

Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language

ICS 25.040.01; 35.060

National foreword

This British Standard is the UK implementation of IEC 62530:2007.

The UK participation in its preparation was entrusted to Technical Committee GEL/93, Design automation.

A list of organizations represented on this committee can be obtained on request to its secretary.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 31 December 2007

© BSI 2007

ISBN 978 0 580 59320 8

Amendments issued since publication

Amd. No.	Date	Comments



IEC 62530

Edition 1.0 2007-11

INTERNATIONAL STANDARD

IEEE 1800™

**Standard for SystemVerilog – Unified Hardware Design, Specification, and
Verification Language**

CONTENTS

IEEE introduction.....	11
1. Overview.....	12
1.1 Scope.....	12
1.2 Purpose.....	12
1.3 Conventions used in this standard	14
1.4 Syntactic description.....	14
1.5 Use of color in this standard	15
1.6 Contents of this standard.....	15
1.7 Examples.....	18
1.8 Prerequisites.....	18
2. Normative references.....	20
3. Literal values.....	22
3.1 Introduction.....	22
3.2 Literal value syntax.....	22
3.3 Integer and logic literals	23
3.4 Real literals	23
3.5 Time literals	23
3.6 String literals.....	23
3.7 Array literals	24
3.8 Structure literals.....	24
4. Data types	26
4.1 Introduction.....	26
4.2 Data type syntax.....	27
4.3 Integer data types	28
4.4 Real and shortreal data types	29
4.5 Void data type.....	29
4.6 Chandle data type.....	29
4.7 String data type	30
4.8 Event data type.....	35
4.9 User-defined types	35
4.10 Enumerations	37
4.11 Structures and unions.....	42
4.12 Class.....	48
4.13 Singular and aggregate types	48
4.14 Casting	48
4.15 \$cast dynamic casting	49
4.16 Bit-stream casting	50
4.17 Default attribute type	53
5. Arrays.....	54
5.1 Introduction.....	54
5.2 Packed and unpacked arrays	54
5.3 Multiple dimensions	55
5.4 Indexing and slicing of arrays.....	56
5.5 Array querying functions	57
5.6 Dynamic arrays	57
5.7 Array assignment	59
5.8 Arrays as arguments.....	60
5.9 Associative arrays.....	61

5.10	Associative array methods	64
5.11	Associative array assignment.....	67
5.12	Associative array arguments	67
5.13	Associative array literals.....	67
5.14	Queues	67
5.15	Array manipulation methods	70
6.	Data declarations.....	76
6.1	Introduction.....	76
6.2	Data declaration syntax.....	76
6.3	Constants.....	77
6.4	Variables	81
6.5	Nets	82
6.6	Scope and lifetime	83
6.7	Nets, regs, and logic.....	84
6.8	Signal aliasing.....	85
6.9	Type compatibility	87
6.10	Type operator.....	90
7.	Classes	92
7.1	Introduction.....	92
7.2	Syntax	92
7.3	Overview.....	93
7.4	Objects (class instance).....	94
7.5	Object properties.....	94
7.6	Object methods	95
7.7	Constructors	95
7.8	Static class properties.....	96
7.9	Static methods.....	97
7.10	This	97
7.11	Assignment, renaming, and copying.....	98
7.12	Inheritance and subclasses	99
7.13	Overridden members.....	99
7.14	Super	100
7.15	Casting	101
7.16	Chaining constructors	101
7.17	Data hiding and encapsulation.....	102
7.18	Constant class properties	102
7.19	Abstract classes and virtual methods	103
7.20	Polymorphism: dynamic method lookup	104
7.21	Class scope resolution operator ::	104
7.22	Out-of-block declarations	105
7.23	Parameterized classes	106
7.24	Typedef class	107
7.25	Classes and structures	108
7.26	Memory management	108
8.	Operators and expressions	110
8.1	Introduction.....	110
8.2	Operator syntax.....	110
8.3	Assignment operators	112
8.4	Operations on logic and bit types	112
8.5	Wild equality and wild inequality.....	113
8.6	Real operators	113

8.7	Size.....	114
8.8	Sign	114
8.9	Operator precedence and associativity	114
8.10	Built-in methods	114
8.11	Static prefixes	115
8.12	Concatenation	116
8.13	Assignment patterns.....	117
8.14	Tagged union expressions and member access.....	122
8.15	Aggregate expressions	123
8.16	Operator overloading.....	123
8.17	Streaming operators (pack/unpack)	125
8.18	Conditional operator	129
8.19	Set membership.....	129
9.	Scheduling semantics.....	132
9.1	Execution of a hardware model and its verification environment	132
9.2	Event simulation	132
9.3	The stratified event scheduler	132
9.4	The PLI callback control points.....	136
10.	Procedural statements and control flow	138
10.1	Introduction.....	138
10.2	Statements.....	138
10.3	Blocking and nonblocking assignments	139
10.4	Selection statements.....	140
10.5	Loop statements	147
10.6	Jump statements	149
10.7	Final blocks.....	149
10.8	Named blocks and statement labels	150
10.9	Disable	151
10.10	Event control.....	152
10.11	Level-sensitive sequence controls	154
10.12	Procedural assign and deassign removal	154
11.	Processes.....	156
11.1	Introduction.....	156
11.2	Combinational logic.....	156
11.3	Latched logic.....	157
11.4	Sequential logic.....	157
11.5	Continuous assignments	157
11.6	fork...join.....	158
11.7	Process execution threads	159
11.8	Process control.....	159
11.9	Fine-grain process control	161
12.	Tasks and functions	164
12.1	Introduction.....	164
12.2	Tasks	164
12.3	Functions.....	166
12.4	Task and function argument passing	168
12.5	Import and export functions.....	171
13.	Random constraints.....	174
13.1	Introduction.....	174

13.2	Overview.....	174
13.3	Random variables	177
13.4	Constraint blocks	179
13.5	Randomization methods	192
13.6	In-line constraints—randomize() with.....	195
13.7	Disabling random variables with rand_mode()	195
13.8	Controlling constraints with constraint_mode()	196
13.9	Dynamic constraint modification.....	197
13.10	In-line random variable control	198
13.11	Randomization of scope variables—std::randomize().....	199
13.12	Random number system functions and methods	200
13.13	Random stability	202
13.14	Manually seeding randomize	204
13.15	Random weighted case—randcase	205
13.16	Random sequence generation—randsequence.....	206
14.	Interprocess synchronization and communication.....	214
14.1	Introduction.....	214
14.2	Semaphores	214
14.3	Mailboxes.....	215
14.4	Parameterized mailboxes	218
14.5	Event	219
14.6	Event sequencing: wait_order()	221
14.7	Event variables.....	222
15.	Clocking blocks	224
15.1	Introduction.....	224
15.2	Clocking block declaration	224
15.3	Input and output skews	226
15.4	Hierarchical expressions	227
15.5	Signals in multiple clocking blocks	227
15.6	Clocking block scope and lifetime.....	227
15.7	Multiple clocking blocks example	228
15.8	Interfaces and clocking blocks.....	228
15.9	Clocking block events.....	230
15.10	Cycle delay: ##	230
15.11	Default clocking.....	231
15.12	Input sampling	232
15.13	Synchronous events	232
15.14	Synchronous drives.....	233
16.	Program block.....	236
16.1	Introduction.....	236
16.2	The program construct.....	236
16.3	Eliminating testbench races	238
16.4	Blocking tasks in cycle/event mode.....	239
16.5	Programwide space and anonymous programs.....	239
16.6	Program control tasks	240
17.	Assertions.....	242
17.1	Introduction.....	242
17.2	Immediate assertions.....	242
17.3	Concurrent assertions overview	244
17.4	Boolean expressions	245

17.5	Sequences.....	247
17.6	Declaring sequences	250
17.7	Sequence operations	253
17.8	Manipulating data in a sequence.....	270
17.9	Calling subroutines on match of a sequence.....	274
17.10	System functions.....	275
17.11	Declaring properties.....	275
17.12	Multiclock support.....	288
17.13	Concurrent assertions.....	296
17.14	Clock resolution	302
17.15	Binding properties to scopes or instances.....	308
17.16	Expect statement	310
17.17	Clocking blocks and concurrent assertions.....	311
18.	Coverage	314
18.1	Introduction.....	314
18.2	Defining the coverage model: covergroup.....	314
18.3	Using covergroup in classes	317
18.4	Defining coverage points	319
18.5	Defining cross coverage.....	325
18.6	Specifying coverage options	329
18.7	Predefined coverage methods	334
18.8	Predefined coverage system tasks and functions	335
18.9	Organization of option and type_option members	335
18.10	Coverage computation	336
19.	Hierarchy	338
19.1	Introduction.....	338
19.2	Packages.....	338
19.3	Compilation unit support	343
19.4	Top-level instance.....	344
19.5	Module declarations.....	345
19.6	Nested modules	345
19.7	Extern modules	347
19.8	Port declarations	348
19.9	List of port expressions.....	349
19.10	Time unit and precision	350
19.11	Module instances	351
19.12	Port connection rules	356
19.13	Name spaces	357
19.14	Hierarchical names	358
20.	Interfaces.....	360
20.1	Introduction.....	360
20.2	Interface syntax.....	361
20.3	Ports in interfaces.....	365
20.4	Modports.....	366
20.5	Interfaces and specify blocks.....	372
20.6	Tasks and functions in interfaces.....	373
20.7	Parameterized interfaces.....	379
20.8	Virtual interfaces.....	381
20.9	Access to interface objects.....	385
21.	Configuration libraries.....	388

21.1	Introduction.....	388
21.2	Libraries	388
22.	System tasks and system functions.....	390
22.1	Introduction.....	390
22.2	Type name function	390
22.3	Expression size system function	391
22.4	Range system function.....	391
22.5	Shortreal conversions.....	392
22.6	Array querying system functions	392
22.7	Assertion severity system tasks	394
22.8	Assertion control system tasks.....	395
22.9	Assertion system functions	395
22.10	Random number system functions.....	396
22.11	Program control	396
22.12	Coverage system functions	396
22.13	Enhancements to Verilog system tasks.....	396
22.14	\$readmemb and \$readmemh	398
22.15	\$writememb and \$writememh	398
22.16	File format considerations for multidimensional unpacked arrays.....	399
22.17	System task arguments for multidimensional unpacked arrays.....	400
23.	Compiler directives.....	402
23.1	Introduction.....	402
23.2	`define macros.....	402
23.3	`include	403
23.4	`begin_keywords and `end_keywords	403
24.	Value change dump (VCD) data.....	406
24.1	Introduction.....	406
24.2	VCD extensions	406
25.	Deprecated constructs	408
25.1	Introduction.....	408
25.2	Defparam statements.....	408
25.3	Procedural assign and deassign statements.....	408
26.	Direct programming interface (DPI).....	410
26.1	Overview.....	410
26.2	Two layers of the DPI	411
26.3	Global name space of imported and exported functions.....	412
26.4	Imported tasks and functions	413
26.5	Calling imported functions	419
26.6	Exported functions	421
26.7	Exported tasks.....	421
26.8	Disabling DPI tasks and functions.....	422
27.	SystemVerilog VPI object model	424
27.1	Introduction.....	424
27.2	Module (supersedes 26.6.1 of IEEE Std 1364).....	426
27.3	Interface	427
27.4	Modport	427
27.5	Interface task and function declaration	427
27.6	Program.....	428

