cadence™

SystemVerilog Reference

Product Version 8.1
May 2008

Printed in the United States of America.

Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA

Contents

1
Overview of SystemVerilog 13
Availability of Constructs within Simulators 13
SystemVerilog in Simulation 13
SystemVerilog VPI Extensions 13
SystemVerilog Assertions 14
SystemVerilog Coverage 14
SystemVerilog with AMS 14
SystemVerilog Examples 14
Language Support 15
Getting Help 15
About Online Help 15
Getting Help on Commands to Run Tools 17
Getting Help on Tool Messages 18
Other Documentation 18
Customer Support 18

2
Compiling SystemVerilog Constructs 21
Using ncvlog 21
Using the irun Utility 21
SystemVerilog and the PLI tf_nodeinfo() Interface 22

3
List of Supported Constructs 23

4
Convenience Enhancements 29
Literal Value Assignments 29

Matching End Names 29
Time Unit and Time Precision 31
.name Implicit Port Connection 33
Dot Star (.*) Implicit Port Connection 34

5
Data Types 37

Data Types Overview 37
Overview of Verilog Data Types 38
Primitive Data Types 38
User-Defined Data Types 39
logic Data Type 40
bit Data Type 41
byte, shortint, int, longint Data Types 42
Chandle Data Type 43
Strings 43
String Operators 44
String Methods 45
Strings and System Tasks 47
Using Strings with Classes 48
Using Strings with Packages 49
Using Strings within begin...end Blocks 49
Declaring a Fixed Array of Strings 50
Declaring Arrays and Queues of Strings 51
Using Elements of a Dynamic Array of Strings 52
Using Strings as Parameters and localparams 52
Limitations on Strings 54
typedef Declaration 55
Limitations 55
Creating a New Data Type Definition 55
Handling Data Type Visibility 58
enum Data Type 59
Limitations 59
Declaring an Enumeration 59
Specifying Enumeration Constants 60

Treating Enumeration Objects as Bit Vectors . . . 61
Enumeration Type Checking . . . 62
Enumeration Type Methods . . . 63
Structures . . . 65
Packed Structures . . . 66
Unpacked Structures . . . 67
Debugging Structures . . . 71
uwire Nets . . . 71
Casting . . . 73
Limitations . . . 74

6
Arrays . . . 75
Packed and Unpacked Arrays . . . 76
Limitations on Packed and Unpacked Arrays . . . 78
Array Querying Functions . . . 79
Dynamic Arrays . . . 81
Access Methods for Dynamic Arrays . . . 81
Accessing Out-of-Bound Elements of a Dynamic Array . . . 83
Passing Dynamic Arrays by Reference to Tasks and Functions . . . 83
Fixed Arrays of Dynamic Arrays . . . 84
Limitations on Dynamic Arrays . . . 84
Associative Arrays . . . 87
Access Methods for Associative Arrays . . . 88
Limitations for Associative Arrays . . . 90
Queues . . . 92
Access Methods for Queues . . . 93
Limitations for Queues . . . 94
Array Locator Methods . . . 97
Limitations on Array Locator Methods . . . 98
Debugging Queues and Arrays . . . 99

7
Data Declarations . . . 101
Declaring Variables with Initializers . . . 101

Declaring Local Variables in Unnamed Blocks . . . 101
Continuous Assignments to Variables . . . 102
Restrictions on Continuous Assignments to Variables . . . 103
Limitations in the Current Release . . . 105

8
Classes . . . 107
Declaring a Class Data Type . . . 107
Working with Constructors . . . 109
Inheritance . . . 110
Protecting Class Members . . . 111
Additional Features . . . 112
Limitations . . . 113
Debugging Classes . . . 114

9
Operators and Expressions . . . 115
Supported Operators . . . 115
Assignment Operators . . . 115
Wild Equality and Wild Inequality Operators . . . 116
Assignment Patterns . . . 116
Limitations . . . 118
Aggregate Expressions . . . 119

10
Procedural Statements . . . 121
Unique and Priority Decision Statements . . . 121
do...while Loop . . . 124
for Loop . . . 124
foreach Loop . . . 125
return, break, continue Jump Statements . . . 126
final Blocks . . . 127
iff Event Control Qualifier . . . 128
always_* Blocks . . . 128

fork...join . . . 129
fork...join . . . 129
fork...join_none . . . 129
fork...join_any . . . 131
wait fork . . . 132
disable fork . . . 132

11
Tasks and Functions . . . 135
Multiple Statements in Tasks and Functions . . . 135
Function Output Arguments . . . 136
Default Direction in Task and Function Declarations . . . 136
Void Functions . . . 137
Discarding Function Return Values . . . 137
Passing Task and Function Arguments by Reference . . . 138
Limitations for Passing Task and Function Arguments by Reference . . . 140
Specifying Default Argument Values for Tasks and Functions . . . 141
Limitations for Default Argument Values . . . 142
Passing Task and Function Arguments by Name . . . 142
Optional Arguments for Tasks and Functions . . . 143

12
Random Constraints . . . 145
Random Variables . . . 146
Limitations on Random Variables . . . 147
Constraint Blocks . . . 148
Limitations on Constraint Blocks . . . 148
External Constraint Blocks . . . 148
Inheritance . . . 149
Set Membership . . . 149
Distribution . . . 149
Implication . . . 151
if...else Constraints . . . 152
Iterative Constraints . . . 153
Global Constraints . . . 153

solve...before Constraints . . . 154
Static Constraint Blocks . . . 154
Functions in Constraints . . . 155
Randomization Methods . . . 155
The randomize() Method . . . 155
pre_randomize() and post_randomize() . . . 157
In-Line Constraints (randomize() with) . . . 157
Activating and Inactivating Random Variables with rand_mode() . . . 157
Limitations . . . 158
Activating and Inactivating Constraints with constraint_mode() . . . 158
In-Line Random Variable Control . . . 159
Randomization of Scope Variables (std::randomize()) . . . 160
Specifying Constraints . . . 162
Limitations . . . 164
Random Number System Functions and Methods . . . 164
The $urandom Function . . . 165
The $urandom_range Function . . . 166
The srandom() Method . . . 166
Additional System Functions and Methods . . . 167
Random Stability . . . 167
Random Weighted Case (randcase) . . . 167
Random Sequence Generator (randsequence) . . . 169
Declaring a randsequence Block . . . 169
if...else Production Statements . . . 170
Case Production Statements . . . 170
Repeat Production Statements . . . 171
Limitations . . . 172
Debugging Random Constraints . . . 172

13
Interprocess Synchronization and Communication . . . 173

Semaphores . . . 173
Limitations on Semaphores . . . 175
Mailboxes . . . 175
Mailbox Methods . . . 176

Limitations on Mailboxes . . . 178
Events . . . 179
Non-Blocking Event Trigger . . . 179
Persistent Trigger . . . 181
Event Variables . . . 182

14
Clocking Blocks . . . 185
Declaring a Clocking Block . . . 185
Defining Default Skews and Clocking Direction . . . 187
Defining Clocking Items . . . 188
Using Hierarchical Expressions . . . 188
Defining Default Clocking Blocks . . . 189
Specifying Cycle Delays and Clocking Drives . . . 190
Debugging Clocking Blocks . . . 191

15
Program Blocks . . . 193
Declaring a Program Block . . . 193
Supported Constructs for Program Blocks . . . 194
Unsupported Constructs . . . 194
Nesting Program Blocks . . . 195
Working with Variable Assignments . . . 196
Referencing Program Block Variables . . . 196
Instantiating Program Blocks . . . 197
New Program Design Unit . . . 197
Understanding the $exit() Control Task . . . 197

16
Assertions . . . 201
Immediate Assertions . . . 201
Concurrent Assertions . . . 201

17
Hierarchy 203
Packages 203
Declaring a Package 205
Referencing Data in a Package 206
Controlling Visibility of Names within Packages: The import Statement 206
Debugging Packages 209
Compilation Units 210
Supported External Declarations 211
Limitations on Compilation Units 212
Explicitly Referencing External Declarations 212
Port Declarations 212

18
Interfaces 215
Declaring an Interface 216
Creating Design Units 219
Using the Interface as a Module Port 219
Limitations on Interfaces 220
Interface Array Ports 221
Supported Uses for Interface Array Ports 221
Using Arrays of Interfaces in Interface Array Ports 222
Limitations on Interface Array Ports 224
Referencing an Interface 224
Working with Modports 225
Defining a Modport 228
Selecting Which Modport to Use 228
Limitations on modports 229
Declaring Tasks and Functions in Interfaces 230
Virtual Interfaces 230
Limitations on Virtual Interfaces 234
Working with Interfaces and Timing 234

19
System Functions 235
Out-of-Module Reference ($root) 235
Expression Size System Function ($bits) 236

20
Compiler Directives 239
`define 239
`begin_keywords and `end_keywords 240
Limitations 242
Reserved Keywords for IEEE 1800 243
`remove_keyword and `restore_keyword 245
ncvlog -rmkeyword 245
`remove_keyword and `restore_keyword Compiler Directives 246
Limitations 247

21
Direct Programming Interface 249
Importing Functions and Tasks using DPI 249
pure and context Properties 250
Importing C Functions and Tasks 251
Importing SystemC Functions and Tasks 255
Exporting SystemVerilog Functions and Tasks using DPI 258
Exporting Functions and Tasks to C 259
Exporting SystemVerilog Functions and Tasks to SystemC 262
Using typedef with SystemC Data Types 265
Tasks That Consume Time 266
Using DPI with the Incisive Simulator 269
Using the irun Utility with DPI 269
Using NC-Verilog with DPI 271
Disabling DPI Tasks and Functions 275
Debugging DPI Import and Export Functions 276
DPI Accessor Functions 277
DPI Examples 278

Using DPI with C 279
Using DPI with SystemC 281
Using scSetScopeByName in SystemVerilog 282
Unpacked Structs as Formal Arguments to DPI-C Import Functions 283
Unpacked Structs as Formal Arguments to DPI-SC Import Functions 284

Index 287

1

Overview of SystemVerilog

The IEEE 1800 standard for SystemVerilog documents a large set of extensions to the existing IEEE Verilog-2001 standard. This set of enhancements provides new capabilities for modeling hardware at the RTL and system level, along with a powerful set of new features for verifying model functionality.

This reference guide tells you how to enable the SystemVerilog constructs and describes the constructs in the IEEE 1800 standard that are supported by the NC-Verilog Simulator. For information on the NC-Verilog simulator, refer to the *Cadence NC-Verilog Simulator Help.*

Availability of Constructs within Simulators

Cadence offers several simulators. All of the features described in this book are supported within the NC-Verilog Simulator, Incisive Simulator, the Incisive Design Team Simulator, and the Incisive Enterprise Simulator. However, some of the constructs described in this document are not available with the Incisive HDL Simulator. These constructs include: classes, semaphores, program blocks, and clocking blocks. For more information on the Incisive HDL Simulator, refer to the *Design Team Family Technology Overview*.

SystemVerilog in Simulation

For information on how to simulate a design that contains SystemVerilog constructs, including information on how to view and debug SystemVerilog constructs using Tcl or SimVision, refer to *SystemVerilog in Simulation*.

SystemVerilog VPI Extensions

Because the SystemVerilog VPI standard is still evolving, Cadence does not support SystemVerilog VPI extensions in the current release.

Existing user and third-party PLI and VPI applications that already work for designs without SystemVerilog language extensions will continue to work on those designs. However, these existing applications may fail if applied to designs that contain SystemVerilog constructs.

SystemVerilog Assertions

Note: SystemVerilog assertions are available only if you have an Incisive license.

Support for SystemVerilog assertions is documented *Assertion Writing Guide* and in the *SVA Quick Reference Guide*.

SystemVerilog Coverage

Note: SystemVerilog coverage is available only if you have an Incisive license.

Support for SystemVerilog coverage is documented in the “Functional Coverage” chapter of the *ICC User Guide*.

SystemVerilog with AMS

You can simulate a design that contains both AMS and SystemVerilog code, but with the following limitations:

- If an AMS scope is instantiated inside a SystemVerilog scope, it cannot have any non-digital ports.
- If a SystemVerilog scope is instantiated inside an AMS scope, it cannot have non-digital objects connected to its ports.

You can use the multi-step invocation mode (`ncvlog`, `ncelab`, `ncsim`) to simulate mixed AMS/SystemVerilog designs by using separate invocations of `ncvlog` with the appropriate option and then compiling the SystemVerilog and AMS portions of the design.

SystemVerilog Examples

This document contains small examples for each of the supported SystemVerilog constructs. For complete examples, refer to the following:

- *SystemVerilog Engineering Notebook*—Documents examples for various SystemVerilog constructs. You can download the examples and run them using the NC-Verilog simulator.

- *SystemVerilog DPI Engineering Notebook*—Documents examples for SystemVerilog DPI. You can download the examples and run them using the NC-Verilog simulator.
- *Examples Reference Guide*—Lists the examples located within your installation.

Language Support

This document refers to the following terms:

- *Verilog* or *Verilog-2001*—Refers to the IEEE Std. 1364-2001 standard for the Verilog Hardware Description Language
- *IEEE 1800*—Refers to the IEEE 1800 Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language.

Getting Help

This section describes:

- About Online Help on page 15
- Getting Help on Commands to Run Tools on page 17
- Getting Help on Tool Messages on page 18
- Other Documentation on page 18

About Online Help

Documentation for SystemVerilog is provided in HTML and in PDF format.

The online documentation system consists of:

- The Cadence documentation window.

  This window lets you find and open any of the books shipped with the products that you ordered. You can list books by product name, by product family, or by document type. For example, you can list all manuals, all Product Notes documents, or all Known Problems and Solutions documents. When you select a document, it is opened in your Web browser. The system automatically starts the browser, if necessary.
- Manuals in HTML format.

Each HTML document has both hyperlinked cross-references and a toolbar with buttons that let you navigate through the documentation system. Using the buttons on the toolbar, you can redisplay the documentation window, move forward and backward through chapters, display the Table of Contents, open a PDF file for printing, or open the search page. You can also send an e-mail directly to Cadence publications with comments about the document that you are viewing.

- A PDF (Portable Document Format) file for each document so that you can print the entire document or sections of a document.
- A powerful search tool that lets you search for information in documents for a product family or for specific products. You can also search individual documents.

Click the *Help* button on the toolbar of any document to open the *Cadence Documentation User Guide*, which contains a complete description of how to use the online documentation system.

Invoking the Documentation System

There are two ways to open the online help for the simulator:

- From the Cadence documentation window.

  To invoke the Cadence documentation window on UNIX or Linux, use the Cadence Help command.

  ```
  % cdnshelp
  ```

  On the Cadence documentation window, click on a category name to show the documents in that category. Then double-click a manual title to load that manual into your web browser.

- From the Help menu on the graphical user interface.

  If you are using a graphical user interface, such as NCLaunch or the SimVision analysis environment, pull down the *Help* menu and select the name of the online manual that you want to view. For example, if you are simulating a Verilog design, select *NC-Verilog Help* to open the online help for the NC-Verilog simulator.

  Help can also be accessed from forms on the graphical interface. Click on the *Help* button on the bottom right of the form to get online help.

  To open the Cadence documentation window, click the Library button at the top of any document displayed in the browser.

Printing Documents From the Online Documentation System

The *View/Print PDF* button opens a PDF file in Adobe Acrobat® Reader. You can print the entire document or print a section of the document by specifying a range of page numbers.

A hyperlinked list of contents (a "bookmark" list) is available for navigating the print document online. Hyperlinked cross-references in the HTML version are reformatted to include page numbers in the printed copy, so that you can find the referenced page easily when reading the printed version.

Searching Documents

The built-in search mechanism lets you search various groupings of books or individual books. You can:

- Search all installed documents.
- Search all documents for specific products or product families.
- Search one or more specific books.

The Search tool lets you perform many different types of full-text search queries. You can search for text phrases or exact words, use Boolean AND, OR, or NOT operators, use special operators such as CASE (for case-sensitive searches) or NEAR (to search for words near each other), or use wildcard characters for substitution.

Getting Help on Commands to Run Tools

You can display a list of options for any of the simulator tools and utilities by typing the tool or utility name followed by the `-help` option.

The `-help` option displays a list of the command options for the specified tool with a brief description of each option.

Syntax:

```
% tool_name -help
```

Examples:

```
% ncvlog -help
% ncvhdl -help
% ncelab -help
% ncsim -help
% ncupdate -help
```

Getting Help on Tool Messages

Use the *nchelp* utility to display extended help on the brief messages generated by the compiler, elaborator, and simulator.

Syntax:

```
% nchelp [options] tool_name message_code
```

You can enter the *message_code* argument in lowercase or in uppercase.

Examples:

```
% nchelp ncvlog BADCLP
% nchelp ncvlog badclp
% nchelp ncelab cuvwsp
% nchelp ncsim NOSNAP
```

Other Documentation

A wealth of other documentation related to Cadence products is available on SourceLink, a technical support service for Cadence software users. The service is available to all customers who have a software support services agreement. SourceLink contain product information, datasheets, information on what's new in the latest release, application notes, white papers, information about Cadence services, such as training, customer support, and methodology services, and so on.

http://sourcelink.cadence.com

Customer Support

There are several ways that you can get help with your Cadence product:

- Customer support

  Cadence is committed to keeping your design teams productive by providing answers to technical questions, the latest software updates, and education services to keep your skills updated. For information on Cadence support, go to the following web site:

  http://www.cadence.com/support

- SourceLink

  Customers with a maintenance contract with Cadence can obtain current information on the tools at the following web site:

http://sourcelink.cadence.com

- Feedback about documentation

  Contact Cadence Customer Support to file a PCR if you find:

  - An error in the manual
  - An omission of information in a manual
  - A problem using the Cadence Help documentation system

2

Compiling SystemVerilog Constructs

This section describes the ways you can compile a SystemVerilog design using the NC-Verilog simulator.

Using ncvlog

To compile a design that contains SystemVerilog constructs, use the `-sv` option with `ncvlog`:

```
% ncvlog -sv systemverilog_source_files
```

Note: SystemVerilog has added many new keywords. Therefore, Verilog legacy code that uses SystemVerilog keywords as identifiers will not compile if you use the `-sv` switch. In these cases, you can either:

- Specify the set of reserved keywords in effect, using the SystemVerilog `` `begin_keywords `` and `` `end_keywords `` compiler directives.
- Remove and restore specific keywords, using the compiler command-line option, `ncvlog -rmkeyword`, or the Cadence `` `remove_keyword `` and `` `restore_keyword `` compiler directives.

Refer to "`begin_keywords and `end_keywords" on page 240 and "`remove_keyword and `restore_keyword" on page 245 for more information.

See *NC-Verilog Simulator Help* for more details on simulating with NC-Verilog.

Using the irun Utility

With the *irun* utility, you can run the simulator by specifying all input files and all command-line options on a single command line. *irun* determines the language of a file by its extension, and then maps the file to its appropriate compiler. For example:

```
% irun -linedebug -access rwc -gui vlogfile1.v vlogfile2.v systemv1.sv systemv2.sv
```

In this example, *irun* will compile the `.v` files using `ncvlog` and the `.sv` files using `ncvlog -sv`. After the input files have been compiled, *irun* automatically invokes *ncelab* to elaborate

the design. In the example command line, the `-access` option is passed to the elaborator to provide read access to simulation objects. After the elaborator has generated a simulation snapshot, `ncsim` is invoked with SimVision.

For each file type there is a command-line option that you can use to change, or add to, the list of defined file extensions. For example, the default extensions for SystemVerilog files are `.sv`, `.svp`, `.SV`, and `.SVP`. You can add `.mysv` to the list of SystemVerilog file extensions by using the `-sysv_ext` option. For example:

```
irun top.sv dut.v test.mysv -sysv_ext .sv, .svp, .SV, .SVP, .mysv
```

When you use an extension option, the built-in defaults are removed, and you must specify all of the different extensions to be recognized.

Note: Compiling files using the `irun` utility is an alternative to using the "`begin_keywords and `end_keywords" on page 240 directives when you want to distinguish Verilog files from SystemVerilog files. However, this utility does not automatically support files that contain a mixture of Verilog and SystemVerilog. For those cases, you can use the `` `begin_keywords `` and `` `end_keywords `` compiler directives within the `.sv` file, and compile the file using `irun`.

For more information on *irun*, refer to the *irun User Guide*. For information on using the *irun* utility with DPI, refer to "Using the irun Utility with DPI" on page 269.

SystemVerilog and the PLI tf_nodeinfo() Interface

The PLI `tf_nodeinfo()` interface is not compatible with SystemVerilog designs. Therefore, you cannot compile SystemVerilog designs with the `-nomempack` option, or elaborate SystemVerilog designs with the `-arr_access` option.

For more information on these options, refer to the *NC-Verilog Simulator Help*.

3

List of Supported Constructs

This lists the SystemVerilog constructs that are supported in the current release. The following table summarizes the supported constructs, with a reference to the relevant section number(s) in the IEEE 1800 standard. Refer to the construct's relevant section within this book for information on what the current release supports and any limitations that might apply.

Note: The section numbers shown in this table are the section numbers where most of the information in the LRM can be found. In many cases, a section may contain information related to more than one construct, and information related to a particular construct may be in more than one section.

Table 3-1 SystemVerilog Constructs Supported in the Current Release

IEEE 1800 Section	SystemVerilog Construct
3.3	Literal Value Assignments on page 29
3.5 and 10.3 (for time units) 19.10 (for `timeunit` and `timeprecision`)	Time Unit and Time Precision on page 31
4	Data Types on page 37 ■ `logic`, `bit` ■ `byte`, `shortint`, `int`, `longint` ■ `typedef` declaration ■ Chandle data type ■ String data type ■ Enumerated data types ■ Structure data types
4.2	uwire Nets on page 71

Table 3-1 SystemVerilog Constructs Supported in the Current Release

IEEE 1800 Section	SystemVerilog Construct
5	Arrays on page 75 ■ Packed and unpacked arrays ■ Array querying functions ■ Dynamic arrays ■ Associative arrays ■ Queues ■ Array locator methods
6.6	Declaring Local Variables in Unnamed Blocks on page 101
6.7 and 11.5	Continuous Assignments to Variables on page 102
7	Classes on page 107 **Note:** Classes are supported in the NC-Verilog simulator, the Incisive Simulator, the Incisive Design Team Simulator, and the Incisive Enterprise Simulator. Classes are not available with the Incisive HDL Simulator. For more information on the Incisive HDL Simulator, refer to the *Design Team Family Technology Overview*.
8	Operators and Expressions on page 115 ■ Assignment operators ■ Increment and decrement operators ■ Wild equality operators ■ Assignment patterns ■ Aggregate Expressions
10.10	`iff` event control qualifier

10.4	Unique and Priority Decision Statements on page 121
	`unique if, unique case`
	`priority if, priority case`

10.5.1	do...while Loop on page 124

Table 3-1 SystemVerilog Constructs Supported in the Current Release

IEEE 1800 Section	SystemVerilog Construct
10.5.2	for Loop on page 124
10.5.3	foreach Loop on page 125
10.6	`return, break, continue Jump Statements` on page 126
10.7	final Blocks on page 127
11.2, 11.3, and 11.4	`always_* Blocks` on page 128
11.6	fork...join on page 129
11.8.2	`disable fork` on page 132
11.8.1	`wait fork` on page 132
12.2 and 12.3	Multiple Statements in Tasks and Functions on page 135
12.3	Default Direction in Task and Function Declarations on page 136
12.3.2	Discarding Function Return Values
12.3	Function Output Arguments on page 136 `function func(input integer a, input integer b, output integer x, output integer y);`
12.3.1	Void Functions on page 137 `function void myprint (integer a);`
12.4.5	Optional Arguments for Tasks and Functions on page 143
12.4.4	Passing Task and Function Arguments by Name on page 142
12.4.2	Passing Task and Function Arguments by Reference on page 138
12.4.3	Specifying Default Argument Values for Tasks and Functions on page 141

Table 3-1 SystemVerilog Constructs Supported in the Current Release

IEEE 1800 Section	SystemVerilog Construct
13	Random Constraints on page 145 ■ Random variables ■ Constraint blocks ■ Randomization methods ■ `rand_mode()` and `constraint_mode()` methods ■ In-line random variable control ■ Randomization of scope variables ■ Random number system functions and methods ■ Random stability ■ Random weighted case ■ Random sequence generator
14.2	Semaphores on page 173
14.3	Mailboxes on page 175
14.5.2	Non-Blocking Event Trigger on page 179
14.5.4	Persistent Trigger on page 181
14.7	Event Variables on page 182
15	Clocking Blocks on page 185 **Note:** Clocking blocks are supported in the NC-Verilog Simulator, the Incisive Simulator, the Incisive Design Team Simulator, and the Incisive Enterprise Simulator. Clocking blocks are not available with the Incisive HDL Simulator. For more information on the Incisive HDL Simulator, refer to the *Design Team Family Technology Overview*.

Table 3-1 SystemVerilog Constructs Supported in the Current Release

IEEE 1800 Section	SystemVerilog Construct
16	Program Blocks on page 193 **Note:** Program blocks are supported in the NC-Verilog Simulator, the Incisive Simulator, the Incisive Design Team Simulator, and the Incisive Enterprise Simulator. Program blocks are not available with the Incisive HDL Simulator. For more information on the Incisive HDL Simulator, refer to the *Design Team Family Technology Overview*.
17	SystemVerilog Assertions **Note:** SystemVerilog assertions are available only if you have an Incisive license. Support for SystemVerilog assertions is documented in the *Assertion Writing Guide* and in the *SVA Quick Reference Guide*.
18	SystemVerilog Coverage **Note:** SystemVerilog coverage is available only if you have an Incisive license. Support for SystemVerilog coverage is documented in the “Functional Coverage” chapter of the *ICC User Guide*.
19.0	Hierarchy on page 203 ■ Packages ■ Port declarations ■ Compilation units
19.11.3	.name Implicit Port Connection on page 33
19.11.4	Dot Star (.*) Implicit Port Connection on page 34
19.4	Out-of-Module Reference ($root) on page 235
20	Interfaces on page 215
22.4	Expression Size System Function ($bits) on page 236
23.2	`define on page 239
23.4	`begin_keywords and `end_keywords on page 240

Table 3-1 SystemVerilog Constructs Supported in the Current Release

IEEE 1800 Section	SystemVerilog Construct
26	Direct Programming Interface on page 249

10.8 (for `begin...end` and `fork...join`)
12.2 (for tasks)
12.3 (for functions)
7.2 (for classes)
15.2 (for clocking blocks)
16.2 (for program blocks)
19.2 (for packages)
19.5 (for modules)
20.2 (for interfaces)
	Matching End Names on page 29
	For example:
	`begin: blockA`
	`...`
	`end: blockA`

Not applicable (This is a Cadence extension)	`remove_keyword and `restore_keyword on page 245

4

Convenience Enhancements

Convenience enhancements are constructs that help make it easier to model in Verilog.

Literal Value Assignments

Literal value assignments are described in Section 3.3 of the IEEE 1800 standard.

SystemVerilog adds the ability to specify unsized literal single-bit values with a preceding apostrophe (`'`), but without the base specifier. This enhancement lets you fill a vector of any width with any logic value, without having to specify the vector size of the literal value. All bits of the vector on the left-hand side of the assignment are set to the specified value.

- `'0` – Fill all bits with `0`
- `'1` – Fill all bits with `1`
- `'Z` or `'z` – Fill all bits with `Z`
- `'X` or `'x` – Fill all bits with `X`

For example:

```
reg [127:0] data;

data = '1;    // Sets all bits of data to 1
data = 'z;    // Sets all bits of data to Z
data = 'X;    // Sets all bits of data to X
```

Matching End Names

SystemVerilog lets you specify a matching ending name with named blocks of code. This helps make the code more readable and easier to maintain.

The end name is preceded by a colon, and the specified name must be exactly the same as the name with which it is paired. A warning message is issued if the names are different.

You can specify a matching end name after the following keywords:

- `end`

  ```
  begin : block_identifier
  ...
  end : block_identifier
  ```

  See Section 10.8 of the IEEE 1800 standard.

- `join`

  ```
  fork : block_identifier
  ...
  join : block_identifier
  ```

  See Section 10.8 of the IEEE 1800 standard.

- `endtask`

  ```
  task task_identifier
  ...
  endtask : task_identifier
  ```

  See Section 12.2 of the IEEE 1800 standard.

- `endfunction`

  ```
  function function_identifier
  ...
  endfunction : function_identifier
  ```

  See Section 12.3 of the IEEE 1800 standard.

- `endclass`

  ```
  class class_identifier
  ...
  endclass : class_identifier
  ```

  See Section 7.2 of the IEEE 1800 standard.

- `endclocking`

  ```
  clocking clocking_identifier
  . . .
  endclocking: clocking_identifier
  ```

  See Section 15.2 of the IEEE 1800 standard.

- `endprogram`

  ```
  program program_identifier
  . . .
  endprogram: program_identifier
  ```

  See Section 16.2 of the IEEE 1800 standard.

- `endpackage`

```
package package_identifier
. . .
endpackage: package_identifier
```

See Section 19.2 of the IEEE 1800 standard.

- `endmodule`

```
module module_identifier
...
endmodule : module_identifier
```

See Section 19.5 of the IEEE 1800 standard.

- `endinterface`

```
interface interface_identifier
...
endinterface : interface_identifier
```

See Section 20.2 of the IEEE 1800 standard.

Time Unit and Time Precision

Time unit and time precision are described in Sections 3.5 and 10.3 (for time units), and Section 19.10 (for `timeunit` and `timeprecision`) of the IEEE 1800 standard.

In Verilog-2001, time values are specified with a number without a time unit. For example,

```
initial
  #5 clock = 1;

always
  #50 clock = ~clock;
```

The time unit, and the time precision, can be specified with the `` `timescale `` compiler directive. You can specify this directive in one or more files, and you can specify directives with different time unit and time precision values for different modules in the design. When the source files are compiled, and a `` `timescale `` directive is encountered, that directive remains in effect until another `` `timescale `` directive is encountered. Therefore, the time units and precision that is used for a source file without a `` `timescale `` directive depends on the order in which the source files are compiled. This can cause simulation results to vary for different simulation runs.

SystemVerilog provides two enhancements to control the specification of time units of time values, which remove ambiguity and the file order dependency problem associated with the `` `timescale `` directive.

- A time unit can be specified with a time value.

The time unit can be `s`, `ms`, `us`, `ns`, `ps`, or `fs`. There can be no whitespace between the time value and the time unit. For example:

```
0.1ns
40ps
#10ns clock = ~clock;
r = <= #1ns a;
```

Note: The SystemVerilog LRM states that the time unit can also be `step`. This is not supported in the current release.

See Sections 3.5 and 10.3 of the IEEE 1800 standard for details.

- Time units and time precision can be specified within a module with the keywords `timeunit` and `timeprecision`.

  As with the `` `timescale `` directive, the units that can be specified with the `timeunit` and `timeprecision` keywords are `s`, `ms`, `us`, `ns`, `ps`, or `fs`, and the units can be specified in multiples of 1, 10, or 100. There can be no whitespace between the time value and the time unit.

  These declarations can be specified within a module, package, or interface. The declarations must appear immediately after the module, package, or interface declaration. For example:

```
module test #(parameter MSB = 3, LSB = 0)
             (output reg [MSB:LSB] x,
             input wire [MSB:LSB] a,
             input wire [7:0] b,
             input wire enable);

timeunit 1ns;
timeprecision 10ps;

initial
  begin
    $display($time,,,"x=%d", x );
    #20 $finish;
  end

endmodule
```

  The scope of a `timeunit` or `timeprecision` declaration is limited to the design unit in which it is declared. There can be only one time unit and one time precision for the design unit. The `timeunit` and/or `timeprecision` declaration can be repeated as later items, but the values must match the original values exactly.

  See Section 19.10 of the IEEE 1800 standard for details.

In the current release, the time unit and precision for a time value is determined according to the following search order:

1. Use the time unit specified as part of the time value.

2. Use the time unit and precision specified with the `timeunit` and `timeprecision` keywords.
3. Use the time unit and precision specified with the `` `timescale `` compiler directive that is currently in effect.
4. Use the simulator's default time unit and precision.

.name Implicit Port Connection

Module instantiation using implicit `.name` port connections is described in Section 19.11.3 of the IEEE 1800 standard.

In Verilog-2001, you can instantiate a module using named port connections. The syntax requires you to specify the port name used in the module declaration, followed by the name used in the instantiating module. For example:

```
module top;
  wire data, clk, clr, q, qb;

  flop u1 (.data(data), .clock(clk), .clear(clr), .q(q), .qb(qb));
  ...
endmodule

module flop (input data, clock, clear, output q, qb);
  ...
  ...
endmodule
```

When you connect module instance ports in this way, the width of the port does not have to match the width of the net or variable connected to the port.

SystemVerilog simplifies this syntax for cases where the name of the port matches the name of the net or variable connected to the port. If the names match, and if the data types on each side of the port are compatible, you can specify only the port name. For example, in the code shown above, the port `data` is connected to the net `data`. This connection can be specified as `.data`.

Verilog named port connections must be used if the names do not match. Implicit `.name` port connections can be combined with named port connections. In the example above, the port connections could be written as follows:

```
  flop u1 (.data, .clock(clk), .clear(clr), .q, .qb);
```

Note: The SystemVerilog LRM specifies that the size of the port must match the size of the net or variable connected to the port. In the Cadence implementation, this restriction is not imposed. A warning is issued if the size of the net or variable does not match the size of the port.

As in Verilog-2001, you cannot mix positional port connections and named port connections in a module instantiation.

Dot Star (.*) Implicit Port Connection

Module instantiation using implicit `.*` port connections is described in Section 19.11.4 of the IEEE 1800 standard.

In addition to using `.name` implicit port connections in the port list of a module instantiation (see ".name Implicit Port Connection" on page 33), SystemVerilog also provides the `.*` construct to further simplify the syntax for connecting ports by name.

The `.*` implicit port connection is used in the port list of a module instantiation as a shorthand for named port connections. Each unconnected port in the module definition is connected to a variable, wire, or interface with the same name declared in the instantiating module.

As with `.name` implicit port connections, the name of the port must match the name of the net or variable connected to the port, and the data types connected together must be compatible. Verilog-2001 named port connections must be used for any connections that cannot be inferred by `.*`.

Note: The SystemVerilog LRM specifies that the size of the port must match the size of the net or variable connected to the port. In the Cadence implementation, this restriction is not imposed. A warning is issued if the size of the net or variable does not match the size of the port.

Note: In the current release, the following restrictions apply to the use of `.*` implicit port connections:

- `.*` cannot be used inside a generate block.
- `.*` can be used only in the instantiation of a Verilog or SystemVerilog module or interface. For example, the construct cannot be used in the instantiation of an analog block, a VHDL block, or a SystemC® block.

The following example uses the `.*` syntax in the module instantiation:

```
module top;
  wire data, clk, clr, q, qb;

  flop u1 (.*, .clock(clk), .clear(clr));
  ...
endmodule

module flop (input data, clock, clear, output q, qb);
  ...
  ...
endmodule
```

You can use only one `.*` token in the port list. When the implicit `.*` port connection is mixed with named port connections, as in the example shown above, or with `.name` implicit port connections, you can place the `.*` token anywhere in the port list.

As in Verilog-2001, you cannot mix positional port connections with named port connections in a module instantiation.

5

Data Types

Data Types Overview

Data types are described in Section 4 of the IEEE 1800 standard.

In Verilog-2001, all logic values manipulated during simulation are 4-state. That is, the logic values in the simulation belong to the set `0`, `1`, `X` (unknown), and `Z` (high impedance). The logic value represented by a variable or net always belongs to this set of four values at any time.

Verilog 2001 defines the following data types for storing integers:

Data type	Description	Default
`reg`	4-state, user-defined vector size	unsigned
`integer`	32-bit 4-state integer	signed
`time`	64-bit 4-state integer	unsigned

SystemVerilog introduces one new 4-state data type (`logic`) and several 2-state data types. In the current release, the following 4-state and 2-state data types have been implemented:

Data type	Description	Default
`logic`	1-bit 4-state integer, user-defined vector size See “logic Data Type” on page 40.	unsigned
`bit`	1-bit 2-state integer, user-defined vector size See “bit Data Type” on page 41.	unsigned
`byte`	8-bit 2-state integer or ASCII character See “byte, shortint, int, longint Data Types” on page 42.	signed

Data type	Description	Default
`shortint`	16-bit 2-state integer	signed
`int`	32-bit 2-state integer	signed
`longint`	64-bit 2-state integer	signed

Overview of Verilog Data Types

Verilog data objects have two attributes:

- The *kind* of the object (parameter, variable, or net)

  The object kind indicates what you can do with the object. For example, only parameters can be modified with `defparam` statements, and only variables can be assigned by procedural assignments.

- The *data type* of the object (integer, real, scalar bit, bit vector, and so on)

  The data type of an object indicates the values that the object can take on. For example, an object of type `real` can take on the value `3.14`, while an object of a bit vector type can take on the value `4'b0xz1`.

These two attributes of an object are largely orthogonal. For example, a net can be of almost any data type, and a bit vector can be the data type of almost any kind of object.

A data type is a set of values, and the Verilog data types fall into two groups:

- Primitive Data Types, whose values cannot be defined in terms of other values or data types.
- User-Defined Data Types, whose values are constructed from other values or data types.

Primitive Data Types

The Verilog data types include a small number of *primitive* data types that serve as a basis for the value system. The values of a primitive data type cannot be defined in terms of other values or data types.

Verilog-2001 has two primitive data types: a 4-state bit type, and a real type. SystemVerilog introduces a name, `logic`, for the existing 4-state data type. SystemVerilog also introduces a 2-state bit type called `bit`. Thus, the extended language has three primitive data types: `logic`, `bit`, and `real`. These primitive data type names are all reserved words.

The following variable declarations use the primitive data type names:

```
real realvar;      // a real-valued variable
logic logicvar;    // a 4-state variable
bit bitvar;        // a 2-state variable
```

See "logic Data Type" on page 40 for details on the `logic` data type. See "bit Data Type" on page 41 for details on the `bit` data type.

User-Defined Data Types

In addition to primitive data types, the data type system includes data types that have values constructed from other values. These data types are called *user-defined* data types because you must describe how the values are constructed. For example, a Verilog bit vector is a user-defined data type because you must include a range to indicate the number of bits and how to index the vector.

You can describe the characteristics of a user-defined data type directly in an object declaration. For example, the following declarations define two 8-bit–wide 4-state variables called `byte_var1` and `byte_var2`:

```
logic [7:0] byte_var1;
logic [7:0] byte_var2;
```

SystemVerilog enhances the language by introducing the `typedef` declaration, which you can also use to describe the characteristics of a user-defined data type. A `typedef` declaration gives the data type a name that you can use in other declarations. The following example shows how to define the same two variables using a `typedef` declaration:

```
typedef logic [7:0] bits8;    // The name of this data type is bits8
bits8 byte_var1;
bits8 byte_var2;
```

See "typedef Declaration" on page 55 for details on the `typedef` declaration.

Some user-defined data types are so common that they have been given special predefined status in the language. Their names are reserved words, and you can use them without specifying the construction of their values. The Verilog-2001 predefined data types are `integer`, `time`, and `realtime`. The `integer` data type represents a 4-state 32-bit signed integer. SystemVerilog introduces new predefined data types, `byte`, `shortint`, `int`, and `longint` to represent 2-state signed integers of 8, 16, 32, and 64 bits, respectively. See "byte, shortint, int, longint Data Types" on page 42 for more information on these new data types.

logic Data Type

In Verilog-2001, all logic values manipulated during simulation are 4-state. That is, the logic values in the simulation belong to the set `0`, `1`, `X` (unknown), and `Z` (high impedance). The logic value represented by a variable or net always belongs to this set of four values at any time.

SystemVerilog gives this 4-state data type a name: `logic`. This keyword is simply the name of the 4-state bit type; it does not imply an object kind. The `logic` keyword can be used in any context in which a data type is allowed when you want to declare a 4-state object. For example, `logic` is used to explicitly state the data type in the following parameter, variable, and net declarations:

```
parameter logic p = 1'b0;    // 1-bit wide 4-state parameter
logic v;                     // 1-bit wide 4-state variable
logic [63:0] v2;              // 64-bit wide 4-state variable
wire logic w;                // 1-bit wide 4-state net
```

These declarations are equivalent to the following Verilog-2001 declarations:

```
parameter p = 1'b0;
reg v;
reg [63:0] v;
wire w;
```

You can also use the `logic` data type on ports. Output ports that are declared as a `logic` data type are considered variables by default. In the following example, port `answer` is declared as a `logic` data type and is considered a variable by default.

```
module mymod(result, one, two);
  output logic result;
  input wire one;
  input wire two;

  assign result = one & two;
endmodule
```

Note: When you use the `logic` data type on an output port, a variable is added to its connections. The added variables are then subject to continuous assignments. The continuous assignments for these added variables can affect optimization, and can cause performance degradation (depending on the number of levels in your design, and the number of continuous assignments). To prevent the addition of variables to a port's connections, you can declare the port to also be a wire. For example:

```
...
output wire logic inputA;
...
```

Refer to "Continuous Assignments to Variables" on page 102 for more information on continuous assignments.

logic and reg Data Types

In Verilog-2001, the `reg` keyword is special in that it implies both a data type (4-state logic) and an object kind (variable). Unlike other keywords that imply data type, such as `integer`, `reg` cannot be used to declare a parameter or a net.

In SystemVerilog, `reg` is a data type with the same semantics as `logic`. Both keywords specify that an object is 4-state, without saying anything about what kind of object it is. That is, `reg` does not imply that an object is a variable, as opposed to a net or parameter. For example, the following declarations are supported:

```
reg [31:0][7:0] mdv;
typedef reg signed [31:0] myIntT;
struct packed { reg [1:9] m1; reg m2; } packedStructVar;
enum reg [1:0] {high, moderate, low, off} eVar = off;

function reg func(input x);
      ...
endfunction

parameter reg x = 0;    // Same as: parameter logic x = 0;
module m(inout reg x);  // Module port is a net, not a variable
```

Note: The IEEE 1800 standard (Section 6.5) states that the `reg` keyword cannot immediately follow a net type keyword, such as `wire` or `tri`. The `reg` keyword can be used in a net or port declaration if there are lexical elements between the net type and `reg` keywords. For example:

```
wire reg x;               // Illegal. reg follows net type wire
inout tri reg y;          // Illegal. reg follows net type tri
wor scalared reg[3:0] z;  // Legal. scalared separates wor and reg
```

In these cases, use the `logic` keyword instead of `reg`.

bit Data Type

SystemVerilog adds support for a 2-state logic data type called `bit`. This keyword is simply the name of the 2-state bit type. As with `logic`, it does not imply an object kind. The `bit` data type differs from `logic` in that `bit` only stores the 2-state values of 0 and 1.

You can declare parameters and variables to be of type `bit`. Nets cannot be declared as `bit`.

The `bit` data type is used in exactly the same way the `logic` type is used. For example:

```
parameter bit p = 0;    // parameter p can only be 0 or 1
bit v;                  // 1-bit wide 2-state variable
bit [63:0] v2;           // 64-bit wide 2-state variable
```

Treatment of 2-state objects varies, depending on the object kind.

- `bit` parameters are initialized to the value specified in the parameter declaration. You can override the initial value in the same way that 4-state parameters are overridden.
- `bit` variables are initialized to `0`.
- It is possible to assign an `X` or a `Z` to a `bit` variable. `X` and `Z` values are converted to 0.

byte, shortint, int, longint Data Types

In Verilog-2001, the `integer` data type is a 32-bit vector of 4-state values intended for holding signed integer numbers. The Cadence data type extensions allow `integer` to be used as a general-purpose data type. You can use it in any context in which a data type is allowed. For example:

```
wire integer w;     // wire of type integer
input integer p;    // input port of type integer
```

SystemVerilog introduces new 2-state data types for storing integer numbers:

- `byte`

  A vector of eight 2-state bit values for holding either a signed integer number or a single ASCII character.

- `shortint`

  A vector of 16 2-state bit values for holding a signed integer number.

- `int`

  A vector of 32 2-state bit values for holding a signed integer number.

- `longint`

  A vector of 64 2-state bit values for holding a signed integer number.

Nets cannot be declared as a 2-state data type.

A variable declared as a 2-state data type is initialized to 0.

When an object declared as a 2-state data type is assigned a value, any `X` or `Z` values in the elements of the new value are converted to 0.

The following are some example declarations using the new data types:

```
int v;                 // A variable of type int
input int p2;           // input port of type int
parameter int w2 = 289; // A parameter of type int
shortint errors;       // A variable of type shortint
parameter longint reset_count = 0; // A parameter of type longint

enum byte {steady, rising, falling} barometer; // An 8-bit 2-state enumeration
                                               // variable
```

Chandle Data Type

The `chandle` data type is described in Section 4.6 of the IEEE 1800 Standard.

SystemVerilog adds a special `chandle` data type, which is used to store and pass C or C++ pointers in and out of DPI imported or exported tasks and functions. The syntax for a `chandle` declaration is as follows:

`chandle` *`variable_name`*;

where *`variable_name`* is a valid identifier. Chandles are initialized to the value `null` (value of zero on the C side).

For example, the following uses chandles to pass C pointers as arguments to imported DPI functions:

```
import "DPI-C" function chandle func10_dpi_ch ( inout chandle hndl);
```

Note: A `chandle` pointer cannot be relocated by the simulator, because it points directly into user memory. When you restart a snapshot, `chandle` pointers restore the value they had when the snapshot was last saved. This might cause unexpected behavior if your DPI code tries to deference a `chandle` pointer after a restore operation, because the `chandle` pointer might have changed since the last save operation. A warning message is issued when you try to restore a snapshot that contains `chandle` pointers.

Note: In the current release, you cannot declare chandles inside unpacked structures.

Strings

The string data type is described in Section 4.7 of the IEEE 1800 standard.

SystemVerilog introduces a `string` data type, which represents a variable-length text string. The syntax for a string data type is as follows:

`string` *`string_identifier`* [= *`initial_value`*];

where the *`initial_value`* can be a string literal or an empty string `""`.

The `string` data type has the following characteristics:

- A string literal can be assigned to a `string` data type.
- The length of the `string` data type can vary during simulation.

  When a string literal is assigned to an integral variable or an unpacked array of bytes of a different size, the string literal is truncated. The `string` data type eliminates this situation. When a value is assigned to a `string` variable, its length adjusts accordingly.
- A single character of a `string` variable is of type `byte`.
- The indexes of a `string` variable are numbered from `0` to *`N`*`-1`, where *`N`* is the length of the string. The `0` corresponds to the first character of the string, and `N-1` corresponds to the last character of the string.
- If a string literal contains the special "`\0`" character, this character is ignored.

String Operators

The following table describes the SystemVerilog string operators that are supported in the current release.

Operator	Description
`s1 == s2`	The *equality* operator checks whether two strings are equal.
`s1 != s2`	The *inequality* operator checks whether two strings are different.
`s1[index]`	The *indexing* operator returns the ASCII code for the given index. If the given index is out of range, the operator returns 0.
`{s1, s2,...,sN}`	The *concatenation* operator can take `strings` or string literals. If at least one operand is of type `string`, then the expression evaluates to the concatenated string and is of type `string`. If all operands are string literals, then the expression behaves like a Verilog concatenation of integral types. If the result is then used in another expression that involves `string` types, the expression is converted to type `string`.

