



Verilog-AMS

Language Reference Manual

**Analog & Mixed-Signal Extensions
to
Verilog HDL**

**Version 1.4
Cleanup Committee Working Draft**

July 13, 1999

Open Verilog International

Copyright© 1996-1999 by Open Verilog International, Inc. All rights reserved.

No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means --
- graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and
retrieval systems --- without the prior written approval of Open Verilog International.

Additional copies of this manual may be purchased by contacting Open Verilog International at the address
shown below.

Notices

The information contained in this draft manual represents the definition of the Verilog-AMS hardware description language as proposed by OVI (Analog and Mixed-Signal TSC) as of July 1999. Open Verilog International makes no warranties whatsoever with respect to the completeness, accuracy, or applicability of the information in this draft manual to a user's requirements. This language is not yet fully defined and is subject to change. It is suitable for learning how to do analog and mixed-signal modeling and as a vehicle for providing feedback to the standards committee. Verilog-AMS should not be used for production design and development.

Open Verilog International reserves the right to make changes to the Verilog-AMS hardware description language and this manual at any time without notice.

Open Verilog International does not endorse any particular simulator or other CAE tool that is based on the Verilog-AMS hardware description language.

Suggestions for improvements to the Verilog hardware description language and/or to this manual are welcome. They should be sent to the address below.

Information about Open Verilog International and membership enrollment can be obtained by inquiring at the address below.

Published as: Verilog-AMS Language Reference Manual
 Version 1.4, July 1999.

Published by: Open Verilog International
 15466 Los Gatos Blvd., #109071
 Los Gatos, CA 95032
 Phone: (408) 358-9510
 Fax: (408) 358-3910

Printed in the United States of America.

Verilog® is a registered trademark of Cadence Design Systems, Inc.

The following people contributed to the creation, editing, and review of this document.

Ramana Aisola	Motorola	aisola@analog-dse.sps.mot.com
Graham Bell	Viewlogic	gbell@viewlogic.com
William Bell	Veribest	wrbell@veribest.com
Kevin Cameron	Consultant	
Ed Chang	Consultant	edcheng@best.com
Raphael Dorado	Apteq	raf@apteq.com
John Downey	Viewlogic	jdowney@viewlogic.com
Dan FitzPatrick	Apteq	dkf@apteq.com
Vassilios Gerousis	Motorola	gerousis@udslaz.sps.mot.com
Ian Getreu	Analogy	iang@analogy.com
Kim Hailey	Consultant	kim@santolina.com
Steve Hamm	Motorola	hamm@adtx.sps.mot.com
Graham Helwig	Motorola	ghelwig@asc.corp.mot.com
William Hobson	Cadence	wmh@cadence.com
Ken Kundert	Cadence	kundert@cadence.com
Oskar Leuthold	GEC Plessy	leuthold@sv.gpsemi.com
S. Peter Liebmann	Antrim	SPL@antrim.com
Ira Miller	Motorola	rjx340@email.sps.mot.com
Tom Reeder	Viewlogic	treeder@viewlogic.com
Steffen Rochel	Simplex	steffen@simplex.com
Jon Sanders	Cadence	jons@cadence.com
James Spoto	Rockwell	james.spoto@rss.rockwell.com
Richard Trihy	Cadence	trihy@cadence.com
Yatin Trivedi	Seva Technologies	trivedi@seva.com
Frank Weiler	Avant!	frankw@avanticorp.com
Alex Zamfirescu	Veribest	a.zamfirescu@ieee.org
Amir Zarkesh	Transcendent	amir@tdes.com