27.7	Instance	429
27.8	Instance arrays (supersedes 26.6.2 of IEEE Std 1364)	430
27.9	Scope (supersedes 26.6.3 of IEEE Std 1364)	431
27.10	IO declaration (supersedes 26.6.4 of IEEE Std 1364)	432
27.11	Ports (supersedes 26.6.5 of IEEE Std 1364)	433
27.12	Reference objects	434
27.13	Nets (supersedes 26.6.6 of IEEE Std 1364)	438
27.14	Variables (supersedes 26.6.7 and 26.6.8 of IEEE Std 1364)	441
27.15	Variable select (supersedes 26.6.8 of IEEE Std 1364)	444
27.16	Variable drivers and loads (supersedes 26.6.23 of IEEE Std 1364)	444
27.17	Typespec	445
27.18	Structures and unions	446
27.19	Named events (supersedes 26.6.11 of IEEE Std 1364)	447
27.20	Parameter (supersedes 26.6.12 of IEEE Std 1364)	448
27.21	Class definition	449
27.22	Class variables and class objects	450
27.23	Constraint, constraint ordering, distribution	452
27.24	Constraint expression	453
27.25	Module path, path term (supersedes 26.6.15 of IEEE Std 1364)	453
27.26	Task and function declaration (supersedes 26.6.18 of IEEE Std 1364)	454
27.27	Task and function call (supersedes 26.6.19 of IEEE Std 1364)	455
27.28	Frames (supersedes 26.6.20 of IEEE Std 1364)	456
27.29	Threads	457
27.30	Clocking block	458
27.31	Assertion	459
27.32	Concurrent assertions	460
27.33	Property declaration	460
27.34	Property specification	461
27.35	Sequence declaration	462
27.36	Sequence expression	463
27.37	Multiclock sequence expression	464
27.38	Simple expressions (supersedes 26.6.25 of IEEE Std 1364)	465
27.39	Expressions (supersedes 26.6.26 of IEEE Std 1364)	466
27.40	Atomic statement (supersedes atomic stmt in 26.6.27 of IEEE Std 1364)	468
27.41	Event statement (supersedes event stmt in 26.6.27 of IEEE Std 1364)	469
27.42	Process (supersedes process in 26.6.27 of IEEE Std 1364)	469
27.43	Assignment (supersedes 26.6.28 of IEEE Std 1364)	469
27.44	Event control (supersedes 26.6.30 of IEEE Std 1364)	470
27.45	Waits (supersedes wait in 26.6.32 of IEEE Std 1364)	470
27.46	If, if-else (supersedes 26.6.35 of IEEE Std 1364)	470
27.47	Case, pattern (supersedes 26.6.36 of IEEE Std 1364)	471
27.48	Expect	471
27.49	For (supersedes 26.6.33 of IEEE Std 1364)	472
27.50	Do-while, foreach	472
27.51	Alias statement	473
27.52	Disables (supersedes 26.6.38 of IEEE Std 1364)	473
27.53	Return statement	473
27.54	Attribute (supersedes 26.6.42 of IEEE Std 1364)	474
27.55	Generates (supersedes 26.6.44 of IEEE Std 1364)	475
28.	SystemVerilog assertion API	478
28.1	Requirements	478
28.2	Static information	478
28.3	Dynamic information	479

28.4	Control functions	482
29.	SystemVerilog code coverage control and API	484
29.1	Requirements	484
29.2	SystemVerilog real-time coverage access	484
29.3	FSM recognition	489
29.4	VPI coverage extensions	492
30.	SystemVerilog data read API	496
30.1	Introduction	496
30.2	Requirements	496
30.3	Extensions to VPI enumerations	497
30.4	VPI object type additions	498
30.5	Object model diagrams	500
30.6	Usage extensions to VPI routines	500
30.7	VPI routines added in SystemVerilog	503
30.8	Reading data	504
30.9	Optionally unloading data	504
30.10	Reading data from multiple databases and/or different read library providers	504
30.11	VPI routines extended in SystemVerilog	517
30.12	VPI routines added in SystemVerilog	518
Annex A (normative)	Formal syntax	522
Annex B (normative)	Keywords	562
Annex C (normative)	Std package	564
Annex D (normative)	Linked lists	566
Annex E (normative)	Formal semantics of concurrent assertions	574
Annex F (normative)	DPI C layer	586
Annex G (normative)	Include file svdpi.h	618
Annex H (normative)	Inclusion of foreign language code	628
Annex I (normative)	sv_vpi_user.h	632
Annex J (informative)	Glossary	642
Annex K (informative)	Bibliography	646
Annex L (informative)	List of participants	647
Index		650

Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language

Sponsor

**Design Automation Standards Committee
of the
IEEE Computer Society**

and the

IEEE Standards Association Corporate Advisory Group

Approved 20 March 2006

American National Standards Institute

Approved 8 November 2005

IEEE-SA Standards Board

Abstract: This standard provides a set of extensions to the IEEE 1364™ Verilog® hardware description language (HDL) to aid in the creation and verification of abstract architectural level models. It also includes design specification methods, embedded assertions language, testbench language including coverage and an assertions application programming interface (API), and a direct programming interface (DPI). This standard enables a productivity boost in design and validation and covers design, simulation, validation, and formal assertion-based verification flows.

Keywords: assertions, design automation, design verification, hardware description language, HDL, PLI, programming language interface, SystemVerilog, Verilog, Verilog programming interface, VPI

IEEE Introduction

The purpose of this standard is to provide the electronic design automation (EDA), semiconductor, and system design communities with a well-defined and official IEEE unified hardware design, specification, and verification standard language. The language is designed to coexist and enhance the hardware description languages (HDLs) presently used by designers while providing the capabilities lacking in those languages.

SystemVerilog is a unified hardware design, specification, and verification language that is based on the Accellera SystemVerilog 3.1a extensions to the Verilog HDL [B1]^a, published in 2004. Accellera is a consortium of EDA, semiconductor, and system companies. IEEE Std 1800 enables a productivity boost in design and validation and covers design, simulation, validation, and formal assertion-based verification flows.

SystemVerilog enables the use of a unified language for abstract and detailed specification of the design, specification of assertions, coverage, and testbench verification that is based on manual or automatic methodologies. SystemVerilog offers application programming interfaces (APIs) for coverage and assertions, a vendor-independent API to access proprietary waveform file formats, and a direct programming interface (DPI) to access proprietary functionality. SystemVerilog offers methods that allow designers to continue to use present design languages when necessary to leverage existing designs and intellectual property. This standardization project will provide the VLSI design engineers with a well-defined IEEE standard that meets their requirements in design and validation and enables a step function increase in their productivity. This standardization project will also provide the EDA industry with a standard to which they can adhere and which they can support in order to deliver their solutions in this area.

Notice to users

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: [http://stan-](http://standards.ieee.org/reading/ieee/updates/errata/index.html)

[dards.ieee.org/reading/ieee/updates/errata/index.html](http://standards.ieee.org/reading/ieee/updates/errata/index.html). Users are encouraged to check this URL for errata periodically.

Interpretations

Current interpretations can be accessed at the following URL: <http://standards.ieee.org/reading/ieee/interp/index.html>.

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents or patent applications for which a license may be required to implement an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

^aThe numbers in brackets correspond to the numbers in the bibliography in [Annex K](#).

Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language

1. Overview

1.1 Scope

This standard specifies extensions for a higher level of abstraction for modeling and verification with the Verilog[®] hardware description language (HDL). These additions extend Verilog into the systems space and the verification space. SystemVerilog is built on top of IEEE Std 1364^{TM1} for the Verilog HDL. This standard includes design specification methods, embedded assertions language, testbench language including coverage and assertions application programming interface (API), and a direct programming interface (DPI).

Throughout this standard, the following terms apply:

- *Verilog* refers to IEEE Std 1364 for the Verilog HDL.
- *Verilog-2001* refers to IEEE Std 1364-2001 [B4]² for the Verilog HDL.
- *Verilog-1995* refers to IEEE Std 1364-1995 [B3] for the Verilog HDL.
- *SystemVerilog* refers to the extensions to the Verilog standard (IEEE Std 1364) as defined in this standard.

1.2 Purpose

SystemVerilog is built on top of IEEE Std 1364. SystemVerilog improves the productivity, readability, and reusability of Verilog-based code. The language enhancements in SystemVerilog provide more concise hardware descriptions, while still providing an easy route with existing tools into current hardware implementation flows. The enhancements also provide extensive support for directed and constrained-random testbench development, coverage-driven verification, and assertion-based verification.

¹Information on references can be found in [Clause 2](#).

²The numbers in brackets correspond to the numbers in the bibliography in [Annex K](#).

SystemVerilog adds extended and new constructs to Verilog, including the following:

- Extensions to data types for better encapsulation and compactness of code and for tighter specification
 - C data types: **int**, **typedef**, **struct**, **union**, **enum**
 - Other data types: bounded queues, logic (0, 1, X, Z) and bit (0, 1), tagged unions for safety
 - Dynamic data types: string, classes, dynamic queues, dynamic arrays, associative arrays including automatic memory management freeing users from deallocation issues
 - Dynamic casting and bit-stream casting
 - Automatic/static specification on a per-variable-instance basis
- Extended operators for concise description
 - Wild equality and inequality
 - Built-in methods to extend the language
 - Operator overloading
 - Streaming operators
 - Set membership
- Extended procedural statements
 - Pattern matching on selection statements for use with tagged unions
 - Enhanced loop statements plus the **foreach** statement
 - C-like jump statements: **return**, **break**, **continue**
 - **final** blocks that execute at the end of simulation (inverse of **initial**)
 - Extended event control and sequence events
- Enhanced process control
 - Extensions to **always** blocks to include synthesis consistent simulation semantics
 - Extensions to **fork...join** to model pipelines and for enhanced process control
 - Fine-grain process control
- Enhanced tasks and functions
 - C-like void functions
 - Pass by reference
 - Default arguments
 - Argument binding by name
 - Optional arguments
 - Import/export functions for DPI
- Classes: object-oriented mechanism that provides abstraction, encapsulation, and safe pointer capabilities
- Automated testbench support with random constraints
- Interprocess communication synchronization
 - Semaphores
 - Mailboxes
 - Event extensions, event variables, and event sequencing
- Clarification and extension of the scheduling semantics
- Cycle-based functionality: **clocking** blocks and cycle-based attributes that help reduce development, ease maintainability, and promote reusability
 - Cycle-based signal drives and samples
 - Synchronous samples

- Race-free program context
- Assertion mechanism for verifying design intent and functional coverage intent
 - Property and sequence declarations
 - Assertions and coverage statements with action blocks
- Extended hierarchy support
 - Packages for declaration encapsulation with import for controlled access
 - Compilation-unit scope nested modules and extern modules for separate compilation support
 - Extension of port declarations to support interfaces, events, and variables
 - \$root to provide unambiguous access using hierarchical references
- Interfaces to encapsulate communication and facilitate communication-oriented design
- Functional coverage
- DPI for clean, efficient interoperation with other languages (C provided)
- Assertion API
- Coverage API
- Data read API
- Verilog procedural interface (VPI) extensions for SystemVerilog constructs
- Concurrent assertion formal semantics

1.3 Conventions used in this standard

This standard is organized into clauses, each of which focuses on a specific area of the language. There are subclauses within each clause to discuss individual constructs and concepts. The discussion begins with an introduction and an optional rationale for the construct or the concept, followed by syntax and semantic descriptions, followed by examples and notes.