`s1 < s2`
`s1 <= s2`
`s1 > s2`
`s1 >= s2`
	The *comparison* operators use the `compare()` string method. If the given condition is true, these relational operators return 1. Both operands can be of type string, or one of them can be a string literal.

String Methods

SystemVerilog provides built-in methods for working with strings. The current release supports the following string methods:

Method	Syntax	Description
`len()`	`str.len()`	Returns the length of the specified string
`putc()`	`str.putc(i, c)`	Replaces the `i`th character of the string with the given integral value `c`.
`getc()`	`str.getc(i)`	Returns the ASCII code of the `i`th character of the specified string
`toupper()`	`str.toupper()`	Returns the uppercase of the specified string
`tolower()`	`str.tolower()`	Returns the lowercase of the specified string
`itoa()`	`str.itoa(i)`	Stores the ASCII representation of `i` into the string
`atoi()`	`str.atoi()`	Returns the integer corresponding to the ASCII decimal representation in `str`
`atobin()`	`str.atobin()`	Returns the binary value corresponding to the ASCII binary representation in `str`
`atohex()`	`str.atohex()`	Returns the hexadecimal value corresponding to the ASCII hexadecimal representation in `str`
`atooct()`	`str.atooct()`	Returns the octal value corresponding to the ASCII octal representation in `str`
`hextoa()`	`str.hextoa(i)`	Stores the ASCII hexadecimal representation of the `i`th character of the specified string
`octtoa()`	`str.octtoa(i)`	Stores the ASCII octal representation of the `i`th character of the specified string
`bintoa()`	`str.bintoa(i)`	Stores the ASCII binary representation of the `i`th character of the specified string
`compare()`	`str.compare(s)`	Compares `str` and `s`
`icompare()`	`str.icompare(s)`	Compares `str` and `s`, but the comparison is case insensitive.

Method	Syntax	Description
`atoreal()`	`str.atoreal()`	Returns the real number corresponding with the ASCII decimal representation in `str`
`realtoa()`	`str.realtoa(r)`	Stores the ASCII real representation of `r` into `str`.
`substr`	`str.substr(i, j)`	Returns a new string that is a substring formed using the characters in position `i` and `j` of `str`. If `i < 0`, `j < i`, or `j >= str.len()`, then this method returns an empty string.

Example 5-1 Using String Methods and Operators

The following example illustrates the supported string methods and operations.

```
module top;

//Declares a string data type.
string mystring = "hello world";
string newstring= "he\0llo big world"; //The \0 character will be ignored.
string string_pi = "3.1415";
real real_pi;
int i;

initial begin
   $display ("Value of string is: %s", mystring);
   i = mystring.len();                //Returns the length of the string.
   $display("%d characters long", i);

   //Displays the ASCII code for the given characters.
   $display("%s", mystring.getc(0));
   $display("%s", mystring.getc(1));
   $display("%s", mystring.getc(2));
   $display("%s", mystring.getc(3));
   $display("%s", mystring.getc(4));

   //Changes the string to uppercase
   mystring = mystring.toupper();
   $display ("My new string in uppercase: %s", mystring);

   //Displays the string in lowercase
   $display("My new string in lowercase: %s", mystring.tolower());
   $display("First string is:%s", mystring);
   $display("Second string is:%s", newstring);

   //Compares two strings
    if (newstring == mystring)
     $display("The strings are the same.");
    else
     $display("These strings are not the same.");
    if (newstring != mystring)
```

```
     $display("Again, they are not the same.");
    else
     $display("These strings are the same.");

    //Uses the indexing operator to replace characters
    mystring[0] = "Y";
    mystring[5] = "W";
    $display("New string %s", mystring);

    //substr() method extracts “WORLD” from “YELLOWWORLD”
    $display(“Short string is %s”, mystring.substr(6,10));

    //Converts string to a real value
    real_pi = string_pi.atoreal();
    $display(“String pi is %s”, string_pi);
    $display(“Real pi is %f”, real_pi);

end

endmodule
```

This example produces the following output:

```
Value of string is: hello world
11 characters long
 h
 e
 l
 l
 o
My new string in uppercase: HELLO WORLD
My new string in lowercase: hello world

First string is: HELLO WORLD
Second string is: hello big world

These strings are not the same.
Again, they are not the same.

New string YELLOWWORLD

Short string is WORLD

String pi is 3.1415
Real pi is 3.141500
```

Strings and System Tasks

The following system tasks have been enhanced such that they can accept `string` variables as arguments: `$display`, `$write`, `$strobe`, `$monitor`, `$fdisplay`, `$fwrite`, `$fstrobe`, `$fmonitor`, `$sscanf`, and `$fopen`.

Using Strings with Classes

The current release supports strings within classes as:

- Default class members (public or automatic)
- Static, local, or protected class members
- Local and global constant class members

You can perform the following operations on a string that is a member of a class:

- Initializing the string within a constructor
- Passing the string as an argument to an automatic or static function or task that is within the class. The actual string can be a member of the same class or can be declared outside the class.
- Using the string with any of the supported operators and methods. See "String Operators" on page 44 and "String Methods" on page 45.
- Using the string as an argument to system tasks and functions
- Using the string as the return type for an automatic or static function that is declared inside the class
- Using the string with the `this` and `super` keywords
- You can declare a function that returns the string as `extern`.
- A virtual function can return the string and take the string as an argument.

The following example illustrates how to use strings within a class.

Example 5-2 Using Strings with Classes

```
module top;
class demo_class;
  static string s; //Static class member
  string s1;
  string s2;
  string s3;
  function new();
    s1 = “first_string”; //Initialized within a constructor
    s2 = “second_string”;
  endfunction
  function string concat_string(); //Function that returns a string
    return {s1.toupper(),"",s2.toupper()}; //Uses string methods and operators
  endfunction
  task format_string(string s1); //String passed as an argument
   $sformat(s, “Output of %s”, s1); //String as an argument of a system task
  endtask
endclass
```

```
demo_class dc;
initial begin
  dc = new;
  dc.s3 = {dc.s1,"", dc.s2};
  if (dc.s3.toupper() == dc.concat_string())
    $display(“Correct”);
  else
    $display(“Not Correct”);
  dc.format_string(“format_string”);
  $display(dc.s);
end
endmodule
```

This should produce the following output:

```
Correct
Output of format_string
```

Using Strings with Packages

A string can also be declared as a public, static, local or protected member of a class that is declared inside a package. The following is supported for strings that are members of a class declared within a package:

- Initializing the string inside a constructor
- Passing the string as an argument to a task or function inside the class
- Using the strings as an argument to system tasks and functions
- The `this` and `super` keywords can be applied to this type of string.
- A virtual function can return this type of string and take the string as an argument.
- Using any of the supported operators and methods on this type of string. See “String Operators” on page 44 and “String Methods” on page 45.
- Declaring a function that returns this type of string within a class.

Using Strings within begin...end Blocks

Strings can be declared as static within `begin...end` blocks for the following types of statements:

`initial`	`always`	`case`, `casex`, and `casez`
`repeat`	`while`	`forever`
`for`	`fork...join`	`do...while`

You can perform the following operations on strings that are declared within a `begin...end` block:

- Assigning a string literal or string variable to another string
- Passing a string as an argument to tasks or functions
- Using the string with any of the supported operators and methods. See "String Operators" on page 44 and "String Methods" on page 45.

Declaring a Fixed Array of Strings

You can declare a fixed array of type string within the following scopes: modules, program blocks, tasks, functions, packages, and classes.

You can perform the following operations on the index of a fixed array of strings:

- Assigning the string index to a string literal or string variable.
- Using the string index as an argument to system tasks or functions.
- Passing the string index as an argument (`input`, `output`, or `inout`) to a user-defined task or function
- Using the string index with any of the supported operators and methods. See "String Operators" on page 44 and "String Methods" on page 45.

The following example illustrates how to use a fixed array of strings.

Example 5-3 Fixed Array of Strings

```
module top;
  string s1[10];
  string s = "one";
  function string func(string s2);
   return s2.substr(1,4);
  endfunction
initial begin
  #0;
  s1[0] = "zero"; //String literal is initialized to the index of the string array
  s1[1] = s; //String variable is initialized to the index of the string array
  $display("Print index %c", s1[0][1]); //Uses the indexing operator
  if(s1[0]==s1[1]) //Uses the equality operator
    $display("Incorrect");
  else
    $display("Relational operator is working correctly");
  s1[2] = {s1[0],"",s1[1]}; //Uses concatenation operator
  $display("Print concatenation result", s1[2]);
  $display("Length of first index:%d", s1[1].len()); //Uses the len method
  s1[3] = s1[2].toupper(); //Uses the toupper method
  $display("Print toupper", s1[3]);
  s1[4] = func(s1[3]); //Index of string array is passed as an argument to a
```

```
                        //function that returns a string that is assigned to the
                        //index of a string array.
  $display("After function return", s1[4]);
  s1[5] = "hi";
  $sformat(s1[5],"%s","hello"); //Index of string array as an argument to a system
                                //task
  $display("after sformat", s1[5]);
end
endmodule
```

This example should produce the following output:

```
Print index e
Relational operator is working correctly
Print concatenation result zero one
Length of first index: 3
Print toupper ZERO ONE
After function return ERO
After sformat hello
Index 3 changed
```

Declaring Arrays and Queues of Strings

You can declare fixed arrays, dynamic arrays, queues, and associative arrays of type `string` within the following scopes: modules, program blocks, tasks, functions, packages, and classes.

You can perform the following operations on the index of an array or queue of strings:

- Assigning the string index to a string literal or string variable.
- Using the string index as an argument to system tasks or functions.
- Passing the string index as an argument (`input`, `output`, or `inout`) to a user-defined task or function
- Using the string index with any of the supported operators and methods. See "String Operators" on page 44 and "String Methods" on page 45.

An element of an array or queue of strings is a string. The current release supports the same functionality for strings as for the elements of an array or queue of strings. In turn, whatever is not supported for strings is also not supported for elements of an array of strings. This applies to only the *elements* of an array or queue of strings. The *entire* array or queue is subject to the same limitations as defined for that type of array. See "Arrays" on page 75.

Using Elements of a Dynamic Array of Strings

An element of a dynamic array of strings is a string. The current release supports the same functionality for strings and for elements of a dynamic array of strings. In turn, whatever is not supported for strings is also not supported for elements of a dynamic array of strings.

Note: This applies only to elements of a dynamic array of strings. The entire dynamic array, however, is subject to the same limitations as defined for dynamic arrays. See "Arrays" on page 75.

For example, you have the following on an element of a dynamic array of strings:

```
string dyn_arr[];
string s1;
...
dyn_arr = new[10];
...
```

You can perform the following operations on the dynamic array of strings, because they are supported for the `string` data type.

- Bit select on an element of a dynamic array of strings

  ```
  dyn_arr[0] = "abc";
  ```

- Pass it as an `inout`/`input`/`output` parameter to a function

  ```
  func(dyn_arr[0])
  ```

- Assign an element of the dynamic array of strings to a string

  ```
  s1 = dyn_arr[0];
  ```

- Functions can return an element of a dynamic array of strings

  ```
  s1 = func(dyn_arr[0]);
  ```

- Event controls are supported for elements of a dynamic array of strings

  ```
  always @ (dyn_arr[0])
  ```

Using Strings as Parameters and localparams

A string can be declared as parameter and localparam inside a module. In the current release, a string parameter can be:

- Assigned while instantiating a module.
- Assigned to other string variables.
- An argument to system tasks/functions.
- Passed as an argument to user defined functions/tasks.

- Can be used with the supported String Operators and String Methods—except for the `itoa()`, `hextoa()`, `octtoa()`, and `bintoa()` methods as it is illegal to change string parameter values at simulation time.

Example 5-4 Using String Parameters

```
module sub;
  parameter string test = "hello"; // Declares a string parameter
  string test1;
  function string func(string s);
    func = s;
    $display(func);
  endfunction
  initial begin
   $display(test);
   test1 = test; // String parameter is assigned to a string variable
   $display(test1);
   if (test1 == test) // Equality operator
   $display("correct");
   test1 = {test, "hello"}; // Concatenation operator
   $display(test1);
   $display("byte : %c",test[1]); // Indexing operator
   $display("len : ",test.len()); // len method
   $display("getc : ",test.getc(0)); // getc method
   $display("toupper : ",test.toupper()); // toupper method

   if (!test1.compare(test)) // compare method
     $display("correct");
     $display("atoi : ", test.atoi()); // atoi method
     test1 = func(test); // string parameter passed as an argument
     $display(test1);
     test1 = "";
     $sformat(test1,"%s",test); // String parameter passed as an argument to
                                // sformat
     $display(test1);
  end
endmodule

module top;
  sub #("hi")t1();
endmodule
```

This example produces the following simulation results:

```
hi
hi
correct
hihello
byte : i
len : 2
getc : 104
toupper : HI
atoi :   0
hi
hi
hi
```

Limitations on Strings

The following summarizes the features in the LRM that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- The replication and dot operators are not supported.
- Declaring the `string` data type or assigning a `string` literal or `string` variable to a `string` is supported within a module, program block, task, function, package, or class. The current release does not support strings within a structure.

```
typedef struct{
  int s1;
  string s; //Unsupported
} mystruct;
```

- Strings cannot be used in out-of-module references.

```
interface myint;
  string s4 = “My string”;
endinterface

module top1;
  myint ins();
  initial begin
   $display(“Value of my string:%s”, ins.s4); //Causes elaboration error
  end
endmodule
```

- You cannot use string indexing on class selects that are an argument to system tasks for functions. For example (where `c1` is a class):

```
$sformat(str,"%x", c1[2].s); //Invalid
```

- If a string is a member of a dynamic or fixed array of a class, then you cannot use it as an argument to a system task or function. For example:

```
c c1[10];
$sformat(str, "%s", c1[2].s); //Invalid
```

- Strings cannot be declared in a `for` loop that is inside a `generate` statement. For example:

```
module test;

integer j=0;
genvar i;

generate
for(i=4; i>1; i=i-1) begin : gen1
typedef logic [0:7] w_type;

logic [7:0] A;
string B ; // Invalid

end
```

```
endgenerate
...
endmodule
```

- You cannot use string members of a class in port mapping or with the scope resolution operator.
- String variables cannot be passed by reference to a task or function.

typedef Declaration

A `typedef` declaration lets you describe a data type in a declaration that gives the data type a name that you can use in other declarations. The advantage of using a `typedef` is that you can characterize the data type in a single place and then refer to that description in as many other declarations as you like. By referring to a common definition for the data type, you can guarantee that a set of objects share the same data type characteristics, such as width, array size, and signedness. Another common use for naming data types is to provide descriptive information for the object declarations that use them. This can help to make the code more self-documenting and easier to read.

Limitations

This section summarizes the features in the SystemVerilog standard that are not supported in the current release.

- The LRM states that a type can be used before it is defined. For example:

  ```
  typedef foo;
  foo f = 1;
  typedef int foo;
  ```

  This form is not supported.

- The LRM also allows references to type identifiers defined within an interface through ports, provided that they are locally redefined before being used. This is not supported.

Creating a New Data Type Definition

A `typedef` declaration begins with the keyword `typedef`. The nature of the rest of the declaration depends on what kind of data type you want to declare. The syntax is essentially the same as that of an object declaration, with the data type name appearing in the location where you would normally put the object name.

In the following example, a type named `bits8` is used to declare two variables with the same data type characteristics (4-state, 8-bits wide). In this example, the name `bits8` is a synonym for `logic [7:0]`.

```
typedef logic [7:0] bits8;    // The name of this data type is "bits8"
    bits8 byte_var1;
    bits8 byte_var2;
```

The following declarations illustrate the syntax for different kinds of data types:

```
// A scalar type called logic_type
typedef logic logic_type;

// An unsigned vector type called vector_type
typedef logic [31:0] vector_type;

// A signed vector type called signed_vector_type
typedef logic signed [31:0] signed_vector_type;

// An array type called array_type
typedef logic [7:0] array_type [15:0][15:0];
```

A `typedef` declaration does not create a new data type. It simply introduces a new name for a data type. The data type itself is defined by the form of its values and the associated operators. The fact that a data type has a name does not imply any special treatment. The same compatibility and conversion rules apply to its values.

In the Cadence implementation, you can use a data type name in the declaration of parameters, variables, nets, ports, functions, and tasks. The following code fragments show where the data type name appears in the different kinds of Verilog declarations:

```
// The name of this 32-bit bit vector data type is addressT
typedef logic [31:0] addressT;

// Variable
addressT v1;

// Net
wire addressT w1;

// Port
inout addressT p1;

// Parameter
parameter addressT default_value = 32'h00000000;

// Task or function argument
input addressT value_in;

// Function
function addressT checksum;
```

Example 1

In this example, a `typedef` declaration is used to define a set of shared data type characteristics in a single place. The data type name is then used in other declarations and in another `typedef` declaration.

```
localparam BUS_WIDTH = 32;    // address & data are same size
localparam DMA_BURST = 4;     // how many data words are transferred

// Create a name, busT, for data & address buses
typedef logic [BUS_WIDTH:1] busT;

// Declare two variables of type busT
busT IOaddress_reg, IOdata_reg;

// Declare two nets of type busT
wire busT IOaddress, IOdata;

// Declare a data type called dmaT constructed from type busT
typedef busT dmaT [DMA_BURST-1:0];
```

The shared width characteristics of the data and address buses are captured in the data type definition of `busT`. This data type name is then used to declare two variables (`IOaddress_reg` and `IOdata_reg`), two nets (`IOaddress` and `IOdata`), and another data type (the array type `dmaT`).

Example 2

In this example, the data type names declared in `typedef` declarations do not define any data type characteristics, but provide descriptive information for the object declarations that use them.

```
typedef integer flagsT;
typedef flagsT maskT;

typedef enum maskT {
    SIGBUS  = 32'h00000001,
    SIGSEGV = 32'h00000002,
    SIGTRAP = 32'h00000004,
    SIGILL  = 32'h00000008,
    SIGFPE  = 32'h00000010
} flag_bitT;

task set_flag ( inout flagsT flags, input flag_bitT flag_bit );
    flags = flags | flag_bit;
endtask

task clear_flag ( inout flagsT flags, input flag_bitT flag_bit );
    flags = flags & ~flag_bit;
endtask
```

These declarations define a set of flag bits and utility routines for setting and clearing a given flag bit. The `flagsT` data type is simply another name for `integer`, and the `maskT` data type

is simply another name for `flagsT`. Thus, we have three different names for the same data type.

The new names are intended to provide additional descriptive information for the object declarations that use them. These distinct names are used as the enumeration base type and the task argument types to make the semantic nature of these items very clear.

Handling Data Type Visibility

Like macros, data type names must be declared before they are used. A data type declaration can appear outside of a module declaration, or in any location in which an object declaration is allowed.

The information about a data type (that is, the information in its declaration) must be known at the time that you compile declarations that use it. Since hierarchical references cannot be resolved until the design hierarchy has been created, you cannot use a hierarchical name to refer to a data type.

The following visibility rules apply to data type names:

- If a data type name is declared inside a block, the name is visible only within that block.
- If a data type name is declared inside a module, the data type name is visible only within that module.
- If a data type name is declared outside of a module, the data type declaration is treated lexically in the order in which it is encountered in the description. That is, the data type name is visible in any module that follows the data type declaration.

  The following example shows a data type that is declared outside of a module so that it can be used in declaring and instantiating a module with ANSI-style parameter declarations:

```
typedef enum { FALSE, TRUE } booleanT;

module test #( parameter booleanT do_random_test = FALSE );
    ...
    ...
endmodule

module top;
    test #(TRUE) t1 ();
    ...
    ...
endmodule
```

Like other forms of declaration, the name of a data type must be unique in the immediate scope in which it is declared. Two data type declarations that introduce the same name within the same scope are not allowed, even if the data types they describe are identical. This is true

even if the declarations appear in the scope by including the same file with `` `include ``. Two data type declarations introducing the same name may appear in separate compilation units.

enum Data Type

SystemVerilog introduces enumerated data type declarations. Enumerated data types let you declare a variable that has a list of valid values. Each value in the list has an associated user-defined name. You can then use these meaningful names to manipulate the values that the variable can have.

Using enumerations can make the code easier to read and to debug. Like constant names defined with `` `define `` macros, you can use the constant names in your Verilog code. However, unlike `` `define `` macros, you can also see the values as names in the waveform viewer, use the constant names in Tcl commands, and see the values as names in the output of Tcl commands.

Limitations

This section summarizes the features in the SystemVerilog standard that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- Hierarchical references to enumeration constants are not supported.
- The shorthand notations for specifying a range of names in an enumerated list (Section 4.10.2) are not supported.
- The LRM (Section 4.10) states that if a value is explicitly specified for an enumeration constant, the value is a constant expression that can include references to parameters, local parameters, genvars, other named constants, and so on. In the current release, the constant expression is restricted to be a simple number (optionally preceded by a +/-).

Declaring an Enumeration

The definition of an enumerated data type is introduced by the keyword `enum`. The definition includes a list of the enumeration constants, and, optionally, a data type and/or vector width.

As with all data types, you can define an enumeration in an object declaration, as shown in the following examples:

```
enum { clear, warning, error } status;

enum { clear, warning, error } status = clear;
```

```
enum int { overflow, underflow, io_error } error_codes;

enum logic [1:0] { overflow, underflow, io_error } error_codes;
```

You can also define an enumeration in a `typedef` declaration. For example:

```
typedef enum { overflow, underflow, io_error } error_codes_t;

typedef enum bit { FALSE, TRUE } boolean;

typedef enum bit { FALSE=1'b0, TRUE=1'b1 } boolean;

typedef enum logic [2:0] { MOVop, SUBop, ANDop, ADDop, ORop, LDop, XORop } opcode_t;
```

Depending on how you declare an enumeration, it is a bit or a bit vector.

- If you do not include a range or a data type name, the base type defaults to `int`. For example, the following declaration:

  ```
  typedef enum { overflow, underflow, io_error } error_codes_t;
  ```

  is equivalent to:

  ```
  typedef enum int { overflow, underflow, io_error } error_codes_t;
  ```

- If you specify the `logic` or `bit` type without a range, as in the following declaration, the enumeration is a scalar.

  ```
  typedef enum bit { FALSE, TRUE } boolean;
  ```

- If you specify a range, as in the following declaration, the enumeration is a vector.

  ```
  typedef enum logic [2:0] { MOVop, SUBop, ANDop, ADDop, ORop, LDop, XORop }
  opcode_t;
  ```

The following code fragments show different ways to declare the enumeration `error_code` as a 32-bit signed bit vector:

```
// The default is int
typedef enum { OVERFLOW, UNDERFLOW, MEMLIMIT, BUSERR } error_code;

// Explicitly including the data type name
typedef enum int { OVERFLOW, UNDERFLOW, MEMLIMIT, BUSERR } error_code;

// With a bit type, range, and signed specification
typedef enum logic signed [31:0] { OVERFLOW, UNDERFLOW, MEMLIMIT, BUSERR }
        error_code;

// With another user-defined data type
typedef logic signed [31:0] my_integer;
typedef enum my_integer { OVERFLOW, UNDERFLOW, MEMLIMIT, BUSERR } error_code;
```

Specifying Enumeration Constants

Enumeration constants are names for a set of constant values. These constants belong to the same namespace as the enclosing declaration. This means that if two enumeration

declarations are visible in the same place, all of their enumeration constant names must be unique.

The value represented by a name in the enumeration list is the same type as the enum type itself, which defaults to `int`. The first name in the list has a value of `0`, the second name has a value of `1`, the third name has a value of `2`, and so on. For example, in the following enumeration, the enumeration constants `suspended` and `active` have the values `0` and `1`, respectively.

```
enum { suspended, active } process_status;
```

You can explicitly declare a value for names in the enumerated list. For example:

```
typedef enum logic [2:0] {WAIT=3'b001, LOAD=3'b010, READY=3'b100} states_m;
```

If you do not specify a value for a name, the value of the previous name in the list is incremented by one. In the following example, the value of `A` is `1`, the value of `B` is `2`, the value of `C` is `5`, and the value of `D` is `6`.

```
enum {A=1, B, C=5, D} enum_list;
```

Each name in the list must have a unique value. For example:

```
enum {E=1, F, G=2, D} enum_list2;
```

causes the following compilation error, because both `B` and `C` have the value `2`:

```
ncvlog: *E,SVEDUP (test.sv,8|14): Enumeration constant 'G' has the same value as
enumeration constant 'F'. For a given enumeration type, each enumeration constant
must have a unique value.
```

You can assign the values `X` or `Z` to a name if the data type is a 4-state type. For example:

```
enum logic {H=0, I=1, J=1'bx, D=1'bz} enum_list3;
```

If you assign the value `X` or `Z` to a name, the following name must have an explicit value assigned. For example, the following is an error because `D` does not have an explicit value.

```
enum logic {K=0, L=1, M=1'bx, D=1'bz} enum_list4;
```

Treating Enumeration Objects as Bit Vectors

Enumeration data types are a special form of bit or bit vector. That is, depending on how you declare the enumerated data type, the enumeration objects are bits or bit vectors. For example, in the following declaration, the enumeration constants are scalars.

```
enum logic { OVERFLOW, UNDERFLOW } error_code;
```

In the following declaration, the enumeration constants are vectors.

```
typedef enum logic [2:0] {
        idle = 3'b001,
        read_cycle = 3'b010,
        write_cycle = 3'b100 } fsm_states;
```

```
fsm_states state = idle;
```

All vector operations and semantics apply to vector enumeration objects. For example, enumeration objects that are vectors can be the subject of bit-selects and part-selects. The normal semantics for initialization, assignment (except for type checking, see "Enumeration Type Checking"), and value conversion (sign extension and truncation) also apply. For example, the variable `state`, in the example shown above, will start simulation as all X's, and transition to the value of `idle` in the first simulation cycle.

The value of an enumeration constant with an explicit encoding is determined by the standard vector assignment semantics, as though the value were assigned to an object of that data type. If there is a width mismatch between the value and the vector width, truncation or extension of the value will occur. An error is issued only when an application of the assignment changes the value. For example:

```
enum logic[1:0]{
  WAIT = 1,
  LOAD = 2,
  READY = 4} stateA;
```

In this example, the values of `WAIT` and `LOAD` do not cause an error message because they can be represented in two bits (unsigned). However, the value of `READY` cannot be represented in this way, thus causing the following compilation error message for this enumeration constant:

```
ncvlog: *E,SVECTR (test.sv,14|10): Truncation occurred converting the enumeration
constant expression for 'READY' into a value of this enumeration type.
```

Enumeration Type Checking

SystemVerilog enumeration data types are strongly typed, in that when an `enum` object is assigned a value, the data type of that value must match the data type of the `enum` object. For example:

```
enum {s, m, l} sizes;
typedef enum {red, green, blue } colorT;
colorT color;
int i;
initial begin
 color = green;  //Valid. Both have the same data type.
 color = m; //Invalid. Data type mismatch.
...
end
```

When an `enum` object is in an expression, it is automatically converted to the underlying numeric type for the `enum`. For example:

```
i = green + 1; //green is automatically converted to int 1. Therefore, i = 2.
```

An `enum` variable cannot be assigned a value that is outside the enumeration set, unless you use an explicit cast. For example:

```
color = colorT'(1); //Valid. Static cast converts 1 to colorT.
```

Enumeration Type Methods

SystemVerilog provides a set of methods that you can use to display information about an enumeration.

Method	Description
`first()`	Returns the value of the first member in an enumeration.
`last()`	Returns the value of the last member in an enumeration.
`next()`	Based on the given value, this method returns the value of the next member in the enumeration. If the given value is the last member in the enumeration, this method returns the value of the first member. If the given value is not a member of the enumeration, this method returns the default initial value. This method also takes an optional parameter, which indicates how many values to go forward.
`prev()`	Based on the given value, this method returns the value of the previous member in the enumeration. If the given value is the first member in the enumeration, this method returns the value of the last member. If the given value is not a member of the enumeration, this method returns the default initial value. This method also takes an optional parameter, which indicates how many values to go backward.
`num()`	Returns the number of elements in an enumeration.
`name()`	Returns the string representation of the given enumeration value. If the given value is not a member of the enumeration, this method returns the default initial value.

For example:

```
module top;

typedef enum byte {seattle, austin, calcutta = 15, louisville, london = 100 }
Cities;

Cities city = city.first();
```

```
initial
 forever begin
 $display("%n has the internal value %d", city, city);
 if (city == city.last()) break;
 city = city.next();
end

initial begin
 $display("Total number of cities is %d", city.num());
 city = calcutta;
 $display("%n + 2 is %n", city, city.next(2));
 city = louisville;
 $display ("%n - 3 is %n", city, city.prev(3));
end
endmodule
```

This produces the following output:

```
seattle has the internal value    0
austin has the internal value    1
calcutta has the internal value   15
louisville has the internal value   16
london has the internal value  100
Total number of cities is           5
calcutta + 2 is london
louisville - 3 is seattle
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

In the current release, the following is not supported:

- The SystemVerilog LRM allows method calls to omit the parentheses ( ) that follow the method name. However, the Cadence implementation requires parentheses:

  ```
  city.next;    //Valid in SystemVerilog, but not in the current release
  city.next();  //Supported
  ```

- Cascaded enumeration methods are not supported. For example:

  ```
  city = city.first().next(); //Not supported
  ...
  city = city.first(); //Use this instead
  city = city.next();
  ```

- Enumeration methods are not supported for array elements or members of a structure. For example:

  ```
  Cities cityArray[10];
  city = cityArray[2].next(); //Not supported

  ...
  city = cityArray[2]; //Use this instead
  city = city.next();
  ```

- Enumeration methods are not supported for hierarchical references or OOMRs. For example:

  ```
  module m;
    enum {cold, hot} temp = hot;
  endmodule
  ```

```
module top;
  m i();
  initial $display(i.temp); //OOMR to enum variable is supported
  initial $display(i.temp.first()); //enum method on OOMR not supported
endmodule
```

causes the following error message:

```
ncelab: *E,NLMETH (./test.sv,8|30): Built-in method calls are not presently
supported on non-local objects (i.e., referenced with hierarchical names),
array elements or structure members.
```

Structures

Structures are described in Section 4.11 of the IEEE 1800 standard.

Verilog does not have a convenient way to group related data objects under a common name. SystemVerilog introduces C-like *structures* to Verilog that can group related data objects, where each member in the structure is called a *field*. Fields can be standard data types (such as `time`, `int`, and `logic`), user-defined data types, or other structures.

Structures are declared using the `struct` keyword, and a simple structure declaration looks like:

```
struct {
structure_fields;
...
} structure_name;
```

In SystemVerilog, there are two kinds of structures: *packed* and *unpacked*. By default, structures are *unpacked*. You can explicitly declare a structure as *packed* by using the `packed` keyword. A packed structure consists of bit fields that are packed together in memory without gaps. The following lists some of the differences between packed and unpacked structures:

- Packed structures, unlike unpacked structures, can be treated as normal vectors. The first member in a packed structure has the most significance, while the following members have decreasing significance. Packed structures can be indexed like normal vectors (for example, `packedStruct[3]`). Packed structures can also be used as a whole, with arithmetic and logical operators.
- Packed structures can have only integral values that can be represented as a vector (such as `int` and `byte`). Packed structures cannot contain unpacked structures, `real` or `shortreal` variables, unpacked unions, or unpacked arrays. SystemVerilog cannot pack a structure if any of the fields cannot be represented as a vector. These restrictions do not apply to unpacked structures.

You can reference the whole group of fields using the structure's name, or access a member of the structure using its field name. Structure members are accessed as follows:

<structure_name>.<field_name>

For example, the following assigns a value of 0 to `field1` in structure `test1_var`:

```
test1_var.field1 = 0;
```

Structures can contain data objects of different types and sizes. For example, the following illustrates a simple declaration for a packed structure:

```
typedef struct packed{
   integer field1;
   logic[7:0] field2;
}test1;
```

Packed Structures

The Cadence implementation supports the following functionality for packed structures:

- Packed structures can also be applied to variables, parameters, and nets. For example:

  ```
  test1 test1_var;

  parameter test1 test1_parameter = 2;

  wire test1 test1_wire;
  ```

- SystemVerilog unpacked arrays of packed structures.
- SystemVerilog packed arrays of packed structures. For example:

  ```
  wire struct packed { logic [2:0][1:0] a1; } [7:0] myArray;
  ```

- Mixing two-state and four-state packed structures. If any data type within a packed structure is 4-state, then the whole structure is 4-state. When reading 2-state members, there is an implicit conversion from 4-state to 2-state. When writing 2-state members, there is an implicit conversion from 2-state to 4-state. For example:

  ```
  struct packed { logic m1; bit m2; } v;
  ```

- Assignment patterns to members of packed structures. See "Assignment Operators" on page 115 for information on assignment patterns.

Limitations on Packed Structures

This section summarizes the features in the SystemVerilog standard that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- Out-of-module references to members of a packed structure are not supported. For example, given the following code, an out-of-module reference of the form `top.mystruct.member` is not supported.

```
module top;
  struct packed {
    int member;
  } mystruct;
endmodule
```

Declaring a Packed Structure

Packed structures are defined using the `packed` keyword. For example:

```
struct packed {
    logic m1;
    logic m2;
    } packed_test;
```

You can create a data type from a structure, using the `typedef` keyword. When you declare a structure as a user-defined type, storage is not allocated. In order for fields within a structure to store values, you must declare a variable of that data type. For example, the previous example can look like this:

```
//Structure definition
typedef struct packed {
    logic m1;
    logic m2;
    } p_test;

//Variable of the data type
p_test packed_test;
```

You can define a structure where its fields are other structures. For example:

```
struct packed {
    p_test x;
    p_test y;
} coordinate;
```

Unpacked Structures

The Cadence implementation supports the following functionality for unpacked structures:

- Unpacked structure members with the following data types:
  - Any data type allowed for members of a packed structure
  - The `real` data type
  - Unpacked structures
  - A fixed-size unpacked array data type

- Variables of an unpacked structure type. For example:

```
struct { integer i1, i2; } mystruct;
```

- Member selection for unpacked structure variables (for example, `struct.`*`member`*)
- Unpacked structures as task and function arguments and function return types.
- Unpacked structure assignments for variables. These variable assignments are subject to the type-compatibility requirements enforced by the SystemVerilog LRM. For example:

```
//Unpacked structure data types
typedef struct { real x; real y;} cartesianCoordinateT;
typedef struct { real magnitude; real angle; } polarCoordinateT;

//Variables of the unpacked structure data type
cartesianCoordinateT cartesian;
polarCoordinateT polar1, polar2;

//Conversion function
function polarCoordinateT convert(input cartesianCoordinateT x);
...
endfunction

polar1 = polar2;               //Supported
polar1 = cartesian;            //Causes compilation error due to type mismatch
polar1 = convert(cartesian);   //Supported
```

- Assignment patterns to members of packed structures. For example:

```
struct { integer age, weight; }
       alfred = '{weight: 155, age: 36 },
       ella =   '{27, 155},
       ginger = '{age:27, default:155};
```

See "Assignment Operators" on page 115 for more information on assignment patterns.

- Conditional, logical equality (==), and case equality (===) operators on unpacked structures. For example:

```
struct { byte b; real r; } v1, v2, v3;
...

// Assigns v1 to either v2 or v3, depending on the how v1 compares to v3 using
// logical equality
v1 = (v1 == v3) ? v2 : v3;
```

- Arrays of unpacked structures. For example:

```
typedef struct{
   integer length, width, depth;
} dimensionsT;

dimensionsT arrayOfDimensions[250]; //Supported
```

Using Unpacked Structures in Classes

In the current release, unpacked structures are supported within classes. The following summarizes the supported functionality:

- Unpacked `struct` variables can be declared within classes; this includes classes that are declared within packages or modules.
- User-defined unpacked `struct` types can be used within a class.
- Unpacked `struct` members can be declared within a class.
- You can assign a class' `struct` member to another class `struct` member of the same type:

```
C cc1, cc2;
cc1.st1 = cc2.st1;
```

- Within a class, you can declare a function that returns a `struct` and that has `input`, `output`, `inout`, or `ref` arguments that are structures.
- You can use the logical equality operators (== and `!=`) with class `struct` variables:

```
if (cc.s1 == cc1.st1)
```

- Unpacked structures can be declared as static members of a class:

```
class C;
  static struct {
    ...
  }st1;
endclass
```

- Unpacked structures can also be declared as public, local, or protected members of a class.
- You can nest unpacked structures within a class:

```
class C;
  struct {
   int a;
   struct {
     real b;
   }st2;
  }st1;
endclass
```

Limitations on Unpacked Structures

The following list summarizes the features in the SystemVerilog standard that are not supported in the current release.

- The following data types are not supported for members of an unpacked structure:

- `string`
- associative arrays
- dynamic arrays
- classes
- queues
- events

- The Cadence implementation does not support initializing members of an unpacked structure. For example:

```
typedef packed {
   logic [3:0] ab1 = 4'b0101;  // Not supported
   }myInitType;
myInitType myInitVar;
```

- Nets of an unpacked structure data type are not supported.

```
wire struct {integer i1, i2; } wu; //Not supported
```

- Parameters of an unpacked structure data type are not supported.

```
parameter struct {integer a1, a2; } ab; //Not supported
```

- Unpacked structure variables are not supported within constant functions
- Net ports that have an unpacked structure data type are not supported.

```
module m3(i1, i2, i3, o1, o2, o3};
   typedef struct {integer m1; logic [7:0] m2; } upsT;
   input upsT i1;       //Not supported, i1 is a net port
   input wire upsT i2;  //Not supported, i2 is a net port
   input var upsT i3;   //Supported, i3 is a variable port
   output upsT o1;      //Supported, o1 is a variable port
   output wire upsT o2; //Not supported, o2 is a net port
   output var upsT o3;  //Supported, o3 is a variable port
endmodule
```

- Out-of-module references to an unpacked structure variable or to members of an unpacked structure are not supported. This includes unpacked structures that are declared within classes.
- Unpacked structures are not supported within packages.
- Non-blocking assignments are not supported on class objects that contain unpacked structures. Non-blocking assignments are also not supported on unpacked structures or unpacked structure members that are elements of a class.
- Unpacked `struct` members cannot be used in sensitivity lists:

```
always @ (cc.st1)
```

- You cannot use class struct variables with relational operators (<, <=, >, >=).

```
if (cc.st1 < cc1.st1)
```

- You cannot use the `$bits()` function on class `struct` variables.

Debugging Structures

For information on how to debug structures using the Tcl command-line interface or the SimVision analysis environment, refer to *SystemVerilog in Simulation*.

uwire Nets

Verilog net types, such as `wire` and `trireg`, determine how the value of a net is computed from its drivers. The current release supports a new net type, `uwire`, for single-driver nets.

Note: Cadence implemented this new net type in the IUS5.4 production release, before it was approved by the IEEE for inclusion in the SystemVerilog standard. Cadence called this new net type `wone`. Beginning with the first IUS5.4 hotfix release (IUS5.4-S1), if you try to use the `wone` keyword, an error is generated telling you to use the equivalent `uwire` net type instead.

You can use the `uwire` net type to enforce a restriction that a given net has at most one driver. A `uwire` net behaves like a single-driver `wire` net. The value of a `uwire` net is the value of its driver, if it has one. If the net has no driver, its value is `Z`. During elaboration, a check is performed for multiple drivers for the `uwire` net. The elaborator generates an error if multiple drivers are detected.

A `uwire` net cannot be connected to a bidirectional terminal of a tran network.

It is an error to connect a `uwire` net to an AMS `wreal` net.

The following design defines a `uwire` net called `toggle`. It has two drivers: a gate output in module `top`, and an implicit continuous assignment for the connection to the output port of module `toggle_driver`:

```
`timescale 1ns/1ns

module top;

    uwire toggle;

    toggle_driver td (toggle);

    not #1 delay_toggle (toggle, toggle);

    initial
```

```
    begin
        $monitor($stime, "toggle = %b", toggle);
        #10 $finish;
    end
endmodule

module toggle_driver (output reg toggle = 1'b0);

    always #1 toggle = ~toggle;

endmodule
```

Because the `uwire` net has two drivers, the elaborator generates the following error:

```
ncelab: *E,UWIREM: Multiple drivers detected on a 'uwire' net (top.toggle).
```

You do not need to specify the `uwire` net type on all of the ports connected to your net. The `uwire` net type has precedence over all other net types except for `supply0` and `supply1`.

The coercion of a net to `uwire` can cause a change in behavior for some types of nets. In such cases, the elaborator issues a warning. For example, connection of a `uwire` to a `trireg` results in the following kind of warning:

```
ncelab: *W,UWIREC (./test.v,7|23): Incompatible port connection to 'uwire' net
top.toggle; 'uwire' dominates, and 'trireg' semantics are not in effect.
```

When different net types are connected through a module port, and one or both of the net types is `uwire`, the resulting net type is determined as follows:

Net Type 1	Net Type 2	Resolution	Warning?
`uwire`	`uwire`	`uwire`	No
`uwire`	`wire/tri`	`uwire`	No
`uwire`	`wand/triand`	`uwire`	Yes
`uwire`	`wor/trior`	`uwire`	Yes
`uwire`	`trireg`	`uwire`	Yes
`uwire`	`tri0`	`uwire`	Yes
`uwire`	`tri1`	`uwire`	Yes
`uwire`	`supply0`	`supply0`	No
`uwire`	`supply1`	`supply1`	No

You can use the `` `default_nettype `` compiler directive to make the `uwire` net type the default for all of your nets.

Casting

Casting is described in Section 4.14 of the IEEE 1800 Standard.

SystemVerilog adds the ability to change the data type of a value using a cast (`'`) operation. The current release supports a subset of the functionality described in the LRM.

In the current release, type casts have the following syntax:

`simple_type` `'` (*`expression`*)

where

- *`simple_type`* can be a user-defined data type or a built-in data type (such as `int` and `time`).
- *`expression`* is an arbitrary expression.

In this type of cast, the value of *`expression`* is converted to the data type specified by *`simple_type`*.

For example:

```
typedef enum { red=0, green=1, blue=2} colorT;
colorT v;
v = colorT'(1); //Type cast
$display("v: %s, v.name()); //Displays the value 'green'
```

In this example, the type cast `colorT' (1)` converts the value `1` to the data type `colorT`, which is equivalent to `green`.

Another example:

```
typedef logic [3:0] vec4T;
reg [7:0] vec8V;
vec8V = 8'b11111111;
$display("vec8v truncated to vec4T: %b", vec4T'(vec8V)); // Displays value "1111"
```

In this example, the static type cast `vec4T'(vec8V)` converts the value of `vec8V` (namely, `8'b11111111`) to a vector of width 4.

In the current release, type casts have the following limitations:

- *`simple_type`* and *`expression`* must be an integral data type. Integral data types include `shortint`, `int`, `longint`, `byte`, `bit`, `logic`, `reg`, `integer`, `time`, packed array data types, packed structure data types, or enumeration data types.
- *`simple_type`* cannot be a reference to a type parameter.

Limitations

The SystemVerilog data type extensions are described in Chapter 4 of the IEEE 1800 standard. Section 4.2 shows the data type syntax. This section summarizes the syntax constructs in the SystemVerilog LRM that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- The following data type alternatives are not supported in the current release:
  - union
  - ps_covergroup_identifier
- `shortreal` is not allowed as a non_integer_type.

6

Arrays

Arrays are described in Section 5 of the IEEE 1800 standard.

Arrays are used to hold elements of a declared data type. In Verilog-2001, arrays can be multidimensional and can be declared for all data types. Verilog-2001 arrays use the following syntax:

data_type vector_width array_name array_dimension

For example:

```
reg [4:0] student_id [0:7]; // Array of 8 student_ids. Each student_id is
                            // 5 bits wide.
integer msk[0:63];          // Array of 64 integer values
wire array_w[7:0][5:0];     // Multidimensional array of wires
```

In SystemVerilog, array declarations use the following syntax:

data_type packed_dimensions array_name unpacked_dimensions

where

- *packed_dimensions*—Refers to the dimensions just before the name of the array.
- *unpacked_dimensions*—Refers to the dimensions just after the name of the array.

For example:

```
integer chk [3:0][7:0][3:0];  // Multidimensional unpacked array
bit [7:0][3:0] chk1;          // Multidimensional packed array
```

SystemVerilog enhances arrays by:

- Allowing unpacked arrays of any data type (including the `event` data type).
- Allowing multidimensional packed arrays.
- Adding *dynamic arrays*, which provide the ability to change the size of one of the dimensions in an unpacked array. Storage for a dynamic array is allocated during simulation.

- Adding *associative arrays*, which are best used when the size of a collection of variables is unknown or when data space is limited. Storage for an element in an associative array is not allocated until that element is accessed.
- Adding *queues,* which are used to collect elements of a declared data type. Queues are declared like arrays, but use $ for the range.

Note: The LRM describes “Array Manipulation Methods” in Section 5.15. The current release does not support array manipulation methods.

Packed and Unpacked Arrays

Packed and unpacked arrays are described in Section 5.2 of the IEEE 1800 standard.

In Verilog, vectors can consist of single-bit data types, such as `reg` or `wire`, and the vector range is declared just before the signal name. In SystemVerilog, vector declarations are called *packed arrays*. For example, the following declares a packed array called `addr` that is 41 bits wide:

```
reg [0:40] addr; // One-dimensional packed array
```

Packed arrays are used to divide a vector into its subfields so that the subfields can be accessed as array elements. Packed arrays can be made up of single-bit types, other packed arrays, or packed structures. Packed arrays are always represented as a contiguous set of bits.

SystemVerilog enhances arrays in that you can declare multidimensional packed arrays. For example, the following declares an array of four 6-bit sub-arrays:

```
bit [3:0][5:0] addr2; // Two-dimensional packed array
```

The Cadence implementation supports the following functionality for packed arrays:

- Packed arrays whose element type is another packed array. Specifically, the Cadence implementation supports multidimensional vectors.

  ```
  logic [8:1] [16:1] v1 [63:0] [31:0];
  reg [128:1] x;
  ...
  x = v1[4][7];
  ...
  ```

- Packed arrays of enumerations. In the following example, p is a packed array of enumeration vectors.

  ```
  typedef enum bit [7:0] {x, y, z} t;
  parameter t [31:0] p = 0;
  ```

- Packed structures with packed array members. In the following example, `w1` is a packed structure with a multidimensional packed array member:

```
wire struct packed {
logic [2:0][1:0] m;
} w1;
```

- Packed arrays of packed structures. In the following example, `w3` is a packed array of packed structures, where each structure has a two-dimensional packed array member.