Table of Contents

1	Verilog-AMS Overview	1-1
1.1	Overview	1-1
1.2	Mixed-signal language features	1-2
1.3	Systems	1-3
1.3.1	Conservative systems	1-3
1.3.2	Kirchhoff's laws	1-5
1.3.3	Signal-flow systems	1-6
1.3.4	Mixed conservative/signal flow systems	1-6
1.3.5	Natures, disciplines and nodes	1-9
1.4	Conventions used in this document	1-9
1.5	Contents	1-10
2	Lexical Conventions.....	2-1
2.1	Lexical tokens	2-1
2.2	White space	2-1
2.3	Comments	2-1
2.4	Operators	2-2
2.5	Numbers	2-2
2.5.1	Integer constants	2-3
2.5.2	Real constants	2-3
2.5.3	Scale factors for real constants	2-4
2.6	Identifiers, keywords, and system names	2-5
2.6.1	Escaped identifiers	2-5
2.6.2	Keywords	2-6
2.6.3	System tasks and functions	2-7
2.6.4	Compiler directives	2-8
3	Data Types	3-1
3.1	Integer and real datatypes	3-1
3.2	Parameters	3-2
3.2.1	Type Specification	3-4
3.2.2	Value Range Specification	3-5
3.2.3	Parameter Arrays	3-5
3.3	Genvars	3-6
3.4	Nodes	3-6
3.4.1	Natures	3-7
3.4.2	Disciplines	3-10
3.4.3	Node Declaration	3-14
3.4.4	Implicit Nodes	3-15
3.5	Default Discipline	3-15

3.5.1	Discipline Precedence	3-17
3.6	Node Compatibility	3-17
3.7	Branches	3-20
3.7.1	Branch Declaration	3-20
3.7.2	Accessing Node and Branch Signals	3-21
3.7.3	Accessing Attributes	3-22
3.8	Namespace	3-23
3.8.1	Nature and Discipline	3-23
3.8.2	Access Functions	3-23
3.8.3	Node	3-23
3.8.4	Branch	3-23

4 Expressions 4-1

4.1	Operators	4-1
4.1.1	Operators with real operands	4-2
4.1.2	Binary operator precedence	4-3
4.1.3	Expression evaluation order	4-4
4.1.4	Arithmetic operators	4-4
4.1.5	Relational operators	4-5
4.1.6	Equality operators	4-6
4.1.7	Logical operators	4-6
4.1.8	Bit-wise operators	4-6
4.1.9	Shift operators	4-7
4.1.10	Conditional operator	4-8
4.1.11	Event or	4-8
4.1.12	Concatenations	4-8
4.2	Built-In Mathematical Functions	4-9
4.2.1	Standard Mathematical Functions	4-9
4.2.2	Transcendental Functions	4-10
4.2.3	Error Handling	4-11
4.3	Signal Access Functions	4-11
4.4	Analog Operators	4-12
4.4.1	Restrictions on analog operators	4-12
4.4.2	Vector or Array Arguments to Analog Operators	4-13
4.4.3	Analog Operators and Equations	4-13
4.4.4	Time Derivative Operator	4-14
4.4.5	Time Integral Operator	4-14
4.4.6	Circular Integrator Operator	4-15
4.4.7	Delay Operator	4-17
4.4.8	Transition Filter	4-18
4.4.9	Slew Filter	4-22
4.4.10	Last_Crossing Function	4-23
4.4.11	Laplace Transform Filters	4-23
4.4.12	Z-Transform Filters	4-26

4.4.13	Limited Exponential	4-29
4.4.14	Constant vs Dynamic Arguments	4-29
4.5	Analysis Dependent Functions	4-30
4.5.1	Analysis	4-30
4.5.2	AC Stimulus	4-32
4.5.3	Noise	4-32
4.6	User defined functions	4-34
4.6.1	Defining an analog function	4-34
4.6.2	Returning a value from an analog function	4-35
4.6.3	Calling an analog function	4-36
5	Signals	5-1
5.1	Analog Signals	5-1
5.1.1	Access Functions	5-1
5.1.2	Probes and Sources	5-2
5.1.3	Examples	5-3
5.1.4	Port Branches	5-6
5.1.5	Switch Branches	5-6
5.1.6	Unassigned Sources	5-7
5.2	Signal Access for Vector Branches	5-8
5.3	Contribution statements	5-10
5.3.1	Branch Contribution Statements	5-10
5.3.2	Indirect Branch Assignments	5-13
6	Analog Behavior	6-1
6.1	Analog procedural block	6-1
6.2	Block statements	6-2
6.2.1	Block names	6-3
6.3	Procedural assignments	6-3
6.4	Conditional statement	6-4
6.4.1	Analog Conditional Statements	6-5
6.5	Case statement	6-5
6.5.1	Analog case statements	6-6
6.5.2	Constant expression in case statement	6-7
6.6	Looping statements	6-7
6.6.1	Repeat and while statements	6-7
6.6.2	For statements	6-8
6.7	Events	6-9
6.7.1	Event detection	6-10
6.7.2	Event OR operator	6-10
6.7.3	Event Triggered Statements	6-11
6.7.4	Global events	6-11
6.7.5	Monitored events	6-13
6.8	Announcing Discontinuity	6-15