The terminology conventions used throughout this standard are as follows:

- The word *shall* is used to indicate mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted (*shall* equals *is required to*).
- The word *should* is used to indicate that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited (*should* equals *is recommended that*).
- The word *may* is used to indicate a course of action permissible within the limits of the standard (*may* equals *is permitted to*).
- The word *can* is used for statements of possibility and capability, whether material, physical, or causal (*can* equals *is able to*).

1.4 Syntactic description

The main text uses the following conventions:

- *Italicized* font when a term is being defined
- Constant-width font for examples, file names, and references to constants, especially 0, 1, x, and z values
- **Boldface constant-width** font for Verilog and SystemVerilog keywords, when referring to the actual keyword

The formal syntax of SystemVerilog is described using Backus-Naur Form (BNF). The following conventions are used:

- Lowercase words, some containing embedded underscores, denote syntactic categories. For example:
`module_declaration`
- **Boldface-red** characters denote reserved keywords, operators, and punctuation marks as a required part of the syntax. For example:
`module    =>    ;`
- A vertical bar (|) separates alternative items, unless it appears in boldface-red, in which case it stands for itself. For example:
`unary_operator ::=
 + | - | ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~`
- Square brackets ([]) enclose optional items. For example:
`input_declaration ::= input [ range ] list_of_variables ;`
- Braces ({ }) enclose a repeated item, unless it appears in boldface-red, in which case it stands for itself. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus, the following two rules are equivalent:
`list_of_param_assignments ::= param_assignment { , param_assignment }`
`list_of_param_assignments ::=
 param_assignment
 | list_of_param_assignment , param_assignment`
- If the name of any category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, “*msb_index*” and “*lsb_index*” are equivalent to “index.”

1.5 Use of color in this standard

This standard uses a minimal amount of color to enhance readability. The coloring is not essential and does not affect the accuracy of this standard when viewed in pure black and white. The places where color is used are the following:

- Cross references that are hyperlinked to other portions of this standard are shown in underlined-blue text (hyperlinking works when this standard is viewed interactively as a PDF file).
- Syntactic keywords and tokens in the formal language definitions are shown in **boldface-red text**.
- Some figures use a minimal amount of color to enhance readability.

1.6 Contents of this standard

A synopsis of the clauses and annexes is presented as a quick reference. There are 31 clauses and 11 annexes. All clauses, as well as [Annex A](#) through [Annex I](#), are normative parts of this standard. [Annex J](#) and [Annex K](#) are included for informative purposes only.

[Clause 1](#) describes the contents of this standard and the conventions used in this standard.

[Clause 2](#) lists references to other standards that are required in order to implement this standard.

[Clause 3](#) describes the lexical tokens used in SystemVerilog source text and their conventions. It describes how to specify and interpret the lexical tokens.

[Clause 4](#) describes enhancements to Verilog data objects and data types, including new variable data objects and types, net data type extensions, string types, enumerated types, user-defined types, structures, and unions.

[Clause 5](#) describes SystemVerilog arrays, including packed and unpacked arrays, dynamic arrays, associative arrays, queues, and various array methods.

[Clause 6](#) describes declaring net, variable and constant data, enhanced rules on writing to variables, signal aliasing, and type compatibility.

[Clause 7](#) describes the object-oriented programming capabilities in SystemVerilog. Topics include defining classes, dynamically constructing objects, inheritance and subclasses, data hiding and encapsulation, polymorphism, and parameterized classes.

[Clause 8](#) describes new operators, rules on operations with SystemVerilog data types, operations on arrays, operator methods, and operator overloading.

[Clause 9](#) describes SystemVerilog enhanced simulation scheduling semantics.

[Clause 10](#) describes enhancements to Verilog decision statements and looping constructs, new procedural statements, **final** blocks, statement and block labels, enhanced event types and event control.

[Clause 11](#) describes specialized procedural blocks for modeling combinational logic, latched logic, and sequential logic; enhancements to Verilog continuous assignments; and process control.

[Clause 12](#) describes numerous enhancements to Verilog tasks and functions, plus the syntax for importing functions from a foreign language and exporting tasks and functions to a foreign language.

[Clause 13](#) describes generating random numbers, constraining random number generation, dynamically changing constraints, and seeding random number generators (RNGs) and randomized **case** statement execution.

[Clause 14](#) describes built-in semaphore and mailbox classes and describes enhanced Verilog event type and operators.

[Clause 15](#) defines **clocking** blocks, input and output skews, cycle delays, and default clocking.

[Clause 16](#) describes the testbench program construct, the elimination of testbench race conditions, and program control tasks.

[Clause 17](#) describes immediate assertions, concurrent assertions, properties, sequences, sequence operations, multiclock sequences, clock resolution, and assertion binding.

[Clause 18](#) describes coverage groups, coverage points, cross coverage, coverage options, and coverage methods.

[Clause 19](#) describes packages, compilation units (\$unit), top-level instance (\$root), nested modules, extern modules, enhanced port declarations, time unit and precision, port connection rules, and name spaces.

[Clause 20](#) describes interface syntax, interface ports, modports, interface tasks and functions, parameterized interfaces, virtual interfaces, and accessing objects within interfaces.

[Clause 21](#) describes enhancements to Verilog configurations.

[Clause 22](#) describes several extensions to Verilog system tasks and system functions, including a \$type-name function, \$bits size function, range function, array querying functions, assertion control tasks, random number functions, program control tasks, coverage functions, and enhancements to Verilog system tasks and system functions.

[Clause 23](#) describes extensions to the Verilog ‘define and ‘include directives, and a new directives for controlling keywords compatibility.

[Clause 24](#) describes extensions to the value change dump (VCD) file to support SystemVerilog data objects and data types.

[Clause 25](#) covers the possible deprecation of the Verilog **defparam** statement and the Verilog procedural **assign/deassign** statements.

[Clause 26](#) describes SystemVerilog’s direct interface to foreign languages.

[Clause 27](#) describes enhancements to the VPI object diagrams to support SystemVerilog constructs.

[Clause 28](#) describes the assertion API in SystemVerilog.

[Clause 29](#) describes the coverage API in SystemVerilog.

[Clause 30](#) describes enhancements to the VPI to support accessing design and simulation data from files.

[Annex A](#) defines the formal syntax of SystemVerilog, using BNF.

[Annex B](#) lists the SystemVerilog keywords.

[Annex C](#) describes the system type definitions for mailbox, semaphore, randomize, and process.

[Annex D](#) defines a List package that implements a list data-structure, analogous to the standard template library (STL).

[Annex E](#) describes a formal semantics for SystemVerilog concurrent assertions.

[Annex F](#) defines the C-language layer for the SystemVerilog DPI.

[Annex G](#) defines the standard `svdpi.h` include file for use with SystemVerilog DPI applications.

[Annex H](#) describes common guidelines for the inclusion of foreign language code into a SystemVerilog application.

[Annex I](#) provides a listing of the contents of the `sv_vpi_user.h` file, which extends the Verilog `vpi_user.h` include file.

[Annex J](#) defines terms that are used in this standard.

[Annex K](#) lists reference documents that are related to this standard.

1.7 Examples

Several small SystemVerilog code examples are shown throughout this standard. These examples are informative. They are intended to illustrate the usage of SystemVerilog constructs in a simple context and do not define the full syntax.

1.8 Prerequisites

This standard presupposes a working knowledge of the Verilog HDL and the Verilog programming language interface (PLI). Some clauses of this standard presuppose a working knowledge of the C programming language.

2. Normative references

The following referenced documents are indispensable for the application of this standard. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEEE Std 1364™, IEEE Standard for Verilog Hardware Description Language.^{3, 4, 5}

IEEE Std 754™, IEEE Standard for Binary Floating-Point Arithmetic.

³IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, Piscataway, NJ 08854, USA (<http://standards.ieee.org/>).

⁴This IEEE standards project was not approved by the IEEE-SA Standards Board at the time this publication went to press. For information about obtaining a draft, contact the IEEE.

⁵The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

3. Literal values

3.1 Introduction

NOTE—The lexical conventions for SystemVerilog literal values are extensions of those for Verilog. SystemVerilog adds literal time values, literal array values, literal structures, and enhancements to literal strings.⁶

3.2 Literal value syntax

```
time_literal5 ::= // from A.8.4  
    unsigned_number time_unit  
    | fixed_point_number time_unit  
time_unit ::= s | ms | us | ns | ps | fs | step  
number ::= // from A.8.7  
    integral_number  
    | real_number  
integral_number ::=  
    decimal_number  
    | octal_number  
    | binary_number  
    | hex_number  
decimal_number ::=  
    unsigned_number  
    | [ size ] decimal_base unsigned_number  
    | [ size ] decimal_base x_digit { _ }  
    | [ size ] decimal_base z_digit { _ }  
binary_number ::= [ size ] binary_base binary_value  
octal_number ::= [ size ] octal_base octal_value  
hex_number ::= [ size ] hex_base hex_value  
sign ::= + | -  
size ::= non_zero_unsigned_number  
non_zero_unsigned_number1 ::= non_zero_decimal_digit { _ | decimal_digit }  
real_number1 ::=  
    fixed_point_number  
    | unsigned_number [ . unsigned_number ] exp [ sign ] unsigned_number  
fixed_point_number1 ::= unsigned_number . unsigned_number  
exp ::= e | E  
unsigned_number1 ::= decimal_digit { _ | decimal_digit }  
string_literal ::= " { Any_ASCII_Characters } "// from A.8.8
```

Syntax 3-1—Literal values (excerpt from [Annex A](#))

⁶Notes in text, tables, and figures are given for information only and do not contain requirements needed to implement this standard.