```
wire struct packed {
logic [2:0][1:0] m;
} [7:0] w3;
```

- Verilog arrays of packed arrays.

- Assignment patterns for assigning to elements of a packed array. See "Assignment Patterns" on page 116 for information on assignment patterns.

Unpacked arrays refer to the dimensions that come right after the array name, which is similar to the Verilog style of array declarations. Unlike packed arrays, an unpacked array can be of any type, and may or may not be represented as a contiguous set of bits. For example:

```
wire n [0:32];        // One-dimensional unpacked array of 33, one-bit nets
int m [7:0][3:0]; // Two-dimensional unpacked array of 32-bit int variables
```

The Cadence implementation supports the following functionality for unpacked arrays:

- Unpacked arrays whose size is specified by a single, positive number.

  Instead of a range, SystemVerilog allows the use of a single, positive number to denote the size of an unpacked array, where `[size]` is the same as `[0:size-1]`. For example:

```
logic v[4];                  // Same as: logic v[0:3];
wire [5:0] w[10][0:4][20];   // Same as: wire [5:0] w[0:9][0:4][0:19];

parameter aSize = 3000;
typedef int myArray[aSize];  // Same as: typedef int myArray[0:aSize-1];

logic v1[0];                 // Illegal. Array size must be positive.
```

  Note: This feature is not available for packed array dimensions. For example, the following declaration is invalid:

```
logic [8] eightBits;         // Invalid. [8] is a vector dimension.
```

- Assignments between unpacked array variables. The current release enforces the SystemVerilog type-compatibility requirements for assignments to unpacked arrays. For example:

```
integer fourintA[4];
integer fourintB[4];
integer fiveint[5];
int fourint[4];
```

```
...
task showSum(input integer x[4]);
...
endtask
...
fourintA = fourintB;  //Valid
showSum(fourintB);    //Valid
fourintA = fiveint;   //Invalid
showSum(fourint);      //Invalid
```

- Unpacked arrays as arguments to tasks and functions, and as function return types. For example, the following function accepts an unpacked array input argument and returns an unpacked array function.

```
typedef reg [31:0] registerSetT[16];
registerSet myregs;
function registerSetT negateRegisters(input registerSetT regs);
...
endfunction
...
myRegs = negateRegisters(myRegs);
```

- Conditional, logical equality (==), and case equality (===) operations on unpacked arrays.
- Assignment patterns for assigning to elements of an unpacked array. For example:

```
int upa [0:3];
intial upa = '{0,1,2,3}; //Assignment to an unpacked array
```

  See “Assignment Patterns” on page 116 for information on assignment patterns.

- Unpacked arrays that are involved in assignments; function and task argument passing and value return; and conditional and equality operations cannot be given by an out-of-module reference, or be variable inside a class.

Limitations on Packed and Unpacked Arrays

This section summarizes the features in the SystemVerilog standard that are not supported in the current release.

- In Verilog, you can select only a single element of an array. SystemVerilog enhances arrays by allowing the selection of one or more contiguous elements of an array. This selection is called a *slice*. Selecting an array slice in SystemVerilog is similar to performing a constant part select or indexed part select in Verilog.

  The Cadence implementation supports only SystemVerilog slices of packed arrays, where the array is a one-dimensional array of scalars. The implementation supports slicing only the last dimension of a multidimensional vector.

  The implementation does not support slices on unpacked arrays.

For example:

```
module top;
  logic [7:0][4:0] mdv;
  logic arr [9:0];
  logic [1:0] twoScalars;
  logic [9:0] tenScalars;

  initial begin
    twoScalars = mdv[7][1:0];  // Valid, slice of a one-dimensional
                               // packed array of scalars
  tenScalars= mdv[7:6];        // Invalid, slice does not occur
                               // at last dimension
  twoScalars= arr[7:6];         // Invalid, slice of an unpacked array
  end
endmodule
```

Array Querying Functions

Array querying functions are described in Section 5.5 of the IEEE 1800 standard.

The current release supports the following SystemVerilog system functions, which are used to return information about the dimensions of a given array or integral data type, or of data objects of such a data type.

In the current release:

- Array query functions are supported for fixed arrays and integral data types. They are not supported for dynamic arrays.
- Array query functions can be used in class objects and to access class properties. They are also supported for packed arrays in class objects. However, they are not supported for unpacked arrays in class objects.

`$left`	Returns the left bound (most significant bit) of the dimension.
`$right`	Returns the right bound (least significant bit) of the dimension.
`$low`	Returns the minimum of `$left` and `$right` of the dimension.
`$high`	Returns the maximum of `$left` and `$right` of the dimension.
`$increment`	Returns 1 if `$left` is greater than or equal to `$right`, and `-1` if `$left` is less than `$right`.

`$size`	Returns the number of elements in the dimension. This is also equal to `$high - $low + 1`.
`$dimensions`	For packed, unpacked, dynamic, and static arrays, this function returns the number of dimensions in the array. For the `string` data type, and other non-array data types that are equivalent to simple bit vector types, this function returns 1. For all other types, this function returns zero. **Note:** In the current release, you cannot use array querying functions with dynamic arrays.
`$unpacked_dimensions`	For static and dynamic arrays, this function returns the number of unpacked dimensions. For all other types, this function returns zero. **Note:** In the current release, you cannot use array querying functions with dynamic arrays.

Note the following for array-querying functions:

- The `$dimensions` and `$unpacked_dimensions` functions take one argument—the array identifier. For example:

  ```
  //Returns the number of dimension for myarr, or zero if it is integral
  a = $dimensions(myarr);
  ```

  All of the other functions take two arguments: the array identifier (required) and dimension (optional). For example:

  ```
  //Returns the most significant bit of myarr's second dimension
  a = $left(myarr, 2);
  ```

  Note: If the dimension is not specified, it defaults to 1. If you specify a dimension that is out of range, this function returns `x`.

- All of these functions return an integral data type.
- When used with fixed arrays, these functions can act as constant functions and can be passed as an elaboration parameter.

For example, the following code:

```
module test();

  logic [6:9] [10:14] word;
  parameter p = $left(word, 2);
  int a = $left(word, 1) ;
```

```
  initial begin
    void'($size(word, 2));
    $display("a = %d \n", a);
    $display("Size = %d \n", $size(word, 1));
    $display("Left bound = %d \n", $left(word, 1));
    $display("Right bound = %d \n", $right(word, 1));
    $display("Low = %d \n", $low(word, 1));
    $display("High = %d \n", $high(word, 1));
    $display("Increment = %d \n", $increment(word, 1));
    $display("Dimensions = %d \n", $dimensions(word));
    $display("Unpacked dimensions = %d \n", $unpacked_dimensions(word));
  end
endmodule
```

produces the following simulation results:

```
a =                            6
Size =                         4
Left bound =                   6
Right bound =                  9
Low =                          6
High =                         9
Increment =                   -1
Dimensions =                   2
Unpacked dimensions =          0
```

Dynamic Arrays

Dynamic arrays are described in Section 5.6 of the IEEE 1800 standard.

SystemVerilog enhances Verilog arrays with the addition of *dynamic arrays*. A dynamic array is one dimension of an unpacked array, whose number of elements can be set or changed during simulation. Storage for a dynamic array is allocated during simulation. The syntax for dynamic array declarations is as follows:

```
data_type array_name[];
```

For example:

```
int x[];         // Dynamic array of ints
bit [4:0] y[];   // Dynamic array of 5-bit vectors
string dynstr[]; // Dynamic array of strings

typedef int da[];// User-defined dynamic array
da d;
```

Access Methods for Dynamic Arrays

SystemVerilog offers the following built-in methods for use with dynamic arrays:

- `new[]`—A function that creates a dynamic array of the specified size and initializes the newly-created array elements with the elements of the specified array, or to their initial default value. The syntax for the `new[]` function is as follows:

  ```
  dyn_array = new [size][(old_dyn_array)]
  ```

  where `[size]` is an expression that specifies the number of elements in the array, and must be a non-negative integral expression. The index of a dynamic array is always `[0:size-1]`.

  `[old_dyn_array]` is an optional argument. When specified, the elements of *`dyn_array`* are initialized to the elements of *`old_dyn_array`*. Otherwise, the elements of *`dyn_array`* are initialized to their initial default value. *`old_dyn_array`* must be a dynamic array of the same type as *`dyn_array`*, but can have a different size.

  For example:

  ```
  integer myaddr[];     // Declares the dynamic array
  myaddr = new[50];     // Creates a 50-element array, with an index of 0 to 49,
                        // and array elements are initialized to x.
                        // The index of a dynamic array is always [0:size-1].
  myaddr= new[60](myaddr); //Resizes the array, while preserving its previous
                           content
  ...
  integer newaddr[];
  newaddr = new[70](myaddr); // Copies the content of myaddr into newaddr
  ```

- `size()`—A method that returns the size of the dynamic array.

  The built-in `size()` method returns the current size of the dynamic array, and returns zero if the array is empty. The syntax for the `size()` method is as follows:

  ```
  array_name.size()
  ```

  For example:

  ```
  int ab[];
  ...
  ab = new[10];               // Creates a 10-element array
  $display(“%d”, ab.size());  // Displays 10
  ```

- `delete()`—A method that removes all storage for a dynamic array.

  The built-in `delete()` method removes all storage within a given dynamic array, resulting in an empty array. The syntax for the `delete()` method is as follows:

  ```
  array_name.delete()
  ```

  For example:

  ```
  ab.delete();                // Deletes the array created in the last example
  $display(“%d”, ab.size());  // Displays zero
  ```

Accessing Out-of-Bound Elements of a Dynamic Array

If you try to write to an out-of-bound index of a dynamic array, the simulation issues a warning message. If you try to read an out-of-bound index of a dynamic array, the simulator does not issue an error message, but displays the default value. For example:

```
module top;
  string dyn_arr[]; //Dynamic array of strings
  initial begin
    dyn_arr = new[10];
    dyn_arr[0] = “ABC”;
    $display(“value = %d\n”, dyn_arr[0]); //Displays ABC
    #5;
    dyn_arr[13] = “ABC”; //Writes to an out-of-bound index. Causes warning.
    #display(“value=%d\n”, dyn_arr[14]); //Reads an out-of-bound index.
                                         //Displays a null string, which
                                         //is the default value for strings.
    end
endmodule
```

Passing Dynamic Arrays by Reference to Tasks and Functions

Passing a piece of a dynamic array by reference to a task or function is not supported. You can pass only whole objects by reference to a task or function.

```
module top;
  string dyn_ar[];
  initial begin
   dyn_arr = new[10];
   func(dyn_arr); //Calls a time-consuming task
   $display(“size = %d\n”, dyn_arr.size());
  end
  initial
  $monitor(“value - %s time =”, dyn_arr[3], $time);
  task automatic func(ref string dyn_arr[]); //Passes dynamic array by reference
   dyn_arr[3] = “abc”; //Task changes the value of dyn_array
                       //The new value is visible to the module
   #2;
   dyn_arr = new[20]; //Resizes the array
   #3;
   dyn_arr.delete(); //Deletes the dynamic array
   #5;
  endtask
endmodule
```

This example should produce the following simulation results:

```
value-    abc time =     0
value-        time =     2
size =    0
```

Fixed Arrays of Dynamic Arrays

You can also declare a fixed array of dynamic arrays. For example, the following declares a fixed array called `fa_da` where each element is a dynamic array, and each element in the dynamic array is of type `int`:

```
int fa_da[0:3][]; // Fixed array of dynamic arrays
```

For example:

```
module top;
   string fada[0:3][]; //Declares a fixed array of dynamic arrays
   int i,j,k;

initial begin
  k=100;
   for(i=0;i<2;i++)
   begin
    fada[i]=new[4]; // Allocates a dynamic array of size 4
    $display("Value is %d",fada[i].size()); //Displays the size of fada
    for(j=0;j<2;j++)
     begin
      fada[i][j]="Hello";
       $display("Value is %s",fada[i][j]); //Displays the value of fada
     end
   end
end
endmodule
...
ncsim> run
Value is 4
Value is Hello
Value is Hello
Value is 4
Value is Hello
Value is Hello
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

Limitations on Dynamic Arrays

This section summarizes the features in the SystemVerilog standard that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- The Cadence implementation supports dynamic arrays for:
  - `bit`
  - `logic`
  - `byte`
  - `shortint`

- `int`
- `longint`
- `integer`
- `string`
- enumerated data types
- packed arrays of `bit`, `logic`, and `reg`
- classes

The Cadence implementation does not support the following complex data types for dynamic arrays: structures, dynamic arrays within a structure, and multidimensional arrays.

- Dynamic arrays are supported:
  - On module ports
  - At the top level of a module, interface, package, class, or program block
  - In global compilation units
  - Within tasks, functions, and class methods

  Dynamic arrays are not supported within structures. For example, if a structure is defined within a module, a dynamic array declaration within the structure is not currently allowed.
- Dynamic arrays are also supported within `begin...end` blocks that are in the following:
  - Modules, program blocks, and interfaces. This includes `if...else`, `while`, and `for` loops declared within these scopes.
  - Tasks and functions
  - `for generate, if generate, case generate`
  - `fork...join`, `repeat`, and `forever` loops
  - `always` blocks
  - `final` statements
  - `casex` and `casez` statements
  - Named blocks
- Dynamic arrays can also be declared as public, `static`, `local`, or `protected` members of a class that is declared within a package.

- Multidimensional dynamic arrays are not supported. For example:

```
reg [0:7] arr[];    // Supported
int int_array[];    // Supported
reg [0:7] arr[][];  // Unsupported
int int_arr[][];    // Unsupported
```

- Assigning a dynamic array to a dynamic array of a different element type is not supported:

```
int da_int;
int da_int1;
bit [0:31] da_bit[];
...
da_int = da_int1; //Supported
da_int = da_bit; //Not supported
```

- Assigning a fixed-size array to a dynamic array is not supported in the current release. For example:

```
module test_top;
  int arrA[0:7];
  int dynarr[];
  initial begin
    dynarr  = new[8];
    dynarr  = arrA;    // Unsupported
  end
endmodule
```

- Assignments made to a part select of a dynamic array are not supported in the current release.

```
module test_top;
  int arrA[0:7];
  int dynarr[];
  int dynarr1[];
  initial begin
    dynarr  = new[8];
    dynarr1 = new[8];
    dynarr[0:2]  = arrA[0:2];    // Unsupported
    dynarr1[0:2] = dynarr[0:2];  // Unsupported
  end
endmodule
```

- The Cadence implementation specifies an upper limit of `(2**31-1)` for dynamic arrays. In the following example, the parser issues a warning if N is more than `(2**31-1)`. In this case, the elaborator and simulator will continue with undefined behavior.

```
dyn_arr = new[N];
```

- Dynamic arrays of unpacked structures are not supported.

```
typedef struct {
  integer a, b, c;
}dimensionsT;

dimensionsT dynUpacked[]; //Not supported
```

- The Cadence implementation does not support OOMRs or hierarchical references. You can reference a dynamic array only from the scope in which it was declared. However, if a dynamic array is declared within a package, you can reference it from other scopes.
- Passing a part select or element of a dynamic array by reference to a task or function is not supported. You can pass only whole objects by reference to a task or function. However, the dynamic arrays cannot be passed as `input`, `output`, or `inout` types to a task or function. For example:

  ```
  task automatic func(ref dynarr[]); //Legal
  task automatic func(ref dynarr[0:2]); //Not supported
  task automatic func(ref dynaarr[0]); //Not supported
  ```

- You cannot use an event control on an entire dynamic array or on a part select of a dynamic array. However, you can use an event control on an individual element of a dynamic array:

  ```
  int da[];
  always @(da)        //On entire array. Not supported.
  always @(da[0:2])   //On part select of the array. Not supported.
  always @(da[0])    //On an array element. Supported.
  ```

  You cannot use an event control on a part select of a fixed array of dynamic arrays.
- All of these limitations apply also to dynamic arrays of strings and classes.

Associative Arrays

Associative arrays are described in Section 5.9 of the IEEE 1800 standard.

SystemVerilog enhances Verilog arrays with the addition of *associative arrays*. An associative array is declared using a data type as its special array size. The syntax for declaring an associative array is as follows:

data_type array_id [*index_type*]

where:

- *data_type* is the data type of the array elements.
- *array_id* is the array name.
- *index_type* specifies the data type that will be used as an index.

The following are examples of associative array declarations:

```
logic my_array[integer];  //Associative array with an integer index
...
typedef int foo;
foo myfoo[int];           // Associative array constructed from a typedef

typedef int a_t[string];  // User-defined associative array
a_t a;
```

Storage for members of an associative array is allocated when that member is created. Associative arrays are best used when you have limited data space, or when the size of the collection of variables is unknown.

For a list of unsupported features, refer to "Limitations for Associative Arrays" on page 90.

Access Methods for Associative Arrays

This section uses the following example to describe the built-in methods that provide access into associative arrays:

```
module MyMod;
  int myArr[byte];
  int myVar;
  byte myIndex;
  byte i;
  initial begin
    myVar = 5;
    myIndex = 45;
    myArr[34] = 1;
    myArr[-66] = myVar;
    myArr[myIndex] = 13;
  end
endmodule
```

- `num()` method—Returns the number of entries for an associative array. Returns zero for empty arrays. For example, the following prints "`3 items in my array`":

  ```
  myVar = myArr.num();
  $display("%d items in my array", myVar);
  ```

- `delete()` method—Removes elements in an associative array. The syntax for this method is as follows:

  `array_name``.delete([input `*`index`*`]);`

  where *`index`* is an optional index of the appropriate type. If *`index`* is specified, then the `delete()` method removes the entry at the specified index. If *`index`* is unspecified, then the `delete()` method removes all of the elements in the array.

  For example:

  ```
  myArr.delete(34); //Deletes entry whose index is 34
  ```

- `exists()` method—Indicates whether an array element exists for a particular index. Returns 1 if the element exists; otherwise, it returns 0. The syntax for this method is as follows:

  ```
  array_name.exists(input index);
  ```

  where *index* is an index of the appropriate type.

  For example, the following returns displays “Index 45 exists”, because an element exists with an index of 45:

  ```
  if (myArr.exists(45))
    $display(“Index 45 exists”);
  else
      $display(“Index 45 does not exist”);
  ```

- `first()` method—Assigns to the given index variable the value of the first or smallest index in the associative array. Returns 0 if the array is empty; otherwise, the method returns 1. The syntax for this method is as follows:

  ```
  array_name.first(ref index);
  ```

  where *index* is an index of the appropriate type.

  For example:

  ```
  if (myArr.first(i));
    $display ("The first index of the array is %d", i);
  ```

- `last()` method—Assigns to the given index the value of the last or largest index in the associative array. Returns 0 if the array is empty; otherwise, the method returns 1. The syntax for this method is as follows:

  ```
  array_name.last(ref index);
  ```

  where *index* is an index of the appropriate type.

  For example:

  ```
  if (myArr.last(i));
    $display ("The last index of the array is %d", i);
  ```

- `next()` method—Locates an entry with an index that is greater than the specified index. If it finds an entry, the method assigns the index of the located entry to the index variable, and then returns 1. If it cannot find an entry, the method returns 0, and the index variable is unchanged. The syntax for this method is as follows:

  ```
  array_name.next(ref index);
  ```

  where *index* is an index of the appropriate type.

  For example:

  ```
  if (myArr.first(i));
    do
     $display ("%d in the current index", i);
    while
      (myArr.next(i));
  ```

- `prev()` method—Locates an entry with an index that is smaller than the specified index. If it finds an entry, the method assigns the index of the located entry to the index variable, and then returns 1. If it cannot find an entry, the method returns 0, and the index variable is unchanged. The syntax for this method is as follows:

  ```
  array_name.prev(ref index);
  ```

  where *`index`* is an index of the appropriate type.

  For example:

  ```
  if (myArr.last(i));
    do
     $display ("%d in the current index", i);
    while
     (myArr.prev(i));
  end
  ```

Limitations for Associative Arrays

This section summarizes the features in the SystemVerilog standard that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- Although the LRM states that an associative array can use any data type allowed for fixed-size arrays, the Cadence implementation supports only the following:
  - `int`
  - `integer`
  - `logic`
  - `bit`
  - `reg`
  - `shortint`
  - `byte`
  - `longint`
  - `time`
  - `class`
  - `string`
  - enumerated data types
  - User-defined packed types

The Cadence implementation does not support the following complex data types for associative arrays: structures, multidimensional arrays, events, and queues.

- The Cadence implementation supports associative array declarations on module ports and in modules, interfaces, classes, packages, programs, tasks, functions, global compilation units, and class methods.

  Associative arrays are not supported within generates (unless they are enclosed in a `begin...end` block).

- Associative arrays are also supported within `begin...end` blocks that are in the following:

  - Modules, program blocks, and interfaces. This includes `if...else`, `while`, and `for` loops declared within these scopes.

    Associative arrays are not supported in any other sub-scopes of a module, interface, or program.

  - Tasks and functions
  - `for generate, if generate, case generate`
  - `fork...join`, `repeat`, and `forever` loops
  - `always` blocks
  - `final` statements
  - `casex` and `casez` statements
  - Named blocks

- Associative arrays can also be declared as public, `static`, `local`, or `protected` members of a class that is inside a package, module, interface or program block.

- The Cadence implementation supports strings, integers, user-defined packed types, class handles, and wildcards (*), as index types to associative arrays. However, the user-defined types described in LRM Section 5.9.7 are not supported. The Cadence implementation does not support complex user-defined types.

- The Cadence implementation supports limited indexing into associative arrays, through bit-selects. A bit select can be a constant or dynamic value. For example:

  ```
  a = myArray[0];
  myArray[1] = a;
  ```

  If you try to read a value that does not exist, the simulator returns the default initial value for the array type (as specified in Table 5-1 of the LRM). If you specify an invalid index during a write operation, the simulator ignores the write.

- Section 5.13 of the LRM describes a concatenation syntax and specifying default value for associative arrays. These features are not supported in the current release.

```
int a1[int] {default:1};  // Unsupported
```

- The Cadence implementation does not support OOMRs or hierarchical references. You can reference an associative array from only the scope in which it was declared. However, if an associative array is declared within a package, you can reference it from other scopes.
- In the Cadence implementation, you can update values in an associative array through either calls to the methods discussed in "Access Methods for Associative Arrays" on page 88, or through standard blocking assignments. You can only use blocking assignments to assign a single bit-select item; blocking assignments cannot be a part of a complex expression, such as concatenation or part selects.

  The Cadence implementation does not support other methods of updating values, which include forces, continuous assignments, or non-blocking assignments.
- Passing a piece of an associative array by reference to a task or function is not supported. You can pass only whole objects by reference to a task or function.
- Assignments made to a part select of an associative array are not supported in the current release. For example:

```
int aa[int];
int aa1[int];
initial begin
  aa[0:3] = aa1[0:3]; //Not supported. Part-select assignment.
end
```

- You cannot use an event control on an entire associative array or on a part select of an associative array. However, you can use an event control on an individual element of an associative array:

```
int aa[byte];
always @(aa)        //Not supported.On entire array.
always @(aa[0:2])   //Not supported.On part select of the array.
always @(aa[0])     //Supported.On an array element.
```

Queues

Queues are described in Section 5.14 of the IEEE 1800 standard.

For examples that you can download and run, refer to the *SystemVerilog Engineering Notebook*.

SystemVerilog introduces the queue construct, which is a variable-sized collection of elements of a declared data type. Queues are similar to one-dimensional, unpacked arrays whose size can increase or decrease automatically.

A queue is declared like an array, but uses a dollar sign ($) for its range. The maximum size of an array can be limited by specifying an optional *constant_expression* as its last index. The syntax for declaring a queue is as follows:

data_type queue_id [$[:*constant_expression*]]

For example:

```
int myArray[0:63];   // A standard array of 64 integers
int myQueue[$];      // A queue of integers
bit q1[$:63];        // A queue with a maximum size of 64 bits

typedef int foo;     // A queue constructed from a typedef
foo q2[$];

typedef int q_t[$];  // A user-defined queue
q_t q;
```

Queue members are identified by numbers that represent their position in the queue. Zero represents the leftmost (first) member of the queue, and $ represents the rightmost (last) member.

Access Methods for Queues

This section describes the built-in methods that provide access into queues.

- `size()` method—Returns the number of elements in a queue. Returns 0 for empty queues. For example:

  ```
  int ww55[$];
  q = ww55.size();  // Returns 0
  ```

  Note: When used in a constraint expression, the built-in `.size()` method of a queue has the appearance of a state variable. The constraint will not set the size of a `rand` queue. This is a limitation outlined by the LRM.

- `insert()` method—Inserts the specified item at the specified index position. For example, the following inserts element `0` at position `15` within the queue called `ww55`.

  ```
  ww55.insert(0,15);
  ```

- `delete()` method—Deletes the element at the specified position. For example, the following deletes the element at position `0` within the queue called `ww55`:

  ```
  ww55.delete(0);
  ```

  The argument to the `delete()` method is optional. If the argument is missing, then the entire contents of the queue is deleted.

```
ww55.delete();      //Deletes contents of myq
```

- `pop_front()` method—Removes and returns the first element in a queue. For example, the following removes and returns the first element in `ww55`:

  ```
  q = ww55.pop_front();
  ```

- `pop_back()` method—Removes and returns the last element in a queue. For example, the following removes and returns the last element in `ww55`.

  ```
  q = ww55.pop_back();
  ```

- `push_front()` method—Inserts the specified element at the beginning of the queue. For example, the following inserts `1` at the beginning of queue `ww55`:

  ```
  ww55.push_front(1);
  ```

- `push_back()` method—Inserts the specified element at the end of the queue. For example, the following inserts `15` at the end of queue `ww55`:

  ```
  ww55.push_back(15);
  ```

Limitations for Queues

This section summarizes the features in the SystemVerilog standard that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- The Cadence implementation supports only the following data types for queues:
  - `int`
  - `integer`
  - `logic`
  - `bit`
  - `reg`
  - `shortint`
  - `byte`
  - `longint`
  - `time`
  - `class`
  - `string`
  - enumerated data types

- User-defined packed types

  The Cadence implementation does not support the following complex data types for queues: structures, multidimensional arrays, and events.

- The Cadence implementation supports queue declarations on module ports and within modules, interfaces, classes, packages, program blocks, tasks, functions, global compilation units, and class methods. Queues are not supported within generates (unless enclosed in a `begin...end` block).

- Queues are also supported within `begin...end` blocks that are in the following:

  - Modules, program blocks, and interfaces. This includes `if...else`, `while`, and `for` loops declared within these scopes.

    Queues are not supported in any other sub-scopes of a module, interface, or program.

  - Tasks and functions

  - `for generate, if generate, case generate`

  - `fork...join`, `repeat`, and `forever` loops

  - `always` blocks

  - `final` statements

  - `casex` and `casez` statements

  - Named blocks

- Queues can also be declared as public, `static`, `local`, or `protected` members of a class that is inside a package, module, interface or program block.

- The Cadence implementation supports limited indexing into queues, through bit-selects. A bit select can be a constant or dynamic value. For example:

  ```
  a = myQueue[0];
  myQueue[1] = a;
  ```

  If you try to read a value that does not exist, the simulator returns the default initial value for the queue (as specified in Table 5-1 of the LRM). If you specify an invalid index during a write operation, the simulator ignores the write.

- Section 5.14.1 of the LRM specifies a concatenation-like syntax for initializing queues and part-select operations. The Cadence implementation does not support these features. For example, the following is not supported:

  ```
  int q1[$] = {1,2,3};  // Invalid
  ```

- The Cadence implementation does not support OOMRs or hierarchical references. You can reference a queue only from the scope in which it was declared. However, if the queue is declared within a package, you can reference it from other scopes.

- In the Cadence implementation, you can update values in a queue through either calls to the methods discussed in "Access Methods for Queues" on page 93, or through standard blocking assignments. You can use blocking assignments to assign only a single bit-select item; blocking assignments cannot be a part of a complex expression, such as concatenation or part selects.

  The Cadence implementation does not support other methods of updating values, which include forces, continuous assignments, or non-blocking assignments.

- Passing a piece of a queue by reference to a task or function is not supported. You can pass only whole objects by reference to tasks and functions.

- You can pass a queue or a part select of a queue by value to: a task, function, method, or another queue that has the same element type and maximum size.

```
package p;
  typedef reg [0:45] q_type[$];  //queue typedef
   function void copy_print_q (q_type q); //passes queue by value to a function
   ...
   endfunction
...
endpackage
```

- You can use $ in operations on a queue's index. For example:

```
myQueue[$+1] = 1;
e = myQueue[$-1];
```

  However, the following is not supported:

```
myqueue = myqueue {myqueue[0:pos-1], e, myqueue[pos:$]};
          //Not supported. $ in concatenation expressions.
```

- Assigning an entire queue to another queue is supported, but they must be compatible (in that they have the same element type and maximum size):

```
int q[$];
int q1[$];
...
q = q1; //Whole assignment.
```

  Note: Entire queues are supported only in assignments. You cannot use entire queues in other places like a system task, system function, or in a sensitivity list.

- Part selects of a queue are not supported on the left-hand side of an assignment. Part selects of queues are supported on the right-hand side of an assignment, provided the element types on the right and left sides are equivalent.

```
int q[$];
int q1[$:9];
```

```
int q2[$];
...
q = q1[1:7]; //Illegal. Maximum sizes are different.
q[0:3] = q1[0:3]; //Not supported. Part select on left side.
q = q2[0:5]; //Supported. Right side and of equivalent types.
```

- Indexed part selects of queues are not supported.
- You cannot use an event control on an entire queue or on a part select of a queue. However, you can use an event control on an individual element of a queue.

```
int q[$];
always @(q)      //Not supported. On entire queue.
always @(q[0:2]) //Not supported. On part select.
always @(q[0])   //Supported. On an element of the queue.
```

Array Locator Methods

Array locator methods are described in Section 5.15 of the IEEE 1800 Standard.

SystemVerilog provides array locator methods, which are built-in methods used to search through the indices of an array for a given expression. In SystemVerilog, array locator methods operate on any unpacked array, including queues, but have a queue return type.

Note: In the current release, array locator methods operate only on queues.

Array locator methods have the following syntax:

queue_id.*array_method* (*iterator_argument*) with (*expression*)

where:

- *queue_id* specifies the queue to traverse.
- *array_method* specifies the array method.
- *iterator_argument* specifies the name of the variable to use in the `with` expression. This variable is implicitly declared and its scope is the `with` expression. This variable is optional. If you do not designate a variable, the method uses the default name `item`.
- *expression* is used to iterate over the elements of the given array.

For example:

```
int IQ[$], qi[$];

qi = IQ.find(x) with (x > 5);       //Searches for all elements greater than 5
qi = IQ.find_index with (item == 3); //Searches for indices of items equal to 3
```

Table 6-1 Array Locator Methods

Method	Description
`find()`	Finds all of the elements that satisfy the given expression. Traverses the array from front to back.
`find_first()`	Finds the first element that satisfies the given expression. Traverses the array from front to back.
`find_first_index()`	Returns the index of the first element that satisfies the given expression. Traverses the array from front to back.
`find_index()`	Returns the indexes of all the elements that satisfy the given expression. Traverses the array from front to back.
`find_last()`	Returns the last element that satisfies the given expression. Traverses the array from back to front.
`find_last_index()`	Returns the index of the last element that satisfies the given expression. Traverses the array from back to front.

Example 6-1 Using Array Locator Methods

```
module top;
  int IA[$], qi[$];
  initial begin
      IA.push_front(100);
      IA.push_back(100);
      IA.push_back(70);
      qi = IA.find with (item > 70); //Returns a queue of 100,100
      qi = IA.find_first with (item > 50); //Returns a queue of 100
      qi = IA.find_last with (item > 50); //Returns a queue of 70
      qi = IA.find_index with (item > 70); //Returns a queue of 0,1
      qi = IA.find_first_index with (item > 50); //Returns a queue of 0
      qi = IA.find_last_index with (item > 50); //Returns a queue of 2
  end
endmodule
```

Limitations on Array Locator Methods

- In the current release, array locator methods are supported only for queues.
- In the current release, only the following array locator methods are supported: `find()`, `find_index()`, `find_first()`, `find_first_index()`, `find_last()`, and `find_last_index()`. You must use the with expression with these methods and the with expression must evaluate to a boolean value.

  The current release does not support the `min()`, `max()`, `unique()`, and `unique_index()` methods.

Debugging Queues and Arrays

For information on how to debug queues and arrays using the Tcl command-line interface or the SimVision analysis environment, refer to *SystemVerilog in Simulation*.

7

Data Declarations

Data declarations are described in Chapter 6 of the IEEE 1800 standard.

Declaring Variables with Initializers

Declaring variables with initializers is described in Section 6.4 of the IEEE 1800 standard.

Variables can be declared in the following ways:

```
int myint, myint2; //Short form; data type followed by the instances
var myvar; //Using the keyword var. Data type is optional. Defaults to type logic.
int a = 0; //With an initializer
```

In SystemVerilog, initialization values for static variables are executed *before* `initial` or `always` blocks; this is unlike Verilog, where static variables behave like any other `initial` block. Also, initializers in SystemVerilog need not be simple constants.

In the current release, static variables with an initializer that are declared within a procedural scope (such as tasks, functions, and named/unnamed blocks) must use the `static` keyword:

```
task mytask;
static int b = 1; //static keyword is required
...
endtask
```

This limitation was introduced in the Accellera 3.1a LRM, but was removed in the IEEE 1800 standard.

Declaring Local Variables in Unnamed Blocks

Declaring local variables in unnamed blocks is described in Section 6.6 of the IEEE 1800 standard.

In Verilog-2001, you can declare local variables in named `begin...end` or `fork...join` blocks. A local variable declared in a named block can then be referenced using a hierarchical

path. For example, in the following code, there are two variables named `i`, which are referenced by the hierarchical names `foo.i` and `foo.loop.i`.

```
module foo (...);
integer i;
...
initial
  begin: loop
    integer i;
    for (i = 0; i < 10; i = i + 1)
      begin
      ...
      end
  end
```

In SystemVerilog, you can declare local variables in unnamed blocks, as well as in named blocks. For example:

```
initial
  begin
    integer i;
    for (i = 0; i < 10; i = i + 1)
      begin
      ...
      end
  end
```

Variables declared in an unnamed block are visible to the unnamed block and any nested blocks below it. However, because the block has no name, the variables cannot be referenced using hierarchical paths.

Continuous Assignments to Variables

See Sections 6.7 and 11.5 of the IEEE 1800 standard for details on continuous assignments to variables.

In Verilog-2001, a net can be written by one or more continuous assignments, primitive outputs, or through module ports. The left-hand side of a continuous assignment can only be a net data type, such as `wire`. The continuous assignment is a *driver* of the net, and nets can have any number of drivers. A net cannot be procedurally assigned. Variables, on the other hand, cannot be used on the left-hand side of continuous assignments. Variables can only be used on the left-hand side of procedural assignments.

SystemVerilog removes this restriction and permits continuous assignments to variables, in addition to nets. You can assign to a variable with a continuous assignment, and whenever any of the inputs in the right-hand side expression of the assignment changes, the expression is evaluated and the result becomes the new value of the variable.

In the following example, port `answer` is a variable that is the target of a continuous assignment. Whenever `left` or `right` changes value, the variable `answer` is updated automatically with the new value of `left & right`.

```
module and2(answer, left, right);
  output logic answer;
  input wire left;
  input wire right;

  assign answer = left & right;
endmodule
```

Note: Using the `logic` data type on a port can affect optimization and performance.

A variable can also be connected to an output port in a module instantiation or to the output of a primitive. Semantically, this is as though there was an implicit continuous assignment in which the variable is being continuously assigned the value of the output port.

In the following example, variable `result` in module `top` is passed to an output port in a module instantiation. There is no explicit continuous assignment to `result`, but one is implied by the connection to the output port in the module instantiation. The variable `result` is automatically updated with each change in the value of output port `answer` in the instanced module.

```
module and2(answer, left, right);
  output answer;
  ...
endmodule

module top;
  wire left;
  wire right;
  reg result;
  and2 and2 (result, left, right);
endmodule
```

The SystemVerilog LRM requires that a variable can have only a single source for its value. If a variable is used on the left-hand side of a continuous assignment, that assignment is the only one permitted for that variable. See “Restrictions on Continuous Assignments to Variables” on page 103 for more information.

See “Limitations in the Current Release” on page 105 for information on restrictions on continuous assignments to variables in the current release.

Restrictions on Continuous Assignments to Variables

The LRM states that you cannot have multiple continuous assignments to the same variable, mix continuous assignments and procedural assignments for the same variable, or use a variable on the left-hand side of a continuous assignment and connect the same variable to the output port of a module.

For an atomic variable (that is, a scalar or real variable), two rules are applicable:

- There can be at most one continuous assignment to the variable.

```
logic v1;
...
assign v1 = 0;   // Continuous assignment to variable v1.
assign v1 = 1;   // Error. Second continuous assignment to v1.
...
```

- If there is a continuous assignment to the variable, the variable cannot also have any procedural assignments (blocking and non-blocking procedural assignments, procedural continuous assignments, or declared variable initialization).

```
logic v2;
...
assign v2 = 0;  // Continuous assignment to variable v2.
initial
   begin
     v2 = 1;    // Error. Mix of continuous and procedural assignments for v2.
   end
...

logic v3, v4;
...
assign v3 = 0;  // Continuous assignment to variable v3.
always @(v4)
   v3 <= v4;    // Error. Mix of continuous and procedural assignments for v3.
...

logic v7 = 1;   // Treated as a procedural assignment.
...
assign v7 = 0;  // Error. Mix of continuous and procedural assignments for v7.
...
```

For whole vectors, the rules for scalars given above apply. If the vector is not updated as a whole, each element of the vector can have its own continuous assignment. No element may have multiple continuous assignments. It is illegal to use both continuous assignments and procedural assignments to the elements of a vector.

```
logic [1:0] v1;
...
// This is legal. One continuous assignment for index 1 and another for index 0.
assign v1[1] = 0;
assign v1[0] = 0;
...

logic [1:0] v2;
wire w;
...
assign v2[1] = 0; // Continuous assignment for index 1.
always @(w)
  v2[2] <= w;     // Error. Mix of continuous and procedural assignments
                  // for the elements of v2, even though the vector indexes
                  // are different.
...

parameter integer p = 1;
logic [1:0] v3;
```

```
...
assign v3[p:0] = 0;   // Counts as a continuous assignment for each element
                      // of the part select.
assign v3[1] = 0;     // An error if the value of parameter p after elaboration
                      // is 1 because index 1 would then have multiple
                      // continuous assignments.
```

For (unpacked) arrays, the rules are similar to those for vectors, with the exception that the individual array elements can be updated in different ways. For example, one element may be updated with a continuous assignment, while another element may be the subject of a procedural assignment. For any individual array element, however, at most one continuous assignment can be used, and a continuous assignment cannot be combined with a procedural assignment.

```
logic v1[1:0];
...
// This is legal. One continuous assignment for index 1 and one for index 0.
assign v1[1] = 0;
assign v1[0] = 0;
...

logic v2[1:0];
wire w;
...
assign v2[1] = 0;  // Continuous assignment for index 1.
always @(w)
   v2[2] <= w;     // Legal. Mixing continuous and procedural assignments for the
                   // elements of v2 is OK as long as the indexes are different.
...

parameter integer p = 1;
logic v3[1:0];
...
initial
   begin
      v3[p] = 0;  // Procedural assignment.
   end
assign v3[1] = 0; // An error if the value of parameter p after elaboration is 1
                  // because index 1 would have a procedural assignment and a
                  // continuous assignment.
```

Limitations in the Current Release

In the current release, there are two restrictions on the implementation of continuous assignments to variables.

- Delays on continuous assignments to variables are not supported. Any delay specified in a continuous assignment is ignored.
- If a variable is driven by a continuous assignment and the variable is forced and then released, the continuous assignment is not immediately re-evaluated upon release. The variable retains its existing value, in the same way that the variable would if it was driven by procedural assignments rather than by a continuous assignment.

8

Classes

Classes are described in Chapter 7 of the IEEE 1800 standard.

SystemVerilog introduces a new `class` data type, which is used in object-oriented (OO) programming. A class is a user-defined data type that can encapsulate data members and methods. Data members and methods are used together to define the functionality and characteristics of an object.

Classes bring the following aspects of OO programming to SystemVerilog: inheritance, encapsulation, polymorphism, and abstract-type modeling.

This chapter provides a basic overview of SystemVerilog classes. For more information, refer to Chapter 7 of the IEEE 1800 standard.

Note: Classes are supported in the NC-Verilog simulator, the Incisive Simulator, the Incisive Design Team Simulator, and the Incisive Enterprise Simulator. SystemVerilog classes are not available with the Incisive HDL Simulator. For more information on the Incisive HDL Simulator, refer to the *Design Team Family Technology Overview*.

Declaring a Class Data Type

The following is an example of a simple class:

```
module myModule;
   class MyClass;
      integer p1;

      task myTask(input integer i);
        p1 = i;
      endtask

      function integer myFunc();
        myFunc = p1;
      endfunction
   endclass:MyClass
endmodule
```

This example defines a class called `MyClass`, which has one data member called `p1` and two methods called `myTask()` and `myFunc()`.

Note: In the current release, classes cannot be declared globally. Class data types and class variables can be declared within a module or a package. For more information on packages, refer to "Packages" on page 203.

To use `MyClass`, you must create a variable using `MyClass` as its data type. For example:

```
module test_top;

   class MyClass;
   ...
   endclass:MyClass

   //Creates variables of MyClass
   MyClass c1;
   ...

endmodule
```

This creates a class variable called `c1`, whose value is an *instance handle* to an instance of class `MyClass`. An instance handle is a value that points to or represents a particular class object. A class object can have only one instance handle, and an instance handle can point to only one class object. However, multiple class variables can have the same instance handle.

Variables must also be initialized. For example:

```
module test_top;
   class MyClass;
   ...
   endclass:MyClass
   MyClass c1;
   initial begin
      //Initializes the variable
      c1 = new;
   end
   ...
endmodule
```

This example uses the `new` function to initialize the variable `c1` to an instance of class `MyClass`.

Tip

In SystemVerilog, uninitialized variables are given a default value of null. To determine whether a variable is uninitialized, you can check its value versus null. For example:

```
    if (c1 == null) c1 = new;
```

For more information on the new function, see "Working with Constructors" on page 109 and Section 7.7 of the IEEE 1800 standard.

You can also combine the creation of a class variable and its initialization. For example, the following statement declares and initializes the variable:

```
MyClass c1 = new;
```

You can use the variable to access class data members and methods within a class object. For example, the following illustrates various ways to use the variable c1 to access commands within MyClass:

```
  c1.p1 = 5;     //Access data member p1 in MyClass
  c1.myTask(2);  //Accesses myTask() in MyClass and assigns 2 to input b
```

Note: Refer to Sections 7.8 and 7.9 of the IEEE 1800 standard for information on how you can declare static class data members and methods.

Working with Constructors

Class constructors are described in Section 7.7 of the IEEE 1800 standard.

This section describes the new function, which was introduced in "Declaring a Class Data Type" on page 107. The new function, which is also called a *class constructor*, is used to initialize an object at the time of its declaration. For example:

```
ClassA c = new;
```

This declaration creates a new instance of class ClassA and initializes the instance using the new function. Every class has a built-in new function. However, you can also specify any special initialization requirements by defining your own new function within a class. For example:

```
class ClassA;
    ...
    integer a;
    function new();
      a = 0;   // Insert special initialization here
    endfunction
endclass:ClassA
```

You can also customize an instance at run time by passing arguments to a new function. For example:

```
ClassA c = new(5, 2);
```

where the corresponding class declaration can look like:

```
class ClassA;
    integer a1, a2;
    function new(input int arg1, input int arg2);
      a1 = arg1;
      a2 = arg2;
    endfunction
endclass:ClassA
```

Inheritance

Class inheritance is described in Section 7.12 of the IEEE 1800 standard.

In SystemVerilog, you can create new classes that are based on existing classes. For example, the following declares a class called `ExtClass` that contains a variable of `Class_AB`:

```
class Class_AB;
   integer p1;
   task t (input integer i);
    p1 = i;
   endtask

   virtual function integer f();
     f = p1;
   endfunction
endclass: Class_AB

class ExtClass;
   Class_AB ext_ab;   // Variable of type Class_AB
   ...
endclass:ExtClass
```

Although this example is legal, SystemVerilog offers a more efficient way of extending a class—through the keyword `extends`. When you extend a class, the derived class inherits all of the data members and methods of the parent class. However, a derived class can add its own data members and methods. In the following example, `ExtendClass` extends `Class_AB`, inherits its data members, and adds properties of its own:

```
class ExtendClass extends Class_AB;
   integer p2;
   task t (input integer i);
     p1 = 2 * i;
     p2 = 4 * i;
   endtask

   virtual function integer f ();
     f = this.p1 + 1;
   endfunction
endclass:ExtendClass
```

Virtual class members are described in “Protecting Class Members” on page 111.

Note: Section 4.15 of the IEEE 1800 standard describes `$cast` dynamic casting, which is used to cast a handle from a base class to a derived class. The current release supports `$cast` dynamic casting (Section 4.15 of the LRM) only on classes.

Protecting Class Members

This section describes the various keywords you can use to protect data members against accidental modification.

- `local`—Local members are available only to methods within the same class. Local members are not available to subclasses, but can be accessed from different instances within the same class. For example (taken from Section 7.17 of the LRM):

  ```
  class Packet;
     local integer i;
     function integer compare (Packet other);
        compare = (this.i == other.i);
     endfunction
  endclass
  ```

  For more information on the `local` keyword, see Section 7.17 of the IEEE 1800 standard.

- `protected`—Protected members are similar to local members, but are visible inside subclasses.

  For more information on the `protected` keyword, see Section 7.17 of the IEEE 1800 standard.

- `const`—Use the `const` keyword to declare read-only members. For example, the following defines `a1` as a `const` and as a protected member:

  ```
  class ObsC;
      const protected int a1;

      ....
  endclass
  ```

  There are two types of `const` declarations: *global* and *instance*.

  - Global constants assign a value to a member at the time of the declaration. For example, the following declares `const a1` with an initial value as a part of its declaration:

    ```
    class ObsC;
        const protected int a1 = 2;

        ....
    endclass
    ```

    A global `const` can be assigned a value only within its declaration.

- Instance constants assign a value within the `new()` function—also called the constructor—of the class. For example:

```
class ObsC;
    const protected int a1;
    ...
    function new();
      a1 = 2;
    endfunction
endclass
```

  For more information on the `const` keyword, see Section 7.18 of the IEEE 1800 standard.

- `virtual`—Use the `virtual` keyword to specify that a class can be extended by other classes, but cannot be instantiated. For example, the following declares a virtual class called `virClass` that can be extended by derived classes, but cannot be instantiated:

```
virtual class virClass;
   ...
endclass
```

  For more information on the `virtual` keyword, see Section 7.19 of the IEEE 1800 standard.

Note: These keywords do not have a predefined order. However, you can specify them only once per member, and you cannot have a member that is declared both as `protected` and as `local`.

Additional Features

For more information on the following aspects of SystemVerilog classes, refer to the specified sections in the IEEE 1800 standard. The Cadence implementation of classes supports all of these features:

- Overridden Members (Section 7.13)—Describes how SystemVerilog classes handle overridden members.
- Super (Section 7.14)—Describes how to refer to members of a parent class from a derived class.
- Casting (Section 7.15)—Describes how to assign a subclass variable to a variable in a higher class.
- Chaining Constructors (Section 7.16)—Describes the initialization sequence within inherited classes.
- Polymorphism (Section 7.20)—Describes how a base-class variable can hold a sub-class object and reference the sub-class methods directly.

- Class Scope Resolution Operator (Section 7.21)—Describes how to uniquely identify members of a class using the `::` operator.
- Typedef Class (Section 7.24)—Describes how to declare a class variable before declaring the class itself.
- Classes and Structures (Section 7.25)—Describes the difference between classes and structures.

  Note: This section also describes global class definitions, which is unsupported in the current release. Instead, the current release supports packages. Refer to "Packages" on page 203.
- Memory Management (Section 7.26)—Describes SystemVerilog's automatic memory management system.

