

Section 3

SystemVerilog Coverage API

3.1 Requirements

This chapter defines the Coverage Application Programming Interface (API) in SystemVerilog 3.1/draft 2.

3.1.1 SystemVerilog API

The following criteria are used within this API.

- 1) This API shall be similar for all coverages
There are a wide number of coverage types available, with possibly different sets offered by different vendors. Maintaining a common interface across all the different types enhances portability and ease of use.
- 2) At a minimum, the following types of coverage shall be supported:
 - a) statement coverage
 - b) toggle coverage
 - c) fsm coverage
 - i) fsm states
 - ii) fsm transitions
 - d) assertion coverage
- 3) Coverage APIs shall be extensible in a transparent manner, i.e., adding a new coverage type shall not break any existing coverage usage.
- 4) This API shall provide means to obtain coverage information from specific sub-hierarchies of the design without requiring the user to enumerate all instances in those hierarchies.

3.1.2 Naming conventions

All elements added by this interface shall conform to the Verilog Procedural Interface (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., `vpiCoverageStmt`.
- All callback names shall start with `cb`, followed by initially capitalized words with no separators, e.g., `cbAssertionStart`.
- All *function names* shall start with `vpi_`, followed by all lowercase words separated by underscores (`_`), e.g., `vpi_control()`.

3.1.3 Nomenclature

The following terms are used in this standard.

Statement coverage — whether a statement has been executed or not, where *statement* is anything defined as a statement in the LRM. *Covered* means it executed at least once. Some implementations also permit

querying the execution count. The granularity of statement coverage can be per-statement or per-statement block (however defined).

FSM coverage — the number of states in a finite state machine (FSM) that this simulation reached. This standard does not require FSM automatic extraction, but a standard mechanism to force specific extraction is available via pragmas.

Toggle coverage — for each bit of every signal (wire and register), whether that bit has both a 0 value and a 1 value. *Full coverage* means both are seen; otherwise, some implementations can query for *partial coverage*. Some implementations also permit querying the toggle count of each bit.

Assertion coverage — for each assertion, whether it has had at least one success. Implementations permit querying for further details, such as attempt counts, success counts, failure counts and failure coverage.

These terms define the “primitives” for each coverage type. Over instances or blocks, the coverage number is merely the sum of all contained primitives in that instance or block.

3.2 SystemVerilog real-time coverage access

| This section ...

3.2.1 Predefined coverage constants in SystemVerilog

The following predefine ``defines` represent basic real-time coverage capabilities accessible directly from SystemVerilog.

— Coverage control

```
`define SV_COV_START      0
`define SV_COV_STOP       1
`define SV_COV_RESET      2
`define SV_COV_QUERY      3
```

— Scope definition (hierarchy traversal/accumulation type)

```
`define SV_COV_MODULE     10
`define SV_COV_HIER       11
```

— Coverage type identification

```
`define SV_COV_ASSERTION  20
`define SV_COV_FSM_STATE  21
`define SV_COV_STATEMENT  22
`define SV_COV_TOGGLE     23
```

— Status results

```
`define SV_COV_OVERFLOW   -2
`define SV_COV_ERROR      -1
`define SV_COV_NOCOV      0
`define SV_COV_OK         1
`define SV_COV_PARTIAL    2
```

3.2.2 Built-in coverage access system functions

| This section ...

3.2.2.1 \$coverage_control

```
$coverage_control(control_constant,
                  coverage_type,
                  scope_def,
                  modules_or_instance)
```

This function enables, disables, resets or queries the availability of coverage information for the specified portion of the hierarchy. The return value is a `defined name, with the value indicating the success of the action.

`SV_COV_OK

the request is successful. When querying, if starting, stopping, or resetting this means the desired effect occurred, coverage is available. A successful reset clears all coverage (i.e., using a ...get() == 0 after a successful ...reset()).

`SV_COV_ERROR

the call failed with no action, typically due to errors in the arguments, such as a non-existing module or instance specifications.

`SV_COV_NOCOV

coverage is not available for the requested portion of the hierarchy.

`SV_COV_PARTIAL

coverage is only partially available in the requested portion of the hierarchy (i.e., some instances have the requested coverage information, some don't).

Starting, stopping, or resetting coverage multiple times in succession for the same instance(s) has no further effect if coverage has already been started, stopped, or reset for that/those instance(s).

The hierarchy(ies) being controlled or queried are specified as follows.