3.3 Integer and logic literals

Literal integer and logic values can be sized or unsized, and they follow the same rules for signedness, truncation, and left-extending as Verilog.

SystemVerilog adds the ability to specify unsized literal single-bit values with a preceding apostrophe ('), but without the base specifier. All bits of the unsized value are set to the value of the specified bit. In a self-determined context, these literals have a width of 1 bit, and the value is treated as unsigned.

```
'0, '1, 'X, 'x, 'Z, 'z    // sets all bits to this value
```

3.4 Real literals

The default type is **real** for fixed-point format (e.g., 1.2), and exponent format (e.g., 2.0e10).

A cast can be used to convert literal **real** values to the **shortreal** type (e.g., `shortreal'(1.2)`). Casting is described in [4.14](#).

3.5 Time literals

Time is written in integer or fixed-point format, followed without a space by a time unit (**fs ps ns us ms s step**). For example:

```
0.1ns
40ps
```

The time literal is interpreted as a **realtime** value scaled to the current time unit and rounded to the current time precision.

NOTE—While s, ms, ns, ps, and fs are the usual SI unit symbols for second, millisecond, nanosecond, picosecond, and femtosecond, due to lack of the Greek letter μ (mu) in coding character sets, 'us' represents the SI unit symbol for microsecond, properly μs.

3.6 String literals

A string literal is enclosed in quotes and has its own data type. Nonprinting and other special characters are preceded with a backslash. SystemVerilog adds the following special string characters:

```
\v vertical tab
\f form feed
\a bell
\x02 hex number
```

A string literal must be contained in a single line unless the new line is immediately preceded by a \ (back slash). In this case, the back slash and the new line are ignored. There is no predefined limit to the length of a string literal.

A string literal can be assigned to an integral type, as in Verilog. If the size differs, it is right justified.

```
byte c1 = "A" ; bit [7:0] d = "\n" ;
bit [0:11] [7:0] c2 = "hello world\n" ;
```

A string literal can be assigned to an unpacked array of bytes. If the size differs, it is left justified.

```
byte c3 [0:12] = "hello world\n" ;
```

Packed and unpacked arrays are discussed in [Clause 5](#). The difference between string literals and array literals is discussed in [3.7](#).

String literals can also be cast to a packed or unpacked array, which shall follow the same rules as assigning a literal string to a packed or unpacked array. Casting is discussed in [4.14](#).

SystemVerilog also includes a **string** data type to which a string literal can be assigned. Variables of type string have arbitrary length; they are dynamically resized to hold any string. String literals are packed arrays (of a width that is a multiple of 8 bits), and they are implicitly converted to the string type when assigned to a string type or used in an expression involving string type operands (see [4.7](#)).

3.7 Array literals

Array literals are syntactically similar to C initializers, but with the replicate operator ({ { } }) allowed.

```
int n[1:2][1:3] = '{ {0,1,2}, {3{4}}};
```

The nesting of braces must follow the number of dimensions, unlike in C. However, replicate operators can be nested. The inner pair of braces in a replication is removed. A replication expression only operates within one dimension.

```
int n[1:2][1:6] = '{2{ {3{4, 5}}}}; // same as '{ {4,5,4,5,4,5}, {4,5,4,5,4,5}}
```

Array literals are array assignment patterns or pattern expressions with constant member expressions (see [8.13.1](#)). An array literal must have a type, which may be either explicitly indicated with a prefix or implicitly indicated by an assignment-like context.

```
typedef int triple [1:3];  
$mydisplay(triple'{0,1,2});
```

Array literals can also use their index or type as a key and use a default key value (see [8.13.1](#)).

```
triple b = '{1:1, default:0}; // indexes 2 and 3 assigned 0
```

3.8 Structure literals

Structure literals are structure assignment patterns or pattern expressions with constant member expressions (see [8.13.2](#)). A structure literal must have a type, which may be either explicitly indicated with a prefix or implicitly indicated by an assignment-like context.

```
typedef struct {int a; shortreal b;} ab;  
ab c;  
c = '{0, 0.0}; // structure literal type determined from  
              // the left-hand context (c)
```

Nested braces should reflect the structure. For example:

```
ab abarr[1:0] = '{ {1, 1.0}, {2, 2.0}};
```

The C-like alternative '{1, 1.0, 2, 2.0}' for the preceding example is not allowed.

Structure literals can also use member name and value or use data type and default value (see [8.13.2](#)):

```
c = '{a:0, b:0.0};' // member name and value for that member
c = '{default:0};' // all elements of structure c are set to 0
d = ab'{int:1, shortreal:1.0};' // data type and default value for all members
// of that type
```

When an array of structures is initialized, the nested braces should reflect the array and the structure. For example:

```
ab abarr[1:0] = '{1, 1.0}, {2, 2.0}';
```

Replicate operators can be used to set the values for the exact number of members. The inner pair of braces in a replication is removed.

```
struct {int X,Y,Z;} XYZ = '{3{1}}';
typedef struct {int a,b[4];} ab_t;
int a,b,c;
ab_t v1[1:0] [2:0];
v1 = '{2{'{3{a,'{2{b,c}}}}}}';

/* expands to '{3{'{a,{2{b,c}}}}}', '{3{{a,'{2{b,c}}}}}' */

/* expands to '{'{'{a,'{2{b,c}}}', 'a,'{2{b,c}}}', 'a,'{2{b,c}}}',
'{'{a,'{2{b,c}}}', 'a,'{2{b,c}}}', 'a,'{2{b,c}}}' } */

/* expands to '{'{'{a,'{b,c,b,c}}', 'a,'{b,c,b,c}}', 'a,'{b,c,b,c}}',
'{'{a,'{b,c,b,c}}', 'a,'{b,c,b,c}}', 'a,'{b,c,b,c}}}' */
```

4. Data types

4.1 Introduction

NOTE—To provide for clear translation to and from C, SystemVerilog supports the C built-in types, with the meaning given by the implementation C compiler. However, to avoid the duplication of **int** and **long** without causing confusion, **int** is 32 bits and **longint** is 64 bits in SystemVerilog. The C **float** type is called **shortreal** in SystemVerilog so that it is not be confused with the Verilog **real** type.

Verilog has data objects that can take on values from a small number of predefined value systems: the set of 4-state logic values, vectors and arrays of logic values, and the set of floating point values. SystemVerilog extends Verilog by introducing some of the data types that conventional programming languages provide, such as enumerations and structures.

In extending the type system, SystemVerilog makes a distinction between an object and its data type. A data type is a set of values and a set of operations that can be performed on those values. Data types can be used to declare data objects or to define user-defined data types that are constructed from other data types.

The Verilog logic system is based on a set of four state values: 0, 1, X, and Z. Although this 4-state logic is fundamental to the language, it does not have a name. SystemVerilog has given this primitive data type a name, **logic**. This new name can be used to declare objects and to construct other data types from the 4-state data type.

The additional strength information associated with bits of a net is not considered part of the data type.

SystemVerilog adds **string**, **chandle**, and **class** data types and enhances the Verilog **event** type.

Verilog provides arbitrary fixed-length arithmetic using 4-state logic. The 4-state type can have bits at X or Z, however. Therefore, it may be less efficient than an array of bits because the operator evaluation must check for X and Z, and twice as much data must be stored. SystemVerilog adds a **bit** data type that can only have bits with 0 or 1 values. See [4.3.2](#) on 2-state data types.

User-defined types are introduced by **typedef** and must be defined before they are used. Data types can also be parameters to modules or interfaces, making them like class templates in object-oriented programming. One routine can be written to reverse the order of elements in any array. Such a routine is impossible in C and in Verilog.

Structures and unions are complicated in C because the tags have a separate name space. SystemVerilog follows the C syntax, but without the optional structure tags.

See also [Clause 5](#) on arrays.

4.2 Data type syntax

```

data_type ::=
    integer_vector_type [ signing ] { packed_dimension }
    | integer_atom_type [ signing ]
    | non_integer_type
    | struct_union [ packed [ signing ] ] { struct_union_member { struct_union_member } }
      { packed_dimension }13
    | enum [ enum_base_type ] { enum_name_declaration { , enum_name_declaration } }
    | string
    | chandle
    | virtual [ interface ] interface_identifier
    | [ class_scope | package_scope ] type_identifier { packed_dimension }
    | class_type
    | event
    | ps_covergroup_identifier
    | type_reference28

enum_base_type ::=
    integer_atom_type [ signing ]
    | integer_vector_type [ signing ] [ packed_dimension ]
    | type_identifier [ packed_dimension ]24

enum_name_declaration ::=
    enum_identifier [ [ integral_number [ : integral_number ] ] ] [ = constant_expression ]

class_scope ::= class_type ::

class_type ::=
    ps_class_identifier [ parameter_value_assignment ]
    { :: class_identifier [ parameter_value_assignment ] }

integer_type ::= integer_vector_type | integer_atom_type
integer_atom_type ::= byte | shortint | int | longint | integer | time
integer_vector_type ::= bit | logic | reg
non_integer_type ::= shortreal | real | realtime
net_type ::= supply0 | supply1 | tri | triand | trior | trireg | tri0 | tri1 | uwire | wire | wand | wor
signing ::= signed | unsigned
simple_type ::= integer_type | non_integer_type | ps_type_identifier
struct_union_member26 ::=
    { attribute_instance } [ random_qualifier ] data_type_or_void list_of_variable_decl_assignments ;
data_type_or_void ::= data_type | void
struct_union ::= struct | union [ tagged ]
type_reference ::=
    type ( expression27 )
    | type ( data_type )

variable_decl_assignment ::=
    variable_identifier { variable_dimension } [ = expression ]
    | dynamic_array_variable_identifier [ ] [ = dynamic_array_new ]
    | class_variable_identifier [ = class_new ]
    | [ covergroup_variable_identifier ] = new [ ( list_of_arguments ) ]16

```

// from [A.2.2.1](#)

// from [A.2.4](#)

Syntax 4-1—Data types (excerpt from [Annex A](#))

4.3 Integer data types

SystemVerilog offers several integer data types, representing a hybrid of both Verilog and C data types, as shown in Table 4-1.

Table 4-1—Integer data types

shortint	2-state SystemVerilog data type, 16-bit signed integer
int	2-state SystemVerilog data type, 32-bit signed integer
longint	2-state SystemVerilog data type, 64-bit signed integer
byte	2-state SystemVerilog data type, 8-bit signed integer or ASCII character
bit	2-state SystemVerilog data type, user-defined vector size
logic	4-state SystemVerilog data type, user-defined vector size
reg	4-state Verilog data type, user-defined vector size
integer	4-state Verilog data type, 32-bit signed integer
time	4-state Verilog data type, 64-bit unsigned integer

4.3.1 Integral types

The term *integral* is used throughout this standard to refer to the data types that can represent a single basic integer data type, **packed array**, **packed struct**, **packed union**, **enum**, or **time**.