Limitations

The following summarizes the features in the LRM that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- Parameterized classes (Section 7.23) are unsupported.
- Global class definitions (Section 7.25) are unsupported. Instead, the current release supports packages. Refer to "Packages" on page 203.
- Out-of-module references (OOMRs) to class variables are not supported. For example:

```
package p;
   class class_type;
   endclass:class_type
endpackage

import p::class_type;

module top;
   class_type v;
   ...
   initial
     v = bot.c   // Not supported in the current release
   ...
endmodule

module bot (...);
```

```
   class_type c = new;
   ...
endmodule
```

References to class variables in a package are supported. For example:

```
package p;
   class class_type;
   endclass:class_type

   class_type c;
   ...
endpackage

import p::*;

module top;
   class_type v;
   ...
   initial
     v = c   // Supported in the current release
   ...
endmodule
```

- Class variables passed to OOMR tasks or functions or to tasks or functions declared in a package are not supported in the current release. For example:

```
package p;
   class class_type;
   endclass:class_type
endpackage

import p::class_type;

module top;
   class_type v;
   ...
   initial
     bot.mytask(v)   // Class variable passed to OOMR task not supported in the
                     // current release
   ...
endmodule

module bot (...);
   task (input class_type arg_mytask);
   ...
   endtask
...
endmodule;
```

Debugging Classes

For information on how to debug classes using the Tcl command-line interface or the SimVision analysis environment, refer to *SystemVerilog in Simulation*.

9

Operators and Expressions

Operators are described in Section 8.2 of the IEEE 1800 standard.

Verilog does not have increment and decrement operators, and lacks the C assignment operators, such as +=. SystemVerilog adds these operators to the language.

Supported Operators

The current release supports the following:

Assignment Operators on page 115	= += -= *= /= %=&= |= ^= <<= >>= <<<= >>>=
Increment and Decrement Operators	++ and --
Wild Equality and Wild Inequality Operators on page 116	==? !=?
Aggregate Expressions on page 119	

Note: This table summarizes the supported operators. If an operator is supported in the current release but has limitations, a link to its documentation is provided. If the current release supports the operator without limitations, refer to the IEEE 1800 standard for its documentation.

Assignment Operators

SystemVerilog includes the simple assignment operator, =, and the following C assignment operators and special bit-wise assignment operators:

+= -= *= /= %=

&= |= ^= <<= >>=

```
<<<=    >>>=
```

These operators combine an operation with the assignment. For example:

```
abc += xyz;     // Add right-hand side to left-hand side and then assign
                // Same as: xyz = xyz + abc;

i *= 3;        // Same as i = i * 3;

a[i] += 2;     // Same as a[i] = a[i] + 2;
```

These assignment operators can be used as statements, but not in expressions.

Wild Equality and Wild Inequality Operators

The wild equality and wild inequality operators are described in Section 8.5 of the IEEE 1800 standard.

SystemVerilog adds a new equality operator (==?) called the *wildcard equality operator*. This operator is similar to the == operator in that they perform different kinds of bit-wise comparisons. The ==? operator treats Xs and Zs on the right-hand side of the operand as wildcards. Bit positions with Xs and Zs on the right-hand side will automatically match as a result of a wildcard, essentially masking them out of the comparison. The remaining bit positions will participate in a logical comparison (as with ==).

The wildcard inequality operator is `!=?`.

Note: The Accellera 3.1a LRM introduced symmetric operators (=?= and !?=). These operators, which have been removed in the IEEE 1800 standard, have been retained in the Cadence implementation for backward compatibility, but their use is discouraged.

Examples:

```
reg [3:0] op;
op = 4'b100X;

if (op ==? 4'b1XXX)   // true
if (op ==? 4'b1000)   // unknown
if (op ==? 4'b100X)   // true
if (op ==? 4'b000X)   // false
if (op !=? 4'b000X)   // true
```

Assignment Patterns

Assignment patterns are described in Section 8.13 of the IEEE 1800 standard.

SystemVerilog introduces a construct called an *assignment pattern* that can be used to describe patterns of assignments to array elements and structure members. In an assignment pattern, you specify the correspondence between a collection of expressions and the elements of an array or the members of a structure. An assignment pattern uses the following syntax, and can be used on the left-hand or right-hand side of an assignment like syntax:

```
typeName'{key:value;{,key:value}}
```

or

```
'{key:value;{,key:value}}
```

Note: The current release does not support assignment patterns on the left-hand side of assignments.

An assignment pattern consists of braces, keys, and values, with the whole construct prefixed with a simple apostrophe (to distinguish the construct from a Verilog concatenation) or with a full type qualification (`typeName'`). Full type qualifications can be used whenever an expression syntax is allowed. On the other hand, you can prefix an assignment pattern with a simple apostrophe only when the data type can be determined by the target of the assignment pattern, or by the data being assigned. For example:

```
typedef struct packed {int a, b;} packT;
  packT p1 = packT'{a:0, b:1};   //Legal. Provides the datatype.
  packT p2 =      '{a:0, b:1}; //Legal. Datatype from LHS.
```

In the current release, assignment patterns can consist of array literals and structure literals (Sections 3.7 and 3.8 of the LRM). Array and structure literals are assignment patterns that use constant member expressions. The current release does not support assignment patterns that use variables.

There are six different techniques for specifying an assignment pattern:

- Indexed—Uses an index value to locate the member for which to assign the value. For example:

  ```
  typedef logic [1:0] t;
  ...
  t p1 = t'{1:0, 0:1};
  ```

  Associations by index value do not apply to structures.

- Type—Assigns the value to subelements with the specified data type. For example, the following assigns the value `t'{0,0}` to all subelements whose data type is `t`:

  ```
  typedef t [4:2] t2;
  t2 p2 = t2'{t:t'{0,0}};
  ```

- Default—Uses the keyword `default` to assign a value to all of the subelements of the data object that have not been matched by either an index or type key. For example:

```
t p3 = '{default:1};      //Sets all elements to 1
t p4 = '{1:0, default:1}; //Sets all elements, except the element at index 1,
                          //to 1
```

- Positional—Assigns values by position. The first value is assigned to the first subelement, the second value is assigned to the second subelement, and so on. For example:

```
t p5 = t'{0, 1};    //Is the same as the following assignment
t p6 = t'{1:0, 0:1};
```

- Member—Assigns the value to the subelement with the specified member name. For example:

```
typedef struct packed {bits8 m; logic [23:0] n; } t3;
t3 p7 = '{n:5, m:3}; //Assigns 5 to the structure member named n, and 3 to
                     //the structure member named m
```

  Associations by member name do not apply to arrays.

- Replication—Copies the given expression a constant number of times. For example:

```
t pi = t'{2{0}}; //Same as t'{0,0};
```

Mixing association styles in one assignment pattern is not allowed in the LRM. For example, when assigning to a structure, you cannot mix associations by member name (`member:value`) and positional notation, as in the following example:

```
typedef logic [7:0] bits8;
typedef struct packed { bits8 m; logic [23:0] n; } t;
...
parameter t pc = '{m:3, 5};  // Illegal
```

Limitations

The following assignment pattern features in the SystemVerilog standard are not supported in the current release.

- The following types of assignment patterns are not supported:
  - Associations by type
  - Replication
- Assignment patterns can be used only as the right-hand side of an assignment. Using an assignment pattern as the left-hand side of an assignment is not supported. For example:

```
logic a, b, c, d;
typedef logic [3:0] t;
'{a, b, c, d} = t; //Unsupported
```

- Variables are not supported within assignment patterns. In the current release, assignment patterns can only use constant values. For example:

```
int a, b;
int [0:1] ints = '{a,b}; //Not supported
```

- The assignment pattern syntax cannot be used as port expressions. For example:

```
typedef struct packed {int a; byte b;} packT;

module top;
  bottom b (‘{a:0, b:1}); //Not supported
endmodule
```

  However, you can use the assignment pattern as arguments to tasks and functions. For example:

```
module bottom (input packT packer);
  task taskT (output packT out, input packT in);
  out = in;
  endtask

  packT one, two;
  initial task(one, '{416, 73}); //Supported
endmodule
```

- Using a full type qualification (`typeName’`) is required for:
  - Assignment patterns assigned to out-of-module references
  - Port connection expressions

Aggregate Expressions

Aggregate expressions are discussed in Section 8.15 of the IEEE 1800 Standard.

Aggregate expressions, data types, and data objects are used to reference a collection of singular values. In SystemVerilog, aggregate expressions can be used in unpacked structure and array data objects; unpacked structure and array constructors; as well as multi-element slices of unpacked arrays.

In the current release, aggregate expressions can be:

- Copied through assignments
- Used as arguments to tasks and functions
- Compared (using equality or inequality operators) against other aggregate expressions of the same type
- Used with conditional operators

However, for these uses, the current release supports only unpacked structures and fixed-size arrays of the following data type elements:

- Integral data types currently supported in NC-Verilog.
- `real`
- unpacked structures
- unpacked fixed-size arrays, but the element data type must be one of the first three data types listed above

The current release does not support out-of-module references to these variables, and aggregates from array slicing or a member of an unpacked structure.

10

Procedural Statements

SystemVerilog procedural statements are described in Chapter 10 of the IEEE 1800 standard.

In Verilog, procedural statements are introduced using the following:

`initial`	Enables the statement at the beginning of the simulation, and executes it only once
`final`	Executes the statement once at the end of simulation.
`always`	This includes, `always_comb`, `always_latch`, `always_ff`.
`task`	Execute the statement whenever the task is called.
`function`	Execute the statement whenever the function is called, and return a value.

SystemVerilog has the following types of control flow within a process

- Selection, loops and jumps
- Task and function calls
- Sequential and parallel blocks
- Timing control

Unique and Priority Decision Statements

Unique and priority decision statements are described in Section 10.4 of the IEEE 1800 standard.

According the Verilog-2001 standard, `case` statements must evaluate the case selection statements in the order in which they are listed. `if...else` decisions must also be evaluated in source code order. This implies that there is a priority to the case selection items

or series of `if...else...if` statements. To maintain the priority ordering in the hardware implementation, priority-encoded logic is required. Often, however, the order of the decision statements is not important, and synthesis tools may optimize out the priority-encoded logic if the tool determines that the branches of the decision are mutually exclusive (unique). Strict coding guidelines must be followed to avoid mismatches between simulation and how synthesis tools interpret the model.

In addition, the Verilog-2001 standard does not require that these decision statements always execute a branch of code.

SystemVerilog adds the keywords `unique` and `priority`, which can be used before `if` and `case`/`casex`/`casez` statements.

- The `unique` keyword indicates that the order of the decision statements is not important and that they can be evaluated in parallel.

  Examples:

```
always_comb
unique case(a)
  0:       $display("0");
  1:       $display("1");
  default: $display("Value is not 0 or 1");
endcase

always_comb
unique if (a == 0) $display("0");
else if (a == 1) $display("1");
else $display("Value is not 0 or 1");
```

  The SystemVerilog LRM specifies that when a `case` or `if` statement is specified as `unique`, the software tools will verify that all of the decision conditions are mutually exclusive, and that they must generate an error if more than one condition is true, or can be true. Tools are also required to generate an error if the `case` or `if` statement is evaluated and no branch is executed. For example, the simulator generates an error message for the following code because both the first and second conditions are true.

```
a = 3;
b = 4;

unique if (a < b)
  c = 1;
else if (a < 5)
  c = 2;
else if (a > b)
  c = 3;
```

- The `priority` keyword indicates that the order of the decision statements is important, and that tools must maintain the priority encoding. The selection items or series of `if...else...if` statements are evaluated in order, and the first match is used.

  Examples:

```
reg [2:0] a;

priority if (a[2:1] == 0) $display("0 or 1");
else if (a[2] == 0) $display("2 or 3");
else $display("4 to 7");

reg [2:0] a;

priority casez(a)
  3'b00?:  $display("0 or 1");
  3'b0??:  $display("2 or 3");
  default: $display("4 to 7");
endcase
```

The SystemVerilog LRM specifies that when a `case` or `if` statement is specified as `priority`, there must be at least one true condition. Tools must generate a run-time error if the `case` or `if` statement is evaluated and no branch is executed. For example, the simulator generates an error message for the following code because neither condition is true.

```
a = 3;

priority if (a = 0)
  c = 0;
else if (a = 1)
  c = 1;
```

Suppressing Semantic Checks for Unique/Priority Statements

By default, the simulator performs the checks that SystemVerilog requires for `unique` and `priority` constructs. This semantic checking can have an impact on simulation performance.

Use the `ncelab -svperf` command-line option to enhance performance by disabling the checking for `unique` and/or `priority` violations.

Examples:

The following option is the default. Checking is performed for both `unique` and `priority` constructs, and error messages are generated for all violations.

```
ncelab -svperf -up              // Same as -svperf -u-p
```

The following option disables checking of both `unique` and `priority` constructs.

```
ncelab -svperf +up              // Same as -svperf +u+p
```

The following option disables checking of `unique` constructs, but enables checking of `priority` constructs.

```
ncelab -svperf +u               // Same as +svperf+u-p
```

do...while Loop

The `do...while` loop is described in Section 10.5.1 of the IEEE 1800 standard.

Verilog provides `for`, `while`, `repeat`, and `forever` loops. SystemVerilog enhances loops by providing the `do...while` loop and `foreach` loop.

In the Verilog `while` loop, the condition expression is tested at the beginning of the loop. If the expression evaluates to false, the loop is not executed. If the expression starts out as false, the loop is not executed at all.

The `do...while` loop construct has the following syntax.

```
do statement_or_statement_block while (condition)
```

For a `do...while` loop, the condition is evaluated after the statement or statement block has been executed. This means that the statement(s) in the loop will be executed at least once whenever the loop is encountered. For example:

```
i=0;
  do begin
    a[i] = i;
    i = i + 1;
  end
  while (i < 10);
```

for Loop

The enhanced `for` loop is described in Section 10.5.2 of the IEEE 1800 standard.

In Verilog-2001, the variable used to control a `for` loop must be declared prior to the loop. This can cause problems if you inadvertently use the same loop control variable to control loops in two or more concurrent procedural blocks because one loop could modify the variable while another loop is still using it. To avoid this situation, you must either declare different variables at the module level, or declare local variables within named `begin...end` blocks.

Verilog-2001 also allows only one initial assignment statement and one step assignment statement in a `for` loop.

SystemVerilog enhances `for` loops in two ways:

- The loop control variable can be declared within the `for` loop itself. For example, in the following code, two loop control variables with the same name are declared.

  ```
  module foo;
  ...
    initial
  ```

```
    begin
      for (integer i = 0; i < 10; i++)
      ...
    end

  initial
    begin
      for (integer i = 15; i >= 0; i--)
      ...
    end
...
endmodule
```

This creates a local variable within the loop that other loops cannot affect.

Variables declared in a `for` loop initialization statement are automatic variables. Because they are created when the loop is invoked, and destroyed when the loop exits, they cannot be used outside of the `for` loop in which they are declared, and they cannot be referenced using a hierarchical name.

- Multiple initializer and step statements are allowed. The multiple initial assignment statements and step assignment statements are separated by commas. For example:

  ```
  for (integer i = 0, integer j = 0; i < 10; i++, j++)
  ```

In the current release, the following is not supported:

- `genvar` variables are not supported within `for generate` loops
- modports within `for` loops

foreach Loop

The `foreach` loop is described in Section 10.5.3 of the IEEE 1800 standard.

Verilog provides `for`, `while`, `repeat`, and `forever` loops. SystemVerilog enhances loops by providing the `foreach` loop. The `foreach` construct specifies iteration over the elements of an array:

```
foreach (array_identifier [loop_variables]) statement
```

The keyword `foreach` is used for this loop. This construct specifies the following:

- *`array_identifier`*—Designates array, which can be a fixed-size, dynamic or associative array. According to the LRM, the `array_identifier` must be a simple identifier (for example, hierarchical identifiers are not allowed).
- *`loop_variables`*—This is a list of loop variables, where each variable corresponds to a dimension of the array.

Loop variables cannot use the same identifier as the array, and the number of loop variables must not exceed the number of array dimensions. If a loop variable is not specified, then there is no iteration over the dimensions of the array.

The type of the loop variable automatically matches the type of the array index.

The `foreach` construct is similar to the Verilog `repeat` loop. However, the `repeat` loop takes a number to designate how many times the loop should be executed, while the `foreach` loop uses the array bounds to specify the repeat count.

For example:

```
module myforeach;
  int a[5][3][4];

  initial begin
    foreach (a[lv]) begin
    $display ("Value of lv is %d",lv);
    end
    #1 $finish;
  end
endmodule
```

The `foreach` causes `lv` to iterate from 0 to 4 to produce the following simulation output:

```
Value of lv is 0
Value of lv is 1
Value of lv is 2
Value of lv is 3
Value of lv is 4
```

The current release has the following limitations:

- The LRM states that loop variables are implicitly declared as automatic variables.

  The current release does not support automatic variables outside of automatic tasks and functions. Therefore, loop variables are declared as automatic only when the foreach construct is within an automatic task or function. Otherwise, the loop variable is implicitly declared as a static variable.

- The LRM also states that loop variables are implicitly declared as read-only variables.

  This is not enforced in the current release.

  The *`array_identifier`* cannot be an associative array with wildcard indexes. This limitation is expected to be part of the next revision of the standard.

return, break, continue Jump Statements

Jump statements are described in Section 10.6 of the IEEE 1800 standard.

Verilog-2001 provides the `disable` statement, which is used to jump to the end of a named statement group or to exit from a task. The `disable` statement requires adding block names, and can create code that is non-intuitive.

SystemVerilog includes the C jump statements `return`, `break`, and `continue`.

- The `return` statement can be used only in a task or function. This statement can be used to:
  - Exit a task.
  - Exit a non-void function and return a value. The `return` must have an expression of the correct type.

    `return` *`expression`*`;`
  - Exit a void function.

    ```
    return;
    ```
- The `break` statement can be used only in a loop. This statement jumps out of the loop.

  ```
  break;
  ```
- The `continue` statement can be used only in a loop. This statement jumps to the end of the loop and executes the loop control, if present. Named `begin...end` blocks are not required.

  ```
  continue;
  ```

A block of code cannot be disabled if it contains a `return`, `break`, or `continue` that can exit the block.

final Blocks

Final blocks are described in Section 10.7 of the IEEE 1800 standard.

A `final` block is similar to an `initial` block, except that it executes when simulation ends, without delays. The only statements allowed within a `final` block are those that are legal in functions.

Note: The end of simulation does not cause an implicit call to `$finish`.

A `final` block can be used to display statistical information about the simulation.

Example:

```
final
   begin
      $display(“Ending Simulation Time: %d”, $time);
```

```
    $display("Clock Cycles: %d", nCycles);
  end
```

iff Event Control Qualifier

The `iff` event control qualifier is described in Section 10.10 of the IEEE 1800 standard.

System Verilog adds an `iff` qualifier to the @ event control. This construct provides conditional qualification of the event control.

The event expression is evaluated when the expression before the `iff` qualifier changes value, but it is triggered only if the expression after the `iff` qualifier is true.

Examples:

```
// Event expression is evaluated when d changes value, and triggers
// if enable is equal to 1.
always @(d iff enable == 1)
  q <= d;

// Event expression is evaluated at posedge clk, and triggers if enable
// is equal to 1 and preload changes value.
always @(posedge clk iff enable == 1, preload)
  q <= d;
```

always_* Blocks

Specialized procedural blocks are described in Sections 11.2, 11.3, and 11.4 of the IEEE 1800 standard.

System Verilog adds three specialized procedural blocks that reduce the ambiguity of the general purpose Verilog `always` block. These specialized blocks can be used to indicate design intent to simulation, synthesis, and formal verification software tools.

- `always_comb` – Indicates that the intent of the procedural block is to model combinational logic.
- `always_latch` – Indicates that the intent of the procedural block is to model latched logic behavior.
- `always_ff` – Indicates that the intent of the procedural block is to model sequential logic behavior.

fork...join

The SystemVerilog enhancements to `fork...join` are described in Section 11.6 of the IEEE 1800 standard.

The Verilog `fork...join` block statement spawns multiple processes that execute in parallel. Each statement is a separate process, and statements that follow a `fork...join` are blocked from execution until all of the spawned processes have completed execution.

SystemVerilog enhances the `fork...join` statement by adding `fork...join_any` and `fork...join_none` blocks. These additions provide three options for specifying when the parent (forking) process is to resume execution.

`fork...join`	Statements that follow a `fork...join` are blocked from execution until all of the spawned processes have completed execution.
`fork...join_any`	Statements that follow a `fork...join_any` are blocked from execution until any one of the spawned processes has completed execution.
`fork...join_none`	Statements that follow a `fork...join_none` are not blocked from execution while the spawned processes are executing.

The current release also includes extensions to `fork...join` and `fork...join_none`.

fork...join

The `fork...join` statement has been extended so that the statement is now allowed in automatic tasks and functions, including inside local scopes. However, waiting on automatic variables with `wait` statements or event controls is not supported.

fork...join_none

The `fork...join_none` statement has been extended such that it is allowed in functions.

In SystemVerilog, delaying statements (such as delay controls, event controls, `wait` statements, and task calls) are not allowed in functions. Before this release, `fork...join_none` statements were not allowed within functions because they are not useful without delaying statements. This restriction has been removed.

In the current release:

- `fork...join_none` statements are allowed inside functions.
- Delaying statements are allowed inside `fork...join_none` statements that are within functions.
- Tasks whose delays only only occur within a `fork...join_none` are considered non-time-consuming tasks by the direct programming interface (DPI).

When called by the same type of process, the behavior of a `fork...join_none` statement in a function is the same as the behavior of a `fork...join_none` statement in a task. As with tasks, the process that called the function is considered the parent thread of all the threads created by the `fork...join_none` statement.

The following restrictions apply to `fork...join_none`:

- Only the following processes can call a `fork...join_none` statement: `initial` blocks, `always` blocks, variable declaration initializers, and processes created by `fork`.

  In particular, `fork...join_none` statements cannot be executed by the following (doing so will cause a run-time error): non-blocking assignments, regular and procedural continuous assignments, `force`, nonblocking event triggers, `final` blocks, particular system tasks (`$monitor`, `$strobe`, `$async$*`), or an evaluation of a user-defined system task or function argument by PLI. For example:

```
module top;

function f;
input i;
  fork
    #10 $display("in f");
  join_none
  f = i;
endfunction

integer r, x;
event e;

initial
  begin
  //Subprocesses that produce run-time errors

  // Nonblocking assignments
  r <= f(x);
   r <= #3 f(x);
  r <= #(f(x)) 0;
  r <= @(f(x)) 0;
  r <= repeat(2) @x f(x);
  r <= repeat(2) @(f(x)) 0;
  r <= repeat(f(x)) @x 0;
  r[f(x)] <= 0;

  // Nonblocking event triggers
  ->> #(f(x)) e;
  ->> @(f(x)) e;
```

```
  ->> repeat(2) @(f(x)) e;
  ->> repeat(f(x)) @x e;

  // System tasks that create subprocesses
  $monitor(f(x));
  $strobe(f(x));

  // Force statement
  force r = f(x);

  // Procedural continuous assignment
  assign r = f(x);
  end

  always @e $display(r);

  endmodule
```

- The Verilog `disable` statement will not disable subprocesses created by a `fork...join_none`. However, these subprocesses can be disabled using the `disable fork` statement.
- The current release supports `fork...join_none` subprocesses inside local scopes (`begin...end` or `fork...join`) that declare their own automatic variables, and local automatic scopes inside `fork...join_none` subprocesses. However, you cannot use `return`, `break`, or `continue` statements to exit these kinds of scopes. For example, the following illustrates an unsupported `break` statement within a local automatic scope.

```
task automatic t;
  int i;
   for (i = 0; i < 10; i++)
    begin: loop1
     int j; //Local automatic variable
     j = i;
     fork
      #1 $display(j); //j is visible to the following join_none subprocess
      join_none
      if (j == last)
         break; //Unsupported. Uses break to leave scope loop1
    end: loop1
endtask
```

- Waiting on automatic variables with `wait` statements or event controls is not supported.

fork...join_any

In the current release, the following restrictions apply to `fork...join_any`:

- The Verilog `disable` statement will not disable subprocesses created by a `fork...join_any`. However, these subprocesses can be disabled using the `disable fork` statement.
- The current release supports `fork...join_any` subprocesses inside local scopes (`begin...end` or `fork...join`) that declare their own automatic variables, and local

automatic scopes inside `fork...join_any` subprocesses. However, you cannot use `return`, `break`, or `continue` statements to exit these kinds of scopes.

- Waiting on automatic variables with `wait` statements or event controls is not supported.

wait fork

The `wait fork` construct is described in Section 11.8.1 of the IEEE 1800 standard.

Use the SystemVerilog `wait fork` statement to ensure that all spawned processes have completed their execution.

In the following example, the `getvalue` task waits for all four processes to complete before returning to its caller.

Example 10-1 Using the wait fork Construct

```
task automatic getvalue;
  fork
    taskA(); //Start taskA and taskB at the same time
    taskB();
  join_any   //Block until either taskA or taskB completes

  fork
    taskC(); //Start taskC and taskD at the same time
    taskD();
  join_none

  wait fork; //Block until all four tasks have completed
endtask
```

Note: The `wait fork` statement waits only for processes or threads that were directly spawned by the waiting process or thread. The `wait fork` statement does not wait for any descendents or subprocesses. In order to wait for all descendents of a process, each descendent must wait for all of its own spawned processes to complete execution before the descendent can terminate.

disable fork

The `disable fork` construct is described in Section 11.8.2 of the IEEE 1800 standard.

SystemVerilog provides the `disable fork` statement, which disables all active threads of a calling process. This includes any subprocesses that were spawned by any of those threads.

Unlike the `disable` statement, which terminates the execution of a named block—regardless of its relationship to the calling process, the `disable fork` statement terminates only the processes that were forked by the calling thread.

In the following example, the `disable fork` statement terminates all three threads forked by the `getvalue` process, regardless of whether they have finished execution:

```
task automatic getvalue;
  input [8:0] value;
  input [8:0] num;

  begin
     #(num) $display(“Delay of %d. Value of %d.”, num, value);
  end
endtask

initial begin
fork
 #1 getvalue (11, 8);
 #1 getvalue (4, 6);
 #1 getvalue (8, 5);
join_none
 #5 disable fork;
 ...
end
```

However, in the following example, the `disable fork` statement is blocked until one of the `getvalue` processes has completed execution. Then, it proceeds to terminate the two outstanding `getvalue` processes.

```
...
initial begin
fork
 #1 getvalue (11, 8);
 #1 getvalue (4, 6);
 #1 getvalue (8, 5);
join_any
 #5 disable fork;
 ...
end
```

11

Tasks and Functions

Tasks and functions are described in Chapter 12 of the IEEE 1800 standard.

Several enhancements to tasks and functions are included in the SystemVerilog LRM.

Multiple Statements in Tasks and Functions

Multiple statements in tasks and functions are described in Sections 12.2 and 12.3 of the IEEE 1800 standard.

In Verilog-2001, you must enclose multiple statements in a task or function using `begin...end`. SystemVerilog does not require you to enclose multiple statements with `begin...end`.

For example, Verilog 2001 requires:

```
function bounds_err(input integer a, input integer lower);
  integer upper;

  begin
    upper = lower + 255;
    if (a > upper || a < lower)
      bounds_err = 1;
    else
      bounds_err = 0;
end

endfunction
```

SystemVerilog infers the `begin...end`, and executes the multiple statements sequentially.

```
function bounds_err(input integer a, input integer lower);
  integer upper;

  upper = lower + 255;
  if (a > upper || a < lower)
    bounds_err = 1;
  else
    bounds_err = 0;

endfunction
```

Function Output Arguments

Function output arguments are described in Section 12.3 of the IEEE 1800 standard.

In Verilog-2001, functions can have only inputs. The only output is the single return value.

SystemVerilog allows the formal arguments of functions to be declared with the same directional specifiers as tasks, so that a function can have any number of outputs in addition to the return value. The arguments can be declared as:

- `input` – Copy the value in at the beginning of the function call.
- `output` – Copy the value out at the end of the function call.
- `inout` – Copy the value in at the beginning of the function call, and out at the end.

Note: SystemVerilog also lets you declare a task or function argument as `ref`. Refer to "Passing Task and Function Arguments by Reference" on page 138.

Examples:

```
function void myfunc;
input integer a;
input integer b;
output integer x;
output integer y;
...
endfunction

function void myfunc(input integer a, input integer b, output integer x, output
integer y);
...
endfunction
```

You cannot call a function with `output` or `inout` arguments from:

- An event expression
- An expression within a procedural continuous assignment
- An expression that is not within a procedural statement

Default Direction in Task and Function Declarations

The default direction in task and function declarations is described in Section 12.3 of the IEEE 1800 standard.

When using ANSI-style declarations in tasks and functions, if the direction of an argument is not specified, the direction is the direction of the previous argument in the list. If the direction

of the first argument is not specified, the default is `input`. In the following example, the formal argument a defaults to `input`. No direction is specified for argument b, so it is also an `input`. Arguments `x` and `y` are both outputs.

```
function void myfunc (integer a, integer b, output integer x, y);
...
endfunction
```

Void Functions

Void functions are described in Section 12.3.1 of the IEEE 1800 standard.

In Verilog-2001, a function must return a value. The return value is specified by assigning a value to an implicitly declared internal variable with the same name as the function. For example:

```
function myfunc (input a, b);
  myfunc = a * b - 1;
endfunction
```

The function call is an operand in an expression. For example:

```
x = y + myfunc(c, d);
```

SystemVerilog allows functions to be declared as data type `void`, which do not have a return value. For example:

```
function void myprint (integer a);
...
endfunction
```

Void functions have the syntax and semantic restrictions of non-void functions, but they are called as statements, like Verilog tasks. For example:

```
myprint(a);
```

Discarding Function Return Values

Discarding function return values is described in Section 12.3.2 of the IEEE 1800 standard.

In SystemVerilog, you can discard a function's return value by casting the function to the `void` data type. For example, the following discards the return value of `myclass.randomize()`:

```
void'(myclass.randomize());
```

Passing Task and Function Arguments by Reference

Passing task and function arguments by reference is described in Section 12.4.2 of the IEEE 1800 standard.

In Verilog-2001, arguments are passed to tasks and functions by value using a copy-in/copy-out mechanism. When a task or function is called, the argument is copied into the task or function. The argument value then becomes local to the task or function, and is not visible outside the task or function. When the task or function is finished executing, the argument is copied out to the caller of the task or function. Unfortunately, passing by value is undesirable when you have large arguments, or if you have programs that need to share data that is not declared as global.

SystemVerilog lets you pass arguments by reference. Instead of copying an argument value, a reference to the original argument is passed to the task or function. Passing by reference offers the following:

- Unlike the pass by value mechanism, changes to a referenced argument are visible outside the task or function. Conversely, changes to a referenced argument outside the task or function are also visible within the task or function.
- Tasks can be reused to generate waveforms and transactions on different variables, which is beneficial in testbench environments.

The syntax for passing argument values by reference is as follows:

```
subroutine (ref type argument);
```

For example:

```
task my_task( ref reg in1, ref reg in2 );
  begin
  ...
  end
endtask
```

The `ref` keyword cannot be used with other direction keywords (`input`, `output`, or `inout`). For example, the following is illegal:

```
task check_status ( ref input integer in1 );
...
```

The LRM states that arguments that are passed by reference must match exactly and that auto-casting, promotion, or conversion is not allowed. For example, you cannot pass an integer by reference when the formal argument is not of type `int`.

In the following example, a module called `submod` defines an array called `my_array` and a variable of type `int` called `my_int`. These objects are then passed as arguments in a call to `mytask`. In `mytask`, the formal arguments are defined as `ref` arguments. Within the module

submod, the formal argument in1 is an alias for my_array, and the formal argument in2 is an alias for my_int.

```
...

module submod;
  parameter LEFT  = 1;
  parameter RIGHT = 0;
  reg  [LEFT:RIGHT] my_vector;
  integer my_int;

  initial begin
    #1 my_task(.in1(my_vector), .in2(my_int)); //Passes objects as arguments
  end

  task my_task(
    ref reg [1:0] in1, //References my_vector using alias in1
    ref integer in2 ); //References my_int using alias int2
    begin
      $display( "in1( %d )", in1 );
      $display( "in2( %d )", in2 );
    end
  endtask
endmodule
```

The current release also supports:

- Passing automatic variables by reference

  The following example passes the value of the automatic variable b:

  ```
  task automatic mytask(ref int a);
    int b;
    b = a;

    if (b > 1)
       mytask(b); //Supported
  endtask
  ```

- Automatic tasks or functions that pass a ref argument by reference

  ```
  task automatic mytask (ref int r, inout int io);
   int depth;
   if (depth > 0) begin
    mytask(r, io); //Valid to pass r by reference
    mytask(io, r); //Valid to pass io and r by reference
  end
  endtask
  ```

- Passing an enum by reference

  ```
  typedef enum {red, green, blue} color;

  task mytask(ref color color_var);
  ...
  endtask
  ```

- Passing variables by reference to automatic tasks

Limitations for Passing Task and Function Arguments by Reference

The following summarizes the features in the LRM that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- `ref` arguments can be scalars and vectors of type `bit`, `logic`, and `reg`. A `ref` argument can also be of type `int`, `shortint`, `longint`, `byte`, `real`, and a packed structure. The Cadence implementation supports passing arrays of type `int`, `shortint`, `longint`, `byte`, `real`, packed structures, enumerations, `logic`, `bit`, and `reg`. For example:

```
ref bit a;       //A scalar bit is valid
ref bit [3:1] a; //A bit vector is valid
ref bit a[0:3];  //A bit array is valid
ref int a;       //A scalar int is valid
ref int a[0:3];  //An int array is valid
```

  Although it is not explicitly stated in the LRM, the Cadence implementation does not support the passing of a member of an array, or a member of a packed structure by reference to a task or function.

- Although it is not explicitly stated in the LRM, the Cadence implementation does not support part selects that travel through `ref` arguments. For example, the following is not supported:

```
module test_top;
...
reg [15:0] a;
task rotate(ref [3:0] op);
...
endtask

rotate(a[15:12]); //Invalid
endmodule
```

- Although it is not explicitly stated in the LRM, the Cadence implementation does not support out-of-module references to `ref` arguments. For example, the following illustrates illegal out-of-module reference (OOMR) access to a task's reference arguments:

```
module test_top;
reg  [31:0] a;
integer b;

initial begin
  #1  my_task(.in1(a), .in2(b));
  #1  test_top.my_task.in1 = 1;   //Illegal OOMR write to ref argument
  #1  b = my_task.in2;            //Illegal OOMR read from ref argument
end

task my_task( ref reg [1:32] in1, ref integer in2 );
 begin
```

```
    //Illegal OOMR read from ref argument
    #1 $display( "in1( %d )", test_top.my_task.in1 );

    //Illegal OOMR read from ref argument
    #1  $display( "in2( %d )", my_task.in2 );
    end
  endtask

  endmodule
```

Specifying Default Argument Values for Tasks and Functions

Default argument values for tasks and functions are described in Section 12.4.3 of the IEEE 1800 standard.

SystemVerilog lets you specify default values for task and function formal arguments with direction `input`, `inout`, or `ref`. Specifying default values is allowed only with ANSI-style declarations.

When the task or function is called, an argument with a default value can be omitted from the call, and the default value will be used for that call. An error is generated if a formal argument without a default value is not passed in a value when the task or function is called.

The following example, taken from the SystemVerilog LRM, declares a task called `read`. Two of the formal arguments have default values.

```
task read(int j = 0, int k, int data = 1);
...
endtask;
```

This task can be called using various arguments, as follows:

```
read( , 5 );        // Same as read( 0, 5, 1 );
read( , 5, );       // Same as read( 0, 5, 1 );
read( 2, 5 );       // Same as read( 2, 5, 1 );
read( , 5, 7 );     // Same as read( 0, 5, 7 );
read( 1, 5, 2 );    // Same as read( 1, 5, 2 );
read( );            // Error because argument k has no default value
```

The default value for a parameter must be a constant expression involving math on constants and locally defined parameters. For example, the following default argument values are legal:

```
module foo();
parameter ONE = 1;

task read(int j = 0, int k, int data = (1 ** ONE));
...
```

The following default expressions are illegal:

```
module foo();
sub_mod sm1();
reg [31:0] a;    // This is a variable

task read(int k, int data = (1 ** sm1.TWO));  // Illegal - OOMR parameter
...
task read(int k, int data = (a + 1));         // Illegal - expression has variable
...
```

Limitations for Default Argument Values

The following summarizes the features in the LRM that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- Declaring a task or function argument as `ref` is not supported in the current release.
- In the current release, when you call a task or function using default parameters, the target task or function must be defined in the same module. That is, default values cannot be passed to tasks and functions that are called with an out-of-module reference.

Passing Task and Function Arguments by Name

Passing task and function arguments by name is described in Section 12.4.4 of the IEEE 1800 standard.

In Verilog-2001, values must be passed to task or function arguments in the same order in which the formal arguments are defined. This can cause errors if values are passed in the wrong order.

SystemVerilog lets you pass arguments by name, as well as by position. Named arguments can be passed in any order. The syntax is the same as that used for named port connections to a module instance.

Example:

```
task read(int j = 0, int k, int data = 1);
...
endtask;
```

The following calls to the `read` task pass arguments by name. If an argument is not specified, the default value is used. A value for argument `k` must be specified because no default value is defined.

```
read( .j(2), .k(5), .data(3) );        // read(2, 5, 3);
read( .k(5), .data(3), .j(2) );        // read(2, 5, 3);
read( .j(2), .k(5) );                  // read(2, 5, 1);
```

```
read( .k(5) );                          // read(0, 5, 1);
```

If both positional and named arguments are specified in a call, all positional arguments must precede the named arguments. For example:

```
read( 2, 5, .data(7) );     //read( 2, 5, 7 );
read( .k(5), 2, 5 );        // Error because named arg precedes positional arg
```

Optional Arguments for Tasks and Functions

Optional arguments for tasks and functions is described in Section 12.4.5 of the IEEE 1800 standard.

In Verilog-2001, tasks and functions are required to have at least one argument. SystemVerilog removes this restriction. For example:

```
module top;
int count;
function int myfunc (input int a = 0);
  ...
endfunction

initial begin
  count = myfunc(3);  //Valid in Verilog-2001 and SystemVerilog
  count = myfunc();   //Valid in SystemVerilog, but not in Verilog-2001
end
endmodule
```

The SystemVerilog LRM also allows function calls with no arguments to omit the parentheses ( ) that follow the task/function name. However, the Cadence implementation requires the parentheses:

```
count = myfunc;     //Valid in SystemVerilog, but not in Verilog-2001 and current
                      release
```

12

Random Constraints

Chapter 13 of the IEEE 1800 standard describes random constraints, which can be used to generate constraint-driven tests. Unlike traditional directed testing, random testing can help create tests for unique, hard-to-find cases.

In SystemVerilog, you can specify a constraint to describe a property of a field. A solver, which can be embedded in the simulator, then processes constraints and chooses a value that satisfies the properties of the constraint. Although the solver chooses a value that satisfies your constraint, the value is still chosen at random.

Constraints are treated like class members, similar to methods. You can specify a constraint in a class, or a derived class. For example, the following defines the class `myClass` with three variables: a, b, and c.

```
class myClass;
  rand bit [3:0] a, b, c;
  constraint c1 {c == a + b;}
endclass
```

`c1` constrains the value of c, such that it is the sum of a and b.

To generate random values for the variables in this class, you can use the `randomize()` method, which is described in Section 13.5.1 of the LRM and in “The randomize() Method” on page 155:

```
task chkSuccess(myClass myc);
  int success;
  success  = myc.randomize();  // Generates random values for a, b, and c
endtask
```

Note: The constraint solver and random number generator (RNG) used in SystemVerilog randomization may include optimizations not included in previous releases. Because of this, the random number stream can differ from release to release.

Random Variables

Random variables are described in Section 13.3 of the IEEE 1800 standard.

You can declare random class variables using the `rand` or `randc` keywords, where:

- The `rand` modifier is used to define standard random variables, where the solver distributes values equally over a range. For example, if the following declaration is not assigned a constraint, so the solver picks a random value between 0 and 3:

  ```
  rand bit [1:0] d;
  ```

  In this example, there is a 1 in 4 chance that the solver repeats a value during successive calls to `randomize()`.

- The `randc` modifier is used to define random-cyclic variables, where the solver goes through all of the values in a random permutation of the specified range. For the following example, the solver generates a random initial permutation using the possible range values for b.

  ```
  randc bit [1:0] e;
  ```

  An initial permutation for this declaration could be: 2, 0, 1, 3. If there are successive calls to `randomize()`, the solver returns the values in this order. When the solver runs out of values within the initial permutation, it automatically generates a new permutation.

  `randc` variables can only be of type `bit` or enumerated types (Section 13.3.2 of the LRM).

The current release supports the following:

- `rand` handles for classes.
- `rand` arrays, where the array element type can be a:
  - handle
  - `integer`, `shortint`, `byte`, or `bit`
  - `bit` vector
- Multidimensional `rand` arrays
- Aliased `rand` handles are also supported. This is where two or more handles refer to the same class instance.

```
module top;
 class class1;
  rand int rint1;
 endclass
 class class2;
  rand class1 ch1a;
```

```
  class1 ch1b;
  constraint con1 { ch1a.rint1 == 200; }
 endclass
  class2 ch2;
  int res;
 initial begin
  ch2 = new;
  ch2.ch1a = new;
  ch2.ch1b = ch2.ch1a;  // ch2.ch1a and ch2.ch1b refer to the same object
  ch2.ch1b.rint1 = 100;
  res = ch2.randomize();
  // ch2.ch1b.rint1 is now 200, even though ch2.ch1b is not rand
  $display("ch2.ch1b.rint1 = %d", ch2.ch1b.rint1);
 end
endmodule
```

- Null `rand` handles

```
module top;
 class class1;
  rand int rint1;
  constraint con1 { rint1 == 200; }
 endclass
 class class2;
  rand class1 ch1a;
  rand class1 ch1b;
 endclass
 class2 ch2;
 int res;
 initial begin
  ch2 = new;
  ch2.ch1a = new;
  // ch2.ch1b is null
  ch2.ch1a.rint1 = 100;
  res = ch2.randomize();  // Does not consider ch2.ch1b, since it is null
  $display("ch2.ch1a.rint1 = %d", ch2.ch1a.rint1);
 end
endmodule
```

Limitations on Random Variables

The following summarizes the features in the LRM that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- The `rand` and `randc` modifiers are supported for integral variables.
- In the Cadence implementation, `rand` is supported for fixed-size arrays, and dynamic arrays. The elements of these arrays can be integral types (such as `integer`, `bit`, and `bit` vector) or class handles.
- `randc` arrays are not supported.
- `rand` is not supported for unpacked structures and associative arrays.

- In the Cadence implementation, `randc` variables have a maximum size of 16 bits and `randc` enums have a maximum size of 32 bits.

Constraint Blocks

Constraint blocks are described in Section 13.4 of the IEEE 1800 standard.

A constraint block is a class member that lists expressions used to limit the range of a variable, or to define the relationship between variables. The following is the simplified syntax for a constraint block.

```
constraint constraint_identifier
      {constraint_expression [;constraint_expression]; }
```

For example:

```
class myClass;
  rand integer len;
  constraint db {len > 0;}  // Specifies that len must be greater than zero
endclass:myClass
```

Limitations on Constraint Blocks

The following summarizes the features in the LRM that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- In a constraint expression, the left and right operands of the * * (exponentiation) operator must be constants.
- Section 13.2 of the LRM states that only 2-state values are supported within constraints, but the LRM does not describe what happens to 4-state variable values. In the Cadence implementation, 4-state values are converted to 2-state values. This supports Section 4.3.2 or the LRM, which specifies that X and Z values be converted to zero.

External Constraint Blocks

External constraint blocks are described in Section 13.4.1 of the IEEE 1800 standard.

The current release supports external constraint blocks.

Inheritance

Inheritance is described in Section 13.4.2 of the IEEE 1800 standard.

The current release supports inheritance in constraint blocks, which is similar to inheritance for class variables, tasks, and functions.

The current release supports constraint inheritance as discussed in the LRM. That is, a constraint in a derived class that uses the same name as a constraint in its parent classes overrides the base class constraints. Otherwise, all constraints are inherited as-is from the parent class.

Set Membership

Set membership constraints are described in Section 13.4.3 of the IEEE 1800 standard.

The current release supports the `inside` operator (defined in Section 8.19 of the SystemVerilog LRM).

For example, the following specifies that the legal random values that can be assigned to `rand` variable a are 3, 4, or 5:

```
class set_example;
  rand integer a;
  constraint setA {a inside {3, 4, 5};}
endclass
```

In the following example, the `randc` variable `my_randc` will cycle through values in the ranges 0 through 20, and 60 through 90:

```
class set_example1;
  randc byte my_randc;
  constraint con1 {my_randc inside {[0:20], [60:90]};}
endclass
```

The following illustrates a set membership constraint, where *`{set}`* is an array:

```
class set_example2;
  int myarray[0:31];
  rand integer b;
  constraint setB {b inside {myarray};}
endclass
```

Distribution

Distribution is described in Section 13.4.4 of the IEEE 1800 standard.

Randomization constraints can specify sets of weighted values called *distributions*. Assuming that no other constraints are specified, the probability of selecting a legal value in the list is proportional to its specified weight.

Defining a Distribution Expression

To define a distribution expression, use the `dist` operator. The syntax is similar to that used for set membership, in that you specify a set of legal values, which is a comma-separated list of single values or ranges. You can then specify a weight for each term in the list using one of the following operators:

- `:=`

  Assigns the weight to the item. If the item is a range, assigns the weight to every value in the range.

- `:/`

  Assigns the weight to the item. If the item is a range, assigns the weight to the range as a whole.

Specifying a weight is optional. If you do not specify a weight, the default is `:= 1`.

In the following example, the set of legal random values that can be assigned to variable b is 100, 200, or 300. Because no weights are specified, the weighted ratio is 1-1-1.

```
class dist_example;
  rand integer b;
  constraint c1 {b dist {100, 200, 300};}
endclass
```

In the following example, the constraint specifies that b is equal to 100, 200, or 300, but with a weighted ratio of 1-2-3.

```
constraint c2 {b dist {100 := 1, 200 := 2, 300 := 3};}
```

The following example specifies that b is equal to 100, 101, 102, 200, or 300 with a weighted ratio of 1-1-1-2-3.

```
constraint c3 {b dist {[100:102] := 1, 200 := 2, 300 := 3};}
```

The following example uses the `:/` operator, which applies the weight to the range as a whole. Variable b must be 100, 101, 102, 200, or 300 with a weighted ratio of 1/3-1/3-1/3-2-3.

```
constraint c4 {b dist {[100:102] :/ 1, 200 := 2, 300 := 3};}
```

The `dist` operator can be used within conditional statements. For example:

```
constraint c1 {if (cmd==READ) addr dist {12'h0 :=1, 12'h0 :=1, 12'h0 :=1};
```

The `dist` operator can also be used on a member of a structure. For example:

```
type struct {
  bit [7:0] data;
  bit [15:0] addr;
  } op;

  rand op foo;

  constraint data_dist {foo.addr dist 100 := 1, 200 := 2, 300 := 3;} //Supported
```

Limitations on Distribution Expressions

The following features in the LRM are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- There are no limitations on the distribution expression, which is to the left of the `dist` operator. The distribution expression can be any legal expression. There are also no limitations on range expressions.

