → property)
0.3	1/8/2003	João	Updated VPI numeric assignments to fit within assigned range for assertions (700-799), minor updates as per last review, added “stepping” API as per discussions with Bassam. Also changed returned expressions from textual representation to vpiHandles to said expressions.

## Table of Contents

1	Requirements .....	2
1.1	Naming conventions .....	2
2	Extensions to VPI enumerations .....	2
3	Static Information .....	3
3.1	Obtaining assertion handles .....	3
3.2	Obtaining static assertion information .....	4
3.2.1	Additional static information .....	5
4	Dynamic Information .....	5
4.1	Placing assertion “system” callbacks .....	5
4.2	Placing assertions callbacks .....	6
5	Control Functions .....	7
5.1	Assertion System control .....	7
5.2	Assertion control .....	8

# 1 Requirements

To provide an API into the SV assertion capabilities providing sufficient capabilities to:

1. enable user's C code to react to assertion events
2. enable 3<sup>rd</sup> party assertion "waveform" dumping tools to be written
3. enable 3<sup>rd</sup> party assertion coverage tools to be written
4. enable 3<sup>rd</sup> party assertion debug tools to be written

The interface should also be readily extensible/adaptable so that it can easily be kept in sync with progress made by the sv-ac committee.

In addition, this interface should not unnecessarily duplicate any existing PLI/VPI interfaces.

## 1.1 Naming conventions<sup>1</sup>

All elements added by this interface will conform to the vpi interface naming conventions:

1. all names will be prefixed by vpi
2. type names will start with "vpi" followed by Capitalized words with no separators, eg vpiAssertCheck
3. all function names will start with "vpi\_" followed by all lowercase words separated by '\_', eg vpi\_get\_assert\_info()

## 2 Extensions to VPI enumerations<sup>2</sup>

1. Object types (reserve range 700-729 for types & properties):
  1. #define vpiProperty 700 /\* temporal assertion \*/
2. Object properties
  1. #define vpiAssertProperty 701
  2. #define vpiAssumeProperty 702
  3. #define vpiRestrictProperty 703
  4. #define vpiCoverProperty 704
  5. #define vpiCheckProperty 705 /\* (non-temporal) assertion \*/
3. Callbacks (reserve range 730-749 for callbacks)
  1. Property related
    - #define cbPropertyStart 730
    - #define cbPropertySuccess 731
    - #define cbPropertyFailure 732
    - #define cbPropertyStepSuccess 733
    - #define cbPropertyStepFailure 734
    - #define cbPropertyDisable 735
    - #define cbPropertyEnable 736
    - #define cbPropertyReset 737

<sup>1</sup> João comment: these rules are meant to be consistent with the naming conventions used by PLI and VPI interfaces

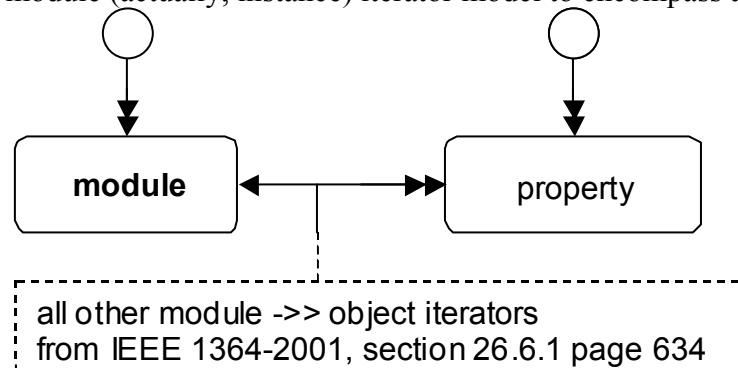
<sup>2</sup> To be merged to the contents of the "vpi\_user.h", described in 1364-2001, Annex G, pages 764-777  
The numbers in the range 700-799 will be reserved for the assertion portion of VPI

- #define cbPropertyKill 738
- 2. “Property System” related
  - #define cbPropertySysInitialized 739
  - #define cbPropertySysStart 740
  - #define cbPropertySysStop 741
  - #define cbPropertySysEnd 742
  - #define cbPropertySysReset 743
- 4. Control constants (reserve range 750-769)
  1. Property related
    - #define vpiPropertyDisable 750
    - #define vpiPropertyEnable 751
    - #define vpiPropertyReset 752
    - #define vpiPropertyKill 753
    - #define vpiPropertyEnableStep 754
    - #define vpiPropertyDisableStep 755
  2. Property stepping related
    - #define vpiPropertyClockSteps 756
  3. “Property System” related
    - #define vpiPropertySysStart 757
    - #define vpiPropertySysStop 758
    - #define vpiPropertySysEnd 759
    - #define vpiPropertySysReset 760

### 3 Static Information

#### 3.1 Obtaining assertion handles

Extend the VPI module (actually, instance) iterator model to encompass assertions.