The term *simple bit vector type* is used throughout this standard to refer to the data types that can directly represent a one-dimensional packed array of bits. The packed vector types of Verilog are simple bit vector types, as are the integral types with predefined widths, such as **byte**. The SystemVerilog packed structure types and multidimensional packed array types are not simple bit vector types, but each is equivalent (see [6.9.2](#)) to some simple bit vector type, to and from which it can be easily converted.

4.3.2 2-state (two-value) and 4-state (four-value) data types

Types that can have unknown and high-impedance values are called *4-state types*. These are **logic**, **reg**, **integer**, and **time**. The other types do not have unknown values and are called *2-state types*, for example, **bit** and **int**.

The difference between **int** and **integer** is that **int** is a 2-state type and **integer** is a 4-state type. The 4-state values have additional bits that encode the X and Z states. The 2-state data types can simulate faster, take less memory, and are preferred in some design styles.

The Verilog keyword **reg** does not always accurately describe user intent. SystemVerilog adds the keyword **logic** as a more descriptive term. The keywords **logic** and **reg** are equivalent types (see [6.9.2](#) for details on type equivalence).

Automatic type conversions from a smaller number of bits to a larger number of bits involve zero extensions if unsigned or sign extensions if signed. Automatic type conversions from a larger number of bits to a smaller number of bits involve truncations of the most significant bits (MSBs). When a 4-state value is automatically converted to a 2-state value, any unknown or high-impedance bits shall be converted to zeros.

4.3.3 Signed and unsigned data types

Integer types use integer arithmetic and can be signed or unsigned. This affects the meaning of certain operators such as '<', etc.

```
int unsigned ui;
int signed si;
```

The data types **byte**, **shortint**, **int**, **integer**, and **longint** default to **signed**. The data types **bit**, **reg**, and **logic** default to **unsigned**, as do arrays of these types.

The **signed** keyword in the preceding example is part of Verilog. The **unsigned** keyword is a reserved keyword in Verilog, but is not utilized.

See also [Clause 8](#) on operators and expressions.

4.4 Real and shortreal data types

The **real**⁷ data type is from Verilog and is the same as a C **double**. The **shortreal** data type is a SystemVerilog data type and is the same as a C **float**.

4.5 Void data type

The **void** data type represents nonexistent data. This type can be specified as the return type of functions to indicate no return value. This type can also be used for members of tagged unions (see [4.11](#)).

4.6 Chandle data type

The **chandle** data type represents storage for pointers passed using the DPI (see [Clause 26](#)). The size of a value of this data type is platform dependent, but shall be at least large enough to hold a pointer on the machine in which the tool is running.

The syntax to declare a handle is as follows:

```
chandle variable_name ;
```

where `variable_name` is a valid identifier. Chandles shall always be initialized to the value **null**, which has a value of 0 on the C side. Chandles are very restricted in their usage, with the only legal uses being as follows:

- Only the following operators are valid on **chandle** variables:
 - Equality (**==**), inequality (**!=**) with another **chandle** or with **null**
 - Case equality (**===**), case inequality (**!==**) with another **chandle** or with **null** (same semantics as **==** and **!=**)
- Chandles can be tested for a boolean value that shall be 0 if the **chandle** is **null** and 1 otherwise.
- Only the following assignments can be made to a **chandle**:
 - Assignment from another **chandle**
 - Assignment to **null**

⁷The **real** and **shortreal** types are represented as described by IEEE Std 754.

- Chandles can be inserted into associative arrays (refer to [5.9](#)), but the relative ordering of any two entries in such an associative array can vary, even between successive runs of the same tool.
- Chandles can be used within a class.
- Chandles can be passed as arguments to functions or tasks.
- Chandles can be returned from functions.

The use of chandles is restricted as follows:

- Ports shall not have the **chandle** data type.
- Chandles shall not be assigned to variables of any other type.
- Chandles shall not be used as follows:
 - In any expression other than as permitted in this subclause
 - As ports
 - In sensitivity lists or event expressions
 - In continuous assignments
 - In untagged unions
 - In packed types

4.7 String data type

SystemVerilog includes a **string** data type, which is an ordered collection of characters. The length of a **string** variable is the number of characters in the collection. Variables of type **string** are dynamic as their length may vary during simulation. A single character of a **string** variable may be selected for reading or writing by indexing the variable. A single character of a **string** variable is of type **byte**. SystemVerilog also includes a number of special methods to work with strings.

Verilog supports string literals, but only at the lexical level. In Verilog, string literals behave like packed arrays of a width that is a multiple of 8 bits. A string literal assigned to a packed array of an integral variable of a different size is either truncated to the size of the variable or padded with zeroes to the left as necessary.

In SystemVerilog, string literals behave exactly the same as in Verilog. However, SystemVerilog also supports the **string** data type to which a string literal can be assigned. When using the **string** data type instead of an integral variable, strings can be of arbitrary length and no truncation occurs. Literal strings are implicitly converted to the **string** type when assigned to a **string** type or used in an expression involving **string** type operands.

The indices of **string** variables shall be numbered from 0 to $N-1$ (where N is the length of the string) so that index 0 corresponds to the first (leftmost) character of the string and index $N-1$ corresponds to the last (rightmost) character of the string. The **string** variables can take on the special value "", which is the empty string. Indexing an empty string variable shall be an out-of-bounds access.

A **string** shall not contain the special character "\0". Assigning the value 0 to a **string** character shall be ignored.

The syntax to declare a **string** is as follows:

```
string variable_name [= initial_value];
```

where `variable_name` is a valid identifier and the optional `initial_value` can be a string literal or the value "" for an empty string. For example:

```
string myName = "John Smith";
```

If an initial value is not specified in the declaration, the variable is initialized to "", the empty string. An empty string has zero length.

SystemVerilog provides a set of operators that can be used to manipulate combinations of string variables and string literals. The basic operators defined on the **string** data type are listed in [Table 4-2](#).

A string literal can be assigned to a **string** or an integral type. If their size differs, the literal is right justified and either truncated on the left or zero filled on the left, as necessary. For example:

```
byte c = "A";                // assign to c "A"
bit [10:0] a = "\x41";       // assigns to a `b000_0100_0001
bit [1:4][7:0] h = "hello" ; // assigns to h "ello"
```

A **string** or a string literal can be assigned directly to a **string** variable. Integral types can be assigned to a **string** variable, but require a cast. When casting an integral value to a string, the **string** variable shall grow or shrink to accommodate the integral value. If the size (in bits) of the integral value is not a multiple of 8, then the integral value is zero filled on the left.

A string literal assigned to a **string** variable is converted according to the following steps:

- All "\0" characters in the string literal are ignored (i.e., removed from the **string**).
- If the result of the first step is an empty string literal, the **string** is assigned the empty string.
- Otherwise, the **string** is assigned the remaining characters in the string literal.

Casting an integral value to a **string** variable proceeds in the following steps:

- If the size (in bits) of the integral value is not a multiple of 8, the integral value is left extended and filled with zeros until its bit size is a multiple of 8. The extended value is then treated the same as a string literal, where each successive 8 bits represent a character.
- The steps described above for string literal conversion are then applied to the extended value.

For example:

```
string s1 = "hello";          // sets s1 to "hello"
bit [11:0] b = 12'ha41;
string s2 = string'(b);      // sets s2 to 16'h0a41
```

As a second example:

```
typedef reg [15:0] r_t;
r_t r;
integer i = 1;
string b = "";
string a = {"Hi", b};

r = r_t'(a);                // OK
b = string'(r);              // OK
b = "Hi";                   // OK
b = {5{"Hi"}};              // OK
a = {i{"Hi"}};              // OK (non constant replication)
r = {i{"Hi"}};              // invalid (non constant replication)
a = {i{b}};                 // OK
a = {a,b};                  // OK
a = {"Hi",b};               // OK
r = {"H",""};               // yields "H\0". "" is converted to 8'b0
```

```

b = {"H", ""};           // yields "H". "" is the empty string
a[0] = "h";             // OK, same as a[0] = "cough"
a[0] = b;               // invalid, requires a cast
a[1] = "\0";            // ignored, a is unchanged

```

Table 4-2—String operators

Operator	Semantics
<code>Str1 == Str2</code>	<i>Equality.</i> Checks whether the two strings are equal. Result is 1 if they are equal and 0 if they are not. Both strings can be of type string . Or one of them can be a string literal. If both operands are string literals, the operator is the same Verilog equality operator as for integer types.
<code>Str1 != Str2</code>	<i>Inequality.</i> Logical negation of <code>==</code>
<code>Str1 < Str2</code> <code>Str1 <= Str2</code> <code>Str1 > Str2</code> <code>Str1 >= Str2</code>	<i>Comparison.</i> Relational operators return 1 if the corresponding condition is true using the lexicographical ordering of the two strings <code>Str1</code> and <code>Str2</code> . The comparison uses the <code>compare</code> string method. Both operands can be of type string , or one of them can be a string literal.
<code>{Str1, Str2, ..., Strn}</code>	<i>Concatenation.</i> Each operand can be of type string or a string literal (it shall be implicitly converted to type string). If at least one operand is of type string , then the expression evaluates to the concatenated string and is of type string . If all the operands are string literals, then the expression behaves like a Verilog concatenation of integral types; if the result is then used in an expression involving string types, it is implicitly converted to the string type.
<code>{multiplier{Str}}</code>	<i>Replication.</i> <code>Str</code> can be of type string or a string literal. Multiplier must be of integral type and can be nonconstant. If multiplier is nonconstant or <code>Str</code> is of type string , the result is a string containing <i>N</i> concatenated copies of <code>Str</code> , where <i>N</i> is specified by the multiplier. If <code>Str</code> is a literal and the multiplier is constant, the expression behaves like numeric replication in Verilog (if the result is used in another expression involving string types, it is implicitly converted to the string type).
<code>Str[index]</code>	<i>Indexing.</i> Returns a byte, the ASCII code at the given index. Indexes range from 0 to <i>N</i> –1, where <i>N</i> is the number of characters in the string. If given an index out of range, returns 0. Semantically equivalent to <code>Str.getc(index)</code> , in 4.7.3 .
<code>Str.method(...)</code>	The dot (.) operator is used to invoke a specified method on strings.

SystemVerilog also includes a number of special methods to work with strings. These methods use the built-in method notation. These methods are described in 4.7.1 through 4.7.15.