`SV_MODULE_COV, "unique module def name"

provides coverage of all instances of the given module (the unique module name is a string), excluding any child instances in the instances of the given module. The module definition name can use special notation to describe nested module definitions.

`SV_COV_HIER, "module name"

provides coverage of all instances of the given module, including all the hierarchy below.

`SV_MODULE_COV, instance_name

provides coverage of the one named instance. The instance is specified as a normal Verilog hierarchical path.

`SV_COV_HIER, instance_name

provides coverage of the named instance, plus all the hierarchy below it.

All the permutations are summarized in Table 3-1 on page 23.

****Revise this xref w/ Stu; also check/revise variable settings, etc.****

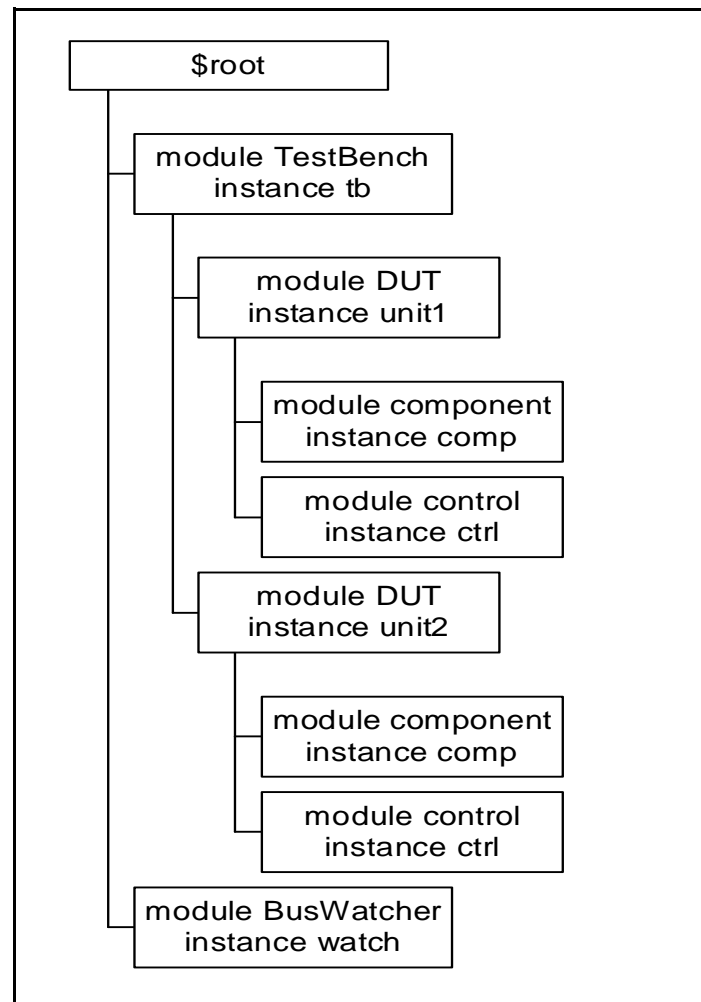
Table 3-1: Instance coverage permutations

Control/query	"Definition name"	instance.name
`SV_COV_MODULE	The sum of coverage for all instances of the named module, excluding any hierarchy below those instances.	Coverage for just the named instance, excluding any hierarchy in instances below that instance.

Table 3-1: Instance coverage permutations (continued)

Control/query	“Definition name”	instance.name
<code>`SV_COV_HIER</code>	The sum of coverage for all instances of the named module, including all coverage for all hierarchy below those instances.	Coverage for the named instance and any hierarchy below it.

NOTE—Definition names are represented as strings, whereas instance names are referenced by hierarchical paths. A hierarchical path need not include any `.` if the path refers to an instance in the current context (i.e., normal Verilog hierarchical path rules apply).

*Example 3-1 Hierarchical instance example*

If coverage is enabled on all instances shown in Example 3-1, then:

```
$coverage_control(`SV_COV_CHECK, `SV_COV_TOGGLE, `SV_COV_HIER, $root)
```

checks all instances to verify they have coverage and, in this case, returns ``SV_COV_OK`.

```
$coverage_control(`SV_COV_RESET, `SV_COV_TOGGLE, `SV_COV_MODULE,
                  "DUT")
```

resets coverage collection on both instances of the DUT, specifically, `$root.tb.unit1` and `$root.tb.unit2`, but leaves coverage unaffected in all other instances.

```
$coverage_control(`SV_COV_RESET, `SV_COV_TOGGLE, `SV_COV_MODULE,
                  $root.tb.unit1)
```