1. Iterate over all properties in the design: use a NULL reference handle (ref) to vpi\_iterate()
 

```

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

- Iterate over all properties in an instance: pass appropriate instance handle as reference handle to `vpi_iterate()`

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

- Obtain assertion by name: extend `vpi_handle_by_name` to also search for assertion names in the appropriate scope(s)

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

**Note** that this operation only works for named assertions! Unnamed assertions cannot be found by name.

**NOTE:** as with all vpi handles, assertion handles are handles to a specific instance of a specific assertion.

**NOTE:** these iterators will return both temporal properties and immediate non-temporal checks.

### 3.2 Obtaining static assertion information

The following information about an assertion is considered to be “static”:

- Assertion name
- Instance in which the assertion occurs
- Module definition containing the assertion
- Assertion directive<sup>3</sup>:
  - assert
  - check
  - assume
  - cover
  - etc as necessary by assertion updates in sv-ac
- Assertion source information  
file, line and column where assertion defined
- Assertion clocking domain/expression<sup>4</sup>

Static information can be obtained directly from an `svaAssertID` without requiring any assertion attempts to be started or completed.

```
typedef struct t_vpi_source_info {
    PLI_BYTE* *fileName;
    PLI_INT32 startLine;
    PLI_INT32 startColumn;
    PLI_INT32 endLine;
    PLI_INT32 endColumn;
} s_vpi_source_info, *p_vpi_source_info;
typedef struct t_vpi_property_info {
    PLI_BYTE8 *name; /* name of property */
    vpiHandle instance; /* instance containing property */
    vpiHandle module; /* module containing assertion */
    vpiHandle clock; /* clocking expression */
    PLI_INT32 directive; /* vpiAssume, ... */
}
```

<sup>3</sup> Exact directives will have to be adjusted as per developments in the sv-ac committee

<sup>4</sup> Specific clocking domain info will have to be adjusted as per developments in the sv-ac committee