4.7.1 Len()

```
function int len()
```

- `str.len()` returns the length of the string, i.e., the number of characters in the string (excluding any terminating character).
- If `str` is "", then `str.len()` returns 0.

4.7.2 Putc()

```
task putc(int i, byte c)
```

- `str.putc(i, c)` replaces the *i*th character in `str` with the given integral value.

- `putc` does not change the size of `str`: If `i < 0` or `i >= str.len()`, then `str` is unchanged.
- if the second argument to `putc` is zero, the string is unaffected.

The `putc` method assignment `str.putc(j, x)` is semantically equivalent to `str[j] = x`.

4.7.3 Getc()

```
function byte getc(int i)
```

- `str.getc(i)` returns the ASCII code of the `i`th character in `str`.
- If `i < 0` or `i >= str.len()`, then `str.getc(i)` returns 0.

The `getc` method assignment `x = str.getc(j)` is semantically equivalent to `x = str[j]`.

4.7.4 Toupper()

```
function string toupper()
```

- `str.toupper()` returns a string with characters in `str` converted to uppercase.
- `str` is unchanged.

4.7.5 Tolower()

```
function string tolower()
```

- `str.tolower()` returns a string with characters in `str` converted to lowercase.
- `str` is unchanged.

4.7.6 Compare()

```
function int compare(string s)
```

- `str.compare(s)` compares `str` and `s`, as in the ANSI C `strcmp` function with regard to lexical ordering and return value.

See the relational string operators in [4.7, Table 4-2](#).

4.7.7 Icompare()

```
function int icode(s)
```

- `str.icode(s)` compares `str` and `s`, like the ANSI C `strcmp` function with regard to lexical ordering and return value, but the comparison is case insensitive.

4.7.8 Substr()

```
function string substr(int i, int j)
```

- `str.substr(i, j)` returns a new string that is a substring formed by characters in position `i` through `j` of `str`.
- If `i < 0`, `j < i`, or `j >= str.len()`, `substr()` returns "" (the empty string).

4.7.9 Atoi(), atohex(), atooct(), atobin()

```
function integer atoi()  
function integer atohex()  
function integer atooct()  
function integer atobin()
```

- `str.atoi()` returns the integer corresponding to the ASCII decimal representation in `str`. For example:

```
str = "123";  
int i = str.atoi(); // assigns 123 to i.
```

The conversion scans all leading digits and underscore characters (`_`) and stops as soon as it encounters any other character or the end of the string. It returns zero if no digits were encountered. It does not parse the full syntax for integer literals (sign, size, tick, base).

- `atohex` interprets the string as hexadecimal.
- `atooct` interprets the string as octal.
- `atobin` interprets the string as binary.

4.7.10 Atoreal()

```
function real atoreal()
```

- `str.atoreal()` returns the real number corresponding to the ASCII decimal representation in `str`. The conversion parses Verilog syntax for real constants. The scan stops as soon as it encounters any character that does not conform to this syntax or the end of the string. It returns zero if no digits were encountered.

4.7.11 Itoa()

```
task itoa(integer i)
```

- `str.itoa(i)` stores the ASCII decimal representation of `i` into `str` (inverse of `atoi`).

4.7.12 Hextoa()

```
task hextoa(integer i)
```

- `str.hextoa(i)` stores the ASCII hexadecimal representation of `i` into `str` (inverse of `atohex`).

4.7.13 Octtoa()

```
task octtoa(integer i)
```

- `str.octtoa(i)` stores the ASCII octal representation of `i` into `str` (inverse of `atooct`).

4.7.14 Bintoa()

```
task bintoa(integer i)
```

- `str.bintoa(i)` stores the ASCII binary representation of `i` into `str` (inverse of `atobin`).

4.7.15 Realtoa()

```
task realtoa(real r)
```

— `str.realtoa(r)` stores the ASCII real representation of `r` into `str` (inverse of `atoreal`).

4.8 Event data type

The **event** data type is an enhancement over Verilog named events. SystemVerilog events provide a handle to a synchronization object. As in Verilog, event variables can be explicitly triggered and waited for. Furthermore, SystemVerilog events have a persistent triggered state that lasts for the duration of the entire time step. In addition, an event variable can be assigned another event variable or the special value **null**. When assigned another event variable, both event variables refer to the same synchronization object. When assigned **null**, the association between the synchronization object and the event variable is broken. Events can be passed as arguments to tasks.

The syntax to declare an **event** is as follows:

```
event variable_name [= initial_value];
```

Where `variable_name` is a valid identifier and the optional `initial_value` can be another event variable or the special value **null**.

If an initial value is not specified, then the variable is initialized to a new synchronization object.

Examples:

```
event done;                // declare a new event called done
event done_too = done;     // declare done_too as alias to done
event empty = null;        // event variable with no synchronization object
```

Event operations and semantics are discussed in detail in [14.5](#).

4.9 User-defined types

<pre>type_declaration ::= typedef data_type type_identifier { variable_dimension } ; typedef interface_instance_identifier . type_identifier type_identifier ; typedef [enum struct union class] type_identifier ;</pre>	<i>// from 4.2.1.3</i>
--	--

Syntax 4-2—User-defined types (excerpt from [Annex A](#))

The user can define a new type using **typedef**, as in C.

```
typedef int intP;
```

This can then be instantiated as follows:

```
intP a, b;
```

A type can be used before it is defined, provided it is first identified as a type by an empty **typedef**:

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

An empty **typedef** shall not be allowed with enumeration values. Enumeration values must be defined before they are used.

User-defined type identifiers have the same scoping rules as data identifiers, except that hierarchical reference to type identifiers shall not be allowed. References to type identifiers defined within an interface through ports are allowed provided they are locally redefined before being used.

```
interface intf_i;  
    typedef int data_t;  
endinterface  
  
module sub(intf_i p)  
    typedef p.data_t my_data_t;  
    my_data_t data;  
    // type of 'data' will be int when connected to interface above  
endmodule
```

User-defined type names must be used for complex data types in casting (see [4.14](#), below), which only allows simple type names, and as type parameter values when unpacked array types are used.

Sometimes a user-defined type needs to be declared before the contents of the type have been defined. This is of use with user-defined types derived from **enum**, **struct**, **union**, and **class**. For an example, see [7.24](#). Support for this is provided by the following forms for **typedef**:

```
typedef enum type_identifier;  
typedef struct type_identifier;  
typedef union type_identifier;  
typedef class type_identifier;  
typedef type_identifier;
```

While an empty user-defined type declaration is useful for coupled definitions of classes as shown in [7.24](#), it cannot be used for coupled definitions of structures because structures are statically declared and there is no support for pointers to structures.

The last form shows that the type of the user-defined type does not have to be defined in the forward declaration.

The actual type definition of a forward **typedef** declaration shall be resolved within the same local scope or **generate** block. Importing a **typedef** from a package into a local scope can also resolve a type definition.

4.10 Enumerations

```
data_type ::=
    ...
    | enum [ enum_base_type ] { enum_name_declaration { , enum_name_declaration } }
enum_base_type ::=
    integer_atom_type [ signing ]
    | integer_vector_type [ signing ] [ packed_dimension ]
    | type_identifier [ packed_dimension ]24
enum_name_declaration ::=
    enum_identifier [ [ integral_number [ : integral_number ] ] ] [ = constant_expression ]
```

// from [A.2.2.1](#)

Syntax 4-3—Enumerated types (excerpt from [Annex A](#))

An enumerated type declares a set of integral named constants. Enumerated data types provide the capability to abstractly declare strongly typed variables without either a data type or data value(s) and later add the required data type and value(s) for designs that require more definition. Enumerated data types also can be easily referenced or displayed using the enumerated names as opposed to the enumerated values.

In the absence of a data type declaration, the default data type shall be `int`. Any other data type used with enumerated types shall require an explicit data type declaration.

An enumerated type defines a set of named values. In the following example, `light1` and `light2` are defined to be variables of the anonymous (unnamed) enumerated `int` type that includes the three members: `red`, `yellow`, and `green`.

```
enum {red, yellow, green} light1, light2; // anonymous int type
```

An enumerated name with x or z assignments assigned to an `enum` with no explicit data type or an explicit 2-state declaration shall be a syntax error.

```
// Syntax error: IDLE=2'b00, XX=2'bx <ERROR>, S1=2'b01, S2=2'b10
enum {IDLE, XX='x, S1=2'b01, S2=2'b10} state, next;
```

An `enum` declaration of a 4-state type, such as integer, that includes one or more names with x or z assignments shall be permitted.

```
// Correct: IDLE=0, XX='x, S1=1, S2=2
enum integer {IDLE, XX='x, S1='b01, S2='b10} state, next;
```

An unassigned enumerated name that follows an `enum` name with x or z assignments shall be a syntax error.

```
// Syntax error: IDLE=2'b00, XX=2'bx, S1=??, S2=??
enum integer {IDLE, XX='x, S1, S2} state, next;
```

The values can be cast to integer types and increment from an initial value of 0. This can be overridden.

```
enum {bronze=3, silver, gold} medal; // silver=4, gold=5
```

The values can be set for some of the names and not set for other names. The optional value of an `enum` named constant is an elaboration time constant expression (see [6.3](#)) and can include references to parameters, local parameters, genvars, other `enum` named constants, and constant functions of these. Hierarchical names and `const` variables are not allowed. A name without a value is automatically assigned an increment

of the value of the previous name. It shall be an error if incrementing the previous value causes an overflow within the significant digits of the enumerated type.

```
// c is automatically assigned the increment-value of 8
enum {a=3, b=7, c} alphabet;
```

If an automatically incremented value is assigned elsewhere in the same enumeration, this shall be a syntax error.

```
// Syntax error: c and d are both assigned 8
enum {a=0, b=7, c, d=8} alphabet;
```

If the first name is not assigned a value, it is given the initial value of 0.

```
// a=0, b=7, c=8
enum {a, b=7, c} alphabet;
```

Any enumeration encoding value that is outside the representable range of the `enum` shall be an error. If any of the `enum` members are defined with a different sized constant, this shall be a syntax error.

```
// Correct declaration - bronze and gold are unsized
enum bit [3:0] {bronze='h3, silver, gold='h5} medal4;

// Correct declaration - bronze and gold sizes are redundant
enum bit [3:0] {bronze=4'h3, silver, gold=4'h5} medal4;

// Error in the bronze and gold member declarations
enum bit [3:0] {bronze=5'h13, silver, gold=3'h5} medal4;

// Error in c declaration, requires at least 2 bits
enum bit [0:0] {a,b,c} alphabet;
```

Type checking of enumerated types used in assignments, as arguments, and with operators is covered in [4.10.3](#). Like C, there is no overloading of literals; therefore, medal and medal4 cannot be defined in the same scope because they contain the same names.