  The weight expressions, however, cannot contain `rand` variables, `randc` variables, or function calls.

- A `rand` variable can have only one distribution constraint.

- If a `:=` weight operator applies to a range, then the range expressions cannot include any `rand` variables.

Implication

Implication is described in Section 13.4.5 of the IEEE 1800 standard.

The implication operator (`->`) allows you to constrain random values only if a condition is successful. The syntax for the constraint is as follows:

`expression` `->` *`constraint_set`*

where:

- *`expression`* is any valid SystemVerilog integral expression

- *`constraint_set`* is any valid constraint or unnamed constraint set

If the *`expression`* is true, then the random numbers must be constrained by *`constraint_set`*. If *`expression`* is false, then the solver ignores the *`constraint_set`*.

The following constraint specifies that payload should be constrained to less than 100 when mode is SMALL, greater than 10000 when mode is LARGE, or unconstrained when mode is neither SMALL nor LARGE:

```
constraint payC {mode == SMALL -> payload < 100; mode == LARGE -> payload > 10000;}
```

Note: The implication operand is bidirectional. In the previous example, the value of payload constrains mode, but the value of mode also constrains payload.

Since the implication operator is bidirectional (a->b), if the expression b is false, then a must also be false.

if...else Constraints

if...else constraints are described in Section 13.4.6 of the IEEE 1800 standard.

Similar to the Implication operator, the if...else style of specifying constraints uses a conditional expression, in that random values are constrained when a condition is met. The syntax for if...else constraints is as follows:

```
if(expression) constraint_set [else constraint_set]
```

where:

- *expression* is any valid SystemVerilog integral expression
- *constraint_set* is any valid constraint or unnamed constraint block. A *constraint_set* can contain a group of constraints.

If the *expression* is true, then the random numbers are constrained by the first *constraint_set*. If *expression* is false, then the random numbers are constrained by the second, optional *constraint_set*.

An if...else statement and an implication statement are equivalent. For example, the following constraints have the same effect, wherein payload should be constrained to less than 100 when mode is SMALL, and greater than 10000 when mode is LARGE:

```
constraint payD {mode == SMALL -> payload < 100; mode == LARGE -> payload > 10000;}

constraint payE {if (mode == SMALL) payload < 100;
                 else if (mode == LARGE) payload > 10000;}
```

Since the else is optional, there might be confusion when associating an if with an else in nested if...else statements. To avoid confusion, SystemVerilog associates an else with the last if that is missing an else. In the following example, the else relates to the second if:

```
constraint payF {if (mode != SMALL) if (mode == LARGE) payload > 10000;
                 else payload < 100;}
```

Iterative Constraints

Iterative constraints are described in Section 13.4.7 of the IEEE 1800 standard.

An iterative constraint expression uses loop variables and indexing expressions to specify iteration over elements in an array. The syntax for an iterative constraint expression is as follows:

```
foreach (array_identifier [loop_variables]) constraint_set

loop_variables::=[index_variable_identifier]{,[index_variable_identifier]}
```

For example, the following constrains the elements of the array called `myArray` in that they must be in the set `(2, 3, 5, 6, 10)`:

```
class myClass;
 rand byte myArray[];
 constraint myCons {foreach (myArray[i]) myArray[i] inside {2, 3, 5, 6, 10};}
 ...
endclass
```

Limitations in the current release:

- The index variable must be an integer type.
- Function calls are supported within `foreach` constraint expressions. However, if the `foreach` expression is within an in-line constraint, it cannot contain function calls that have a `foreach` loop index as an argument.

Global Constraints

Global constraints are described in Section 13.4.8 of the IEEE 1800 standard.
Global constraints are constraint expressions that contain random variables from other objects. In the current release, constraint expressions can contain variables that are declared in modules, packages, and classes. For example, the following illustrates the use of `rand` class handles. This example applies two constraints to the `rand` variable `rand_int_1`, such that its value is between eleven and nineteen.

```
module top;

 class c1;
  rand int rand_int_1;
  constraint con1 { rand_int_1 > 10; }
 endclass

 class c2;
  rand c1 c1h;
  constraint con2 { c1h.rand_int_1 < 20; }
 endclass

 c2 c2h;
```

```
 integer res;

 initial begin
  repeat(5) begin
   c2h = new;
   c2h.c1h = new;
   res = c2h.randomize();
   $display("res = %d, c2h.c1h.rand_int_1 = %d", res, c2h.c1h.rand_int_1);
  end
 end
endmodule
```

solve...before Constraints

The `solve...before` constraint is described in Section 13.4.9 of the IEEE 1800 standard.

Generally, random values are chosen with equal probability; that is, all values have the same chance of being chosen by the solver. However, you can use the `solve...before` keywords in a randomization constraint to specify that a particular combination of legal values occur more often than others. A `solve...before` constraint has the following syntax:

```
constraint constraint_identifier
      {solve identifier_list before identifier_list;}
```

Where, *`identifier_list`* contains only random variables with integral values. (You cannot impose variable ordering on `randc` variables, as `randc` variables are always solved first.) For example:

```
class myclass;
  rand bit r1;
  rand bit [7:0] r2;
  constraint con1 { r1 -> r2 == 1; }
  constraint order {solve r1 before r2; }
endclass
```

In this example, the `solve...before` directive provides a mechanism for specifying that the value of `r1` be chosen independently from the value of `r2`, giving `r1` equal probability of having the value 0 or 1. Without the `solve...before` directive, `r1` and `r2` would be determined together, giving `r1` almost zero probability of being 1.

The current implementation of `solve...before` supports all of the legal uses outlined by the LRM.

Static Constraint Blocks

Static constraint blocks are described in Section 13.4.10 of the IEEE 1800 standard.

The current release supports the `static` keyword, which is used to declare static constraint blocks. When a constraint block is declared as static, the `constraint_mode()` method affects all of the instances of the specified constraint, in all objects.

Functions in Constraints

Functions in constraints are described in Section 13.4.11 of the IEEE 1800 standard.

Function calls are now supported in constraint expressions.

In the current release, when a function is called from a constraint expression:

- The function must return an integral type.
- The arguments to the function call cannot contain `rand` variables.
- The function cannot contain a `randomize` call.

For example:

```
class class1;
   rand int rint1;
   function int fun1(input int arg1);
     fun1 = 2 * arg1;
   endfunction

   constraint c1 { rint1 == fun1(2); }

endclass
```

Note: According to the LRM, the functions that are called in constraints cannot have side effects, but this is not strictly checked.

Randomization Methods

The current release also supports all of the behavior described in Section 13.5.3, “Behavior of Randomization Methods” of the IEEE 1800 standard.

The randomize() Method

The `randomize()` class method is described in Section 13.5.1 of the IEEE 1800 standard.

The built-in `randomize()` method generates random values for all active random variables within an object, and are subject to the active constraints within the object. The `randomize()` function returns 1 if it successfully assigns all the random variables within an object to a valid value. Otherwise, it returns 0.

The following example defines a class with random variables `i1`, `i2`, and `i3`. It then proceeds to validate whether the solver was able to generate valid random values for these variables:

```
module top;
  integer debug;
  integer success, x1;

  class class1;
    rand integer i1, i2, i3;  // Defines random variables
    integer i10;
    constraint c1 { i1 > 10; i1 < 20; }
    constraint c2 { i2 > i1; i2 < 100; }
    constraint c3 { i3 > i2; i3 < i10; }
  endclass

  class1 p1;

  initial begin
    debug = 1;
    p1 = new;
    p1.i10 = 1000;
    for (x1 = 0; x1 < 10; x1 = x1 + 1) begin
      p1.i1 = 0;
      p1.i2 = 0;
      p1.i3 = 0;

      success = p1.randomize();  // Generate random values for i1, i2, and i3

      if (debug == 1) begin  // Validate whether randomize() was successful
        $display("\nsuccess = %d", success);
      end
    end
  end
endmodule
```

Note: When a constraint contains `randc` variables, the solver generates the `randc` variable values first, and these variables become state variables within the constraint. In this case, `randomize()` might fail to find a solution. In the current release, the solver will not repeat its attempts to reach a solution to the constraint.

pre_randomize() and post_randomize()

The `pre_randomize()` and `post_randomize()` methods are discussed in Section 13.5.2 of the IEEE 1800 standard.

The current release supports `pre_randomize()` and `post_randomize()`, which are built-in methods that are automatically called by the `randomize()` method, before and after it computes random variables. You can override these methods when you want to perform operations before or immediately after randomization. If these methods are overridden, they must call their associated parent class methods. Otherwise, pre- and post- randomization processing steps are skipped.

In-Line Constraints (randomize() with)

Declaring in-line constraints with the `randomize()...with` construct (Section 13.6 of the LRM) is supported. For example:

```
res = bus.randomize() with {atype == low;};
```

In-line constraints can refer to the local variables. For example:

```
class c;
  function void do_rand(input int limit);
   int rc;
   rc = randomize(class_property) with {class_property < limit;} //Supported
  endfunction
endclass
```

Activating and Inactivating Random Variables with rand_mode()

The `rand_mode()` method is described in Section 13.7 of the IEEE 1800 standard.

Random variables can be activated or inactivated using the `rand_mode()` method. All random variables are initially active. A random variable that has been inactivated is treated as a static variable and is not randomized by the `randomize()` method.

The `rand_mode()` method can be called as a task or as a function.

- When called as a task, the argument to `rand_mode` (`1` or `0`) determines the operation to be performed. The syntax is as follows:

  `object``[.`*`random_variable`*`].rand_mode(`*`value`*`);`

  where:

- ❑ *object* is the object in which the variable is defined.
- ❑ *random_variable* is the name of the variable to activate or inactivate. If no variable is specified, the action applies to all random variables in the object.
- ❑ *value* can be either 1 or 0. The value 1 activates the specified variable, or all variables if no variable is specified. The value 0 inactivates the specified variable, or all variables if no variable is specified.

■ When called as a function, rand_mode() returns 1 if the variable is active, or 0 if the variable is inactive. The syntax is as follows:

```
object.random_variable.rand_mode();
```

For example:

```
class Foo;
  rand integer p1, p2;
endclass

Foo myFoo = new;
initial begin
  int result;
  myFoo.rand_mode(0);            // Inactivate all variables in object myFoo
  myFoo.p1.rand_mode(1);         // Activate variable p1 in object myFoo
  result = myFoo.p1.rand_mode(); // Sets result to 1 because p1 is enabled
end
```

Limitations

The Cadence implementation supports using the rand_mode() method for packed arrays, packed structures, full unpacked arrays, and elements of fixed-sized unpacked arrays. However, the rand_mode() method is not supported for unpacked structures.

Activating and Inactivating Constraints with constraint_mode()

The constraint_mode() method is described in Section 13.8 of the IEEE 1800 standard.

Constraints can be activated or inactivated using the constraint_mode() method. All constraints are initially active. Constraints that are inactive are not considered by the randomize() method.

The constraint_mode() method can be called as a task or as a function.

- When called as a task, the argument to `constraint_mode` (`1` or `0`) determines the operation to be performed. The syntax is as follows:

  `object``[.`*`constraint_identifier`*`].constraint_mode(`*`value`*`);`

  where:

  - *`object`* is the object where in the variable is defined.
  - *`constraint_identifier`* is the name of the constraint to activate or inactivate. If no constraint is specified, the action applies to all constraints in the object.
  - *`value`* can be either `1` or `0`. The value `1` activates the specified constraint, or all constraints if no constraint is specified. The value `0` inactivates the specified constraint, or all constraints if no constraint is specified.

- When called as a function, `constraint_mode()` returns `1` if the constraint is active, or `0` if the constraint is inactive. The syntax is as follows:

  `object``.`*`constraint_identifier`*`.constraint_mode();`

For example:

```
class c;
  rand bit cm1,cm2;
  constraint con1 {cm1 < 4;}
  constraint con2 {cm2 == cm1;}
endclass

c cd = new;

initial begin
  int result;
  cd.constraint_mode(0);         // Disables all constraints in object cd
  cd.con1.constraint_mode(1);    // Enables constraint con1 in object cd
  result = cd.con1.rand_mode();  // Sets result to 1 because con1 is active
end
```

In-Line Random Variable Control

In-line random variable control is described in Section 13.10 of the IEEE 1800 standard.

You can temporarily control the randomized variables within a class object or instance by calling the `randomize()` method with arguments. When the `randomize()` method is called with arguments, the arguments specify the variables that should be randomized within a class object—all other variables within the object are considered state variables, and are not randomized. For example:

```
module top();
  class c;
```

```
    rand integer p1, p2;
    integer b1, b2;
    ...
  endclass

c pc1;

  initial begin
    byte res;
    pc1 = new;
    res = pc1.randomize();               // Randomize p1 and p2
    res = pc1.randomize(p1);             // Randomize only p1
    res = pc1.randomize(b1);             // Randomize only b1
    res = pc1.randomize(p1, p2, b1, b2); // Randomize p1, p2, b1, and b2
  end
endmodule
```

Randomization of Scope Variables (std::randomize())

Randomization of scope variables is described in Section 13.11 of the IEEE 1800 standard.

SystemVerilog introduces a scope randomize function, `std::randomize()`, which lets you assign unconstrained or constrained random values to variables that are visible in the current scope.

The syntax of the scope randomize function is as follows:

```
[std::]randomize( [variable_identifier_list] )
       [with {constraint_expression [;constraint_expression]; }];
```

Although it is not specifically mentioned in the LRM, each constraint expression must be followed by a semicolon. For example:

```
success = randomize (Bytes) with {Bytes > 0};  // Invalid
success = randomize (Bytes) with {Bytes > 0;}; // Valid
```

Only integral type variables can be randomized. Wires and non-integral types (such as real) are not allowed. The following variable data types are supported:

- `integer`
- `logic`
- `reg`
- `int`

- `bit`
- `byte`
- `shortint`
- `longint`
- enumerated data types
- packed structures

  The Cadence implementation supports packed structures, but with the following limitations:

  - Packed structures must be at the module level. Packed structures that are defined within classes cannot be used in calls to scope randomize.
  - Packed structures that are defined in a package cannot be used in calls to scope randomize.
  - Packed structures cannot be used in distribution expressions or set membership expressions.
  - Nested structures are unsupported.
  - When a member of a packed structure is an array, a bit or part select of that packed structure is unsupported. For example:

```
module test_top;
  integer r;

  typedef struct packed {
    bit [7:0] rl;
  } struct_t;

  struct_t sl;

  initial begin
    r = randomize(s1) with {sl.rl[2:1]==3;}; // Unsupported
    r = randomize(sl) with {sl.rl == 6;};    // Supported
  end
endmodule
```

In the following example, `std::randomize()` is called with two variables as arguments: `wide` and `i`. The function assigns new random values to these variables.

```
logic [127:0] wide;
integer i;
int success;
```

```
success = randomize(wide, i);
```

The scope randomize function returns `1` if all of the random variables have been set to valid values. Otherwise, it returns `0`.

The scope randomize function can be called with no arguments. In this case, the function acts as a checker, and simply returns status.

Note: The scope randomization construct is not guaranteed to give the same result across simulators from different EDA vendors. One EDA vendor can (and probably will) return different sequences of random numbers than will the Cadence simulator. Furthermore, it is not guaranteed that the Cadence simulator will give the same sequence of random numbers from one major release to the next.

Specifying Constraints

Constraints determine the legal values that can be assigned to the local scope variables. To specify constraints, use the `with` clause. Enclose the constraint expression(s) in curly braces.

```
randomize(a, b, c) with {constraint_expression;};
randomize(a, b, c) with {constraint_expression; constraint_expression;};
```

In the following example, the variables `wide` and `i` are given random values, subject to the constraint that `i` must be less than 32.

```
logic [127:0] wide;
integer i;
int success;

success = randomize(wide, i) with {i < 32;};
```

In the following example, the variables `wide` and `i` are given random values, subject to the constraint that `i` must be less than 32, and the `0`th bit of `wide` must be `0`.

```
success = randomize(wide, i) with {i < 32; wide[0] == 0;};
```

The randomization returns 1 if it succeeds, 0 if it is overconstrained. For example, the following call will return 0:

```
success = randomize(i) with {i < 32; i > 32;};
```

The current implementation of the scope randomization construct supports the SystemVerilog constraint expressions described in “Constraint Blocks” on page 148.

Examples:

- Set membership—For example, the set of legal random values that can be assigned to variable a is 3, 4, or 5:

```
module top;
  integer i;
  int success;
  integer a, a2;
  integer num_iterations;

  initial
    begin
      a = 0;
      num_iterations = 8;

      process::self.srandom(20);

      for (i = 0; i < num_iterations; i = i + 1)
        begin
          // a must be 3, 4, or 5.
          success = randomize(a) with { a inside {3, 4, 5}; };
          if (success != 1) $display("Failed");
          if ( !( (a >= 3) && (a <= 5) )) $display("Failed");
        end
    end
endmodule
```

  The following call to the randomize function specifies that the legal values that can be assigned to variable a are: 3, 4, 5, 10, 11, 12, 13, 14, 15, 16.

```
success = randomize(a) with { a inside {3, 4, 5, [10:16]}; };
```

  The negated form of the `inside` operator specifies that the expression is excluded from the set of legal values. The syntax is:

```
with { !(expression inside {set_of_values} ); };
```

  In the following example, the first `inside` operator specifies that a must be 10–16. The second `inside` operator specifies that a cannot be 11–15. Therefore, the set of legal values that can be assigned to a is 10 or 16.

```
success = randomize(a) with {
          a inside { [10:16] };
          !(a inside { [11:15] });
          };
```

- Distribution—For example, the following specifies that the set of legal random values that can be assigned to variable a2 is 100, 200, or 300. Because no weights are specified, the weighted ratio is 1-1-1.

```
success = randomize(a2) with {
          a2 dist {100, 200, 300};
          };
```

The following specifies that a2 is equal to 100, 200, or 300, with a weighted ratio of 1-2-3.

```
success = randomize(a2) with {
          a2 dist {100 := 1, 200 := 2, 300 := 3};
          };
```

The following specifies that a2 is equal to 100, 101, 102, 200, or 300, with a weighted ratio of 1-1-1-2-3.

```
success = randomize(a2) with {
          a2 dist { [100:102] := 1, 200 := 2, 300 := 3 };
          };
```

The following example uses the :/ operator to apply the weight to the range as a whole. Variable a2 must be 100, 101, 102, 200, or 300, with a weighted ratio of 1/3-1/3-1/3-2-3.

```
success = randomize(a2) with {
          a2 dist { [100:102] :/ 1, 200 := 2, 300 := 3 };
          };
```

- Implication—The following specifies that payload should be constrained to less than 100 when mode is SMALL, greater than 10000 when mode is LARGE, or unconstrained when mode is neither SMALL nor LARGE:

```
success = randomize(payload) with {mode == SMALL -> payload < 100;
                                   mode == LARGE -> payload > 10000;};
```

- if...else constraints—The following specifies that payload should be constrained to less than 100 when mode is SMALL, and greater than 10000 when mode is LARGE:

```
success = randomize(payload) with {if (mode == SMALL) payload < 100;
                                    else if (mode == LARGE) payload > 10000;};
```

Limitations

In-line scope randomize constraints have the same limitations as constraints that are declared in classes. Refer to "Limitations on Constraint Blocks" on page 148 for more information.

Random Number System Functions and Methods

Several enhancements to random number system functions and methods are proposed in the SystemVerilog LRM. This section lists the enhancements that are supported in the current release.

The $urandom Function

The `$urandom` function is described in Section 13.12.1 of the IEEE 1800 standard.

SystemVerilog offers a new function called `$urandom`, which supplements the Verilog `$random` task used for generating random numbers. Unlike `$random`, the `$urandom` function generates *unsigned*, 32-bit random numbers and offers thread stability. The syntax for `$urandom` is:

```
value = $urandom [( seed )] ;
```

where *seed* is an optional, integral expression used to determine the sequence of the generated numbers. For example:

```
module top;
  int value;
  initial begin
    value = $urandom (2); // Sets the seed for the current process thread to 2
    value = $urandom;     // Generates a 32-bit random number
  end
endmodule
```

Note that the following example from the IEEE 1800 standard, Section 13.12.1 is incorrect, because `$urandom` is a function.

```
bit [64:1] addr;

$urandom( 254 );               // Initializes the generator
addr = {$urandom, $urandom };  // 64-bit random number
number = $urandom & 15;         // 4-bit random number
```

The second line should be:

```
value = $urandom( 254 ); // Initializes the generator
```

or, you can cast the function call to `void`:

```
void'$urandom(254); // Initializes the generator
```

The number generator generates pseudo-random numbers, in that the sequence of generated numbers can be reproduced exactly, using the same seed. To generate truly random numbers, you can specify a seed using a non-deterministic source, such as the current time of day, or you can read values of seeds from a file.

`$urandom` offers thread stability, in that the random number values do not depend on the order of thread execution. Refer to the SystemVerilog LRM, Section 13.13.2 for more information.

The $urandom_range Function

The `$urandom_range` function is described in Section 13.12.2 of the IEEE 1800 standard.

SystemVerilog offers a new function called `$urandom_range`, which generates a random number within a specified range, and offers thread stability. The syntax for `$urandom_range` is as follows:

```
integer value;
value = $urandom_range(maxval, minval);
```

where *`maxval`* and *`minval`* are unsigned, 32-bit integral expressions that specify the size of the range. In the following example, `$urandom_range` returns a random value from 3 to 6.

```
address = $urandom_range(6,3);
```

If *`minval`* is omitted, zero is used as a default. In the following example, `$urandom_range` returns a random number from 0 to 6.

```
address = $urandom_range(6);
```

If *`minval`* is greater than *`maxval`*, the two values are swapped so that the first value is always larger than, or equal to, the second value. In the following example, `$urandom_range` returns a random value from 2 to 6.

```
address = $urandom_range(2,6);
```

In the following example, `$urandom_range` always returns 4:

```
address = $urandom_range(4,4);
```

The srandom() Method

The `srandom()` method is described in Section 13.12.3 and Section 13.14 of the IEEE 1800 standard.

Random number streams in SystemVerilog are associated with processes (threads). Each thread has its own independent RNG for all randomization calls invoked from that thread, and the RNG is guaranteed to produce the same sequence of random values from one simulation run to the next.

Section 13.13.1 of the LRM describes how each RNG is seeded. The seed for each Verilog process is obtained from the next random number generated by the process's parent process. If the process does not have a parent (for example an `initial` block in a module instance), the seed is obtained from the next random number generated by the RNG of the module instance. You can manually set the RNG seed for subsequent calls to a process's RNG using the SystemVerilog `srandom()` method.

Each class object has its own RNG stream. This stream derives its seed from the next random number that is generated from the process that created the object. You can use the `srandom()` method within a class to manually seed its RNG.

A seed for a stream can be set with a call to the `srandom()` method of the current process, as follows:

```
// Set the seed for the random number stream associated
// with the current process to 300.
process::self.srandom(300);
```

where:

- `process` is a predefined, built-in class that implements fine-grained process control.
- `self` is a static member function of the built-in process class that returns an object representing the current process.
- `srandom` is a built-in member function of every object that manually sets the seed for the random number generation for that object (in this case, the current process).

  Note: While the simulator recognizes this special form for seeding the RNG, the built-in process class is not implemented in the current release, and the static member function `self` is not supported in a general context.

Additional System Functions and Methods

The current release supports the following random number system functions and methods:

- `get_randstate()` (Section 13.12.4 of the LRM)
- `set_randstate()` (Section 13.12.5 of the LRM)

Random Stability

The current release supports all of the properties of random stability, which are described in Section 13.13 of the IEEE 1800 standard.

Random Weighted Case (randcase)

The `randcase` case statement is described in Section 13.15 of the IEEE 1800 standard.

`randcase` is a case statement that randomly selects one of its branches, based on a branch weight. The probability of taking a branch is determined by its weight, divided by the sum total of all weights.

The weights must be non-negative integral expressions, and they cannot be greater than 64 bits wide. Negative signed expressions are treated as unsigned expressions.

Example:

```
randcase
   2 : result = 1;
   5 : result = 2;
   3 : result = 3;
endcase
```

In this example, the first branch is given a weight of 2, the second 5, and the third 3. The sum of all the weights is 10. The probability of selecting each branch is as follows:

- 20% probability of selecting branch `result = 1;`
- 50% probability of selecting branch `result = 2;`
- 30% probability of selecting branch `result = 3;`

If a weight of 0 is specified for a branch, that branch will be ignored. If all branches specify 0 weights, no branch is taken, and a warning is generated.

Weights are not limited to constants, but can be arbitrary expressions. For example:

```
randcase
   a + b : result = 1;
   a - b : result = 2;
   32'b0 : result = 3;
endcase
```

Random number streams in SystemVerilog are associated with processes (threads). Each thread has its own independent RNG for all randomization calls invoked from that thread. The seed for this generator can be set in each process thread by calling `process::self.srandom(`*`new_seed`*`)`. For example:

```
// Set the seed for the random number stream associated
// with the current process to 300.
process::self.srandom(300);
```

See "The srandom() Method" on page 166 for details.

Random Sequence Generator (randsequence)

The `randsequence` construct is described in Section 13.16 of the IEEE 1800 standard.

To determine whether an utterance of a language is valid, parsers use the language's BNF notation to generate a program that can validate the utterance. SystemVerilog offers a different approach, through *random sequence generators*.

A random sequence generator randomly generates a valid representation of a language, which you can then use to stimulate a design under test. To specify a random sequence generator, use the `randsequence` keyword.

Declaring a randsequence Block

A `randsequence` block is composed of *productions*. A production has a name, and contains a list of production items. Production items are defined in the order for which they should be streamed, and can be classified into *terminals* and *non-terminals*. A terminal is an item that cannot be divided, and needs only its associated block of code, while a non-terminal item can be defined in terms of terminals, or other non-terminals.

The following is the simplified syntax for a `randsequence` block:

```
randsequence ([production_identifier])
 production_list;
endsequence
```

For the complete syntax, refer to the IEEE 1800 standard, Syntax 13-12.

The following example defines a `randsequence` block:

```
randsequence(test)
  //Defines test in terms of its non-terminals
  test: one two done;

  //Defines one as a choice between “up” and “down”
  one: up | down;

  //Defines two as a choice between “smile” and “frown”
  two: smile | frown;

  //Defines terminals that display the production name
  done: { $display("done"); };
  up: { $display("up"); };
  down: { $display("down"); };
  smile: { $display("smile"); };
  frown: { $display("frown"); };
endsequence
```

This example has the following possible sequences:

```
up smile done
up frown done
down smile done
down frown done
```

In this example, the production `test` is defined in terms of three non-terminals: `one`, `two`, and `done`. When the non-terminals are generated, they are decomposed into their productions. A production item can contain multiple productions, separated by the `|` symbol. This `|` symbol indicates a group of choices, from which the generator will choose at random. In this example, `one` specifies a choice between `up` and `down`, and `two` specifies a choice between `smile` and `frown`.

The remaining productions in this example are terminals, in that they are specified solely by their code block. In this example, the `done`, `up`, `down`, `smile`, and `frown` terminals display their production name.

if...else Production Statements

`if...else` production statements are described in Section 13.16.2 of the IEEE 1800 standard.

Productions can be made conditional using an `if...else` production statement, which uses the following syntax:

`if(`*`expression`*`)` *`true_productionitem`* `[else` *`false_productionitem`*`]`

where expression is any expression that evalates to a boolean value. The expression is evaluated. If it is true, the *`true_productionitem`* is generated; otherwise, the optional *`false_productionitem`* is generated.

For example:

```
module top;
 int i1 = 100;
 initial begin
 randsequence (P1)
   P1 : if(i1 == 100) P2 else P3;
   P2 : {$display(“P2”);};
   P3 : {$display(“P3”);};
 endsequence
 end
endmodule
```

Case Production Statements

Case production statements are described in Section 13.16.3 of the IEEE 1800 standard.

The `case` production statement, which is used to select a production from a set of alternatives, has the following syntax:

```
case (expression)
  expression {,expression}: production_item1;
  expression {,expression}: production_item2;
  expression {,expression}: production_item3;
   ...
   default: default_production;
endcase
```

The value of the case expression is evaluated and compared to each of the production expressions, in the order they are written. For the first production expression that matches, its corresponding production is generated. If none of the production expressions match the case expression, the optional *`default_production`* is generated. For example:

```
module top;
 int i1 = 100;
 initial begin
  randsequence (P1)
    P1 : case (i1)
     100 : P2;
     default : P2;
     300 : P3;
         endcase;
    P2 : {$display("P2");};
    P3 : {$display("P3");};
  endsequence
 end
endmodule
```

Note: You can have only one default statement within a case production statement.

Repeat Production Statements

Repeat production statements are described in Section 13.16.4 of the IEEE 1800 standard.

To generate a production a set number of times, use the `repeat` production statement. This statement has the following syntax:

```
repeat(expression) production_item
```

where *`expression`* is a non-negative integeral value that specifies how many times to generate the specified *`production_item`*. For example, the following displays "`My sequence`" ten times:

```
module top;
  int i1 = 100;
  initial begin
   randsequence (P1)
    P1 : repeat (10) P2;
    P2 : {$display("P2");};
   endsequence
  end
endmodule
```

A `repeat` production statement cannot be terminated prematurely. To terminate a `repeat` production statement, you can terminate the entire `randsequence` block using a `break` statement.

Limitations

This section summarizes the features in the SystemVerilog LRM that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- Section 13.16.5 of the LRM states that you can use the `rand join` production control to randomly interleave multiple production sequences, while preserving the order of each sequence.

  Cadence does not support interleaved productions.

- Section 13.16.7 states that you can use arguments within productions. The LRM gives the following example:

```
randsequence(main)
 main: first second gen;
 first:  add | dec;
 second: pop | push;
 ...
 gen (string s = "done"): {$display(s);}; //Argument within a production
endsequence
```

  Cadence does not support arguments within productions.

- Section 13.16.7 also states that productions can return values using the `return` statement. The LRM gives the following example:

```
randsequence (bin_op)
 ...
 bit[7:0] value : {return $urandom};            //Returns an 8-bit value
 string operator : add:=5{return “+”;}
                   | dec := 2 {return “-” ;}    //Returns a string
                   | mult := 1 {return “*”;}
                  ;
endsequence
```

  Cadence does not support productions with return values.

Debugging Random Constraints

For information on how to debug random constraints using the Tcl command-line interface, NCSim, or the SimVision analysis environment, refer to *SystemVerilog in Simulation*.

13

Interprocess Synchronization and Communication

SystemVerilog introduces a powerful set of synchronization and communication mechanisms that can be used to control the interactions that occur between dynamic processes used to model a complex system or a highly dynamic, reactive testbench. These additions include semaphore and mailbox built-in classes and enhancements to the named event data type.

Semaphores

Semaphores are described in Section 14.2 of the IEEE 1800 standard.

Note: Semaphores are supported in the NC-Verilog simulator, the Incisive Simulator, the Incisive Design Team Simulator, and the Incisive Enterprise Simulator. SystemVerilog semaphores are not available with the Incisive HDL Simulator. For more information on the Incisive HDL Simulator, refer to the *Design Team Family Technology Overview*.

A semaphore is a built-in class that can be used for synchronization and the mutual exclusion of resources. If you have a shared resource that can only be accessed by a set number of processes *at any given time*, you can use a semaphore to enforce the set of rules to access the resource.

When a semaphore is created, it contains a specific number of keys. When a process wants to use the resource, it must first obtain a key from the semaphore. The waiting queue for a semaphore is first-in first-out. This does not guarantee the order in which processes will arrive at the queue, only that the semaphore will preserve the order of their arrival. Once the maximum number of processes has been reached, all others must wait until a sufficient number of keys is returned.

The following is the prototype for the semaphore class:

```
class semaphore;
  function new(int keyCount = 0);
  task put(int keyCount = 1);
  task get(int keyCount = 1);
  function int try_get(int keyCount = 1);
endclass
```

Semaphores are a part of the built-in `std` package, which means they are implicitly imported into the compilation-unit scope of every compilation unit. This means that semaphores are available in any other scope.

Note: The `semaphore` class can be redefined because the `semaphore` identifier is not a reserved keyword and can be used as a regular identifier.

This built-in class provides the following methods:

- `new()`—The `new()` constructor creates the semaphore, and has a integer argument `keyCount` that is used to create the desired number of keys. The `keycount` argument defaults to `0`. For example, the following creates a semaphore called `sem1` with `4` keys.

```
semaphore wantKeys; //Declares a semaphore data type
...
initial begin
wantKeys = new(4); //Initializes the semaphore with 4 keys
...
```

- `get()`—The `get()` task obtains keys for a semaphore. The number of keys to obtain is passed as an argument to `get()`. The default number of keys is `1`. For example, the following shows a process that asks for all the keys and blocks until they are available.

```
...
wantKeys.get(4); //Attempt to procure all keys
...
```

  A call to `get()` can be time consuming, because if all of the desired keys are not available, `get()` blocks subsequent statements and waits for all of the remaining keys.

- `put()`—The `put()` task returns keys so that other processes can use them. For example:

```
...
wantKeys.get(4);
...
wantKeys.put(4);
...
```

- `try_get()`—The `try_get()` function is used to obtain keys without blocking. Unlike the `get()` task, the `try_get()` function checks key availability without blocking subsequent calls. If the `try_get()` function is successful in procuring the desired number of keys, it returns `1`. Otherwise, it returns `0`. For example:

```
...
if (wantKeys.tryget(4))
  pr();
else
...
```

Limitations on Semaphores

In the current release, the following is supported for semaphores:

- Semaphores are supported in modules, programs, packages, and interfaces.
- Semaphores can also be declared as public, `static`, `local`, or `protected` members of a class.
- Sempahores can be passed as arguments to tasks, functions, and class methods.

Because semaphores are classes, they are subject to the same limitations. In addition to the class limitations, semaphores also have the following limitations:

- Semaphores cannot be used within `generate` statements or declared on ports.
- Queues, static arrays, dynamic arrays, and associative arrays of semaphores are not supported.
- Semaphores cannot be used in the index of an associative array.
- Semaphores cannot be referenced using hierarchical paths, or from a subclass. They can, however, be access through a task or function. For example:

```
class A;
  int i;
  local semaphore s;

  task s_new(int key);
    s = new(key);
  endtask
endclass
```

- The LRM states that semaphores “can be used as base classes for deriving additional higher level classes.” This is not supported in the current release.
- According to the LRM, semaphores can use the `std::` syntax. With this syntax, the code can easily distinguish between semaphores and any overrides of semaphores. The current release does not support this syntax.

Mailboxes

Mailboxes are described in Section 14.3 of the IEEE 1800 standard.

For a complete example that uses a mailbox, which you can download and run, refer to the *SystemVerilog Engineering Notebook*.

Mailboxes provide a form of *direct* communication between processes, where data can be exchanged between a sending process and its designated recipient.

SystemVerilog mailboxes operate like real mailboxes. A process places a message inside a mailbox so that another process can retrieve it. The message stays within the mailbox until it is retrieved. If the receiving process checks the mailbox before the sending process has deposited the message, the receiving process can either wait for the message to arrive or check back at a later time.

A SystemVerilog mailbox is a built-in class. The following is the prototype for the `mailbox` class:

```
class mailbox #(type T = dynamic_singular_type);
  function new(int bound = 0);
  function int num();
  task put(T message);
  function int try_put (T message);
  task get(ref T message);
  function int try_get(ref T message);
  task peek (ref T message);
  function int try_peek (ref T message);
endclass
```

Note: The `ref` arguments used in the `get()`, `try_get()`, `peek()`, and `try_peek()` are subject to the limitations described in "Limitations for Passing Task and Function Arguments by Reference" on page 140.

The following is the syntax for creating a mailbox:

```
mailbox mailbox_name;
```

Example:

```
mailbox mymailbox;
```

In the current release, you can use the Tcl `describe` command on the handle of a mailbox to display the number of messages inside the mailbox.

```
describe -handle 2
```

Mailbox Methods

This built-in class provides the following methods:

- `new()`—The `new()` constructor creates the mailbox, and has an integer argument `bound` that is used to determine whether the mailbox is bounded or unbounded. If the `bound` argument is set to a non-zero number, then the mailbox is bounded and the number represents the maximum number of messages that the mailbox can contain. When a process tries to place a message in a mailbox that has reached its bound limit, the process will be suspended until space is available.

When the bound argument is set to zero, then the mailbox is unbounded and can contain an unlimited number of messages. Unbound mailboxes never suspend a thread in a send operation.

The default bound argument is 0.

Example:

```
mymailbox = new(0);   // Mailbox is unbounded, default.
mymailbox = new(5);   // Specifies that mailbox queue can contain 5 messages.
mymailbox = new(-10); // Generates warning. Default of 0 is used instead.
```

- num()—The num() method returns the number of messages currently in the mailbox.

  Example:

  ```
  i = mymailbox.num();
  ```

- put()—The put() method places a message in the mailbox. If a process tries to use the put() method on a mailbox that is full, the call will block until space is available. For example, the following tries to put a message called msg1 in myMailbox.

  ```
  mymailbox.put(c1); //c1 is a class handle
  ```

  Note: The put() method stores the messages in the mailbox in FIFO order. In other words, the first message that is put in is the first one to pop out.

- try_put()—The try_put() method is similar to put(), except that it is non-blocking. If the mailbox is not full, the method places the message in the mailbox and returns a positive integer. If the mailbox is full, this method returns 0.

  Note: The try_put() method stores the messages in the mailbox in FIFO order.

- get()—The get() method retrieves one message, if one is available, from the specified mailbox's queue. If the message is not available, then the call blocks until the message is available. If the type of the message variable does not match the type of message in the mailbox, then a run-time error occurs. For example:

  ```
  class c;
  endclass

  class d extends c;
  endclass

  class x;
  endclass

  c c1;
  d d1;
  x x1;
  ...
  mymailbox.put(d1);
  mymailbox.get(x1); //Invalid. Types do not match.
  mymailbox.get(c1); // Valid.
  ```

- `try_get()`—The `try_get()` method is similar to `get()`, except that it is non-blocking. If the mailbox is empty, this method returns 0. If the type of the message variable does not match the type of message in the mailbox, this method returns a negative integer. If a message is available, and its type matches the message variable type, then this method retrieves the message and returns a positive integer. For example:

```
j = mymailbox.try_get(c1); //Value of j will be 0.
mymailbox.put(c1);
j = mymailbox.try_get(x1); //x1 does not change and value of j will be -1.
j = mymailbox.try_get(c2); //Value of j will be 1.
```

- `peek()`—The `peek()` method copies one message, if it is available, from the mailbox. This method is useful if you would like to copy a message from a mailbox, without deleting it from the mailbox. If the message is not available, then the call blocks until the message is available.
- `try_peek()`—The `try_peek()` method is similar to the `peek()` method, except that it is non-blocking. If the mailbox is empty, this method returns 0. If the type of the message variable does not match the type of message in the mailbox, this method returns a negative integer. If a message is available, and its type matches the message variable type, then this method copies the message and returns a positive integer.

Limitations on Mailboxes

The following summarizes the features in the SystemVerilog standard that are not supported in the current release. Differences between the specification shown in the LRM and the Cadence implementation are also listed.

- Only the following data types can be passed as arguments to the `put()`, `try_put()`, `get()`, `try_get()`, `peek()`, and `try_peek()` mailbox methods:
  - class handles
  - `shortint`, `int`, `longint`, `byte`, `integer`
  - `bit`, `logic`, `reg`
  - packed arrays of types `bit`, `logic`, and `reg`
- Mailboxes and fixed arrays of mailboxes can be declared within only modules, program blocks, tasks, functions, classes, interfaces, and packages.
- Mailboxes cannot be used as OOMRs.
- Mailboxes cannot be passed as module ports to tasks and functions. You can, however, pass mailboxes as arguments to tasks, functions, and class methods.
- Although mailboxes are a built-in class, they cannot be extended.

- Parameterized mailboxes are not supported.
- Initializing a fixed array of mailboxes is not support.
- If `mb1` and `mb2` are fixed arrays of mailboxes, then `mb1=mb2` is not supported.

Events

In the current release, events:

- Can be passed by value to tasks and functions
- Can be declared within a class or package.

The current release does not support events within unpacked structures.

Non-Blocking Event Trigger

The SystemVerilog nonblocking event trigger is described in Section 14.5.2 of the IEEE 1800 standard.

In Verilog, event triggers act like blocking assignments. They immediately trigger the event at the point at which the trigger is executed and block the execution of subsequent code until the event finishes execution.

The triggering of named events in Verilog can often lead to race conditions. The following example illustrates the problem.

```
module events;
  event e, go;

  always @(go)
    begin
      $display($time,,"Always 1 - triggering e");
      -> e;
    end

  always @(go)
    begin
      $display($time,,"Always 2 - about to wait on e");
      @(e) $display($time,,"Always 2 - wakeup on e");
      $finish;
    end
```

```
  always #5 -> go;

endmodule
```

The run-time output of this description could either be:

```
5 Always 1 - triggering e
5 Always 2 - about to wait on e
10 Always 1 - triggering e
10 Always 2 - wakeup on e
```

or:

```
5 Always 2 - about to wait on e
5 Always 1 - triggering e
5 Always 2 - wakeup on e
```

SystemVerilog introduces a nonblocking event trigger operator (`->>`). The syntax is as follows:

`->>` [*`delay_or_event_control`*] *`hierarchical_event_identifier`*;

The `->>` operator does not block the execution of subsequent code. The statement creates a nonblocking assign update event in the time in which the delay control expires, or the even-control occurs. The triggered event is scheduled to occur at the end of the simulation time slot in the nonblocking assignment region of the simulation cycle.

The following code shows the first `always` block from the example shown above rewritten to use the nonblocking trigger operator.

```
  always @(go)
    begin
      $display($time,,"Always 1 - triggering e");
      ->> e;  // Nonblocking trigger guaranteed to run after both always blocks.
    end
```

The legal outcomes of the above description are:

```
5 Always 2 - about to wait on e
5 Always 1 - triggering e
5 Always 2 - wakeup on e
```

or:

```
5 Always 1 - triggering e
5 Always 2 - about to wait on e
5 Always 2 - wakeup on e
```

Note: Although the order of the `always` blocks is still arbitrary, both orderings cause the wakeup on event e to occur at time 5.

Persistent Trigger

The SystemVerilog persistent trigger is described in Section 14.5.4 of the IEEE 1800 standard.

In Verilog, events have no logic value or duration. Processes can watch for an event to trigger, but if a process is not watching at the exact moment that an event is triggered, the event will go undetected. SystemVerilog introduces the `triggered` property, which enhances the event data type by allowing an event to persist throughout the time step in which the event is triggered. The syntax is as follows:

`hierarchical_event_identifier``.triggered`

The `triggered` event property evaluates to true, as long as the given event is triggered within the current time unit. Otherwise, the triggered event property evaluates to false.

The addition of the `triggered` property helps resolve a common race condition, illustrated in the following example:

```
begin
-> eventA
...
end
...
@(eventA)
...
```

In this example, `eventA` must occur before the rest of the code can execute. This can cause a race condition when the event control and `eventA` occur at the same time. In this case, the `wait` might either unblock, or wait until the next `eventA`. You can rewrite this code using the `triggered` property and a `wait` statement:

```
begin
-> eventA
...
end
...
wait(eventA.triggered)
...
```

This will unblock the calling process as long as the `wait` executes before or at the same time unit as the event trigger.

When the `.triggered` property is used within the `wait()` construct, the condition waits for the *`event`*`.triggered` property to become true. If you reach the `wait()` statement and

the property is true, execution proceeds immediately; otherwise, it will wait for the property to become true.

The event.triggered construct is most useful when used within the wait() construct. While it seems natural to replace @(event) with @(event.triggered), combining the semantics of the .triggered property and edge-activated event control @() could produce undesired results.

Note: The current release does not support the `.triggered` property on OOMRs to a named event. You can only reference a named event from the scope in which it was declared.

Event Variables

Event variables are described in Section 14.7 of the IEEE 1800 standard.

In SystemVerilog, events behave like variables, in that they can be assigned to one another, assigned a special `null` value, or be compared against each other.

If you have two event type variables called `e1` and `e2`, you can:

- Assign one to the other (Section 14.7.1 of the IEEE 1800 standard)

  When an event is assigned to another event, the two events become merged and they share the synchronization queue of the event on the right-hand side of the assignment. However, the assignment affects only subsequent event control or wait operations. For example, if there are processes waiting on an event at the time it is merged with another event, the waiting process will never unblock.

```
e1 = e2;     //Merges the two event variables
             //After this, events waiting on e1 will never unblock
fork
  begin
   -> e1;    //Also triggers e2
  end

  begin
   wait(e1.triggered);  //Unblocks wait process
   ...
  end

  begin
   wait(e2.triggered);  //Also unblocks wait process
  end
```

- Reclaim an event (Section 14.7.2 of the IEEE standard)

You can reset an event type variable by assigning it a special `null` value. This disconnects the event variable from its synchronization queue, thus making the event variable's resources available again. For example:

```
event e1 = null; //Resets e1
```

Event controls and wait operations on a `null` event are undefined, and triggering a `null` event has no effect. For example:

```
@ e1;                 //Undefined
wait (e1.triggered); //Undefined
-> e1;                //No effect
```

- Compare them (Section 14.7.3 of the IEEE standard)

You can use comparison operators (==, `!=`, ===, and `!==`) to compare an event variable against another event variable, or against `null`. Or, you can check for a boolean value that will be 0 if the event is `null`, or 1 otherwise. For example:if (e1 == null)
$display("e1 is null");
if (e1)
e1 = e2; //Merges e1 and e2 if e1 is not null

14

Clocking Blocks

Clocking blocks are described in Section 15 of the IEEE 1800 standard.

Note: Clocking blocks are supported in the NC-Verilog simulator, the Incisive Simulator, the Incisive Design Team Simulator, and the Incisive Enterprise Simulator. Clocking blocks are not available with the Incisive HDL Simulator. For more information on the Incisive HDL Simulator, refer to the *Design Team Family Technology Overview*.

In Verilog-2001, module ports are used to model communication between blocks. SystemVerilog extends this by introducing the interface construct, which encapsulates the communication between blocks. Although an interface can specify the signals and nets through which a testbench communicates with its DUT, it cannot explicitly specify timing and synchronization requirements. To address this, SystemVerilog introduces the *clocking block* construct. A clocking block:

- Identifies a *clocking domain*, which encapsulates clock signals, and the timing and synchronization requirements of blocks wherein the clock is used.
- Separates time-related details from the structural, functional, and procedural elements of a testbench.
- Helps define the testbench based on transactions and cycles (cycle-based methodology), as opposed to signals and transition times (event-based methodology).
- Simplifies the creation of a testbench that does not have race conditions with the DUT.

For a complete example that uses clocking blocks, which you can download and run, refer to the *SystemVerilog Engineering Notebook*.

Declaring a Clocking Block

A simple clocking block declaration is as follows (see the SystemVerilog LRM, Syntax 15-1 for the complete syntax).

clocking [*clocking_identifier*] *clocking_event*;
clocking_items
endclocking[: *clocking_identifier*]

For example, the following defines a clocking block called `c1` that is to be clocked at signal `clk`.

```
module test_top;

wire clk;
wire a, b;

clocking c1 @clk;
  default input #1ns;
  default output #2ns;
  input posedge a;
  output negedge b;
endclocking: c1

endmodule
```

The second line of the clocking block declaration specifies a default input skew of `1ns`, which means the testbench should sample input signals at `1ns` before the clock edge. The third line specifies a default output skew of `2ns`, which means outputs should be driven `2ns` after the clock edge. The fourth and fifth lines illustrate how you can also reference a clock edge to sample a value or drive a stimulus.