resets coverage of only the instance `$root.tb.unit1`, leaving all other instances unaffected.

```
$coverage_control(`SV_COV_STOP, `SV_COV_TOGGLE, `SV_COV_HIER,
                  $root.tb.unit1)
```

resets coverage of the instance `$root.tb.unit1` and also reset coverage for all instances below it, specifically `$root.tb.unit1.comp` and `$root.tb.unit1.ctrl`.

```
$coverage_control(`SV_COV_START, `SV_COV_TOGGLE, `SV_COV_HIER, "DUT")
```

starts coverage on all instances of the module DUT and of all hierarchy(ies) below those instances. In this design, coverage is started for the instances `$root.tb.unit1`, `$root.tb.unit1.comp`, `$root.tb.unit1.ctrl`, `$root.tb.unit2`, `$root.tb.unit2.comp`, and `$root.tb.unit2.ctrl`.

3.2.2.2 \$coverage_get_max

```
$coverage_get_max(coverage_type, scope_def, modules_or_instance)
```

This function obtains the value representing 100% coverage for the specified coverage type over the specified portion of the hierarchy. This value shall remain constant across the duration of the simulation.

NOTE—This value is proportional to the design size and structure, so it also needs to be constant through multiple independent simulations and compilations of the same design, assuming any compilation options do not modify the coverage support or design structure.

The return value is an integer, with the following meanings.

-2 (``SV_COV_OVERFLOW`)

the value exceeds a number that can be represented as an integer.

-1 (``SV_COV_ERROR`)

an error occurred (typically due to using incorrect arguments).

0 (``SV_COV_NOCOV`)

no coverage is available for that coverage type on that/those hierarchy(ies).

+*pos_num*

the maximum coverage number (where *pos_num* > 0), which is the sum of all coverable items of that type over the given hierarchy(ies).

The scope of this function is specified as per `$coverage_control` (see section 3.2.2.1).

3.2.2.3 \$coverage_get

```
$coverage_get(coverage_type, scope_def, modules_or_instance)
```

This function obtains the current coverage value for the given coverage type over the given portion of the hierarchy. This number can be converted to a coverage percentage by use of the equation:

$$\text{coverage\%} = \frac{\text{coverage_get}()}{\text{coverage_get_max}()} * 100$$

The return value follows the same pattern as `$coverage_get_max` (see section 3.2.2.2), but with the *pos_num* number representing the current coverage level, i.e., the number of the coverable items that have been covered in this/these hierarchy(ies).

The scope of this function is specified as per `$coverage_control` (see section 3.2.2.1).

The return value is an integer, with the following meanings.

-2 (`SV_COV_OVERFLOW)

the value exceeds a number that can be represented as an integer.

-1 (`SV_COV_ERROR)

an error occurred (typically due to using incorrect arguments).

0 (`SV_COV_NOCOV)

no coverage is available for that coverage type on that/those hierarchy(ies).

+*pos_num*

the maximum coverage number (where *pos_num* > 0), which is the sum of all coverable items of that type over the given hierarchy(ies).

3.2.2.4 \$coverage_merge

```
$coverage_merge(coverage_type, "name")
```

This function loads and merges coverage data for the specified coverage into the simulator. *name* is an arbitrary string used by the tool, in an *implementation-specific* way, to locate the appropriate coverage database, i.e., tools are allowed to store coverage files any place they want with any extension they want *as long* as the user can retrieve the information by asking for a specific saved name from that coverage database. If *name* does not exist or does not correspond to a coverage database from the same design, an error shall occur. If an error occurs during loading, the coverage numbers generated by this simulation might not be meaningful.

The return values from this function are:

`SV_COV_OK

the coverage data has been found and merged.

`SV_COV_NOCOV

the coverage data has been found, but did not contain the coverage type requested.

`SV_COV_ERROR

the coverage data was not found or it did not correspond to this design, or another error.

3.2.2.5 \$coverage_save

```
$coverage_save(coverage_type, "name")
```

This function saves the current state of coverage to the tool's coverage database and associates it with the file named *name*. This file name shall not contain any directory specification or extensions. Data saved to the database shall be retrieved later by using `$coverage_merge` and supplying the same name. Saving coverage shall not have any effect on the state of coverage in this simulation.

The return values from this function are:

`SV_COV_OK

the coverage data was successfully saved.

`SV_COV_NOCOV

no such coverage is available in this design (nothing was saved).