4.10.1 Defining new data types as enumerated types

A type name can be given so that the same type can be used in many places.

```
typedef enum {NO, YES} boolean;
boolean myvar; // named type
```

4.10.2 Enumerated type ranges

A range of enumeration elements can be specified automatically, via the syntax shown in [Table 4-3](#).

Table 4-3—Enumeration element ranges

name	Associates the next consecutive number with name.
name = C	Associates the constant C to name.
name [N]	Generates N named constants in the sequence: name0, name1,..., nameN-1. N shall be a positive integral number.

Table 4-3—Enumeration element ranges (*continued*)

<code>name[N] = C</code>	Optionally, a constant can be assigned to the generated named constants to associate that constant to the first generated named constant; subsequent generated named constants are associated consecutive values. <i>N</i> shall be a positive integral number.
<code>name [N:M]</code>	Creates a sequence of named constants starting with <code>nameN</code> and incrementing or decrementing until reaching named constant <code>nameM</code> . <i>N</i> and <i>M</i> shall be nonnegative integral numbers.
<code>name[N:M] = C</code>	Optionally, a constant can be assigned to the generated named constants to associate that constant to the first generated named constants; subsequent generated named constants are associated consecutive values. <i>N</i> and <i>M</i> shall be nonnegative integral numbers.

For example:

```
typedef enum { add=10, sub[5], jmp[6:8] } E1;
```

This example defines the enumerated type `E1`, which assigns the number 10 to the enumerated named constant `add`. It also creates the enumerated named constants `sub0`, `sub1`, `sub2`, `sub3`, and `sub4` and assigns them the values 11...15, respectively. Finally, the example creates the enumerated named constants `jmp6`, `jmp7`, and `jmp8` and assigns them the values 16 through 18, respectively.

```
enum { register[2] = 1, register[2:4] = 10 } vr;
```

The example above declares enumerated variable `vr`, which creates the enumerated named constants `register0` and `register1`, which are assigned the values 1 and 2, respectively. Next, it creates the enumerated named constants `register2`, `register3`, and `register4` and assigns them the values 10, 11, and 12.

4.10.3 Type checking

SystemVerilog enumerated types are strongly typed; thus, a variable of type `enum` cannot be directly assigned a value that lies outside the enumeration set unless an explicit cast is used or unless the `enum` variable is a member of a union. This is a powerful type-checking aid that prevents users from accidentally assigning nonexistent values to variables of an enumerated type. The enumeration values can still be used as constants in expressions, and the results can be assigned to any variable of a compatible integral type.

Both the enumeration names and their integer values must be unique. The values can be set to any integral constant value or auto-incremented from an initial value of 0. It is an error to set two values to the same name or to set a value to the same auto-incremented value.

Enumerated variables are type-checked in assignments, arguments, and relational operators. Enumerated variables are auto-cast into integral values, but assignment of arbitrary expressions to an enumerated variable requires an explicit cast.

For example:

```
typedef enum { red, green, blue, yellow, white, black } Colors;
```

This operation assigns a unique number to each of the color identifiers and creates the new data type `Colors`. This type can then be used to create variables of that type.

```
Colors c;  
c = green;  
c = 1;           // Invalid assignment  
if ( 1 == c )    // OK. c is auto-cast to integer
```

In the example above, the value `green` is assigned to the variable `c` of type `Colors`. The second assignment is invalid because of the strict typing rules enforced by enumerated types.

Casting can be used to perform an assignment of a different data type, or an out-of-range value, to an enumerated type. Casting is discussed in [4.10.4](#), [4.14](#), and [4.15](#).

4.10.4 Enumerated types in numerical expressions

Elements of enumerated type variables can be used in numerical expressions. The value used in the expression is the numerical value associated with the enumerated value. For example:

```
typedef enum { red, green, blue, yellow, white, black } Colors;  
  
Colors col;  
integer a, b;  
  
a = blue * 3;  
col = yellow;  
b = col + green;
```

From the previous declaration, `blue` has the numerical value 2. This example assigns `a` the value of 6 (2×3), and it assigns `b` a value of 4 ($3 + 1$).

An **enum** variable or identifier used as part of an expression is automatically cast to the base type of the **enum** declaration (either explicitly or using **int** as the default). A cast shall be required for an expression that is assigned to an **enum** variable where the type of the expression is not equivalent to the enumeration type of the variable.

Casting to an **enum** type shall cause a conversion of the expression to its base type without checking the validity of the value (unless a dynamic cast is used as described in [4.15](#)).

```
typedef enum {Red, Green, Blue} Colors;  
typedef enum {Mo,Tu,We,Th,Fr,Sa,Su} Week;  
Colors C;  
Week W;  
int I;  
  
C = Colors'(C+1);           // C is converted to an integer, then added to  
                           // one, then converted back to a Colors type  
  
C = C + 1; C++; C+=2; C = I; // Illegal because they would all be  
                           // assignments of expressions without a cast  
  
C = Colors'(Su);           // Legal; puts an out of range value into C  
  
I = C + W;                 // Legal; C and W are automatically cast to int
```

SystemVerilog includes a set of specialized methods to enable iterating over the values of enumerated types.

4.10.4.1 First()

The prototype for the `first()` method is as follows:

```
function enum first();
```

The `first()` method returns the value of the first member of the enumeration.

4.10.4.2 Last()

The prototype for the `last()` method is as follows:

```
function enum last();
```

The `last()` method returns the value of the last member of the enumeration.

4.10.4.3 Next()

The prototype for the `next()` method is as follows:

```
function enum next( int unsigned N = 1 );
```

The `next()` method returns the Nth next enumeration value (default is the next one) starting from the current value of the given variable. A wrap to the start of the enumeration occurs when the end of the enumeration is reached. If the given value is not a member of the enumeration, the `next()` method returns the default initial value for the enumeration.

4.10.4.4 Prev()

The prototype for the `prev()` method is as follows:

```
function enum prev( int unsigned N = 1 );
```

The `prev()` method returns the Nth previous enumeration value (default is the previous one) starting from the current value of the given variable. A wrap to the end of the enumeration occurs when the start of the enumeration is reached. If the given value is not a member of the enumeration, the `prev()` method returns the default initial value for the enumeration.

4.10.4.5 Num()

The prototype for the `num()` method is as follows:

```
function int num();
```

The `num()` method returns the number of elements in the given enumeration.

4.10.4.6 Name()

The prototype for the `name()` method is as follows:

```
function string name();
```

The `name()` method returns the string representation of the given enumeration value. If the given value is not a member of the enumeration, the `name()` method returns the empty string.

4.10.4.7 Using enumerated type methods

The following code fragment shows how to display the name and value of all the members of an enumeration:

```
typedef enum { red, green, blue, yellow } Colors;
Colors c = c.first;
forever begin
    $display( "%s : %d\n", c.name, c );
    if( c == c.last ) break;
    c = c.next;
end
```

4.11 Structures and unions

```
data_type ::= // from A.2.2.1
    ...
    | struct_union [ packed [ signing ] ] { struct_union_member { struct_union_member } }
        { packed_dimension }13
struct_union_member26 ::=
    { attribute_instance } [random_qualifier] data_type_or_void list_of_variable_decl_assignments ;
data_type_or_void ::= data_type | void
struct_union ::= struct | union [ tagged ]
```

Syntax 4-4—Structures and unions (excerpt from [Annex A](#))

Structure and union declarations follow the C syntax, but without the optional structure tags before the ‘{’.

```
struct { bit [7:0] opcode; bit [23:0] addr; } IR; // anonymous structure
// defines variable IR

IR.opcode = 1; // set field in IR.
```

Some additional examples of declaring structure and unions are as follows:

```
typedef struct {
    bit [7:0] opcode;
    bit [23:0] addr;
} instruction; // named structure type
instruction IR; // define variable

typedef union { int i; shortreal f; } num; // named union type
num n;
n.f = 0.0; // set n in floating point format

typedef struct {
    bit isfloat;
    union { int i; shortreal f; } n; // anonymous type
} tagged_st; // named structure

tagged_st a[9:0]; // array of structures
```

A structure can be assigned as a whole and passed to or from a function or task as a whole.

Members of a structure data type may be assigned individual default member values by using an initial assignment with the declaration of each member. The assigned expression must be a constant expression.

An example of initializing member of a structure type is as follows:

```
typedef struct {
    int addr = 1 + constant;
    int crc;
    byte data [4] = '{4' {1}};
} packet1;
```

The structure can then be instantiated.

```
packet1 p1;    // initialization defined by the typedef.
               // p1.crc will use the default value for an int
```

If an explicit initial value expression is used with the declaration of a variable, the initial assignment expression within the structure data type shall be ignored. Subclause [3.8](#) discusses assigning initial values to a structure. For example:

```
packet1 pi = '{1,2,'{2,3,4,5}}; //suppresses the typedef initialization
```

Members of unpacked structures containing a union as well as members of packed structures shall not be assigned individual default member values.

The initial assignment expression within a data type shall be ignored when using a data type to declare a net (see [6.5](#)).

A packed structure is a mechanism for subdividing a vector into subfields that can be conveniently accessed as members. Consequently, a packed structure consists of bit fields, which are packed together in memory without gaps. An unpacked structure has an implementation-dependent packing, normally matching the C compiler. A packed structure differs from an unpacked structure in that, when a packed structure appears as a primary, it is treated as a single vector.

One special provision exists in order to simplify the use of unpacked unions: if an unpacked union contains several unpacked structures that share a common initial sequence (see below) and if the unpacked union object currently contains one of these structures, it is permitted to inspect the common initial part of any of them anywhere that a declaration of the complete type of the union is visible. Two structures share a common initial sequence if corresponding members have equivalent types for a sequence of one or more initial members.

Like a packed array, a packed structure can be used as a whole with arithmetic and logical operators. The first member specified is the most significant and subsequent members follow in decreasing significance. The structures are declared using the **packed** keyword, which can be followed by the **signed** or **unsigned** keyword, according to the desired arithmetic behavior. The default is unsigned:

```
struct packed signed {
    int a;
    shortint b;
    byte c;
    bit [7:0] d;
} pack1; // signed, 2-state

struct packed unsigned {
    time a;
    integer b;
```

```
    logic [31:0] c;  
} pack2; // unsigned, 4-state
```

If all data types within a packed structure are 2-state, the structure as a whole is treated as a 2-state vector.

If any data type within a packed structure is 4-state, the structure as a whole is treated as a 4-state vector. If there are also 2-state members in the structure, there is an implicit conversion from 4-state to 2-state when reading those members and from 2-state to 4-state when writing them.

One or more bits of a packed structure can be selected as if it were a packed array, assuming an [n-1:0] numbering:

```
pack1 [15:8] // c
```

Noninteger data types, such as **real** and **shortreal**, are not allowed in packed structures or unions. Neither are unpacked arrays.