The items and events specified within a clocking block declaration are order independent, which means there is no required ordering within the scope of a design unit.

Clocking blocks can be declared within modules, interfaces, and program blocks. Clocking blocks cannot be nested. They cannot be declared within functions, tasks, packages, or outside all declarations in a compilation unit.

A clocking block is both a declaration and an instance of that declaration. You do not need a separate instantiation.

You can access an individual signal within a declared clocking block by using its name, the dot (`.`) operator, and the signal identifier. For example:

```
c_counter.sig //Accesses sig in clocking block c_counter
```

You can access the clocking event of a clocking block directly, by using the clocking block's name. For example:

```
clocking dram@(posedge phi1);
     inout data;
endclocking
...
always @(dram) // Equivalent to @(posedge phi1)
...
```

The following lists the features that are supported in the IEEE 1800 standard, but are not supported in the Cadence implementation.

- The LRM specifies that names for clocking blocks are optional. The Cadence implementation supports only unnamed, *default* clocking blocks that do not contain clocking items. Unnamed, non-default clocking blocks are not allowed and produce an error message in the simulator. See "Defining Default Clocking Blocks" on page 189 for more information on default clocking blocks.
- The LRM allows clocking block declarations within `generate` loop statements. This is not supported in the Cadence implementation.

Defining Default Skews and Clocking Direction

SystemVerilog allows default skews for a single clocking block to be specified on multiple lines or on a single line. For example, you can use:

```
clocking c1 @clk;
     default input #10ns;
     default output #2ns;
     ...
endclocking: c1
```

or

```
clocking c1 @clk;
     default input #10ns output #2ns;
     ...
endclocking: c1
```

The SystemVerilog LRM gives the following syntax for defining default skews and clocking direction on a single line:

```
input [clocking_skew] output [clocking_skew];
```

The Cadence implementation allows the reverse order. For example:

```
output [clocking_skew] input [clocking_skew];
```

A clocking block cannot have multiple default skews with the same type. Although types can have independent delay controls and edge skews, they can have only one of each. For example, you cannot have multiple default `input` skews:

```
clocking c1 @clk;
     default input #10ns;
     default input #2ns; //Invalid
```

However, an edge skew default does not conflict with a delay control skew default. For example, the following specifies that all inputs will have a `1ns` input skew, but will also have a positive edge skew.

```
clocking c1 @clk;
     default input #1ns;
     default input posedge; //Valid
```

Defining Clocking Items

SystemVerilog allows you to declare clocking items among multiple lines. For example, the following lines:

```
clocking c1 @clk;
    input posedge a;
    output negedge a;
endclocking
```

are the same as:

```
clocking c1 @clk;
    input posedge output negedge a;
endclocking
```

When you declare clock items among multiple lines, SystemVerilog treats them as a summation of the various directions and skews. A skew is not unset if a clocking item is declared among multiple lines. For example:

```
clocking c1 @clk;
    input posedge a;
    input a;        //This does not unset anything
endclocking
```

Each direction of a clocking item (input or output) can have only one skew edge, either posedge or negedge, and one delay control. For example, you cannot have multiple skew edges and delay controls for the same direction of a clocking item:

```
clocking c1 @clk;
    input posedge a;
    input negedge a; // Invalid
    output negedge a;
    output #1 a;
    output #2 a;     // Invalid
endclocking
```

Using Hierarchical Expressions

In SystemVerilog, use a *hierarchical expression,* in place of a local port, to specify that the signal that will be associated with the clocking block is specified by its hierarchical name. A hierarchical expression is introduced using the equal sign (=). For example:

```
clocking c1 @clk;
     input a = top.cpu.state;
endclocking
```

You cannot designate multiple hierarchical expressions for the same clocking item. In the following example, the second designation causes an error message because a already has the hierarchical expression `top.m.a`:

```
clocking c1 @clk;
    input a = top.m.a;
    input posedge a = top.m.b; //Invalid
endclocking
```

The following lists differences between the IEEE 1800 standard and the Cadence implementation:

- The Cadence implementation does not support the following use of hierarchical expressions:

  ```
  clocking c @clk;
       input a.b;
  endclocking
  ```

  Instead, the Cadence implementation supports the following:

  ```
  clocking c @clk;
       input in = a.b;
  endclocking
  ```

- The Cadence implementation does not support the concatenation `{...}` syntax used in the example in Section 15.4 of the IEEE 1800 standard. Instead, the Cadence implementation supports the following usage:

  ```
  clocking mem @(clock);
       input instruction = top.cpu.instr;
  endclocking
  ```

Defining Default Clocking Blocks

You can specify a default clocking block for all cycle delay operations that occur within a given module, interface, or program. You can only specify one default clocking block within a module, interface, or program, and that clocking block is valid within only the scope containing the clocking specification. In the following example, the default clocking block `c1` is valid only within the scope of module `m1`:

```
module m1;
     wire a;
     wire clk;
     default clocking c1;
        input a;
     endclocking
endmodule
```

The LRM Syntax 15-4 allows a short-form designation of a default clocking block, where the default is designated separately from its declaration. For example:

```
module top;
     wire a;
```

```
     wire clk;
     default clocking c1; //Designates the default clocking block

     clocking c1 @clk;   //Clocking block declaration
     input a;
     endclocking
endmodule
```

Note: The implied use of the short-form designation is for nested modules and interfaces. The simulator does not support nested design units. Thus, the Cadence implementation supports only clocking blocks in non-nested design units.

Specifying Cycle Delays and Clocking Drives

The ## operator is used in the testbench to delay execution by a specified number of clocking events, or clock cycles. This is called a *cycle delay*. For example:

```
## 2 // Wait 2 clocking events using the default clocking block
```

The makeup of a clocking event depends on the default clocking block. If a default clocking block has not been specified for the current module, interface, or program, then the compiler issues an error message.

Cycle delays can be used in two kinds of statements: `procedural_timing_control_statement` and `clocking_drive.`

The Verilog `procedural_timing_control_statement` thread allows cycle delays to be specified alone. For example:

```
initial
  begin
   ##1;
  end
```

However, the Verilog `procedural_timing_control_statement` thread does not allow cycle delays to exist on the right-hand side of statements. For example:

```
default clocking c1 @clk;
 output a;
endclocking

initial
 begin
  c1.a <= ##1 b; //Not allowed by procedural_timing_control_statement
 end
```

To account for this, SystemVerilog introduces the `clocking_drive` statement, which allows cycle delays to exist on the right-hand side of statements. For example, the following statement is valid in SystemVerilog, and it specifies to remember the value of b, and then drive `Data` two clock cycles later:

```
cl.Data <= ##2 b;
```

The following summarizes the Cadence implementation for clocking drives:

- You can only use cycle delays on the right-hand side of non-blocking assignments, and the left-hand value must be a clocking item. For example:

  ```
  cl.a = ##1 b; //Invalid, the statement must be non-blocking
  a <= ##1 b;   //Invalid, the left-hand side must be a clocking item
  cl.a <= ##1 b; //Valid, granted cl.a is a clocking item
  ```

- When a cycle delay is specified on the left-hand side of a non-blocking assignment, and the left-hand value is a clocking item, then it is considered a clocking drive. Otherwise, it is considered a `procedural_timing_control_statement` thread. For example:

  ```
  ##1 cl.a <= b; //Considered a clocking drive
  ##1 a <= b;    //Considered a procedural_timing_control_statement
                 //followed by a non-blocking assignment
  ```

- A cycle delay, regardless of whether it exists on the left-hand or right-hand side of an assignment, is defined by the clocking block of the signal being driven. For example:

  ```
  ##1 bus.Data <= 8' hz; //Wait 1 bus cycle, then drive Data
  bus.Data <= ##2 8'hz;  //Wait 2 bus cycles, then drive Data
  ```

- When a cycle delay exits on both sides of an assignment, it is considered a `procedural_timing_control`, followed by a clocking drive. For example:

  ```
  ##1 c1.a <= ##2 b;
  ```

  is the same as:

  ```
  ##1;
  c1.a <= ##2 b;
  ```

Debugging Clocking Blocks

For information on how to debug clocking blocks using the Tcl command-line interface or the SimVision analysis environment, refer to *SystemVerilog in Simulation*.

15

Program Blocks

Program blocks are described in Section 16 of the IEEE 1800 standard.

Note: Program blocks are supported in the NC-Verilog simulator, the Incisive Simulator, the Incisive Design Team Simulator, and the Incisive Enterprise Simulator. Program blocks are not available with the Incisive HDL Simulator. For more information on the Incisive HDL Simulator, refer to the *Design Team Family Technology Overview*.

SystemVerilog introduces a *program block* construct. A program block, similar to a module, facilitates the creation of a testbench, but has special syntax and semantic restrictions. A program block:

- Provides an entry point to the execution of testbenches
- Acts as a scope for the data contained in the program block
- Provides a syntactic context that schedules events in the reactive region
- Uses a special `$exit()` system task

For a complete example on that uses program blocks that you can download and run, refer to the *SystemVerilog Engineering Notebook*.

Declaring a Program Block

A simple `program` declaration is as follows (see the SystemVerilog LRM, Syntax 16-1 for the complete syntax):

```
program program_identifier[(port_list)];
program_items
endprogram[: program_identifier]
```

Although program blocks and modules use different keywords, they follow the same general format. For example, port declarations and end labeling are the same in both constructs.

Supported Constructs for Program Blocks

Program blocks are limited in the type of constructs they can contain. Specifically, the simulator supports the following BNF constructs within a program block:

- `class_constructor_declaration`
- `class_declaration`
- `clocking_declaration`
- `concurrent_assertion_item`
- `concurrent_assertion_item_declaration`
- `continuous_assign`
- `covergroup_declaration`
- `data_declaration`
- `function_declaration`
- `genvar_declaration`
- `initial_construct`
- `local_parameter_declaration`
- `module_or_generate_item_declaration`
- `net_declaration`
- `non_port_program_item`
- `overload_declaration`
- `package_or_generate_item_declaration`
- `parameter_declaration`
- `specparam_declaration`
- `timeunits_declaration`

Unsupported Constructs

The IEEE 1800 standard indicates that you cannot include instantiation objects, `generate` blocks, `specify` blocks, `defparams`, `always` blocks, UDPs, modules, interfaces, or other programs within a program block. Specifically, the simulator does not support the following BNF constructs within a program declaration:

- `generated_module_instantiation`
- `specify_block`
- `program_declaration`
- `module_declaration`
- `parameter_override`
- `gate_instantiation`
- `udp_instantiation`
- `module_instantiation`
- `interface_instantiation`
- `program_instantiation`
- `bind_directive`
- `net_alias`
- `final_construct`
- `always_construct`

Nesting Program Blocks

The IEEE 1800 standard states that program declarations can be nested within modules or interfaces. For example:

```
module test(...)
int shared;

program p;
. . .
endprogram: p

program p1;
. . .
endprogram: p1

endmodule:test
```

Note: The Cadence implementation *does not* support program block declarations that are nested within modules, packages, or interface declarations. Also, program block declarations are allowed only at the top-most level.

Working with Variable Assignments

Use blocking assignments (=) to update values of variables that are local to a program block, and use non-blocking assignments (<=) to update non-program variables (for example, module variables). If you use a non-blocking assignment with a program variable, or a blocking assignment with a non-program variable, you will get an error message. For example:

```
module design(input wire A);
 int B, C;
endmodule

program test(output reg A);

 integer int_number;
 reg a;

 initial begin

 //Valid variable assignments
 top.t.int_number = 15;      //Program variable
 top.d.C <= 3;              // Non-program variable

 //Invalid variable assignments
 top.t.a <= int_number + 1;    //Program variable
 top.d.B = 5;                  //Non-program variable

 end

endprogram

module top;
    wire A;
    design d(A);
    test t(A);
endmodule
```

Referencing Program Block Variables

References to program block variables can exist within program blocks. However, the IEEE 1800 standard does not allow references to program block variables from outside a program block.

You can reference program block instances from within modules in the traditional hierarchical fashion. However, you cannot reference program block instances from within program blocks.

Instantiating Program Blocks

Program blocks, modules, and primitives are instantiated in the same way. The current release does not support arrays of instances within program blocks. The current implementation does not support program block instantiation to other languages, such as VHDL or SystemC.

New Program Design Unit

Although program blocks are very similar to modules, the LRM definition classifies them as a different type of Verilog design unit. To account for this, the NC library system has been enhanced to manage the new design unit type.

If you use the `-messages` option when you compile your source files, the output displays `program`. For example:

```
% ncvlog -nocopyright -messages -sv test.v
file: test.v
     program worklib.P
           errors: 0, warnings: 0
...
```

The default view name for a program is `program`. For example, `worklib.P:program`.

You can query the library system for program objects using the *ncls* utility with the `-program` option.

```
% ncls -program
```

Understanding the $exit() Control Task

Aside from normal simulation tasks, like `$stop` and `$finish`, a program can use the `$exit` control task to terminate a program block. The following summarizes the functionality of the `$exit` control task within the Cadence implementation:

- `$exit()` is a system task that can be called only within program blocks.

  You cannot invoke or enable the `$exit()` call from within a function. You can, however, invoke the `$exit()` task from an `initial` block or task within a program block.

- Program blocks can call `$exit()` explicitly, or implicitly.

A program block must contain an `initial` block in order to implicitly call `$exit()`. In the implicit case, the `$exit()` task is called after all `initial` blocks execute their last statement, regardless of whether that statement has events or processes that occur at a later time. In the following example, `top.p1.r` goes through only two transitions (from unset to `1`, and then from `1` to `0`). The remaining transitions are disabled because a call to `$exit()` exists within the program block.

```
program P1;
    integer r;
    initial
        #1 r = 1;
    initial
        #10 $display("Keep this going!");
endprogram

program P2;
    initial
      begin
        #2 top.p1.r = 0;
        #2 top.p1.r = 1;
        $display("P2 - First Initial Block.");
      end
    initial
        #3 $exit();

    initial
        forever @(top.p1.r)
        $display("r: %b",top.p1.r);
endprogram

module top;
P1 p1();
P2 p2();
endmodule
```

- A call to `$exit()` terminates the program block.

  A call to `$exit()` disables all `initial` blocks in the specified program block, and their sub-processes (following the standard rules for disable). In the following example, the call to `$exit()` disables both `initial` blocks, including the first block that has already run.

  ```
  program P1;
          initial
               $display("First");

          initial
              #2 $exit();
  endprogram
  ```

  The simulator does not terminate continuous assignments and tasks/functions that are defined within the program block, but are called from other blocks. The simulator will, however, terminate outstanding non-blocking assignments that are sub-processes of any of the `initial` blocks.

If the `$exit()` call is within a task, the simulator terminates the program block that has the `initial` block with the call to `$exit()`—not the program block in which the task is defined.

- The simulator calls `$finish()` when all program blocks have exited (either implicitly or explicitly).

  In the following example, each program block has an `initial` block that runs with an implicit call to `$exit()`, but only the last implicit call to `$exit()` triggers the `$finish()` call.

```
...
program P2;
      initial
          begin
            $display("In program block P2.");
          end

endprogram

program P;
     initial
          begin
            $display("In program block P.");
          end

endprogram
...
```

- Calling `$exit()` terminates all processes spawned by the current program.

16

Assertions

SystemVerilog assertions are available only if you have an Incisive license. Support for SystemVerilog assertions is documented in the *Assertion Writing Guide* and in the *SVA Quick Reference Guide*.

Immediate Assertions

Support for the SystemVerilog concurrent assertions is documented in the *Assertion Writing Guide* and in the *SVA Quick Reference Guide*.

Concurrent Assertions

Support for the SystemVerilog concurrent assertions is documented in the *Assertion Writing Guide* and in the *SVA Quick Reference Guide*.

17

Hierarchy

Chapter 19 of the IEEE 1800 standard describes the SystemVerilog enhancements for representing design hierarchy.

Packages

Packages are described in Section 19.2 of the IEEE 1800 standard.

For information on how to compile a design with packages, refer to "Compiling a Design with Packages" in *SystemVerilog in Simulation*.

SystemVerilog introduces a package construct to the Verilog language. A package is a new Verilog design unit that contains declarations that can be shared among modules, macromodules, interfaces, programs, or other packages.

In the following simple example, package `global_types` defines some commonly-used types. The package is imported by the module `error_checks`, and the `boolean` type from the package is used as the type of the variable `suppress_warnings`:

```
package global_types;
  typedef enum logic [1:0] { FALSE, TRUE } boolean;
  typedef enum logic [2:0] { H=1'b1, L=1'b0, Z=1'bz, X=1'bx } logic_state;
endpackage
import global_types::*;
module error_checks;
  ...
  boolean suppress_warnings;
  ...
endmodule
```

A common use of a package is to group a type declaration and a set of tasks or functions that operate on that type. In this scenario, a module could then declare objects of the type and use the package tasks/functions to operate on the object data.

A second common use for a package is to define a utility for common use. This sort of package often makes use of persistent state and non-reentrancy in its implementation. The following package example shows a combination of both sorts of use. This example implements a common error reporting utility.

```
package messages;
  typedef [80*8:1] message_type;
  integer error_count = 0;
  integer warning_count = 0;
  integer error_limit = 1;

  task report_warning;
    input message_type message;
    begin
      $display(“Warning at %0t: %0s”, $time, message);
      warning_count = warning_count + 1;
    end
  endtask

  task report_error;
    input message_type message;
    begin
      $display(“Error at %0t: %0s \n”, $time, message);
      error_count = error_count + 1;
      if (error_count == error_limit) end_simulation;
    end
  endtask

  task end_simulation;
    begin
      $display (“ !! ERROR LIMIT EXCEEDED !!”);
      $display (“ Warnings: %d Errors: %d\n”, warning_count, error_count);
      $finish;
    end
  endtask

endpackage

import messages::* ;
module testbench;
  ...
  if (bad_condition) report_error(“Unexpected ...”);
  ...
endmodule
```

A package defines a single, global set of items that can be used by any design unit that imports that package. Unlike modules, packages cannot be used as structural building blocks to create multiple copies. However, a package may build on top of another package by importing the other package.

The declarations in a package are independent of the structural design hierarchy. Packages do not contain hierarchy, nor do they contain references to global typedefs or items declared in modules and primitives. The declarations within a package cannot contain hierarchical references, unless they refer to items created within the package or to items made visible by the importing of another package. Packages cannot reference items defined within

compilation unit scopes. However, structural elements can depend on items in a package. For example, a module can connect a global supply in a package to a lower-level component.

In SystemVerilog, you cannot have multiple packages with the same name, even if the packages are compiled into different libraries. The parser generates an error if more than one package with the same name exists.

Declaring a Package

Section 19.2 of the IEEE 1800 standard describes the package declaration syntax. Not all of the package items specified in this syntax are supported in the current release.

Attribute instances on package declarations and on the items within a package are supported.

The list of declarations supported within a package in the current release is as follows:

- net_declaration
- data_declaration
- task_declaration
- function_declaration
- dpi_import_export
- class_declaration
- class_constructor_declaration
- parameter_declaration
- local_parameter_declaration
- timeunits_declaration
- concurrent_assertion_item_declaration

The following declarations, listed in the LRM, are not supported in the current release:

- anonymous_program
- extern_constraint_declaration
- covergroup_declaration
- overload_declaration

Referencing Data in a Package

There are two ways to use the declarations contained in a package:

- Reference a package declaration by using its *package item reference full name*. The syntax is:

  package_identifier::*item_name*

  In the following example, the `boolean` type and the `logic_state` type are referenced using their package item reference full name.

```
package global_types;
  typedef enum logic [1:0] { FALSE, TRUE } boolean;
  typedef enum logic [2:0] { H=1'b1, L=1'b0, Z=1'bz, X=1'bx } logic_state;
endpackage

module error_checks;
  ...
  global_types::boolean suppress_warnings = global_types::FALSE;
  global_types::logic_state initial_state = global_types::X;
  ...
endmodule
```

- Use the `import` statement to provide direct visibility of identifiers within a package. The `import` statement allows all or selected identifiers declared in a package to be visible within the current scope. If an identifier declared in a package is imported, you can refer to the item by its simple name, without using a package name qualifier.

Note: You cannot use a Verilog out-of-module reference to refer to an item declared in a package. For example, if variable `var` is declared in a package called `pack`, and the variable is imported into module `top`, you cannot use `top.var` to refer to the package item. You must use the package reference name (`pack::var`) or the simple name (`var`).

Controlling Visibility of Names within Packages: The import Statement

The `import` statement provides control over how and which package items are imported.

The `import` statement can be placed:

- Outside a design unit (package, module, UDP, interface, program). The scope of an `import` statement that appears outside of a design unit declaration extends to the end of the compilation unit. Such an `import` statement affects all following design units in the compilation unit.
- Inside any declarative scope of a design unit. The scope of an `import` statement inside a declarative scope extends from where it is declared to the end of the declarative scope.

The `import` statement has two forms: wildcard import and explicit import.

Wildcard import Statement

The syntax for wildcard import is:

```
import package_identifier::*;
```

For example:

```
import IObus_package::*;
```

Example

In the following example, there are three source files:

- `package.v` contains the shared types and declarations.
- `rtl.v` is the RTL code of the design. This must import the package `globals`, as it uses the types and variables declared in this package.
- `test.v` is the testbench code, which also must import the package. In the source file `test.v`, there is a single `import` statement that applies to both the module `stimulus` and the module `testbench`.

```
// File: package.v
package globals;
  typedef enum logic { FALSE, TRUE } boolean;
  integer error_count;
endpackage

// File: rtl.v
import globals::* ;
module rtl;
  ...
endmodule

// File: test.v
import globals::* ;
module stimulus;
  ...
endmodule

module testbench;
  ...
endmodule
```

Importing the package with `::*` provides potential direct visibility of any of its contents in the importing scope. In the following example, `gnd` and `vdd`, if referenced inside the importing module by their simple names, would have their declaration from the package made locally directly visible in the importing module. Since `gnd` is not referenced inside module `top`, its declaration is not imported. Since `vdd` is referenced by `r = vdd` and `globals::`, it is imported. However, the local declaration that follows causes a duplicate symbol error.

```
// File: package.v
package globals;
```

```
  integer gnd;
  reg vdd;
endpackage

module top;
  reg r;

  import globals::*;

  initial
   r = vdd;   // This is globals::vdd; it is imported because there
               // is no local declaration for vdd before this reference.

  reg vdd;   //Declare vdd. Error because there are 2 visible declarations for vdd.

endmodule
```

In the following example, `globals::vdd` is not imported because `vdd` is locally declared before it is referenced.

```
// File: package.v
package globals;
  int gnd;
  reg vdd;
endpackage

module top;
  reg r;

  import globals::*;
  reg vdd; // Declare vdd.

  initial
    r = vdd;      // This is reg vdd; globals::vdd is not imported because vdd is
                  // locally declared before this reference.
endmodule
```

If the same symbol is imported from two or more different packages with a wildcard import, a direct reference to that symbol in the importing design unit is an error because the declaration it refers to is ambiguous. However, a package item reference full name can be used to disambiguate the origin of the declaration.

In the following example, the first `initial` block contains an error because there is a direct reference to `vdd`, which is defined in two packages.

```
package p;
 reg vdd;
endpackage

package q;
 reg vdd;
endpackage

import p::*;
import q::*;
module top;
  reg r;
```

```
  reg r1;
  initial
    r = vdd;     // Error because vdd is ambiguously defined

  initial
    r1 = p::vdd; // vdd from package p

  initial
    r2 = q::vdd; // vdd from package q

endmodule
```

Explicit import Statement

The explicit `import` statement allows precise control of the symbols to be imported. The syntax is as follows:

```
import package_identifier::identifier[,package_identifier::identifier ...];
```

With an explicit import, only the symbols referenced by the import are made directly visible. All other package items are not directly visible. A package reference full name must be used to refer to the package items not made directly visible by the `import` statement.

In the following example, the module `top` explicitly imports two symbols: `error_count` and `vdd`. They are both directly visible inside module `top`, even though only `vdd` is used to initialize register `r`. The declaration `ground` cannot be directly referenced by its simple name because it is not an imported item, and a package item reference full name (`globals::ground`) is used to refer to the `ground` declaration.

```
import globals::error_count, globals::vdd;
module top;
  reg r;
  reg r0 = globals::ground;

  initial
  r = vdd;
endmodule
```

An explicit import is illegal if the imported declaration is already declared in the same scope, or if it is explicitly imported from another package. However, it is legal to import the same declared item from the same package multiple times.

Debugging Packages

For information on how to debug packages using the Tcl command-line interface or the SimVision analysis environment, refer to *SystemVerilog in Simulation*.

Compilation Units

Compilation units are described in Section 19.3 of the IEEE 1800 standard.

For an example that you can download and run, refer to the “Disabling DPI Tasks and Functions” example in the *SystemVerilog DPI Engineering Notebook*.

SystemVerilog adds a concept called *compilation units*. A compilation unit is a collection of one or more source files compiled together.

SystemVerilog extends Verilog by allowing declarations outside of a module, interface, package, or program. Each compilation unit has a *compilation unit scope*, which contains all of the external declarations made across all the files within the compilation unit. Unlike global declarations, which are shared by all the modules that make up a design, compilation unit scope declarations are visible only to the source files that make up the compilation unit.

By default, all the files on a given compilation command line make up a single compilation unit. To create a separate compilation unit for each source file, you must compile each source file separately.

For example:

file_a.v

```
reg r;
module a;
initial begin
r= 1’b0;
```

file_b.v

```
end
endmodule

module b;
initial begin
r = 1;
end
endmodule
```

file_c.v:

```
module c;
assign r = 1’b1;
endmodule
```

Default implementation:

```
% ncvlog -sv file_a.v file_b.v file_c.v
```

Separate compilation units:

```
% ncvlog -sv file_a.v
% ncvlog -sv file_b.v
% ncvlog -sv file_c.v
```

Using the default mode, where every source file on a given command line is grouped under a single compilation unit, the imported package `pkg1` would be accessible by `file_b.v`. However, if `file_a.v` and `file_b.v` are compiled separately, the reference to `r` in `file_b.v` creates an implicit net.

As defined by the LRM, compilation unit scopes cannot access items within other compilation unit scopes.

Supported External Declarations

SystemVerilog extends Verilog by allowing declarations outside of a module, interface, package, or program. For example:

```
<declarations>
module;
endmodule
```

All of these external declarations in a compilation unit make up the *compilation-unit scope* and can by accessed by any of the constructs defined within the compilation unit.

In the current release, compilation-unit scopes can include the following types of external declarations:

- `bind` directives
- classes
- structures
- package import declarations
- `timeprecision` and `timeunit`
- `‘timescale` and `‘include` directives
- task and function declarations
- variable and net declarations
- constant declarations
- parameters
- user-defined types that use `typedef`, `enum`, or `class`

Limitations on Compilation Units

Properties, sequences, and specialized classes are not supported within a compilation unit.

Explicitly Referencing External Declarations

In SystemVerilog, you can explicitly reference a declaration within a compilation unit scope using `$unit` and the class scope resolution operator (Section 7.21 of the LRM). For example:

```
bit b;
task foo;
  int b;
  b = 5 + $unit::b;
endtask
module
...
endmodule
```

Port Declarations

Port declarations are described in Section 19.8 of the IEEE standard.

In Verilog-2001, input ports cannot be declared as variables, while output ports can be declared as variables, but must be connected to a wire. SystemVerilog removes these restrictions.

In SystemVerilog a port can be declared as an interface, or a variable or net of any allowed data type. The syntax for this type of declaration is as follows:

port_direction port_kind data_type

Where *port_kind* can be the net type keywords or the keyword *var*, which are used to declare net and variable assignments.

For example, the following declares two ports of the packed structure type `my_type`.

```
typedef struct packed {
  logic b;
  int i;
} my_type;
...
module mysub(input var my_type in, output my_type out);
  always @(in)
  out = in;
endmodule
```

The following table outlines the default rules used when keywords are omitted from a port declaration.

Port Direction	Port Kind	Data Type	Default
Unspecified	Specified	Unspecified	In a port list, the port direction is inherited from the previous port. For the first port in a list or a standalone declaration, the port direction defaults to `inout`.
Specified	Unspecified	Unspecified	The port defaults to a net of net type `wire`. You can change the default net type using the Verilog `‘default_nettype` compiler directive.
`input` `inout`	Unspecified	Either	The port kind defaults to a net of net type `wire`. You can change the default net type using the Verilog `‘default_nettype` compiler directive.
`output`	Unspecified	Either	The default port kind is based on the port data type. If the data type is not specified, then the port kind defaults to a net of the default net type. If the data type is specified, the port kind defaults to variable.

The current release supports the following variable data types on ports:

- `bit`
- `shortint`
- `int`
- `longint`
- `logic`
- `byte`
- `enum`
- `reg`
- `real`
- `string`
- classes

- packed structures
- queues
- dynamic arrays
- associative arrays
- packed arrays
- unpacked structures
- unpacked arrays

The current release does not support the event data type on ports.

18

Interfaces

Interfaces are described in Section 20 of the IEEE 1800 standard.

For a complete example that uses interfaces, which you can download and run, refer to the *SystemVerilog Engineering Notebook*.

One of the major extensions to the Verilog language proposed in the SystemVerilog LRM is the *interface* construct. This construct was created to encapsulate the communication between blocks of a digital system.

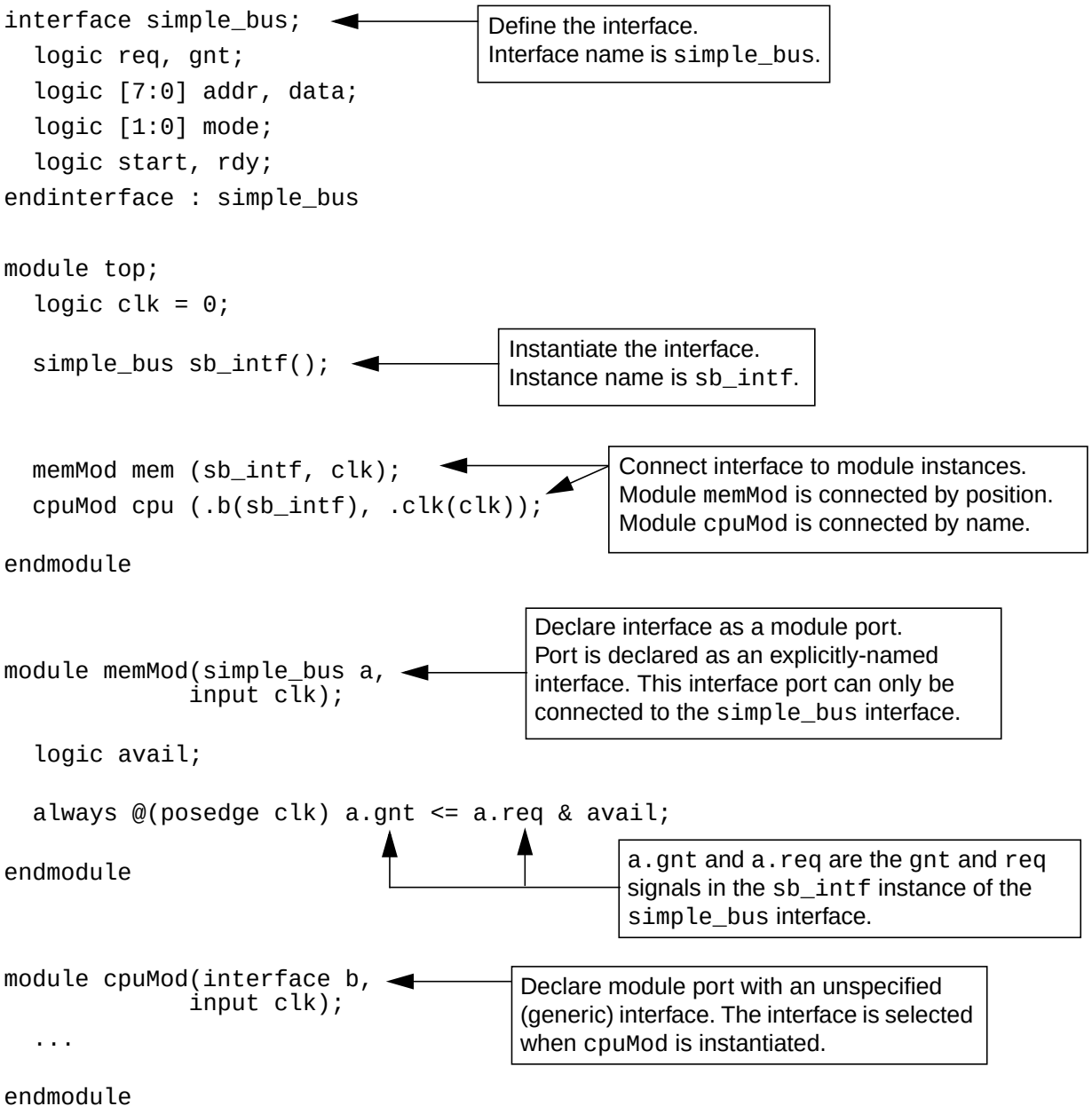
At its lowest level, a SystemVerilog interface is a named bundle of nets or variables that encapsulates the connectivity between blocks. By declaring an interface, you can define a group of signals once in one modeling block. The interface can then be instantiated in the design and can be accessed through a module port as a single item. This eliminates redundant declarations of the same signals in multiple modules, which can significantly reduce the size of a description. Grouping signals together in one place also improves design maintainability. For example, if a change to the port specification is required, the change can be made in one place instead of in multiple modules.

At a higher level, an interface can encapsulate functionality in addition to connectivity. An interface can contain data type declarations, tasks and functions, `initial` and `always` blocks, continuous assignments, and so on. This allows you to define communication protocols, protocol checking routines, and other verification routines in one place.

This section provides details on the functionality provided in the current release for SystemVerilog interfaces.

Figure on page 216 shows an example design with a simple interface that bundles a collection of signals. This example, modified from an example shown in the SystemVerilog LRM, shows the basic syntax for defining, instantiating, and connecting an interface.

Example 18-1 Simple Interface Example



Declaring an Interface

The syntax for an interface declaration is as follows:

```
interface interface_identifier [(port_list)];
```

interface_items
endinterface [: *interface_identifier*]

Any construct that you can use in a module can be used in an interface, with the following exceptions. The IEEE 1800 standard does not allow the following constructs in an interface, and these constructs are not supported in the simulator:

- `defparam` statements
- `specify` blocks
- `alias` statements
- Instances of program blocks, modules, and primitives

Example 18-2 Nested Interface Example

```
interface ITest;
   int a;
   int b;
   always @(a) $display( $time, ":  %m.a = %0d", a );
   always @(b) $display( $time, ":  %m.b = %0d", b );
endinterface

interface ITop;
   ITest   TestIf1();
   ITest   TestIf2();
endinterface

module SubMod(ITest If_test, input int in1);
   initial begin
    #210  If_test.a = (1000 * in1) + 11;
    #10  If_test.b = (1000 * in1) + 22;
   end
endmodule

module TestEnv(ITop If_port);
 SubMod sm1(If_port.TestIf1,1);
 SubMod sm2(If_port.TestIf2,2);
   initial begin
    #110  If_port.TestIf1.a = 103;
    #10  If_port.TestIf2.a = 203;
    #10  If_port.TestIf1.b = 105;
    #10  If_port.TestIf2.b = 205;
   end
endmodule

module test_top();
   ITop main_if();
   TestEnv env(main_if);
   initial begin
     #10  main_if.TestIf1.a = 107;
     #10  main_if.TestIf2.a = 207;
     #10  main_if.TestIf1.b = 109;
     #10  main_if.TestIf2.b = 209;
   end
endmodule
```

Inner interface

Outer interface

Connects the inner interface to the module instance.

Connects outer interface to the module instance.

Connects child interface instances relative to the interface port.
In the current release, you cannot use OOMRs, and you can use only two dot components (the first must be the interface port, and the second must be the child interface).

Connects the outer interface "`main_if`".

When simulated, this example should produce the following results:

```
ncsim> run
  10:  test_top.main_if.TestIf1.a = 107
  20:  test_top.main_if.TestIf2.a = 207
  30:  test_top.main_if.TestIf1.b = 109
  ...
  220:  test_top.main_if.TestIf2.b = 2022
ncsim: *W,RNQUIE: Simulation is complete.
```

Creating Design Units

Although interfaces are very similar to modules, the LRM definition classifies them as a different type of Verilog design unit. To account for this, the NC library system has been enhanced to manage the new design unit type.

If you use the `-messages` option when you compile your source files, the output displays `interface`. For example,

```
% ncvlog -nocopyright -messages -sv source.v
file: source.v
        interface worklib.simple_bus
                errors: 0, warnings: 0
...
...
```

The default view name for an interface is `interface`. For example,

```
worklib.simple_bus:interface
```

The *ncls* utility includes a `-interface` command-line option so that you can query the library system for compiled interfaces. For example:

```
% ncls -nocopyright -interface
        interface worklib.simple_bus:interface (VST)
        interface worklib.simple_bus:interface (SIG) <0x7a6d9d7a>
```

Using the Interface as a Module Port

Interface ports on modules can be declared in the following ways:

- You can declare a module port as a specific type of interface using the following syntax:

  ```
  module module_name (interface_name port_name, other_module_ports);
  ```

  In the example shown in Figure on page 216, module `memMod` contains an explicitly-named interface port.

  ```
  module memMod(simple_bus a, input clk);
  ```

  This port can be connected only to the interface named `simple_bus`.

  You can also use non-ANSI style declarations. For example:

  ```
  module memMod(a, clk);
    simple_bus a;
    input clk;
    ...
  ```

- You can declare a module port as a generic interface port using the following syntax:

```
module module_name (interface port_name, other_module_ports);
```

In the example shown in Figure on page 216, module `cpuMod` contains a generic interface port.

```
module cpuMod(interface b, input clk);
```

This port can be connected to any interface when the module is instantiated.

The LRM states that a generic interface reference "can only be declared by using the list of port declaration style of reference. It shall be illegal to declare such a generic interface reference using the old Verilog-1995 list of port style." The simulator allows these non-ANSI style declarations. For example:

```
module cpuMod(b, clk);
  interface b;
  input clk;
  ...
```

If you use non-ANSI style generic declarations, a warning message is issued saying that this is not official syntax according to the current LRM.

- You can declare interface array ports, where a single dimension array limit is specified after the port name:

  ```
  module module_name (interface port_name array_dimension);
  ```

  In the following example, module `Dut` connects to an explicitly named interface port with a dimension width of 4. This port can be connected only to the interface named `Ifc`:

  ```
  module Dut (Ifc prta [4:1]);
  ...
  ```

  For more information, refer to "Interface Array Ports."

Note: A port that is declared as an interface must be connected to an interface instance. An error is generated if an interface port is left unconnected.

Limitations on Interfaces

The following are allowed in the IEEE 1800 standard, but are not supported in the current release of the simulator:

- Interfaces connected to ports of the interface.
- Gates and UDPs.
- Wires and ports are supported, but the wires and ports declared in an interface must be digital wires. Analog wires within interfaces are not supported.
- Instances of interfaces. In the current release, simple interface instances can be nested within another interface. For example:

```
interface Inner;
  int a;
endinterface

interface Outer;
  Inner inner1();   //Supported
endinterface
```

However, arrays of instances are not supported within interfaces and will generate a parser error. For example:

```
interface Outer;
  Inner inner[1:4]();   //Not supported
endinterface
```

```
ncvlog: *E,INFINS (test.v,13|19): Instances of modules, interface arrays,
program blocks, and primitives are not allowed within interface or program
block definitions.'[SystemVerilog]'.
```

- An interface is instantiated in the same way that modules and primitives are instantiated. Interfaces must be instantiated at the highest point in the hierarchy where they can be used. In the current release, interface instances are limited to single instances. Interface arrays and interfaces in for-generates or in if-generates are not supported.

Interface Array Ports

The current release supports interface array port connections, where you place one dimensional array limits after the port name. For example:

```
module Dut ( Ifc prta [7:1] );
...
endmodule
```

For a complete example that uses interfaces, which you can download and run, refer to the *SystemVerilog Engineering Notebook*.

Supported Uses for Interface Array Ports

In the current release, you can

- Assign a bit select of an interface array port to a virtual interface variable:

  ```
  virtual interface Ifc vi = prta[4];
  ```

- Call a task or function through a bit select of an interface array port:

  ```
  initial prta[5].mytask();
  ```

- Access an interface variable using a bit select of an interface array port:

  ```
  initial prta[6].a = 5;
  ```

- Pass an interface array port to a submodule:

```
    Submod sm1 ( smprta(prta) );
```

Note: All bit selects must be constant expressions. The constant expression index cannot have an X or Z value, and must lie within the defined array limits.

Using Arrays of Interfaces in Interface Array Ports

In the current release, support for interface array port connections follows the Verilog 2001 array of instances port connection rules. For example, the following declares an array of interfaces and passes it to an instance of module `Dut`; this connects the array of interfaces to the interface array port called `prta`:

```
interface Ifc();
...
endinterface

module test_top();
  Ifc ifca [4:1] (); //Array of interfaces instantiation
  Dut dut1(.prta(ifca)); //Module instance
  ...
endmodule

module Dut (Ifc prta [4:1]); //Interface array port
...
endmodule
```

According to the LRM, in interface array port connections, the array widths must match. For example, the following is illegal because the widths of the array of interfaces and the module instance do not match:

```
module top();
  Ifc ifca [3:1] (); //Array of interfaces instantiation
  Dut duta [4:1]( .prt(ifca) ); //Module instance. Width is not the same.
endmodule

module Dut ( Ifc prt );
..
```

The following causes an error, because the array of interfaces and the interface array port have different widths:

```
module top();
  Ifc ifca [4:1] (); //Array of interfaces instantiation
  Dut dut1 ( .prta(ifca) );
endmodule

module Dut ( Ifc prta [5:6] ); //Interface array port. Widths do not match.
...
```

The following also causes an error, because the array of interfaces width is not equal to the module instance width times the interface array port width:

```
module top();
  Ifc ifca [22:1] (); //Should be 21:1
  Dut duta [3:1] ( .prta(ifca) );
```

```
endmodule : top

module Dut ( Ifc prta [7:1] );
...
endmodule
```

You can also pass a bit select of an array of interfaces to the module instance. For example, the following passes a bit select of array `ifca` to module instance `dut1`, thus connecting it to the interface array port `prta`:

```
module top();
  Ifc ifca [4:1] (); //Array of interfaces instantiation
  Dut dut1(.prta(ifca[2]); //Module instance.

module Dut (Ifc prta); //Scalar interface port
...
endmodule
```

However, according to the LRM, you cannot pass a bit select interface array to an interface array port:

```
module top();
  Ifc ifca [4:1];
  Dut dut1(.prta(ifca[4]);

module Dut (Ifc prta [5:6]); //Illegal
...
endmodule
```

The current release also supports implied bit selects of an array of interfaces to module instances:

```
module top();
Ifc ifca [2:4] ();
Dut duta [9:7] (.prt(ifca)); //Connects ifca to interface port of every child
endmodule                    //array using an implied bit select.

module Dut (Ifc prt);
...
endmodule
```

Arrays of interfaces can also connect through implied part selects:

```
module top();
Ifc ifca [6:1] ();
Dut duta [3:1] (.prta(ifca)); //Connects ifca to interface port of every child array
endmodule                    //using an implied part select.

module Dut (Ifc prta [2:1]);
...
endmodule
```

According to the LRM, you cannot pass an interface instance to an interface array port:

```
module top();
  Ifc ifc1();
  Dut dut1 ( .prt(ifc1) ); //Illegal. Passes interface instance ifc1
endmodule : top
```

```
module Dut ( Ifc prt [3:1] );
...
```

Limitations on Interface Array Ports

The following summarizes the features in the LRM that are not supported in the current release.

- Named interface array ports with modports:

  ```
  module Dut (Ifc smprta[6:1].Master);
  ```

- Generic interface array ports with modports:

  ```
  module Dut (interface ifcprta[6:1].Master);
  ```

- Interface array part selects:

  ```
  module top();
    Ifc ifca [4:1] ();
    Dut dut1 ( .prta(ifca[4:3]) ); //Not supported: part select of AOI
  endmodule

  module Dut ( Ifc prta [5:6] );
  ...
  endmodule
  ```

- Interface array ports inside `generate` blocks.

- Whole array assignments to virtual interface array variables:

  ```
  module top();
   Ifc ifca [3:1] ();
   Dut dut1 ( .prta(ifca) );
  endmodule

  module Dut ( Ifc prta [3:1] );
  virtual interface Ifc vidut[3:1] = prta; //Not supported.
  endmodule
  ```

  To work around this, the virtual interface array must be loaded using bit selects:

  ```
  module Dut ( Ifc prta [3:1] );
    virtual interface Ifc vidut[3:1];
    initial begin
     vidut[3] = prta[3];
     vidut[2] = prta[2];
     vidut[1] = prta[1];
    end
  endmodule
  ```

Referencing an Interface

You cannot reference an interface using a hierarchical path. For example, the following is not allowed:

```
module top;
  ...
  memMod mem(some_other_top.the_memory_interface);
  ...
endmodule
```

You can reference objects declared in an interface from any module that declares the interface by using an interface reference. The syntax is as follows:

port_name.interface_signal_name

In the example shown in Figure on page 216, module `memMod` has an interface port with the port name `a`.

```
module memMod(simple_bus a, input clk);

  logic avail;

  always @(posedge clk) a.gnt <= a.req & avail;

endmodule
```

Within the `memMod` module, the signals `gnt` and `req` in the interface are referenced as `a.gnt` and `a.req`, respectively.

Working with Modports

While an interface provides a way to define a group of nets or variables in one place to encapsulate the connectivity between blocks, different modules connected to the interface may require different views of the interface. For example, a particular signal may be an input for one module, while the same signal may be an output for another module.

SystemVerilog provides the `modport` construct, which allows you to customize the interface for the different modules that are connected to it. Using this construct, you can:

- Provide direction information for module ports.
- Specify which signals defined in the interface are accessible to a module.

The following example, modified from examples shown in the SystemVerilog LRM, includes an interface with two modports named `master` and `slave`. This example shows:

- The basic syntax for defining a modport
- The two ways of selecting which modport a module is to use

  The `slave` modport is selected for module `memMod` by specifying the modport when the module is instantiated in module `top`.

The `master` modport is selected for module `cpuMod` by specifying the modport in the module port declaration of module `cpuMod`.

Example 18-3 Interface Example with Modports

```
interface simple_bus;
  logic req, gnt;
  logic [7:0] addr, data;
  logic [1:0] mode;
  logic start, rdy;

  modport master (input gnt, rdy, data,
                  output req, addr, mode, start);

  modport slave (input req, addr, mode, start,
                 output gnt, rdy, data);

endinterface : simple_bus

module top;
  logic clk = 0;

  simple_bus sb_intf();

  memMod mem (.a(sb_intf.slave), .clk(clk));
  cpuMod cpu (.b(sb_intf), .clk(clk));

endmodule

module memMod(simple_bus a,
              input clk);

  logic avail;

  always @(posedge clk) a.gnt <= a.req & avail;

endmodule

module cpuMod(simple_bus.master b,
              input clk);
  ...

endmodule
```

Interface name is `simple_bus`.