`SV_COV_ERROR

some error occurred during the save. If an error occurs, the tool shall automatically remove the coverage database entry for *name* to preserve the coverage database integrity. It is *not* an error to overwrite a previously existing *name*.

NOTES

1—The coverage database format is implementation-dependent.

2—Mapping of names to actual directories/files is implementation-dependent. There is no requirement that a coverage name map to any specific set of files or directories.

3.3 FSM recognition

Coverage tools need to have automatic recognition of many of the common FSM coding idioms in Verilog/SystemVerilog. This standard does not attempt to describe or require any specific automatic FSM recognition mechanisms. However, the standard does prescribe a means by which non-automatic FSM extraction occurs. The presence of any of these standard FSM description additions shall override the tool's default extraction mechanism.

Identification of an FSM consists of identifying the following items:

- 1) the state register (or expression)
- 2) the next state register (this is optional)
- 3) the legal states.

3.3.1 Specifying the signal that holds the current state

****This section reads a bit like a user's guide; convert this into an annex??**

Use the following pragma to identify the vector signal that holds the current state of the FSM:

```
/* tool state_vector signal_name */  
  
**let's define these terms (in the next draft)**
```

where `tool` and `state_vector` are required keywords. This pragma needs to be specified inside the module definition where the signal is declared.

Another pragma is also required, to specify an enumeration name for the FSM. This enumeration name is also specified for the next state and any possible states, associating them with each other as part of the same FSM. There are two ways to do this:

— Use the same pragma:

```
/* tool state_vector signal_name enum enumeration_name */
```

— Use a separate pragma in the signal's declaration:

```
/* tool state_vector signal_name */  
reg [7:0] /* tool enum enumeration_name */ signal_name;
```

In either case, `enum` is a required keyword; if using a separate pragma, `tool` is also a required keyword and the pragma needs to be specified immediately after the bit-range of the signal.

3.3.2 Specifying the part-select that holds the current state

A part-select of a vector signal can be used to hold the current state of the FSM. When `cmView` displays or reports FSM coverage data, it names the FSM after the signal that holds the current state. If a part-select holds the current state in the user's FSM, the user needs to also specify a name for the FSM that `cmView` can use. The FSM name is not the same as the enumeration name.

Specify the part-select by using the following pragma:

```
/* tool state_vector signal_name[n:n] FSM_name enum enumeration_name */
```

3.3.3 Specifying the concatenation that holds the current state

Like specifying a part-select, a concatenation of signals can be specified to hold the current state (when including an FSM name and an enumeration name):

```
/* tool state_vector {signal_name , signal_name, ...} FSM_name enum
   enumeration_name */
```

The concatenation is composed of all the signals specified. Bit-selects or part-selects of signals cannot be used in the concatenation.

3.3.4 Specifying the signal that holds the next state

The signal that holds the next state of the FSM can also be specified with the pragma that specifies the enumeration name:

```
reg [7:0] /* tool enum enumeration_name */
   signal_name
```

This pragma can be omitted if, and only if, the FSM does not have a signal for the next state.

3.3.5 Specifying the current and next state signals in the same declaration

The tool assumes the first signal following the pragma holds the current state and the next signal holds the next state when a pragma is used for specifying the enumeration name in a declaration of multiple signals, e.g.,

```
/* tool state_vector cs */
reg [1:0] /* tool enum myFSM */ cs, ns, nonstate;
```

In this example, the tool assumes signal `cs` holds the current state and signal `ns` holds the next state. It assumes nothing about signal `nonstate`.

3.3.6 Specifying the possible states of the FSM

The possible states of the FSM can also be specified with a pragma that includes the enumeration name:

```
parameter /* tool enum enumeration_name */
   S0 = 0,
   s1 = 1,
   s2 = 2,
   s3 = 3;
```

Put this pragma immediately after the keyword `parameter`, unless a bit-width for the parameters is used, in which case, specify the pragma immediately after the bit-width:

```
parameter [1:0] /* tool enum enumeration_name */
   S0 = 0,
   s1 = 1,
   s2 = 2,
   s3 = 3;
```