```

        s_vpi_source_info sourceInfo;
    } s_vpi_property_info, *p_vpi_property_info;
    int vpi_get_property_info(assert_handle, p_vpi_property_info);

```

This call is used to obtain all the static information associated with an assertion. **Note:** a single call returns all the information for efficiency reasons, as most clients will require most of the data; for efficiency one roundtrip through the API is better than multiple roundtrips. The inputs are a valid handle to an assertion and a pointer to an existing `s_vpi_property_info` datastructure. On success the function returns TRUE and the `s_vpi_property_info` datastructure will be filled in as appropriate. On failure, the function returns FALSE and the contents of the property info datastructure are unpredictable.

### 3.2.1 Additional static information

There are additional items that could be obtained statically from assertion, including:

1. Structure of assertion
2. Set of HDL variables used by assertion
3. Set of HDL expressions used by assertion

*No proposal for these items.*

## 4 Dynamic Information

### 4.1 Placing assertion “system” callbacks

Use `vpi_register_cb()`, setting the `cb_rtn` element to the function to be invoked and the `reason` element of the `s_cb_data` structure to one of the following:

1. `cbPropertySysInitialized`  
Occurs after system has initialized. No assertion specific actions can be performed until after this callback occurs. Note that the property system may initialize before or after `cbStartOfSimulation`
2. `cbPropertySysStart`  
Assertion system has become active and will begin processing property attempts. Will always occur after `cbPropertySysInitialized`. Note that by default the property system will be “started” on simulation startup, but it is possible for a user to delay this with the appropriate use of property system control actions.
3. `cbPropertySysStop`  
Assertion system has been temporarily suspended. While stopped no property attempts will be processed and no property related callbacks will occur. The property system may be stopped and resumed an arbitrary number of times during a single simulation run.
4. `cbPropertySysEnd`  
Occurs when all assertions have completed and no new attempts will start. Once this callback occurs no more property related callbacks will occur and property related actions will have no further effect. Typically occurs after the end of simulation.
5. `cbPropertySysReset`  
Occurs when the assertion system is reset, eg due to a system control action

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

## 4.2 *Placing assertions callbacks*

Use `vpi_register_property_cb()`<sup>5</sup>, whose prototype is as follows:

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

Where event is any of the following:

1. `cbPropertyStart`  
An assertion attempt has started. For most assertions one attempt will start each and every clock tick
2. `cbPropertySuccess`  
When an assertion attempt reaches a success state
3. `cbPropertyFailure`  
When an assertion attempt fails to reach a success state
4. `cbPropertyStepSuccess`  
progress of one “thread” along an attempt. Note that by default step callbacks are **not** enabled on any properties. Note also that step callbacks are enabled on a per-property/per-attempt basis, rather than on a per-property basis.
5. `cbPropertyStepFailure`  
failure to progress along one “thread” along an attempt. Same notes as per `cbPropertyStepSuccess`
6. `cbPropertyDisable`  
Whenever the assertion is disabled (eg as a result of a control action)

---

<sup>5</sup> **NOTE** can’t just use `vpi_register_cb` because the prototype of the callback function is different

7. `cbPropertyEnable`  
Whenever the assertion is enabled
8. `cbPropertyReset`  
Whenever the assertion is reset
9. `cbPropertyKill`  
When an attempt is killed (eg as a result of a control action)

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

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

The callback function will be supplied the following arguments:

- the event that caused the callback,
- the handle for the assertion,
- a pointer to an attempt info structure
- a reference to the user data supplied when the callback was placed.

The attempt info structure contains details relevant to the specific event that occurred:

- a. on disable, enable, reset and kill events the info field absent (a NULL pointer is given as the value of `info`)
- b. on start and success events, only the attempt time field is valid
- c. on a failure event, the attempt time and `detail.failExpr` are valid
- d. on a step callback, the attempt time and `detail.step` elements are valid.

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

Note that placing a step callback will result in the same callback function being invoked for both success and failure steps.

## 5 Control Functions

### 5.1 Assertion System control

Use `vpi_control()`, with one of the following operators and no other arguments:

1. `vpiPropertySysReset`  
discard all attempts in progress for all assertions and restore the entire assertion system to its initial state.

2. `vpiPropertySysStop`  
consider all attempts in progress as unterminated and disable any further assertions from being started.
3. `vpiPropertySysStart`  
restart the property system after it was stopped (eg due to `vpiPropertySysStop`). Once started, attempts will resume on all properties.
4. `vpiPropertySysEnd`  
discard all attempts in progress and disable any further assertions from starting.

## 5.2 Assertion control

Use `vpi_control()` with one of the following operators:

1. For all the following, the second argument must be a valid property handle.
  - a. `vpiPropertyReset`  
discards all current attempts in progress for this assertion and resets this assertion to its initial state
  - b. `vpiPropertyDisable`  
disables the starting of any new attempts for this assertion. Has no effect on any existing attempts. No effect if assertion already disabled. Note that by default all assertions are enabled.
  - c. `vpiPropertyEnable`  
Enables starting new attempts for this assertion. No effect if assertion already enabled. No effect on any existing attempts.
2. For all the following, the second argument must be a valid property handle and the third argument must be an attempt start time (as a pointer to a correctly initialized `s_vpi_time` structure)
  - a. `vpiPropertyKill`  
Discards the given attempts but leaves assertion enabled and does not reset any state used by this assertion (eg `past()` sampling)
  - b. `vpiPropertyDisableStep`  
Disables step callbacks for this assertion. No effect if stepping not enabled or already disabled
3. For the following, the second argument must be a valid property handle, the third argument must be an attempt start time (as a pointer to a correctly initialized `s_vpi_time` structure) and the fourth argument must be a “step control” constant. **Note** that in this release, the only step control constant available is `vpiPropertyClockSteps`, indicating callbacks on a per assertion/clock-tick basis.
  - a. `vpiPropertyEnableStep`  
Enables step callbacks to occur for this property attempt. Note that by default stepping is disabled for all assertions. This call has no effect if stepping already enabled for this property+attempt, other than possibly changing the stepping mode for the attempt iff the attempt has not occurred yet. The stepping mode of any particular attempt cannot be modified after the assertion attempt in question has started.