Define modport `master`.

Define modport `slave`.

For `memMod`, modport `slave` is selected in `memMod` module instantiation.

For `cpuMod`, modport is selected in `cpuMod` definition.

Specify only the interface name. Modport is selected when `memMod` is instantiated.

In `cpuMod` module port declaration, select modport `master`.

Defining a Modport

Modports are defined within an interface using the keyword `modport`. You can define any number of modports in an interface.

In the current release, you can define a modport that specifies the port direction (`input`, `output`, or `inout`) for ports. For example:

```
interface myintf;
  wire a, b, c, d;

  modport master (input a,
                  input b,
                  output c,
                  output d);

  modport slave (output a,
                 output b,
                 input c,
                 input d);
endinterface
```

Modport ports can refer only to nets or variables. Vector sizes or data types are not included in the modport definitions. Only the port direction is specified. All modport ports must have a corresponding net or variable declaration within the interface in which they are declared.

As with module port declarations, you can specify multiple ports within one direction keyword. For example:

```
modport master (input a, b, output c, d);
```

The syntax allows you to define multiple modports with one keyword, as shown in the following example:

```
modport master (input a, b, output c, d),
        slave  (output a, b, input c,d);
```

Selecting Which Modport to Use

You can specify which modport a module interface port should use in two places.

- The modport name can be specified in the module header as part of the module port declaration.

  You can specify an explicitly-named interface and modport using the following syntax:

  `module` *`module_name`* `(`*`interface_name.modport_name port_name, other_ports`*`);`

  In this case, the interface name selects the interface, and the modport name selects the modport.

You can also specify a generic interface and modport using the following syntax:

```
module module_name (interface.modport_name port_name, other_ports);
```

In the example shown in Figure on page 227, the modport for module cpuMod is specified in the port declaration for the module as follows:

```
module cpuMod(simple_bus.master b, input clk);
```

The instance of module cpuMod in module top does not specify the name of the modport. It just connects the module port to the instance of the interface.

```
module top;
  ...
  cpuMod cpu (.b(sb_intf), .clk(clk));
  ...
```

- The modport name can be specified in the port connection with the module instance using the following syntax:

  ```
  module_name instance_name (interface_instance_name.modport_name,
                             other_ports);
  ```

  The module definition can use either an explicitly-named interface port or a generic interface port. For example, in Figure on page 227, the definition of module memMod uses an explicitly-named interface port.

  ```
  module memMod(simple_bus a, input clk);
  ```

  The modport for module memMod is specified in the instantiation statement for the module as follows:

  ```
  memMod mem (.a(sb_intf.slave), .clk(clk));
  ```

Note: You can specify which modport to use in both places. If you do this, the modport identifier must be the same. A warning is generated if you select a modport in one place and select a different modport in the other place. The warning tells you that the modport selected in the module header as part of the module port declaration is being used instead of the modport selected on the module instance.

Limitations on modports

The following are allowed in the IEEE 1800 standard, but are not supported in the current release of the simulator:

- modports inside for loops
- Expressions within modports

Declaring Tasks and Functions in Interfaces

You can declare tasks and functions in an interface using the same syntax and the same statements used for tasks and functions that are defined in a module. SystemVerilog refers to tasks and functions defined in an interface as *interface methods*.

In the current release, you can define interface methods, and then call the methods from modules connected to the interface using an interface reference of the form:

```
interface_port_name.task_function_name(arguments);
```

For example:

```
interface myintf;
  logic start;
  other_interface_signals
...

task mytask;
  ...
endtask : mytask

endinterface : myintf

module mymod(myintf a);
  ...
  always @(a.start)
     a.mytask;
  ...
endmodule
```

See Section 20.6.1 of the IEEE 1800 standard for a more extensive example.

The following SystemVerilog enhancements related to tasks and functions in interfaces are not supported in the current release:

- Defining a task or function in one module and then exporting the task or function through an interface modport to other modules. The `export` construct in a modport is not supported.
- Defining a task or function in one module and then exporting the task or function to an interface without using a modport. The `extern` construct is not supported.
- Exporting a task name from multiple modules into the same interface. The `extern forkjoin` construct is not supported.

Virtual Interfaces

Virtual interfaces are described in Section 20.8 of the IEEE 1800 standard.

In SystemVerilog, you can declare a *virtual interface*, which is a variable that represents an interface instance. Virtual interfaces are meant to separate code that operates on an interface, from the actual code itself. That way, instead of directly manipulating the set of signals within an interface, you are manipulating a virtual set of signals.

The syntax for a virtual interface is as follows:

```
virtual interface virtualinterface_identifier
```

For example:

```
interface Sbus;
  int a;
endinterface


module test_top();
  Sbus sbif1(); //Interface instances
  Sbus sbif2();
    class c;
     virtual interface Sbus sbus = null;  //Virtual interface of type Sbus
                                          //Initialized to null.
     virtual interface Sbus vbus = sbif2; //Virtual interface of type Sbus
                                          //Initialized to an interface instance
                                          //of the same type.

     function new(virtual interface Sbus channel); //Function argument
       sbus = channel;
       $display("My interface");
     endfunction
    endclass

   initial begin
    c myclass;
    myclass = new(sbif1);
   end

endmodule
```

Note: The IEEE 1800 standard does not mention that, when declaring a virtual interface, you must include any specializations for the interface. For example, if there are any parameter values specified on the interface instance, they must be included in the virtual interface declaration.

You can use a virtual interface in the same context as a variable.

The current release supports the following:

- Virtual interfaces that are declared within a class, module, program block, task, package, or function.
- Passing virtual interfaces by value or reference to a task or function.
- Comparing or assigning virtual interfaces to other virtual interfaces, null, or interface instances.

- Dereferencing scalar variables, wires, packed arrays, and unpacked arrays.
- Arrays of virtual interfaces.
- Virtual interfaces that call tasks and functions within an interface. For example:

```
interface bus;
   logic a, b;
   function void msg();
    $display("Can call this function.");
   endfunction
endinterface

module test_top();
...
class test_stim;
   virtual interface bus vi = null;
   function new(virtual bus vbus);
    vi = vbus;
   endfunction
   task run(logic stim);
    ...
    vi.msg(); //Calls function from interface bus
   endtask
endclass
...
endmodule
```

- Nested interface instances. The current release has limited virtual interface support for nested interface instances. To do this, you must:

  a. Define the inner and outer interfaces.

  b. Instantiate the outer interface and its inner children.

  c. Declare the virtual interface variables for the outer and inner interfaces.

  d. Initialize the outer variable such that it points to the desired outer interface instance.

  e. Using a virtual interface select off the outer virtual variable, access the desired interface instance. Then, write this to the inner virtual interface variable.

  This enables access to the inner variables.

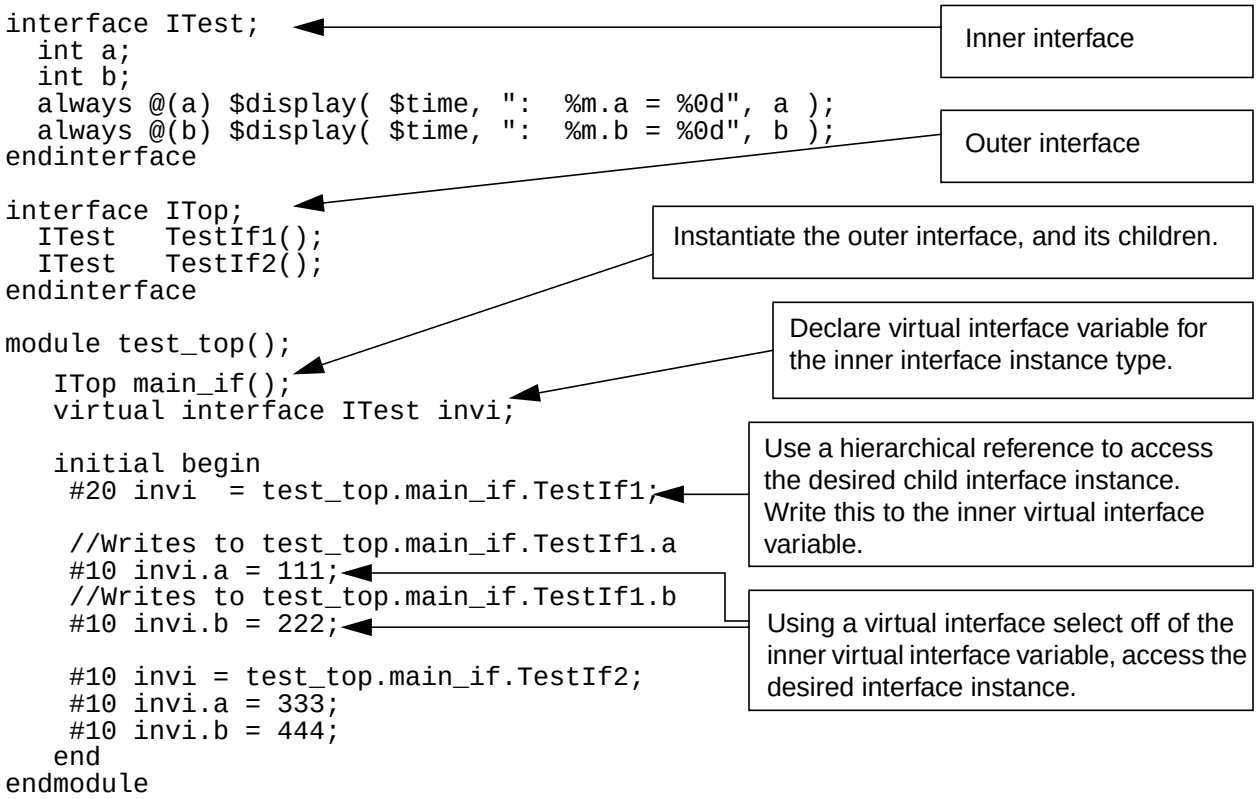
  Note: You can also use a hierarchical reference to access the desired interface instance.

  f. Using a virtual interface select off of the inner virtual interface variable, access the desired interface instance.

  g. If necessary, repeat steps e and f.

  Refer to Example 18-4 on page 233.

Example 18-4 Support for Nested Interface Instances



When simulated, this example should produce the following results:

```
ncsim> run
 30:  test_top.main_if.TestIf1.a = 111
 40:  test_top.main_if.TestIf1.b = 222
 60:  test_top.main_if.TestIf2.a = 333
 70:  test_top.main_if.TestIf2.b = 444
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

However, in the current release:

- You cannot use a virtual interface select to write to a child instance. For example:

  ```
  #10 outvi.TestIf2 = outvi.TestIf1; //Invalid
  ```

- Virtual interfaces cannot directly reference child interface items. For example:

  ```
  #10 outvi.TestIf1.a = 555; //Invalid
  ```

Limitations on Virtual Interfaces

The following summarizes the features in the LRM that are not supported in the current release.

- Parameterized virtual interface declarations are not supported. For example, the following tries to define parameters in a virtual interface declaration:

```
interface Sbus();
  parameter LEFT = 7;
  parameter RIGHT = 0;
  reg[LEFT:RIGHT] a;
endinterface

module top();
  ...
  virtual interface Sbus #(16,0) v1; //Unsupported
endmodule
```

- You can declare virtual interfaces directly within a package.
- Out-of-module references to virtual interfaces is not supported.
- Virtual interfaces cannot reference any of the following datatypes within an interface: queues, dynamic arrays, associative arrays, strings, arrays of classes, or semaphores.

Working with Interfaces and Timing

The IEEE 1800 standard states that, when a module is connected to an interface, signals declared in the interface can be used as terminal descriptors in module path delays and in timing checks described in a `specify` block.

In the current release, you cannot reference a signal in an interface as the terminal node within a `specify` block or in SDF annotation. This includes module path delays, interconnect delays, and timing checks.

19

System Functions

Out-of-Module Reference ($root)

`$root` is described in Section 19.4 of the IEEE 1800 standard.

In Verilog, an out-of-module reference (OOMR) can be ambiguous. For example, if you have an instance `parent` that contains an instance `child` at both the top level and in the current module, the hierarchical path `parent.child.var` can mean the top-level `parent.child.var` or the local `parent.child.var`.

In Verilog-2001 (Section 12.5), this ambiguity is resolved by giving priority to the local scope. An OOMR is resolved by first using a search relative to the enclosing scope. If the object is found using this relative search, no other search is performed, and access to the top-level path is prevented.

SystemVerilog solves this problem by introducing the `$root` qualifier, which forces an OOMR reference to be absolute. `$root` lets you refer explicitly to a top-level instance, or to an instance path starting from the root of the instantiation tree.

Example:

```
$root.parent.child.var   // Item var within instance child within
                          // top-level instance parent
```

Expression Size System Function ($bits)

The `$bits` system function is described in Section 22.3 of the IEEE 1800 standard.

SystemVerilog adds a `$bits` system function that returns the number of bits represented by an expression. The syntax is as follows:

```
$bits(expression);
```

Examples:

```
int bitSize;
reg [31:0] x;
reg [7:0] y [0:31];
reg [3:0] z [7:0] [0:15];
...
...
bitSize = $bits(x);          // bitSize returns 32
bitSize = $bits(y[3]);       // bitSize returns 8
bitSize = $bits(z[3][4]);    // bitSize returns 4
```

When used with fixed-size types, the `$bits` system function can be used as an elaboration-time constant. For example:

```
reg [3:0] a;
reg [$bits(a)-1:0] b;
```

When used with dynamic arrays during simulation, the `$bits()` system function returns the size of the whole dynamic array in bits, which is allocated at that simulation time. For example:

```
logic c[];

initial begin
  $display("$bits(c)=%4d", $bits(c)); //Displays "$bits(c)=0"
  c = new[10];
  $display("$bits(c)=%4d", $bits(c)); //Displays "$bits(c)=10"
  c = new[20];
  $display("$bits(c)=%4d", $bits(c)); //Displays "$bits(c)=20"
end
```

The following lists the current limitations for the `$bits` system function:

- The `$bits` system function cannot be used in constant expressions that involve dynamic arrays. For example:

  ```
  logic c[];
  reg [$bits(c)-1:0] d; //Invalid
  ```

- Declarations that involve `$bits` cannot be evaluated if there is a circular dependency between the declarations. Although the LRM does not explicitly state this, the Cadence

implementation does not support the following instances that can cause circular dependencies:

- Forward references in constant expressions are not supported. For example, the following is not supported:

```
reg [$bits(b)-1:0] a; //Invalid
reg [$bits(a)-1:0] b;
```

- OOMRs within `$bits` constant expressions. For example, the following is not supported:

```
reg [$bits(top.cpu1.regfile2.u1.ctl)-1:0] ctl; //Invalid
```

- You cannot use the `$bits` function on class `struct` variables.

20

Compiler Directives

SystemVerilog compiler directives are described in Chapter 23 of the IEEE 1800 standard.

`define

The `` `define `` compiler directive is described in Section 23.2 of the IEEE 1800 standard.

SystemVerilog enhances the `` `define `` text substitution macro compiler directive.

- The macro text can include `` `" ``. This indicates that the macro expansion should include a quotation mark. For example:

```
`define msg(x) $time,,, `"Value of x is %d`", x
...
...
$display(`msg(count));
```

  In this example, the macro expands to:

```
$display($time,,, "Value of count is %d", count);
```

- The macro text can include `` `\`" ``. This indicates that the macro expansion should include the escape sequence `\"`. For example:

```
`define msg(x) $time,,, `"Value of `\`"x`\`" is %d`", x
...
...
$display(`msg(count));
```

  In this example, the macro expands to:

```
$display($time,,, "Value of \"count\" is %d", count);
```

- The macro text can include ‘‘. This is used to delimit an identifier name without introducing white space. For example:

```
‘define join(a) a‘‘_join
```

  This expands

```
‘join(my)
```

  to

```
my_join
```

`begin_keywords and `end_keywords

The `begin_keywords and `end_keywords compiler directives are described in Section 23.4 of the IEEE 1800 standard.

SystemVerilog has added many new keywords and, unfortunately, these additions can render some Verilog source files illegal. Specifically, Verilog files that have identifiers that match a SystemVerilog keyword will not compile on a SystemVerilog compiler. For example, if you have identifiers such as bit, priority, and do in your Verilog file, it will not compile with a SystemVerilog compiler. SystemVerilog extends the `begin_keywords and `end_keywords compiler directives that were introduced in the 1364-2005 standard by adding the "1800-2005" *version_specifier*, which defines the set of identifiers to use as reserved keywords for a particular block of code.

The `begin_keywords and `end_keywords directives:

- Can surround only modules, primitives, interfaces, programs, or packages. These directives must be specified at the top-level, and cannot be used within any of these constructs.
- Affect only the treatment of identifiers within a block of code. They do not affect semantics, tokens, or other aspects of the language.
- Can be nested. Nested pairs of `begin_keywords and `end_keywords directives are stacked, which means that when the compiler encounters an `end_keyword, it goes back to the version_specifier that was in effect prior to the matching `begin_keywords directive.

These directives have the following syntax:

```
`begin_keywords "version_specifier"
...
`end_keywords
```

where *version_specifier* can be one of the following:

- 1364-1995—Indicates that only the identifiers listed as reserved keywords in the IEEE 1364-1995 standard are considered to be reserved words.
- 1364-2001—Indicates that only the identifiers listed as reserved keywords in the IEEE 1364-2001 standard are considered to be reserved words.
- 1364-2005—Indicates that only the identifiers listed as reserved keywords in the IEEE 1364-2005 standard are considered to be reserved words.

- `1800-2005`—Indicates that only the identifiers listed as reserved keywords in the IEEE 1800 standard are considered to be reserved words.

Examples:

The following does not cause an error, because `priority` is not a keyword in IEEE 1364-2001, so it can be used as an identifier:

```
`begin_keywords "1364-2001"
module test1 (...);
   output priority;   // Valid
   ...
endmodule
`end_keywords
```

The following causes an error because `priority` is a keyword in IEEE 1800, so it cannot be used as an identifier.

```
`begin_keywords "1800-2005"
module test1 (...);
   output priority;   //Invalid
   ...
endmodule
`end_keywords
```

The following example illustrates the use of `` `begin_keywords `` and `` `end_keywords `` with program blocks. The following causes an error because `program` and `endprogram` are not keywords in IEEE 1364-2005:

```
`begin_keywords "1364-2005"
  program p;      //Invalid
   ...
  endprogram: p   //Invalid
`end_keywords
```

To fix the error, use the 1800-2005 *`version_identifier`* instead:

```
`begin_keywords "1800-2005"
  program p;
   ...
  endprogram: p
`end_keywords
```

Limitations

This section summarizes the features of the IEEE 1800 standard that are not supported in the Cadence implementation. Differences between the IEEE 1800 specification and the Cadence implementation are also listed.

- You cannot use the *version_specifier* to expand the set of keywords that are implied by the -sv or -v1995 command options for ncvlog.

  For example, when you invoke ncvlog without any options, the IEEE 1364-2001 keywords are used by default. In this case, you cannot use the 1800-2005 *version_specifier*, but you can use the 1364-1995 *version_specifier*.

  To work around this, you can use the irun utility to compile your Verilog and SystemVerilog files on a single line—without having to use the -sv switch. The irun utility determines the language of a file by its extension, and then maps the file to its appropriate compiler. By default, Verilog files must use the .v extension and SystemVerilog must use the .sv extension. For example:

  ```
  irun vlogfile1.v vlogfile2.v systemv1.sv systemv2.sv
  ```

  In this example, irun will compile the .v files using ncvlog and the .sv files using ncvlog -sv.

  Note: Compiling files using the irun utility is an alternative to using the `begin_keywords and `end_keywords directives when you want to distinguish Verilog files from SystemVerilog files. However, this utility does not automatically support files that contain a mixture of Verilog and SystemVerilog. For those cases, you can use the `begin_keywords and `end_keywords compiler directives within the .sv file, and compile the file using irun.

- The IEEE 1800 specification indicates that the `begin_keywords must be paired with an `end_keywords directive. The specification also indicates that a `begin_keywords is in effect until it reaches its matching `end_keywords directive.

  The Cadence implementation does not require a matching `end_keywords directive. In cases where a matching `end_keywords directive does not exist, the parser issues a warning message, but continues to parse the code using the set of keywords denoted by the last `begin_keyword. When the parser reaches the end of the source file, it will begin parsing the next source file, and the keywords denoted by the last `begin_keywords will remain in effect. The warning message issued by the parser can be upgraded to an error message using the -ncerror command-line option.

- The IEEE 1800 specification does not indicate how the `begin_keywords directive relates to the `resetall directive, which resets all compiler directives to their default values.

In the Cadence implementation, the `` `resetall `` directive does not affect the current set of keywords, as specified by the `` `begin_keywords `` directive.

Reserved Keywords for IEEE 1800

This section lists the set of reserved keywords for the IEEE 1800 standard.

Table 20-1 IEEE 1800 Reserved Keywords

alias	endmodule	matches	small
always	endpackage	medium	solve
always_comb	endprimitive	modport	specify
always_ff	endprogram	module	specparam
always_latch	endproperty	nand	static
and	endspecify	negedge	string
assert	endsequence	new	strong0
assign	endtable	nmos	strong1
assume	endtask	nor	struct
automatic	enum	noshowcancelled	super
before	event	not	supply0
begin	expect	notif0	supply1
bind	export	notif1	table
bins	extends	null	tagged
binsof	extern	or	task
bit	final	output	this
break	first_match	package	throughout
buf	for	packed	time
bufif0	force	parameter	timeprecision
bufif1	foreach	pmos	timeunit
byte	forever	posedge	tran
case	fork	primitive	tranif0
casex	forkjoin	priority	tranif1

Table 20-1 IEEE 1800 Reserved Keywords, *continued*

casez	function	program	tri
cell	generate	property	tri0
chandle	genvar	protected	tri1
class	highz0	pull0	triand
clocking	highz1	pull1	trior
cmos	if	pulldown	trireg
config	iff	pullup	type
const	ifnone	pulsestyle_onevent	typedef
constraint	ignore_bins	pulsestyle_ondetect	union
context	illegal_bins	pure	unique
continue	import	rand	unsigned
cover	incdir	randc	use
covergroup	include	randcase	uwire
coverpoint	initial	randsequence	var
cross	inout	rcmos	vectored
deassign	input	real	virtual
default	inside	realtime	void
defparam	instance	ref	wait
design	int	reg	wait_order
disable	integer	release	wand
dist	interface	repeat	weak0
do	intersect	return	weak1
edge	join	rnmos	while
else	join_any	rpmos	wildcard
end	join_none	rtran	wire
endcase	large	rtranif0	with

Table 20-1 IEEE 1800 Reserved Keywords, *continued*

endclass	liblist	rtranif1	within
endclocking	library	scalared	wor
endconfig	local	sequence	xnor
endfunction	localparam	shortint	xor
endgenerate	logic	shortreal	
endgroup	longint	showcancelled	
endinterface	macromodule	signed	

`remove_keyword and `restore_keyword

The `begin_keywords and `end_keywords compiler directives described in the IEEE 1800 standard and in "`begin_keywords and `end_keywords" on page 240 can be used to specify the complete set of reserved keywords in effect when a design unit is parsed. For example, the following compiler directive specifies that only the identifiers listed as reserved keywords in the IEEE 1364-2001 standard are considered to be reserved words. This compiler directive will remove all of the keywords introduced in SystemVerilog.

```
`begin_keywords "1364-2001"
```

However, in transitioning to SystemVerilog, you may be having difficulty with a limited number of keywords in the new language that conflict with identifiers commonly used in your code. To provide a way to remove and restore specific keywords, Cadence has implemented a compiler command-line option, `ncvlog -rmkeyword`, and the `` `remove_keyword `` and `` `restore_keyword `` compiler directives.

The `-rmkeyword` command-line option and the `` `remove_keyword `` directive can be used to remove a keyword from any set of keywords. That is, their use is not restricted to removing a keyword from the IEEE 1800 set of keywords.

ncvlog -rmkeyword

Using the `-rmkeyword` command-line option is convenient if you want to remove a particular keyword so that it will always be treated as an identifier in all parts of the design. The syntax is as follows:

```
-rmkeyword keyword
```

For example:

```
-rmkeyword logic
```

Only one keyword can be specified with `-rmkeyword`. Use the option multiple times to remove multiple keywords. For example:

```
% ncvlog -rmkeyword logic -rmkeyword do test.v
```

`remove_keyword and `restore_keyword Compiler Directives

Use the `` `remove_keyword `` and `` `restore_keyword `` directives if you need to mix SystemVerilog code with non-SystemVerilog code.

The `` `remove_keyword `` directive removes a specified keyword from the set of keywords. This disables the functionality provided by the keyword, but allows the keyword to be used as an identifier.

Example:

```
`remove_keyword logic
```

You can specify only one keyword. To remove more than one keyword, you must specify each keyword with a different compiler directive. For example:

```
`remove_keyword logic
`remove_keyword do
```

The following syntax generates an error:

```
`remove_keyword logic do    // Illegal syntax
```

The `` `restore_keyword `` directive restores a keyword that was previously removed by `` `remove_keyword ``.

These compiler directives, like the `` `begin_keywords `` and `` `end_keywords `` directives, can only be specified outside of a design element (module, primitive, configuration, interface, program, or package). The `` `remove_keyword `` directive affects all source code that follows the directive, even across code file boundaries.

Interaction with Other Compiler Directives

The `` `remove_keyword `` directive (and the `-rmkeyword` command-line option) takes precedence over `` `begin_keywords `` and `` `end_keywords `` directives. If a keyword is removed, it is removed from all variations of the language specification and cannot be reactivated by starting a new set of keywords using `` `begin_keywords `` or `` `end_keywords ``. The only way to restore a keyword is by using `` `restore_keyword ``.

The `` `remove_keyword `` directive also takes precedence over the `` `resetall `` directive. The set of keywords is not affected by `` `resetall ``.

Limitations

This section lists some limitations on the use of the `-rmkeyword` command-line option and `` `remove_keyword `` and `` `restore_keyword `` directives.

- Removed keywords will still be highlighted as keywords in the SimVision GUI.
- The compiler directives are not specified in any standard, and they might not be supported by other tools. You can hide the directives from other tools by conditionally compiling the code using `` `ifdef INCA `` or `` `ifdef CDS_TOOL_DEFINE `` in the source code.

21

Direct Programming Interface

The Direct Programming Interface (DPI) is described in Section 26 of the IEEE 1800 standard.

SystemVerilog introduces a new foreign language interface, called the Direct Programming Interface (DPI). DPI provides a simple, straightforward, and efficient way to connect SystemVerilog and C language code, and to build a design or testbench with components written in SystemVerilog and C. The current release extends this such that DPI can be used to connect SystemVerilog and SystemC language code.

The following directory contains examples of a SystemVerilog testbench that instantiates a SystemC reference model and uses DPI calls:

`install_dir/tools/systemc/examples/...`

For more information, refer to the *Examples Reference Guide.*

For more DPI examples that you can download and run, refer to the *SystemVerilog DPI Engineering Notebook*.

Importing Functions and Tasks using DPI

With DPI, SystemVerilog code can directly call a C or SystemC function. The functions implemented in C or SystemC are called *imported functions*. The imported function name is imported into the SystemVerilog language using an *import declaration*.

An import declaration specifies the task or function name, the return data type (for functions), and the types and directions of the formal arguments. The number of arguments must match the number of arguments in the C or SystemC function, and the data types of the arguments must be compatible with the C or SystemC function data types.

An import declaration defines a task or function in the scope in which the declaration occurs. You cannot, therefore, import the same task or function name into the same module multiple times. However, the same C or SystemC function can be imported into multiple modules.

Imported tasks and functions are called in the same way that native SystemVerilog tasks and functions are called. Calls of imported tasks and functions are indistinguishable from calls of SystemVerilog tasks or functions.

In the following example, a C function called `hello()` is declared in a SystemVerilog module with an import declaration and then called.

C Function

```
#include<stdio.h>
void hello() {
printf("Greetings from the C function!\n");
}
```

SystemVerilog

```
module top;

import "DPI-C" task hello();
...
initial
if (sig == 1) hello();
...
endmodule
```

pure and context Properties

By default, a C or SystemC function can be imported as either a SystemVerilog function or task. Both can have `input`, `output`, and `inout` arguments. Functions can have a return value, or be declared as `void`. The imported task or function cannot access SystemVerilog data objects other than its actual arguments. Only the actual arguments can be read or written by its call.

Imported C and SystemC functions can be declared as `pure` or `context`. The following is a brief description of these properties. See the IEEE 1800 standard, Sections 26.4.1.3, 26.4.2, and 26.4.3 for a detailed description.

- `pure` functions

  An imported function can be specified as `pure` if the result of the function depends solely on the values of its input arguments. Only non-void functions with no `output` or `inout` arguments can be specified as `pure`. These functions can have no side effects. For example, they cannot read or write anything, access global or static variables, or call other functions.

  Example:

  ```
  import “DPI-C” pure function int calc_parity (input int a);
  ```

  Specifying a function as `pure` can often result in improved simulation performance because more optimizations can be performed.

Because a task does not have a return value or a result, a function imported as a task cannot be specified as `pure`.

- `context` functions

  An imported task or function must be specified as `context` if it accesses SystemVerilog data objects other than its actual arguments (for example, through PLI calls), calls exported tasks or functions, or accesses SystemC portions of the design.

  Example:

  ```
  import "DPI-SC" context function int myclassfunc_func1 ( );
  ```

  Calls of `context` tasks and functions are specially instrumented, and can impair compiler optimizations. Simulation performance can be affected if the `context` property is specified when it is not necessary.

Note: Violating the rules specified above and incorrectly declaring an imported task or function as `pure` or `context` can result in unpredictable simulation behavior.

Importing C Functions and Tasks

The syntax in the LRM for declaring a C function imported as a SystemVerilog function is as follows:

```
import {“DPI” | “DPI-C” } [context | pure] [c_identifier =] function
function_data_type function_identifier ([tf_port_list]);
```

Note: The current release supports both `"DPI"` and `"DPI-C"` strings for specifying DPI tasks/functions. However, "DPI-C" is the recommended usage, as it supports the IEEE 1800 standard.

In the current release, import declarations that use `DPI-C` are allowed only within modules, packages, interfaces, program blocks, and compilation unit scopes.

For example:

```
import “DPI-C” function int calc_parity (input int a);
```

In this example:

- `int` is the data type of the function return value.
- `calc_parity` is the function identifier used in the SystemVerilog code.
- The function has one input, which is of type `int`.

The syntax for declaring a C function imported as a SystemVerilog task is as follows:

```
import {“DPI”|“DPI-C”} [context] [c_identifier =] task task_identifier
([tf_port_list]);
```

For example:

```
import “DPI-C” task calc_task (input int in1, output int out1);
```

Specifying a Local Name for the C Function

By default, the task or function identifier in the import declaration is assumed to be the same as the function identifier in the C code. You can use the *c_identifier* to specify a name to represent the C function name. The specified name must conform to C identifier syntax.

In the following example, the `calc_parity_func` C function is given the name `calc_parity` in SystemVerilog.

```
import “DPI-C” calc_parity_func = function int calc_parity (input int a);
```

Return Types for Imported C Tasks

In the current release, the following data types are supported for imported function return types:

- `void`, `byte`, `shortint`, `int`, `longint`, `real`, `string`, `chandle`
- Scalar values of types `bit` and `logic`

Prior to IUS 8.1, imported tasks did not have return values and C and SystemC functions that corresponded to an imported task had to return a `void` type. This requirement has changed.

Due to the addition of support for the `disable` construct within DPI-based designs, C and SystemC functions that correspond to an imported or exported task are required to return an `int` value. For example, the following defines a C task called `imp_task`, which will be imported into SystemVerilog:

```
int imp_task_c (int x, int y){ /* Return type is int */
..int dis_ret;
  dis_ret = exp_task_c(x,y); /*Return type is int */
  return (dis_ret);
...
}
```

For backward compatibility, you can use `-dpi_void_task` option with `ncelab` or `irun` on existing DPI designs. Designs will not be affected by this new requirement and will behave as they did prior to IUS 8.1. However, you must adhere to this new style of DPI function declaration in order to use the `disable` functionality with DPI-based designs.

See "Disabling DPI Tasks and Functions" on page 275 for more information on the `disable` function.

Formal Arguments for Imported C Functions and Tasks

An imported task or function can have zero or more formal arguments.

By default, each formal argument is assumed to be an input to the C function. You can override this default by explicitly declaring each formal argument as an `input`, `output`, or `inout` argument. For example:

```
import "DPI-C" context task calc_task(input int in1, output int out1);
```

The SystemVerilog data types specified in the import declaration must be compatible with the actual C function data types. No checking is performed to ensure that the data types are compatible, and improper declarations can result in unpredictable behavior and erroneous values.

Section 26.4.6 of the LRM lists the data types that are allowed for formal arguments of imported and exported tasks or functions.

In the current release, the following data types are supported as formal arguments for DPI-C imported functions/tasks.

- `byte`, `shortint`, `int`, `longint`, `real`, `string`, and `chandle`
- Scalar values of types `bit` and `logic`
- One-dimensional packed arrays of types `bit` and `logic`
- One-dimensional unpacked arrays of type `byte` and unsigned `byte`

  Note: This type is not supported as a formal argument of imported SystemC functions and tasks.
- Multi-dimensional packed arrays of types `bit` and `logic`
- One-dimensional open arrays of datatypes specified below are supported:
  - int, shortint, longint
  - string, byte
  - string , byte
  - bit vector, logic vector
  - bit, logic

A formal argument is considered an open array when one or more of its dimensions are unspecified. The actual argument can be a fixed or dynamic array.

- Packed structs of types `bit` and `logic`
- Unpacked structs with members of type `int`, `unsigned bit`, or `logic`. Special rules apply, see "Using Unpacked Structs as Formal Arguments in DPI-C Import Functions" and "Using Unpacked Structs as Formal Arguments in DPI-SC Import Functions" on page 254.

Using Unpacked Structs as Formal Arguments in DPI-C Import Functions

Unpacked structs with members of type `int`, `unsigned bit`, or `logic` can be used as formal arguments to DPI-C imported functions and tasks. They cannot be used as formal arguments to exported functions and tasks. If a SystemVerilog unpacked struct contains a member that is not of type `int`, `unsigned bit`, and `logic` and is used as a formal argument to a DPI-C import function or task, you will get an error message.

For DPI-C imported functions and tasks:

- The formal argument in C code must be a pointer to a C struct.
- The layout of the C and SystemVerilog structs must match. In other words, the structs must have the same number and ordering of fields.

For examples, refer to "Unpacked Structs as Formal Arguments to DPI-C Import Functions" on page 283 or to the *SystemVerilog DPI Engineering Notebook*.

Using Unpacked Structs as Formal Arguments in DPI-SC Import Functions

Unpacked structs with members of type `int`, `unsigned bit`, or `logic` can be used as formal arguments to DPI-SC imported functions and tasks. They cannot be used as formal arguments to exported functions and tasks. If a SystemVerilog unpacked struct contains a member that is not of type `int`, `unsigned bit`, and `logic` and is used as a formal argument to a DPI-SC import function or task, you will get an error message.

For DPI-SC import functions and tasks:

- The formal argument in SystemC code must be a C struct that can be passed by value, pointer, or reference. This means the C struct can contain only C data type fields; SystemC data types are not supported. The SystemVerilog data types used within the SystemVerilog unpacked struct will be mapped to C data types, using the DPI-C mapping rules. For example, `bit` will map to `svBit`, and `logic` will map to `svLogic`. However, the current release does not support mapping `bit` to `bool`, or `logic` to `sc_logic`.

To facilitate the conversion of C types to SystemC types, a library of conversion functions has been added called `sc_dpi_convert` class. For more information on this library, refer to “SystemC and HDL Design Hierarchies” in the *NC-SC Simulator User Guide*.

- If a SystemVerilog unpacked struct is used as an argument to a DPI-SC imported task or function, it cannot contain members that use SystemC data types.
- The layout of the C and SystemVerilog structs must match. In other words, the structs must have the same number and ordering of fields.
- In DPI-SC imports, the names of the structs must match.

  Unlike C, where the linker matches functions by name—regardless of argument types, the C++ linker matches the entire signature of a function, including its argument types.

For an example, refer to “Unpacked Structs as Formal Arguments to DPI-SC Import Functions” on page 284.

Importing SystemC Functions and Tasks

SystemC functions can be imported as a SystemVerilog function or task. To import a task or function, you must:

- Define the following before the first `#include` in your SystemC file:

  ```
  #define NCSC_INCLUDE_TASK_CALLS
  ```

- Declare it using an `import` declaration in SystemVerilog. See “Import Declaration Syntax” on page 256.
- Register the task or function in SystemC using the appropriate `NCSC_REGISTER_DPI*` registration macros. See the “SystemC and HDL Design Hierarchies” chapter of the *NC-SC Simulator User Guide* for more information.

  See “DPI Examples” on page 278 for examples.

You can import the following types of SystemC function:

- Global C++ functions
- Static and non-static member methods of arbitrary classes
- Static and non-static member methods of classes that inherit from `sc_object`

  If a class member method inherits from `sc_object`, you must first specify its scope in SystemC before invoking it from SystemVerilog. See “Setting the SystemC Scope” on page 256.

Import Declaration Syntax

The syntax for declaring a SystemC function imported as a SystemVerilog function is as follows:

```
import “DPI-SC” [context | pure] [c_identifier =] function function_data_type
function_identifier ([tf_port_list]);
```

The “DPI-SC” qualifier is for tasks and functions that are interoperable with SystemC. When you use the “DPI-SC” qualifier in your import declaration:

- SystemC functions and tasks are invoked transparently from SystemVerilog.
- Mapping between the SystemC and SystemVerilog data types is done transparently by the simulator.

For example:

```
import “DPI-SC” context function void scmod_run (input sc_int[31:0] i1);
```

In this example:

- void is the data type of the function return value.
- scmod_run is the function identifier used in the SystemVerilog code.
- The function has one input, which is of type sc_int.

The syntax for declaring a SystemC function imported as a SystemVerilog task is as follows:

```
import “DPI-SC” [context | pure] [c_identifier =] task task_id ([tf_port_list]);
```

For example:

```
import “DPI-SC” task sccalc_task (input int in1, output int out1);
```

In the current release, import declarations that use “DPI-SC” are allowed within modules, packages, interfaces, program blocks, and compilation unit scopes.

Setting the SystemC Scope

sc_object is the base class for all objects in the SystemC design hierarchy. If you import a non-static member method of a class that inherits from sc_object, you must set the scope of its corresponding sc_object instance in SystemC before you can invoke it from SystemVerilog. In NC-Verilog, you can use the scSetScopeByName() function to pass the scope information to SystemC. For example:

```
scSetScopeByName(“sctop.scm”);
```

This specifies any subsequent calls to SystemC sc_object class member methods will use the scope sctop.scm.

Once a scope is set, it persists until another `scSetScopeByName()`is used.

Note: You do not need to set the scope for any other type of imported SystemC function or task (such as global functions, static class member methods, or member methods of an arbitrary class that does not inherit from `sc_object`).

See "DPI Examples" on page 278 for an extensive example.

Return Types for Imported SystemC Functions

For SystemC import declarations, the current release supports the same data types as listed in "Return Types for Imported C Tasks" on page 252.

You can also use SystemC data types as return types for imported functions; however, they require a `typedef`. Refer to "Using typedef with SystemC Data Types" on page 265 for guidelines on using a `typedef` with SystemC data types.

Prior to IUS 8.1, imported tasks did not have return values and C and SystemC functions that corresponded to an imported task had to return a `void` type. This requirement has changed.

Due to the addition of support for the `disable` construct within DPI-based designs, C and SystemC functions that correspond to an imported or exported task are required to return an `int` value. For example, the following defines a C task called `imp_task`, which will be imported into SystemVerilog:

```
int imp_task_c (int x, int y){ /* Return type is int */
  int dis_ret;
  dis_ret = exp_task_c(x,y); /*Return type is int */
  return (dis_ret);
...
}
```

For backward compatibility, you can use `-dpi_void_task` option with `ncelab` or `irun` on existing DPI designs. Designs will not be affected by this new requirement and will behave as they did prior to IUS 8.1. However, you must adhere to this new style of DPI function declaration in order to use the `disable` functionality with DPI-based designs.

Formal Arguments for Imported SystemC Functions and Tasks

For SystemC import declarations, the current release supports the same data types as listed in "Formal Arguments for Imported C Functions and Tasks" on page 253.

You can also use the following SystemC data types as formal arguments for imported functions. The simulator will implicitly map them to a SystemVerilog data type.

Note: Special rules apply for unpacked structs. See "Using Unpacked Structs as Formal Arguments in DPI-SC Import Functions" on page 254.

Table 21-1 Default Data Type Mapping for Formal Arguments

SystemC Data Type	SystemVerilog Data Type
`sc_logic` `sc_lv`	`logic`

`bool`
`sc_bv`
`sc_int`
`sc_uint`
`sc_bigint`
`sc_biguint`
	`bit`

Note: When used as formal arguments, the `sc_lv`, `sc_bv`, `sc_int`, `sc_uint`, `sc_bigint`, and `sc_biguint`. must be declared as vectors.

For example:

```
import "DPI-SC" task task1 (input sc_int[31:0] i1); //Valid
import "DPI-SC" task task1 (input sc_int i1); //Invalid
```

SystemC data types can be used within only `"DPI-SC"` `import` and `export` declarations. Using them in any other context within a SystemVerilog design will result in an error. You can, however, use a `typedef` to override the default data mapping listed in Table 21-1 on page 258. Refer to "Using typedef with SystemC Data Types" on page 265 for guidelines on using a `typedef` with SystemC data types.

Exporting SystemVerilog Functions and Tasks using DPI

A C function that corresponds to a `context import` subroutine can directly call a SystemVerilog task or function. Such SystemVerilog tasks/functions are called *exported* functions/tasks. Exported task/functions can also be invoked from SystemC processes.

For example, SystemVerilog function called `hello_sv()` is declared in SystemVerilog and exported to C using an `export` declaration:

SystemVerilog

```
module top;
export "DPI-C" function hello_sv;
import "DPI-C" context task test();
...
  function void hello_sv(input int a);
  ...
   $display("Hello, world %d\n", a);
  ...
  endfunction
endmodule
```

C Function

```
#include<stdio.h>
extern void hello_sv(int _a1);

 void test(){
 ...
 hello_sv(100);
 ...
 }
```

Exporting Functions and Tasks to C

C functions can call SystemVerilog functions and tasks that are declared using an export declaration.

Note the following for exported functions and tasks:

- In the current release, export declarations are allowed only within modules, packages, interfaces, program blocks, and compilation unit scopes; export declarations must appear in the scope in which the task or function is defined.
- When using DPI to export functions and tasks, you need to include a header file in your C code. You can create your own header file, or generate a header file using the `-dpiheader` switch with `ncelab`. Refer to "Using DPI with the Incisive Simulator" on page 269 for more information.
- Exported functions and tasks are subject to the same argument type and result restrictions as imported functions. See "Formal Arguments for Functions or Tasks Exported to C" on page 262 for more information.

Export Declaration Syntax

The syntax in the LRM for declaring a SystemVerilog function or task that will be exported to C is as follows:

```
export {"DPI" | "DPI-C"} [c_identifier = ] function function_identifier;
export {"DPI" | "DPI-C"} [c_identifier = ] task task_identifier;
```

Note: The current release supports both `"DPI"` and `"DPI-C"` strings for specifying DPI tasks/functions. However, "DPI-C" is the recommended usage, as it supports the IEEE 1800 standard.

By default, the function or task identifier in the export declaration is the same as the function or task identifier in the C code. However, you can use the *c_identifier* to specify a name to represent the C function or task name. The C identifier must conform to C identifier syntax.

For example, the following specifies that the SystemVerilog function `myfunction` will be exported, and that C will use the identifier `my_cfunction`:

```
export “DPI-C” my_cfunction = function myfunction;
```

An export declaration and its corresponding SystemVerilog function definition can appear in any order. However, the export declaration must occur within the scope for which the function or task is defined. For example:

```
module top;
 export "DPI-C" c_name= function sv_name1;
   function int sv_name1( );
      begin
      ...
      end
   endfunction
endmodule
```

Can also look like:

```
module top;
   function int sv_name1( );
      begin
      ...
      end
   endfunction
 export "DPI-C" c_name= function sv_name1;
endmodule
```

A *c_identifier* cannot be used within the same scope for more than one exported function or task. Also, there can be only one export declaration per task or function within the same scope. In the following example, although there are multiple export declarations that correspond to the same C identifier `c_name`, the example is still valid because the declarations exist in different scopes and have the same signatures:

```
module top;

   export "DPI" c_name= function sv_name1;

   function int sv_name1( inout int a1, input real a2, output shortint a3,
                          inout byte a4);
     begin
       $display(" \n Value inside the sv_name1 \
       ...
     end
   endfunction
   ...
endmodule

module mid1;
```

```
   export "DPI" c_name= function sv_name2;

   function int sv_name2( inout int a1, input real a2, output shortint a3,
                          inout byte a4);
     begin
       $display(" \n Value inside the sv_name2 \
       ...
     end
   endfunction
   ...
endmodule
```

The previous example also illustrates how exported functions and tasks can have `input`, `output`, and `inout` arguments.

An exported function or task can be called only from within a `context` imported function or task.

Return Types for Functions Exported to C

In the current release, the following data types are supported for exported function and task return types:

- `void`, `byte`, `shortint`, `int`, `longint`, `real`, `chandle`, and `string`
- Scalar values of type `bit` and `logic`

Due to the addition of support for the `disable` construct within DPI-based designs, C and SystemC functions that correspond to an imported or exported task are required to return an `int` value. For example, the following defines a C task called `imp_task`, which will be imported into SystemVerilog:

```
int imp_task_c (int x, int y){ /* Return type is int */
  int dis_ret;
  dis_ret = exp_task_c(x,y); /*Return type is int */
  return (dis_ret);
...
}
```

For backward compatibility, you can use `-dpi_void_task` option with `ncelab` or `irun` on existing DPI-C designs. Designs will not be affected by this new requirement and will behave as they did prior to IUS 8.1. However, you must adhere to this new style of DPI-C function declaration in order to use the `disable` functionality with DPI-based designs.

See “Disabling DPI Tasks and Functions” on page 275 for more information on the `disable` function.

Formal Arguments for Functions or Tasks Exported to C

The SystemVerilog data types specified in the export declaration must be compatible with the actual C function data types. No checking is performed to ensure that the data types are compatible, and improper declarations can result in unpredictable behavior and erroneous values.

Section 26.4.6 of the LRM lists the data types that are allowed for formal arguments of imported and exported tasks or functions.

In the current release, the following data types are supported as formal arguments for exported functions and tasks:

- `byte`, `shortint`, `longint`, `int`, `real`, `string`, and `chandle`
- Scalar values of types `bit` and `logic`
- One-dimensional packed arrays of types `bit` and `logic`
- One-dimensional unpacked arrays of type `byte` and unsigned `byte`

  Note: This type is not supported as a formal argument of exported SystemC functions and tasks.
- Multi-dimensional packed arrays of types `bit` and `logic`
- Packed structs of types `bit` and `logic`

Exporting SystemVerilog Functions and Tasks to SystemC

SystemC processes can call SystemVerilog functions and tasks that are declared using an `export` declaration. Exported functions/tasks can also be called from `context` imported functions/tasks that are qualified with `"DPI-SC"`. Every exported function must:

- Define the following before the first `#include` in your SystemC file:

  ```
  #define NCSC_INCLUDE_TASK_CALLS
  ```
- Have an `export` declaration. See “Export Declaration Syntax” on page 263.
- Be declared in SystemC as external (using the `extern “C”` keyword), which will indicate that the task or function’s definition resides in another source file. See Example on page 281 for an example.
- Specify its SystemVerilog scope using the `svSetScope()` function in SystemC. `svSetScope()` has the following syntax:

  ```
  svSetScope(svGetScopeFromName(“scope”);
  ```

However, if you are calling an exported function from within an imported function, `svSetScope()` might not be required (see Example on page 281).

- Exported functions can be invoked from the following places in a SystemC design.
  - From a SystemC `end_of_elaboration` callback that is invoked during simulation.

    The callback function can invoke exported functions.
  - From a SystemC `start_of_simulation` callback.

    The callback function can invoke exported functions.
  - From a method process or a thread process. During a process execution, a SystemC design can call exported functions.
  - From the `sc_main()` function.
  - `$call_systemc_function()`and `$call_systemc_process_nb()` can invoke a SystemC function or schedule a SystemC process that can further invoke exported functions.

  You can also call the following accessor functions from each of the places listed above:

`svSetScope`	`svGetNameFromScope`	`svGetScopeFromName`
`svPutUserData`	`svGetUserData`	`svGetScope`

Export Declaration Syntax

The syntax for declaring a SystemVerilog function or task that will be exported to SystemC is as follows:

```
export "DPI-SC" [c_identifier = ] function function_identifier;
export "DPI-SC" [c_identifier = ] task task_identifier;
```

The `"DPI-SC"` qualifier is for tasks and functions that are interoperable with SystemC. When you use the `"DPI-SC"` qualifier in your `export` declaration:

- SystemVerilog functions and tasks are invoked transparently from SystemC.
- Mapping between the SystemC and SystemVerilog data types is done transparently by the simulator.

By default, the function or task identifier in the export declaration is the same as the function identifier in the SystemC code. However, you can use the *`c_identifier`* to specify a name to represent the SystemC function. The C identifier must conform to SystemC identifier

syntax. For example, the following specifies that the SystemVerilog function `myfunction` will be exported and that SystemC will use the identifier `my_scexp`.