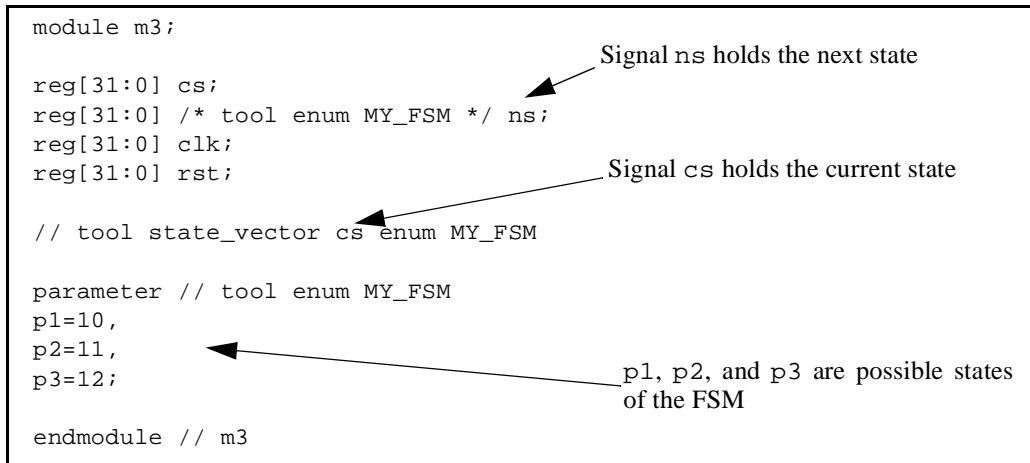
3.3.7 Pragmas in one-line comments

These pragmas work in both block comments, between the `/*` and `*/` character strings, and one-line comments, following the `//` character string, e.g.,

```
parameter [1:0] // tool enum enumeration_name
   S0 = 0,
   s1 = 1,
```

```
s2 = 2,  
s3 = 3;
```

3.3.8 Example



Example 3-2FSM specified with pragmas

3.4 VPI coverage extensions

| This section ...

3.4.1 VPI entity/relation diagrams related to coverage

| This section ...

3.4.2 Extensions to VPI enumerations

— Coverage control

```
#define vpiCoverageStart  
#define vpiCoverageStop  
#define vpiCoverageReset  
#define vpiCoverageCheck  
#define vpiCoverageMerge  
#define vpiCoverageSave
```

— VPI properties

1) Coverage type properties

```
#define vpiAssertCoverage  
#define vpiFsmStateCoverage  
#define vpiStatementCoverage  
#define vpiToggleCoverage
```

2) Coverage status properties

```
#define vpiCovered  
#define vpiCoverMax  
#define vpiCoveredCount
```

3) Assertion-specific coverage status properties

```
#define vpiAssertAttemptCovered
#define vpiAssertSuccessCovered
#define vpiAssertFailureCovered
```

4) FSM-specific methods

```
#define vpiFsmStates
#define vpiFsmStateExpression
```

— FSM handle types (vpi types)

```
#define vpiFsm
#define vpiFsmHandle
```

3.4.3 Obtaining coverage information

■ All ****what??** use `vpi_get ()` along with the appropriate properties and object handles.

coverage type, instance

the number of covered items in the given instance.

vpiCovered, handle

the number of items of the handle type is covered. This is only applicable to: statement handles, signal (wire/reg) handles, assertion handles, and FSM handles.

vpiCoveredCount, handle

the number of times each item of the handle type is covered. This is only easily interpretable when *handle* points to a unique coverable item (otherwise this is the sum of counts of all contained items).

vpiCoveredMax, handle

the total possible coverable items in the given handle. Handle types limited as per above. `vpiCoveredMax` is only really useful when *handle* is a handle to an object potentially containing more than one coverable item.

- Use `vpi_iterate(vpiFsm, instance-handle)` to get the iterator to all FSMs in an instance.
- Use `vpi_handle(vpiFsmStateExpression, fsm-handle)` to get the handle to the signal or expression encoding the FSM state.
- Use `vpi_iterate(vpiFsmStates, fsm-handle)` to get the iterator to all states of an FSM.
- Use `vpi_get_value(fsm_state_handle, state-handle)` to get the value of a state.

3.4.4 Controlling coverage

■ ****Revise similar to Assertions****

`vpi_control ()`

has three arguments: *coverage control* (start, stop, reset, query), *coverage type*, and the *handle* to the appropriate instance or assertion. Statement, toggle, and FSM coverage are not individually controllable (i.e., they are controllable only at the instance level, not on a per statement/signal/FSM). The semantics and behavior are specified as per the equivalent system function `$coverage_control` (see section 3.2.2.1).

`vpi_control ()`

has three arguments: *coverage control* (merge, save), *coverage type*, and *name*. This merges coverage into the current simulation. The semantics and behavior are specified as per the equivalent system functions `$coverage_merge` (see section 3.2.2.4) and `$coverage_save` (see section 3.2.2.5).