A packed structure can be used with a **typedef**.

```
typedef struct packed { // default unsigned  
    bit [3:0] GFC;  
    bit [7:0] VPI;  
    bit [11:0] VCI;  
    bit CLP;  
    bit [3:0] PT ;  
    bit [7:0] HEC;  
    bit [47:0] [7:0] Payload;  
    bit [2:0] filler;  
} s_atmcell;
```

A packed union shall contain members that must be packed structures, or packed arrays or integer data types all of the same size (in contrast to an unpacked union, where the members can be different sizes). This ensures that a union member that was written as another member can be read back. A packed union can also be used as a whole with arithmetic and logical operators, and its behavior is determined by the **signed** or **unsigned** keyword, the latter being the default. One or more bits of a packed union can be selected as if it were a packed array, assuming an [n-1:0] numbering.

If a packed union contains a 2-state member and a 4-state member, the entire union is 4-state. There is an implicit conversion from 4-state to 2-state when reading and from 2-state to 4-state when writing the 2-state member.

For example, a union can be accessible with different access widths:

```
typedef union packed { // default unsigned  
    s_atmcell acell;  
    bit [423:0] bit_slice;  
    bit [52:0] [7:0] byte_slice;  
} u_atmcell;  
  
u_atmcell u1;  
byte b; bit [3:0] nib;  
b = u1.bit_slice[415:408]; // same as b = u1.byte_slice[51];  
nib = u1.bit_slice [423:420]; // same as nib = u1.acell.GFC;
```

With packed unions, writing one member and reading another is independent of the byte ordering of the machine, unlike an unpacked union of unpacked structures, which are C-compatible and have members in ascending address order.

The signing of unpacked structures is not allowed. The following declaration would be considered illegal:

```
typedef struct signed {
    int f1 ;
    logic f2 ;
} sIllegalSignedUnpackedStructType;    // illegal declaration
```

If no initial value is specified in the declaration of a variable of an unpacked union type, then the variable shall be initialized to the default initial value for variables of the type of the first member in declaration order of the union type.

The qualifier **tagged** in a union declares it as a tagged union, which is a type-checked union. An ordinary (untagged) union can be updated using a value of one member type and read as a value of another member type, which is a potential type loophole. A tagged union stores both the member value and a tag, i.e., additional bits representing the current member name. The tag and value can only be updated together consistently using a statically type-checked tagged union expression (see [8.14](#)). The member value can only be read with a type that is consistent with the current tag value (i.e., member name). Thus, it is impossible to store a value of one type and (mis)interpret the bits as another type.

Dynamic types and chandles shall not be used in untagged unions, but may be used in tagged unions.

In addition to type safety, the use of member names as tags also makes code simpler and smaller than code that has to track unions with explicit tags. Tagged unions can also be used with pattern matching (see [10.4](#)), which improves readability even further.

In tagged unions, members can be declared with type **void**, when all the information is in the tag itself, as in the following example of an integer together with a valid bit:

```
typedef union tagged {
    void Invalid;
    int Valid;
} VInt;
```

A value of **VInt** type is either **Invalid** (and contains nothing) or **Valid** (and contains an **int**). Subclause [8.14](#) describes how to construct values of this type and also describes how it is impossible to read an integer out of a **VInt** value that currently has the **Invalid** tag.

For example:

```
typedef union tagged {
    struct {
        bit [4:0] reg1, reg2, regd;
        Add;
    }
    union tagged {
        bit [9:0] JmpU;
        struct {
            bit [1:0] cc;
            bit [9:0] addr;
        } JmpC;
        Jmp;
        Instr;
    }
}
```

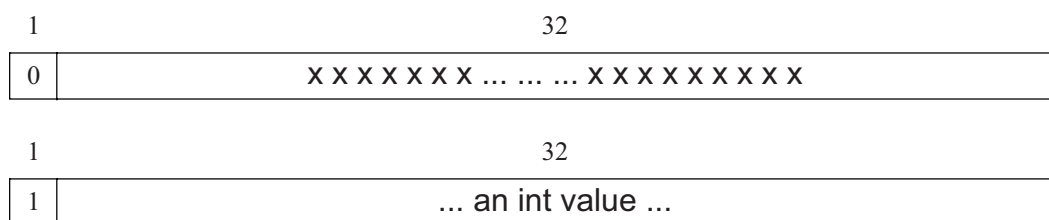
A value of `Instr` type is either an `Add` instruction, in which case it contains three 5-bit register fields, or it is a `Jmp` instruction. In the latter case, it is either an unconditional jump, in which case it contains a 10-bit destination address, or it is a conditional jump, in which case it contains a 2-bit condition-code register field and a 10-bit destination address. Subclause 8.14 describes how to construct values of `Instr` type and describes how, in order to read the `cc` field, for example, the instruction must have opcode `Jmp` and sub-opcode `JmpC`.

When the **packed** qualifier is used on a tagged union, all the members must have packed types, but they do not have to be of the same size. The (standard) representation for a packed tagged union is the following:

- The size is always equal to the number of bits needed to represent the tag plus the maximum of the sizes of the members.
- The size of the tag is the minimum number of bits needed to code for all the member names (e.g., 5 to 8 members would need 3 tag bits).
- The tag bits are always left-justified (i.e., towards the MSBs).
- For each member, the member bits are always right-justified [i.e., towards the least significant bits (LSBs)].
- The bits between the tag bits and the member bits are undefined. In the extreme case of a void member, only the tag is significant and all the remaining bits are undefined.

The representation scheme is applied recursively to any nested tagged unions.

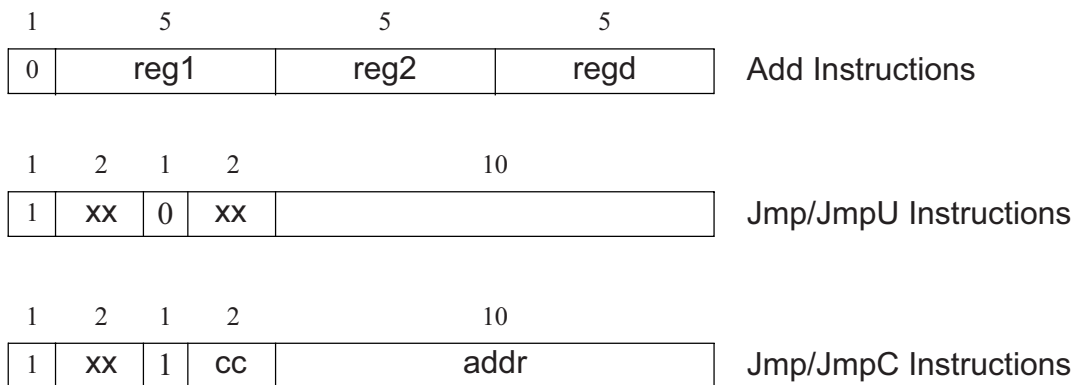
For example: If the `VInt` type definition had the **packed** qualifier, `Invalid` and `Valid` values will have the layouts shown in [Figure 4-1](#), respectively.



tag is 0 for Invalid, 1 for Valid

Figure 4-1—VInt type with packed qualifier

For example: If the `Instr` type had the **packed** qualifier, its values will have the layouts shown in [Figure 4-2](#).



Outer tag is 0 for Add, 1 for Jmp
Inner tag is 0 for JmpU, 1 for JmpC

Figure 4-2—Instr type with packed qualifier

4.12 Class

A class is a collection of data and a set of subroutines that operate on that data. The data in a class are referred to as *class properties*, and the subroutines of the class are called *methods*. The class properties and methods, taken together, define the contents and capabilities of a class instance or object.

```
class_declaration ::=
    [ virtual ] class [ lifetime ] class_identifier [ parameter_port_list ]
        [ extends class_type [ ( list_of_arguments ) ] ];
    { class_item }
endclass [ : class_identifier]                                     // from A.1.2
```

Syntax 4-5—Classes (excerpt from [Annex A](#))

The object-oriented class extension allows objects to be created and destroyed dynamically. Class instances, or objects, can be passed around via object handles, which add a safe-pointer capability to the language. An object can be declared as an argument with direction **input**, **output**, **inout**, or **ref**. In each case, the argument copied is the object handle, not the contents of the object.

A class is declared using the **class...endclass** keywords. For example:

```
class Packet;
    int address;           // Properties are address, data, and crc
    bit [63:0] data;
    shortint crc;
    Packet next;          // Handle to another Packet

    function new();        // Methods are send and new
    function bit send();
endclass : Packet
```

Any data type can be declared as a class member. Classes are discussed in more detail in [Clause 7](#).

4.13 Singular and aggregate types

Data types are categorized as either *singular* or *aggregate*. A singular type shall be any data type except an unpacked structure, unpacked union, or unpacked array (see [Clause 5](#) on arrays). An aggregate type shall be any unpacked structure, unpacked union, or unpacked array data type. A singular variable or expression represents a single value, symbols, or handle. Aggregate expressions and variables represent a set or collection of singular values. Integral types are always singular even though they can be sliced into multiple singular values.

These categories are defined so that operators and functions can simply refer to these data types as a collective group. For example, some functions recursively descend into an aggregate variable until reaching a singular value and then perform an operation on each singular value.

Although a class, as described in [Clause 7](#), is a type, there are no variables or expressions of class type directly, only class object handles that are singular. Therefore, classes need not be categorized in this manner (see [Clause 7](#) on classes).

4.14 Casting

```
constant_cast ::= casting_type ' ( constant_expression ) // from A.8.4  
cast ::= casting_type ' ( expression )  
  
casting_type ::= simple_type | constant_primary | signing // from A.2.2.1  
simple_type ::= integer_type | non_integer_type | ps_type_identifier | ps_parameter_identifier
```

Syntax 4-6—Casting (excerpt from [Annex A](#))

A data type can be changed by using a cast (') operation. In a static cast, the expression to be cast shall be enclosed in parentheses that are prefixed with the casting type and an apostrophe. If the expression is assignment compatible with the casting type, then the cast shall return the value that a variable of the casting type would hold after being assigned the expression. If the expression is not assignment compatible with the casting type, then if the casting type is an enumerated type, the behavior shall be as described as in [4.15](#), and if the casting type is a bit-stream type, the behavior shall be as described in [4.16](#).

```
int' (2.0 * 3.0)  
shortint' { {8'hFA, 8'hCE} }
```

A positive decimal number as a data type means a number of bits to change the size.

```
17' (x - 2)
```

The signedness can also be changed.

```
signed' (x)
```

The expression inside the cast must be an integral value when changing the size or signing. When changing the size, the signing shall pass through unchanged and the result type shall be a one-dimensional packed array with a right bound of zero. When changing the signing, the type of the expression to be cast shall pass through unchanged, except for the signing.