```
export "DPI-SC" my_scexp = function sc_exp;
```

Return Types for Functions Exported to SystemC

For SystemC export declarations, the current release supports the same data types as listed in "Return Types for Functions Exported to C" on page 261.

You can also use SystemC data types as return types for exported functions; however, they require a `typedef`. Refer to "Using typedef with SystemC Data Types" on page 265 for guidelines on using a `typedef` with SystemC data types.

Formal Arguments for Functions or Tasks Exported to SystemC

For SystemC export declarations, the current release supports the same data types as listed in "Formal Arguments for Functions or Tasks Exported to C" on page 262.

You can also use SystemC data types as formal arguments for exported functions. The simulator will map them to a SystemVerilog data type. See Table 21-1 on page 258 for a list of supported SystemC data types.

SystemC data types can be used within only `"DPI-SC"` `import` and `export` declarations. Using them in any other context within a SystemVerilog design will result in an error. You can, however, use a `typedef` to override the default data mapping listed in Table 21-1 on page 258. Refer to "Using typedef with SystemC Data Types" on page 265 for guidelines on using a `typedef` with SystemC data types.

Limitations for Functions and Tasks Exported to SystemC

In the current release:

- Export declarations are allowed only within modules, packages, interfaces, and compilation unit scopes.
- Debugging is not supported for exported functions invoked from a SystemC design.
- Exported tasks can be invoked from only a SystemC thread process. You cannot invoke exported tasks from any other context, such as `sc_main()` or a method process (such as, `SC_METHOD`).

Using typedef with SystemC Data Types

SystemC data types can be used within only `"DPI-SC" import` and `export` declarations. Using them in any other context, without a `typedef`, will result in an error.

If you use a SystemVerilog `typedef` to create a new definition for a SystemC data type, the type definition must satisfy the data type mapping listed in tables 21-2 and 21-3 (these tables apply to both `import` and `export` declarations). Otherwise, you will get a compilation error. If the `typedef` is used outside `"DPI-SC" import` or `export` declarations, the regular SystemVerilog type definition semantics will apply.

Table 21-2 Data Type Mapping for Function Return Types

SystemC Data Type	Valid SystemVerilog Data Types
`sc_logic`	`logic`, `bit`
`bool`	`bit`, `logic`

`sc_int`
`sc_uint`
`sc_bigint`
`sc_biguint`
	`int`, `shortint`, `longint`

Table 21-3 Data Type Mapping for Formal Arguments

SystemC Data Type	SystemVerilog Data Type
`sc_logic` `sc_lv`	`logic`, `bit`
`bool` `sc_bv`	`bit`, `logic`

`sc_int`
`sc_uint`
`sc_bigint`
`sc_biguint`
	`bit`, `int`, `shortint`, `longint`

Note: When using a *typedef* to map `sc_lv`, `sc_bv`, `sc_int`, `sc_unit`, `sc_bigint`, `sc_biguint` to type `bit` or `logic`, they must be declared as vectors. For example:

```
typedef bit sc_int;
import "DPI-SC" task1 (input sc_int[31:0] i1); //Valid
import "DPI-SC" task2 (input sc_int i1); //Invalid
```

```
...
typedef int sc_int;
import “DPI-SC” task3 (input sc_int a1); //Valid
```

Example 1:

```
typedef int sc_int; //Maps sc_int to type int, which is a valid data type
import “DPI-SC” void func1 (input sc_int a);
```

The simulator will map `sc_int` to type `int` before it invokes the SystemC function `func1`.

Example 2:

```
typedef string sc_int; //sc_int cannot be mapped to strings
import “DPI-SC” void func2 (input sc_int a);
```

This example results in a compilation error because the `typedef` maps `sc_int` to an invalid type.

Example 3:

```
typedef bit sc_int;
import “DPI-SC” function void openArrayFunc(input sc_int[7:0] a[]);
```

This example results in an error message because open arrays are not supported for `typedefs` that correspond to SystemC data types.

Tasks That Consume Time

In the current release, SystemVerilog can invoke a DPI import call chain that eventually consumes time in SystemC or in SystemVerilog. Likewise, SystemC can invoke a DPI export call chain that eventually consumes time in SystemVerilog or in SystemC. These types of export call chains can be invoked from only a SystemC thread process. Nested call chains of arbitrary depth are also supported in this release.

To consume time in SystemC, a task that is imported using `“DPI”`, `“DPI-C”`, or `“DPI-SC”` must use the `wait(...)` construct. For more information on the different forms of the `wait(...)` construct, refer to the “SystemC and HDL Design Hierarchies” chapter of the *NC-SC Simulator User Guide* for more information.

Note: Import tasks defined in SystemVerilog program blocks cannot call `wait(...)` in SystemC. This will result in an error message.

To consume time in SystemVerilog, a task that is exported using `“DPI”`, `“DPI-C”`, or `“DPI-SC”` can use the following SystemVerilog construct:

- `fork...join`, `fork...join_none`, and `fork...join_any`
- `wait fork` and `wait`

- Semaphores

  Semaphores provide a `get()` method, which is used to block the execution of a process until a key is available.

- Mailboxes

  Mailboxes provide a `put()` method, which suspends a process until there is enough room in the queue.

- Cycle delays

  Cycle delays can be introduced using default clocking blocks. The ## operator delays the execution by a specified number of clocking events or clocking cycles.

- Blocking and non-blocking assignments that use delay-based timing controls

  The # symbol specifies a delay-based timing control. It specifies how long to wait before executing a statement. For example:

  ```
  #5 a = b;
  a = #5 9 b;
  a <= #4 b;
  ```

- Statements that include event-based timing controls

  The @ symbol specifies an event-based timing control. It specifies that a statement can be executed on a change in signal value or when the specified event has triggered. For example:

  ```
  @ (event_identifier)
  @(*)
  @ a(iff en == 1)
  ```

Note: Disabling a SystemVerilog process that is in the middle of a import call chain is not supported and will result in a fatal error during runtime.

For example, the following call chains can successfully consume time in the current release:

SystemVerilog:

Calls an imported task → Issues `wait(...)`
Calls an imported task → Calls an exported task → Issues @
Calls an imported task → Calls an exported task → Calls an imported task → Issues `wait(...)`

SystemC:

Calls an exported task → Issues @
Calls an exported task → Calls an imported task → Issues `wait(...)`
Calls an exported task → Calls an imported task → Calls an exported task → Issues @

The semantics for consuming time in a DPI call chain is the same as if the call was occurring in one language—the language that originated the DPI call chain. In addition, the DPI call chain can resume execution in the same delta cycle in which the time consuming condition is satisfied (for example, if an event is triggered), without artificial delta delays.

Refer to the “SystemC and HDL Design Hierarchies” chapter of the *NC-SC Simulator User Guide* for more information.

Using DPI with the Incisive Simulator

- Using the irun Utility with DPI on page 269
- Using NC-Verilog with DPI on page 271

Using the irun Utility with DPI

With the *irun* utility, you can run the simulator by specifying all input files and all command-line options on a single command line. *irun* determines the language of a file by its extension, and then maps the file to its appropriate compiler.

Importing C Tasks and Functions using irun

To use the *irun* utility with DPI to import C tasks and functions, use the following command:

```
% irun <design_files> <c_files>
```

Where *design_files* are the design files that contain the DPI `import` statement, which imports the tasks and functions contained in the specified *c_files*. For example:

```
% irun systemv1.sv systemv2.sv cfile1.c cfile2.c
```

In this example, *irun* will compile the `.sv` files using `ncvlog -sv` and the C files using a C compiler. After the input files have been compiled, *irun* automatically invokes *ncelab* to elaborate the design and *ncsim* to simulate the design.

Exporting Tasks and Functions to C using irun

To use the *irun* utility with DPI to export SystemVerilog tasks and functions, use the following command:

```
% irun <design_files> -dpiheader <header_file_name> -cpost <c_files> -end
```

where:

- *design_files*—These are the design files that contain the tasks and functions to export. These files are compiled using `ncvlog -sv` before proceeding to elaboration.

  Note: By default, SystemVerilog files must have the `.sv`, `.svp`, `.SV`, or `.SVP` extension. For more information on how to modify the default extensions, refer to "Compiling SystemVerilog Constructs" on page 21 or the *irun User Guide*.

  header_file_name— In your C files that reference the exported tasks and functions, you must include a header file. The `-dpiheader <header_file_name>` option specifies this header file. If the specified header file does not exist, `ncelab` creates it for you. If

you have your own header file, save the header file in the directory where *irun* will be invoked.

For an example of how to include the header file in your C files, refer to “Debugging DPI Exported Functions and Tasks in *SystemVerilog in Simulation*.

Important

You must create the header file before the C files are compiled.

- *c_files*—These are the C files that reference the exported SystemVerilog tasks and functions. If your C files contain export functionality, they must be specified between the `-cpost` and `-end` options so that the *irun* utility knows to compiles these files with a C compiler *after* elaboration.

For example:

```
% irun top.sv -access +rwc -dpiheader myheader.h -cpost add.c -end
```

In this example, *irun* compiles the `top.sv` file using `ncvlog -sv` and then proceeds to elaboration. During elaboration, all `export` declarations are dumped into the `myheader.h` file. After elaboration, *irun* compiles the *add.c* file using a C compiler and then proceeds to simulation.

For information on how to export SystemVerilog tasks and functions using NC-Verilog, refer to “Exporting Tasks and Functions using NC-Verilog” on page 272.

For more information on `irun`, refer to the *irun User Guide*.

Importing SystemC Tasks and Functions Using irun

To use the *irun* utility with DPI to import SystemC tasks and functions, use the following command:

```
% irun <design_files> <systemc_files> -sysc
```

For example:

```
% irun systemv1.sv system2.sv systemcfile.cpp -sysc
```

In this example, *irun* will compile the `.sv` files using `ncvlog -sv` and the `.cpp` file using the `ncsc_run` compiler interface.

Note: By default, SystemVerilog files must have the `.sv`, `.svp`, `.SV`, or `.SVP` extension. For more information on how to modify the default extensions, refer to “Compiling SystemVerilog Constructs” on page 21 or the *irun User Guide*.

Exporting Tasks and Functions to SystemC using irun

To use the *irun* utility with DPI to export SystemVerilog tasks and functions to SystemC, use the following command:

```
% irun -Wcxx -fPIC <systemc_files> <sv_files> -sysc
```

Where:

- `-Wcxx`—Passes the user-specified arguments to C++.
- `-fPIC`—Is required for all C/C++ designs that have `extern` declarations.
- *`systemc_files`*—These are the SystemC files that reference the exported SystemVerilog tasks and functions.
- *`sv_files`*—Are the SystemVerilog files that contain the tasks and functions to export. These files are compiled using `ncvlog -sv` before proceeding to elaboration.

Note: By default, SystemVerilog files must have the `.sv`, `.svp`, `.SV`, or `.SVP` extension. For more information on how to modify the default extensions, refer to "Compiling SystemVerilog Constructs" on page 21 or the *irun User Guide*.

For example:

```
% irun -Wcxx -fPIC sc.cpp test.sv -sysc
```

For more information on `irun`, refer to the *irun User Guide*.

Using NC-Verilog with DPI

This section describes how to import C tasks and functions to SystemVerilog, and how to export SystemVerilog tasks and functions to C.

To import SystemC functions or to export SystemVerilog functions to SystemC, refer to "Using the irun Utility with DPI" on page 269.

Importing Tasks and Functions using NC-Verilog

This section describes how to use NC-Verilog to import C tasks and functions. To use NC-Verilog with DPI to import C tasks and functions, do the following:

1. Compile the design using the following command:

   ```
   % ncvlog -sv <design_files>
   ```

where *design_files* are the design files that contain the DPI `import` statement, which imports the C tasks and functions contained in the specified *c_files*. For example:

```
% ncvlog -sv top.v
```

2. Elaborate the design using `ncelab`. For example:

   ```
   % ncelab -access +RWC top
   ```

3. In the current directory, create a single shared object library. Refer to "Compiling and Linking C Objects into a Single Shared Object Library" on page 273

4. Simulate the design using `ncsim`. For example:

   ```
   % ncsim -messages top
   ```

   If you compiled and linked your C code into a shared library other than `libdpi.`*ext*, refer to "Simulating the Elaborated Snapshot of Libraries Other than libdpi.ext" on page 274.

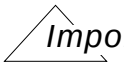
Exporting Tasks and Functions using NC-Verilog

To use NC-Verilog with DPI to export SystemVerilog tasks and functions:

1. Create a header file.

   When using DPI to export tasks and functions, you first need to need to include a header file in your C code. During elaboration, all `export` declarations are dumped into the header file. You can use your own header file or use `ncelab` to generate one for you.

   If you have your own header file, save the header file in the directory where NC-Verilog will be invoked.



   You must create the header file before the C objects are compiled.

   In your C files that reference the exported tasks and functions, include the header file. This is required so that the C symbols that correspond to the exported tasks and functions are externally visible. For an example of how to include the header file in your C files, refer to "Debugging DPI Exported Functions and Tasks in *SystemVerilog in Simulation*.

2. Compile and link your C objects into a single-shared object library that corresponds to your exported tasks and functions. For instructions on how to do this, refer to "Compiling and Linking C Objects into a Single Shared Object Library" on page 273.

3. Go to "Running the Simulator in Multi-Step Mode" on page 273.

Compiling and Linking C Objects into a Single Shared Object Library

If you are exporting SystemVerilog tasks and functions using NC-Verilog, one of the required steps is to compile and link your C objects into a shared library that corresponds to the exported tasks and functions.

Note: Save the shared library in the directory where `ncsim` or `irun` will be invoked, or include the path to the shared library in the library path environment variable (`LD_LIBRARY_PATH` for Solaris and Linux, `SHLIB_PATH` for HP-UX, `LIBPATH` for AIX).

To compile and link your C objects into a shared library using a `gcc` compiler:

```
% gcc -fPIC -shared -o libdpi.so add.c -I $CDS_INST_DIR/tools/inca/include
```

To compile and link your C objects into a shared library using a `cc` compiler:

```
% cc -KPIC -G -o libdpi.so -I $CDS_INST_DIR/tools/inca/include add.c
```

where:

- `add.c` is the C file to compile.
- `libdpi.so` is the shared library to build.

  Note: By default, NC-Verilog looks for a single-shared library called `libdpi.so` (for Solaris, Linux, and AIX) or `libdpi.sl` (for HP-UX). However, the `-sv_root` and `-sv_lib` options of `ncsim` allow you to specify a shared library other than `libdpi.`*`ext`*. These options are discussed in "Running the Simulator in Multi-Step Mode" on page 273.
- `$CDS_INST_DIR` is an environment variable that points to your installation.

Running the Simulator in Multi-Step Mode

If you are using the multi-step invocation mode of NC-Verilog and you have your own header file, do the following:

1. Use the `ncvlog` command with the `-sv` command-line option to compile the SystemVerilog code. For example:

   ```
   % ncvlog -mess test.v -sv
   ```
2. Use the `ncelab` command to elaborate the design. For example:

   ```
   % ncelab -mess top
   ```
3. Simulate the elaborated snapshot. For example:

   ```
   % ncsim -mess top
   ```

If you compiled and linked your C code into a shared library other than `libdpi.ext`, refer to "Simulating the Elaborated Snapshot of Libraries Other than libdpi.ext" on page 274.

If you need to generate a header file using `ncelab`, use the following steps:

1. Use the `ncvlog` command with the `-sv` command-line option to compile the SystemVerilog code. For example:

   ```
   ncvlog -mess test.v -sv
   ```

2. Use the `ncelab` command with the `-dpiheader` command-line option to generate the header file. For example:

   ```
   ncelab -mess top -dpiheader /home/john/myheader.h
   ```

   The `-dpiheader` switch generates a header file that contains declarations for all of the C identifiers that correspond to exported tasks and functions contained in the elaborated snapshot.

   The elaborator saves the header file in the directory specified by the `-dpiheader` switch.

   In your C files that reference the exported the tasks and functions, include the header file. This is required so that the C symbols that correspond to the exported functions are externally visible. For an example of how to include the header file in your C files, refer to "Debugging DPI Exported Functions and Tasks in *SystemVerilog in Simulation*.

Important

   You must generate the header file before you compile your C objects.

3. Compile and link your objects into a single-shared object library that corresponds to your exported tasks and functions. For information on how to do this, refer to "Compiling and Linking C Objects into a Single Shared Object Library" on page 273.

4. Simulate the elaborated snapshot. For example:

   ```
   % ncsim -mess top
   ```

   If you compiled and linked your C code into a shared library other than `libdpi.ext`, refer to "Simulating the Elaborated Snapshot of Libraries Other than libdpi.ext" on page 274.

Simulating the Elaborated Snapshot of Libraries Other than libdpi.ext

If you compiled and linked your C code into a shared library other than `libdpi.ext`, you have to specify the path and name of the shared library using the `-sv_root` and `-sv_lib` options of `ncsim`. For example:

```
% ncsim -mess top -sv_root /a/b/c -sv_lib mylib
```

where:

- `-sv_root` can be a single directory, which will be prepended to any relative path specified by `-sv_lib`.
- `-sv_lib` can be a relative or full path name that corresponds to a shared library. If you provide a relative path, it is prepended by the path specified by `-sv_root`.

For example, the following loads `/a/b/c/mylib.`*`ext`* during simulation:

```
% ncsim -mess top -sv_root /a/b/c -sv_lib mylib
```

Using the `-sv_root` and `-sv_lib` options with `ncsim` allows you to load multiple shared libraries during simulation. For example, the following loads `/x/y/myl.`*`ext`* and `../myl1.`*`ext`* during simulation:

```
% ncsim -mess top -sv_root /x/y/ -sv_lib myl -sv_root ../ -sv_lib myl1
```

Disabling DPI Tasks and Functions

Disabling DPI tasks and functions is described in Section 26.8 of the IEEE 1800 standard.

For an example that you can download and run, refer to the *SystemVerilog DPI Engineering Notebook*.

In SystemVerilog, tasks may be disabled using a `disable` statement. An imported task or function is considered disabled when it or its parent task is the target of a SystemVerilog `disable` statement. In the current release, you can use the `disable` statement to disable process call chains that contain DPI imported tasks/functions and exported tasks.

To account for disabled imported tasks (or the exported task that they call), the C code must follow a certain protocol to programmatically acknowledge that a task has been disabled:

- Imported C or SystemC tasks must return an `int` value of 1 to indicate a disable. Otherwise, they should return a value of 0. If a disabled task does not return a value of 1, the simulator will issue an error message.
- When an imported C or SystemC function is disabled, it should call the `svAckDisabled` API function. If a disabled function does not call this API function before it returns to its parent, the simulator will issue an error message.
- When an exported task is the target of a `disable`, its parent imported task is not considered disabled when the exported task returns. In this case, the exported task will return 0 and any calls to `svIsDisabledState()` will return 0. When both the parent

import task and export task are disabled, the export task must return an integer value of 1.

Imported tasks/functions and exported tasks can also determine whether they are disabled by calling the `svIsDisabledState()` API function.

The following illustrates an imported C task that calls an exported SystemVerilog task.

In SystemVerilog:

```
...
import “DPI-C” context task imp_t1(inout int a);
export “DPI-C” task exp_t1;

task exp_t1(output int a);
...
endtask
```

In C:

```
extern int exp_t1 (int *a);

int imp_t1(int *a);
{
int ret;
...
ret = exp_t1();
...
return ret;
}
```

In this example, the functions `imp_t1` and `exp_t1` return an `int`. The `exp_t1` function will return 0 unless its parent function `imp_t1` is disabled, in which case `exp_t1` will return 1. The `imp_t1` function must check for a disable and acknowledge any disables by also returning 1:

```
int imp_t1(int *a);
{
int ret;
...
ret = svIsDisabledState();
...
return ret;
}
```

According to the LRM, the `disable` statement cannot be used to disable exported functions.

Debugging DPI Import and Export Functions

For information on how to debug DPI import/export functions using the Tcl command-line interface, refer to *SystemVerilog in Simulation*.

DPI Accessor Functions

The following DPI accessor functions are supported in the current release.

Table 21-4 Supported DPI Accessor Functions

svAckDisabledState	svGetArrayPtr	svGetArrElemPtr1
svGetBitArrElem1	svGetBitArrElem1Vec32	svGetBitArrElem1VecVal
svGetBitselBit	svGetBitselLogic	svGetCallerInfo
svGetLogicArrElem1	svGetLogicArrElem1Vec32	svGetLogicArrElem1VecVal
svGetNameFromScope	svGetPartselBit	svGetPartSelectBit
svGetPartSelectLogic	svGetPartselLogic	svGetScope
svGetScopeFromName	svGetSelectBit	svGetSelectLogic
svGetUserData	svHigh	svIsDisabledState
svLeft	svLow	svPutBitArrElem1
svPutBitArrElem1Vec32	svPutBitArrElem1VecVal	svPutBitselBit
svPutBitselLogic	svPutLogicArrElem1	svPutLogicArrElem1Vec32
svPutLogicArrElem1VecVal	svPutPartselBit	svPutPartSelectBit
svPutPartSelectLogic	svPutPartselLogic	svPutSelectBit
svPutSelectLogic	svPutUserData	svRight
svSetScope	svSizeOfArray	

The following DPI accessor functions are not supported in the current release.

Table 21-5 Unsupported DPI Accessor Functions

svGet32Bits	svGetArrElemPtr3	svGetBitArrElem
svGetArrElemPtr2	svGetBitArrElem2Vec32	svGetBitArrElem2VecVal
svGetBitArrElem2	svGetBitArrElem3Vec32	svGetBitArrElem3VecVal
svGetBitArrElem3	svGetBitArrElemVecVal	svGetBits
svGetBitArrElemVec32	svGetLogicArrElem	svGetLogicArrElem2
svGetBitVec32	svGetLogicArrElem2VecVal	svGetLogicArrElem3

svGetLogicArrElem2Vec32	svGetLogicArrElem3VecVal	svGetLogicArrElemVec32
svGetLogicArrElem3Vec32	svGetLogicVec32	svIncrement
svGetLogicArrElemVecVal	svLength	svPutBitArrElem
svPutBitArrElem2	svPutBitArrElem2Vec32	svPutBitArrElem2VecVal
svPutBitArrElem3Vec32	svPutBitArrElem3VecVal	svPutBitArrElemVec32
svPutBitArrElemVecVal	svPutBitVec32	svPutLogicArrElem
svPutLogicArrElem2	svPutLogicArrElem2Vec32	svPutLogicArrElem2VecVal
svPutLogicArrElem3	svPutLogicArrElem3Vec32	svPutLogicArrElem3VecVal
svPutLogicArrElemVec32	svPutLogicArrElemVecVal	svPutLogicVec32
svSizeOfBitPackedArr	svSizeOfLogicPackedArr	
svDimensions	svDpiVersion	
svGet64Bits	svGetArrElemPtr	

DPI Examples

This section provides extensive examples for using DPI with C and SystemC.

For more DPI examples that you can download and run, refer to the *SystemVerilog DPI Engineering Notebook*.

Using DPI with C

In SystemVerilog:

```
module top;
 int top_res;
 bit b1 = 1'b1;
 bit b2 = 1'b1;


 import "DPI-C" context xxx = function int imp_func(input bit bin1, inout bit bin2);
 export "DPI-C" ccc = function exp_func;

 function int exp_func(input bit eb1, output bit eb2);
   if(eb1 != eb2)
    begin
      eb2 = eb1;
      return -43;
    end
   else
      return 54;
 endfunction

 initial
   begin
     #2 top_res = imp_func(b1,b2);
     $display("top_res : %d\n", top_res);
   end

endmodule
```

Imports the C function called xxx and calls it imp_func in SystemVerilog.

Exports the SystemVerilog function called exp_func and calls it ccc in C.

Defines exp_func that will be exported to C.

Invokes the imported C function xxx.

In C:

```
#include <stdio.h>
#include "myheader.h"
  extern int ccc (unsigned char, unsigned char *);

  int xxx (svBit b1, svBit *b2)
  {
  int res;
  printf("Before calling ccc ****** b1: %d, b2 : %d\n", b1, *b2);
  res = ccc(b1,b2);
  printf("Res:%d\n", res);
  printf("After calling ccc ****** b1: %d, b2 : %d\n",b1, *b2);

  printf("\n\nBefore calling ccc ****** b1: %d, b2 : %d\n", b1, *b2);
  res = ccc(b1,b2);
  printf("Res:%d\n", res);
  printf("After calling ccc ****** b1: %d, b2 : %d\n",b1, *b2);
  return res
  }
```

Includes the header file generated by ncelab.

Declares the exported SystemVerilog function xxx as extern.

Defines the imported xxx function.

Invokes the exported function exp_func.

In this example:

- In SystemVerilog, the top module's function imp_func invokes the C function xxx.

- In C, the `xxx` function calls the `ccc` function twice (before and then after probing the values of `b1` and `b2`). The call to the `ccc` function invokes the SystemVerilog function `exp_func` with `b1` and `b2` as `svBit` type parameters. The return value of function `exp_func` is assigned to `res`.

For example:

```
irun -sv test.v testme.c
```

should produce the following simulation result:

```
Before calling ccc ******b1 : 1, b2 :0
Res : -43
After calling ccc ******b1: 1, b2: 1

Before calling ccc ******b1 : 1, b2 :1
Res : 54
After calling ccc ******b1: 1, b2: 1
top_res: 54
```

Using DPI with SystemC

In SystemVerilog (test.sv file):

```
module scmod
  (* integer foreign = "SystemC"; *);
endmodule

module svtop;

  scmod scm();
  import "DPI-SC" context function void scmod_run ( input sc_int[31:0] a);
  export "DPI-SC" function sc_exp;

  function void sc_exp ( input sc_int[31:0] i);
   $display(" Inside sc_exp : value is %d ",i);
  endfunction

 initial begin
  scSetScopeByName("svtop.scm");
  scmod_run(17);
 end
endmodule
```

Imports the SystemC `sc_object` class member method `scmod_run` and uses a SystemC type as a formal argument.

Exports the SystemVerilog function called `sc_exp`.

Defines `sc_exp`, which will be exported to C.

Sets the scope and invokes the imported `sc_object` class member method `scmod_run`.

In SystemC (sc.cpp file):

```
#define NCSC_INCLUDE_TASK_CALLS
#include "systemc.h"

extern "C" {
  extern void sc_exp(sc_int<32> a);
}

class scmod : public sc_module {
  public:
  SC_CTOR(scmod) {
  }
  void scmod_run ( sc_int<32> i1) {
   cout << "Inside SystemC value is " << i1 << endl;
  sc_exp(i1);
  }
};

NCSC_MODULE_EXPORT(scmod)
NCSC_REGISTER_DPI_MEMBER_ALIAS(scmod_run,scmod,scmod_run)
```

`#define` must come before the first `#include` of `systemc.h`.

Declares the exported SystemVerilog function `sc_exp` as `extern`.

Defines the imported `sc_object` class member method `scmod_run`.

Invokes the exported function `sc_exp`.

Registers member method that inherits from `sc_object`.

For example:

```
% irun -Wcxx -fPIC sc.cpp test.sv -sysc
```

produces the following simulation result:

```
Inside SystemC value is 17
Inside sc_exp : value is 17
```

Using scSetScopeByName in SystemVerilog

In SystemVerilog (test.sv file):

```
‘timescale 1 ns/1 ps

  module sctop (* integer foreign = "SystemC"; *);
  endmodule

module top;
  int r1;
  byte b1 = "A";
  sctop sc();
  import "DPI-SC" function int myglobalfunc (input byte a);
  import "DPI-SC" function int myclassfunc (input byte a);
  initial begin
   #10
   r1=myglobalfunc( b1);
   #10
   scSetScopeByName("top.sc");
   r1=myclassfunc(b1);
   $finish;
end
endmodule
```

Imports two SystemC functions.

Calls an imported function. Does not require `scSetScopeByName()`, because `myglobalfunc()` is global.

Sets the scope to `top.sc`.

Calls an imported function. Requires `scSetScopeByName()`, because `myclassfunc()` is a member method of a class that inherits from `sc_object`.

In SystemC (sc.cpp file):

```
#define NCSC_INCLUDE_TASK_CALLS
#include "systemc.h"

int myglobalfunc(char i1) {
  cerr << sc_time_stamp() << " starting myglobalfunc - i1 = " << i1
  <<  endl;
  return 0;
}

class sctop : public sc_module {
public:
  SC_CTOR(sctop) {
  }
  int myclassfunc(char i1) {
  cerr << sc_time_stamp() << " starting myclassfunc - i1 = " << i1
  <<  endl;
  return 0;
}

};

NCSC_MODULE_EXPORT(sctop)
NCSC_REGISTER_DPI(myglobalfunc)
NCSC_REGISTER_DPI_MEMBER_ALIAS(myclassfunc,sctop,myclassfunc)
```

`#define` must come before the first `#include` of `systemc.h`.

Defines the global imported function `myglobalfunc()`.

Defines the sc_object class member method imported function `myclassfunc()`.

Registers global functions.

Registers member methods that inherit from `sc_object`.

For example:

```
% irun sc.cpp test.sv -sysc
```

produces the following results:

```
10 ns starting myglobalfunc - i1 = A
20 ns starting myclassfunc - i1 = A
```

Unpacked Structs as Formal Arguments to DPI-C Import Functions

This example demonstrates the guidelines listed in "Using Unpacked Structs as Formal Arguments in DPI-C Import Functions" on page 254.

In SystemVerilog (test.sv file):

```
module top;

typedef struct {
  int a;
  logic b;
  bit c;
} mystruct ;

import "DPI-C" task t1(input mystruct b);
  mystruct aa;
  initial begin
  aa.a = 10;
  aa.b = 1'bz;
  aa.c = 1'b1;
  t1(aa);
  end
endmodule
```

Layout of SystemVerilog and C structs match.

In C:

```
#include<stdio.h>
#include<svdpi.h>

typedef struct {
  int a;
  svLogic b;
  svBit c;
}myCstruct;

int t1(myCstruct *p) {
  printf("p->a = %d\n",p->a);
  printf("p->b = %d\n",p->b);
  printf("p->c = %d\n",p->c);
}
```

Formal argument in C is a pointer to a C struct.

For example:

```
% irun testme.c test.sv
```

produces the following results:

```
p->a = 10
p->b = 2
p->c = 1
```

Unpacked Structs as Formal Arguments to DPI-SC Import Functions

In SystemVerilog (test.sv file):

```
module top;
 typedef struct {
  int a;
  logic b;
  bit c;
} mystruct;

import "DPI-SC" function void sc_imp_func(
input mystruct a1,
input sc_int[0:31] a2
);
mystruct aa;

initial begin
  aa.a = 10;
  aa.b = 1'bz;
  aa.c = 1'b1;
  sc_imp_func(aa, 32);
end
endmodule
```

Layout and names of SystemVerilog and C structs match.

In SystemC (testme.cpp):

```
#define NCSC_INCLUDE_TASK_CALLS
#include <systemc.h>
#include "sysc/cosim/sc_dpi_convert.h"

typedef struct {
  int a;
  svLogic b;
  svBit c;
} mystruct;

void sc_imp_func(mystruct a1, sc_int<32> a2) {

 int a;
 sc_logic b;
 bool c;

 sc_dpi_convert* converter = sc_dpi_converter();
 a = a1.a;
 // convert from svLogic to sc_logic
 converter->sc_from_logic(b, a1.b);

 // convert from svBit to bool
 converter->sc_from_bit(c, a1.c);
 cerr << "mystruct:" << a << "," << b << "," << c << endl;
}
```

struct contains only C data type fields.

Formal argument in SystemC is a C struct.

sc_dpi_convert class used to convert C types to SystemC types.

For example:

```
% irun testSC.cpp testSC.sv -sysc
```

produces the following results:

```
mystruct:10,Z,1
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

Index

-- operator 115

Symbols

!=? operator 116
$bits system function 236
$cast dynamic casting 111
$dimensions array query function 80
$display 47
$exit() 197
$fdisplay 47
$fmonitor 47
$fopen 47
$fstrobe 47
$fwrite 47
$high array query function 79
$increment array query function 79
$left array query function 79
$low array query function 79
$monitor 47
$right array query function 79
$root 235
$root out-of-module reference 235
$size array query function 80
$sscanf 47
$strobe 47
$unit 212
$unit_#
supported declarations 211
$unpacked_dimensions array query function 80
$urandom function 165
$urandom_range 166
$write 47
** operator 148
++ operator 115
.* implicit port connections 34
.name implicit port connections 33
= operator 115
==? operator 115, 116
->> operator 180
`begin_keyword 240
`begin_keywords 21
`end_keyword 240
`end_keywords 21
`remove_keyword 21
`restore_keyword 21

A

accessor functions, for DPI 277
aggregate expressions 119
aliased rand handles, example of 146
always_comb 128
always_ff 128
always_latch 128
AMS 14
-arr_access 22
array locator methods 97
example using 98
limitations 98
supported methods on queues 98
array of interfaces, example 222
array ports, interfaces 221
array querying functions 79
examples 80
limitations 79
supported functionality 80
arrays 75
debugging 99
in constraints 149
manipulation methods 76
of strings 50, 51
querying functions 79
assertions 14, 27
assignment operators 115
assignment patterns 116
associative arrays 87
access methods 88
limitations 90
of strings 90
supported data types 90
supported index types 91
atobin() string method 45
atohex() string method 45
atoi() string method 45
atooct() string method 45
atoreal() string method 46

B

bintoa() string method 45
blocking assignments, in program blocks 196
break statement 127

C

casting 73
  example of 73
  limitations 73
  syntax 73
casting to void 137
chandles 43
  inside packages 43
  limitations 43
class scope resolution operator 113
classes 107
  debugging 114
  declaring 107
  global class definitions 113
  new function 108
  OOMRs 113
  parameterized classes 113
  static data members 109
  static methods 109
clock drive 190
clocking blocks 185 to ??
  clock drive 190
  clocking direction 187
  cycle delays 190
  debugging 191
  declaring 185
  default blocks 189
  default skews 187
  defining clocking items 188
  engineering notebooks 185
  hierarchical expressions 188
  within modports 228
clocking blocs
  examples 185
clocking direction, in clocking blocks 187
clocking domain 185
clocking items, defining 188
clocking_drive 190
compare() string method 45
compilation units
  $unit 212
  compiling, example 210
  default implementation 210
  limitations 212
  package references 204
  referencing declarations 212
  scope support 211
compilation-unit scopes 211
compiler directives
  `begin_keyword 240
  `define 239
  `end_keywords 240
  `remove_keyword 245
  `restore_keyword 245
compiling C code 273
compiling SystemVerilog 242
  with AMS 14
concurrent assertions 201
constant values, naming 60
constraint blocks 148
  4-state values 148
  distribution 151
  exponentiation operators 148
  inheritance 149
constraint_mode() 158
continue statement 127
continuous assignments to variables 102 to 105
  example 103
  restrictions 103, 105
coverage 14
cycle delay 190

D

data members, in classes 107
data type mapping
  SystemC to SystemVerilog 257
  using typedefs with SystemC types 265
data types 37 to ??
  bit 41
  byte 42
  chandles 43
  enumerated 59
  for storing integers 37
  int 42
  limitations 74
  logic 40
  longint 42
  on ports 212
  primitive 38

shortint 42
string 43
user-defined 39
debugging
arrays 99
classes 114
clocking blocks 191
packages 209
random constraints 172
structures 71
debugging support 13
debuggins
queues 99
declaring local variables in unnamed blocks 101
decrement operator 115
default clocking blocks 189
default skews, in clocking blocks 187
define text substitution macro 239
delete() queue access method 93
direct programming interface (DPI) 249 to 280
accessor functions, list of supported 277
accessor functions, list of unsupported 277
backward compatibility 252, 257
compiling C code 273
context functions 250
creating shared object libraries 273
disabling tasks 252, 257, 275
example using C 279
example using scSetScopeByName 282
example using SystemC 281
examples 278
export declaration 259, 263
formal arguments 262
return values 261
exported functions 259, 262
exported tasks that consume time 266
import declarations 249
formal arguments 253
return values 252
syntax 251, 256
invoking from SystemC designs 263
local names, specifying 252
pure functions 250
required return type 252, 257
using with irun 269
using with the simulator 269
disable construct 275
disable fork 132
disabling DPI tasks 252, 257, 275
distribution constraint
defining 150
limitations 151
do...while loop 124
documentation
additional references 18
printing 17
searching 17
viewing 16
dot operator 54
DPI.*See also* Direct Programming Interface (DPI)
-dpi_void_task 252, 257
dynamic arrays
accessing out-of-bound elements 83
example of passing by reference 83
fixed arrays of 84
of strings 52

E

engineering notebooks
clocking blocks 185
DPI 249
interfaces 215, 221
list of 14
mailboxes 175
program blocks 193
enum keyword 59
enumerated data type 59 to 64
constants 60
declaring 59
displaying member values 63
enumeration objects 61
limitations 59
methods 63
type checking 62
event variables 182
events
comparing 182
in classes 179
merging 182
non-blocking event trigger 179
passing to tasks and functions 179
persistent trigger 181
reclaiming 182
examples

clocking blocks 185
DPI 249
interfaces 215, 221
mailboxes 175
program blocks 193
queues 92
external declarations
in compilation-unit scopes 210
supported 211

F

final blocks 127
find() array locator method 98
find_first() array locator method 98
find_first_index() array locator method 98
find_index() array locator method 98
find_last() array locator method 98
find_last_index() array locator method 98
fixed arrays
of dynamic arrays 84
of strings 50, 51
for generate loop 85, 91, 95
for loop 124
foreach loop 125
fork...join 129
fork...join_any 129
fork...join_none 129
function output arguments 136

G

genvar variables 125
get_randstate() 167
getc string method 45
global class definitions 113
global constraints 153

H

help
on commands 15
on documentation 15
on tool messages 15
hextoa() string method 45
hierarchical identifiers 188

I

icompare() string method 45
if...else constraints 152
iff event control qualifier 128
implication constraints
syntax 151
import statement 206
importing C functions 249
Incisive Design Team Simulator 13
Incisive Enterprise Simulator 13
Incisive HDL Simulator 13
Incisive Simulator 13
increment operator 115
inheritance, in constraint blocks 149
initializing variables 101
insert() queue access method 93
inside operator 149
for rand variables 149
for randc variables 149
support for arrays 149
installation examples 14
interface array ports 221
with modports 224
interface array ports, examples 222
interfaces 215 to ??
array ports 221
declaring 216
declaring tasks and functions 230
engineering notebook 215, 221
examples 215, 221
instantiating 221
modports 225
referencing 224
timing 234
using as a module port 219
interleaved productions 172
irun utility 242
-fPIC option 271
-sysc option 270, 271
-Wcxx option 271
iterative constraints 153
itoa() string method 45

J

jump statements 126

K

keywords
  specifying 240

L

legacy code 21
len() string method 45
literal value assignments 29
localparam of strings 52
lowercase strings 45

M

mailboxes
  as OOMRs 178
  copying a message 178
  creating 176
  engineering notebooks 175
  example 175
  extending 178
  fixed-arrays of mailboxes 178
  limitations 178
  methods 176
  passing as arguments 178
  placing a message 177
  prototype 176
  retreiving a message 177
  returning number of messages 177
  supported scopes 178
matching end names 29
mixed-language support 242
modports 225
  with interface array ports 224
  within for loops 125

N

NC-Verilog Simulator 13
nesting program blocks 195
net type, uwire 71
new function 108
newlink globalconstraints 153
newlink iaport 221
-nomempack 22
non-blocking assignments, in program blocks 196
non-blocking event trigger 179
non-terminals 169
null rand handles, example of 147

O

octtoa() string method 45
operators 115 to 116
  assignment 115
  decrement 115
  increment 115
  wild equality 116

P

packages 203 to ??
  compilation unit references 204
  compiling 203
  debugging 209
  declaring 205
  import statement 206
  referencing data 206
  supported references 204
packed arrays 76
  limitations 78
  supported functionality 76
packed structures 65 to ??
  declaring 67
  limitations 66
parameterized classes 113
parameterized mailboxes 179
parameters of strings 52
parentheses, in enumeration methods 64
parentheses, in function calls 143
persistent trigger 181
pop_back() queue access method 94
pop_front() queue access method 94
port declarations 212
primitive data types 38
priority keyword 122
procedural_timing_control_statement 190
productions 169
program blocks
  $exit() control task 197
  declaring 193
  engineering notebook 193
  example 193
  instantiating 197

restrictions 194, 195
supported constructs 194
syntax 193
variable assignments 196
ps_covergroup_identifier 74
push_back() queue access method 94
push_front() queue access method 94
putc string method 45

Q

querying functions, for arrays 79
queues 92
access methods 93
array locator methods 97
debugging 99
deleting 93
example using array locator methods 98
examples 92
indexing into 95
limitations 94
of strings 94

R

rand 146
rand handles
aliased 146
null 147
rand join 172
rand_mode() 157
randc 146
randcase 167
random constraints
constraint blocks 148
debugging 172
random variables
limitations 147
random weighted case 167
randomization of scope variables 160
randomize() 155
getting different results 145
scope randomization 160
randomize() with 157
randsequence blocks 169 to 170
declaring 169
interleaved productions 172
limitations 172
rand join production control 172
realtoa 46
ref keyword 138
reserved keywords
IEEE 1800-2005 243
removing 245
specifying 245
return statement 127
-rmkeyword 21, 245

S

scope randomization
constraints, specifying 162
supported constraint expressions 162
semaphores 173
limitations 175
methods 174
passing as arguments to tasks and functions 175
prototype 173
set_randstate() 167
shared object library, creating 273
shortreal 74
simulating SystemVerilog designs 13
single-driver nets 71
size() queue access method 93
solve...before constraints 154
srandom() 166
static classes 109
static constraint blocks 154
static keyword 155
static type casts 73
strings 43
as localparams 52
as parameters 52
comparing 44
concatenation 54
displaying length 45
dot operator 54
dynamic arrays of strings 52
example 46, 48, 50
example of fixed array of strings 51
limitations 54
lowercase 45
methods 45
operators 44
replication 54
syntax 43
system tasks 47

uppercase 45
with packages 49
within begin...end blocks 49
structures
debugging 71
packed structures 65 to ??
unpacked structures 67
substr 46
supported constructs 23
sv* accessor functions 277
svAckDisabled 275
svAckDisabledState 277
svGetArrayPtr 277
svGetArrElemPtr1 277
svGetBitArrElem1 277
svGetBitArrElem1Vec32 277
svGetBitArrElem1VecVal 277
svGetBitselBit 277
svGetBitselLogic 277
svGetCallerInfo 277
svGetLogicArrElem1 277
svGetLogicArrElem1Vec32 277
svGetLogicArrElem1VecVal 277
svGetNameFromScope 277
svGetPartselBit 277
svGetPartSelectBit 277
svGetPartSelectLogic 277
svGetPartselLogic 277
svGetScope 277
svGetScopeFromName 277
svGetSelectBit 277
svGetSelectLogic 277
svGetUserData 277
svHigh 277
svIsDisabledState 276, 277
svLeft 277
svLow 277
svPutBitArrElem1 277
svPutBitArrElem1Vec32 277
svPutBitArrElem1VecVal 277
svPutBitselBit 277
svPutBitselLogic 277
svPutLogicArrElem1 277
svPutLogicArrElem1Vec32 277
svPutLogicArrElem1VecVal 277
svPutPartselBit 277
svPutPartSelectBit 277
svPutPartSelectLogic 277
svPutPartselLogic 277
svPutSelectBit 277
svPutSelectLogic 277
svPutUserData 277
svRight 277
svSetScope 277
svSizeOfArray 277
SystemC
data type mapping for typedefs 265
debugging exported functions 263
default data type mapping 257
exported functions and tasks 262
exporting using irun 271
import declaration syntax 256
importing using irun 270
limitations 264
sample call chains 267
setting the scope 256
SystemVerilog code
using legacy code 21
with AMS 14
SystemVerilog examples 14
SystemVerilog VPI extensions 13
SystemVerilog, list of supported constructs 23
-sysv_ext 22

T

tasks and functions 135 to 143
default argument values 141
default direction 136
function output arguments 136
in interfaces 230
multiple statements 135
optional arguments 143
parentheses 143
passing arguments by name 142
passing arguments by reference 138
void functions 137
Tcl support 13
terminals 169
text strings 43
tf_nodeinfo() 22
time-consuming tasks 266
sample call chains 267
support for 266
tolower() string method 45
toupper() string method 45
trigger
non-blocking event 179
persistent 181
typedefs 55

limitations 55

U

uninitialized objects, default value for 108
union 74
unique keyword 122
unique/priority if 121
unpacked arrays 76
limitations 78
supported functionality 77
supported operators 78
unpacked structures
in classes 69
limitations 69
supported functionality 67
uppercase strings 45
user-defined data types 39
uwire net type 72

V

variables
continuous assignments 102
data types on ports 212
declaring with initializers 101
ordering 154
virtual interfaces 230
limitations 234
specializations 231
syntax 231
void
data type 137
functions 137
VPI 13

W

wait fork 132
wild equality operator 116