6.9	Time related functions	6-17
6.9.1	Bounding the time step	6-17
7	Mixed-Signal	7-1
7.1	Fundamentals	7-1
7.1.1	Domains	7-1
7.1.2	Contexts	7-1
7.1.3	Analog and Digital Disciplines	7-1
7.1.4	Nets, Nodes, and Signals	7-1
7.2	Discipline Resolution and Connection Module Insertion	7-1
7.2.1	Discipline Resolution	7-1
7.2.2	Resolution of Discrete-time Disciplines	7-1
7.3	Behavioral Interaction	7-2
7.3.1	Synchronous	7-4
7.3.2	Asynchronous	7-4
7.4	Connect Statement and Connection Module Semantics	7-4
7.5	Automatic Insertion of Connection Modules	7-6
7.5.1	Connection Module Selection and Insertion	7-7
7.5.2	Internal Representation, Driver Receiver Segregation	7-13
7.5.3	Rules for Driver/Receiver Segregation and Connection Module Selection and Insertion	7-16
7.5.4	Instance Names for Auto-Inserted Instances	7-18
7.6	Back Annotation of Parasitics	7-19
7.6.1	Port Names for Verilog Built-in Primitives	7-21
7.7	Driver Access Functions	7-21
7.7.1	driver_update event	7-22
7.7.2	driver_count function	7-22
7.7.3	driver_active function	7-22
7.7.4	driver_local function	7-23
7.7.5	driver_state function	7-23
7.7.6	driver_strength function	7-24
7.7.7	driver_delay function	7-24
7.7.8	driver_next_state function	7-25
7.7.9	driver_next_strength function	7-25
8	Hierarchical Structures	8-1
8.1	Modules	8-1
8.1.1	Top-level modules	8-2
8.1.2	Module instantiation	8-2
8.2	Overriding module parameter values	8-5
8.2.1	Defparam statement	8-5
8.2.2	Module instance parameter value assignment by order	8-6
8.2.3	Module instance parameter value assignment by name	8-7
8.2.4	Parameter override precedence	8-7

8.2.5	Parameter dependence	8-8
8.3	Ports	8-8
8.3.1	Port association	8-8
8.3.2	Port declarations	8-9
8.3.3	Real valued ports	8-10
8.3.4	Connecting module ports by ordered list	8-11
8.3.5	Connecting module ports by name	8-11
8.3.6	Port connection rules	8-12
8.3.7	Inheriting Port Natures	8-13
8.3.8	Multi-disciplinary example	8-14
8.4	Hierarchical names	8-14
8.5	Scope rules	8-16
22	Using VPI routines	22-1
22.1	The VPI interface	22-1
22.1.1	VPI callbacks	22-1
22.1.2	VPI access to Verilog HDL objects and simulation objects	22-2
22.1.3	Error handling	22-2
22.2	VPI object classifications	22-2
22.2.1	Accessing object relationships and properties	22-2
22.2.2	Delays and values	22-3
22.3	List of VPI routines by functional category	22-3
22.4	Key to object model diagrams	22-7
22.4.1	Diagram key for objects and classes	22-7
22.4.2	Diagram key for accessing properties	22-8
22.4.3	Diagram key for traversing relationships	22-9
22.5	Object data model diagrams	22-10
22.5.3	Nature, Discipline	22-13
22.5.4	Scope, task, function, IO declaration	22-14
22.5.5	Ports	22-15
22.5.6	Nodes	22-16
22.5.7	Branches	22-17
22.5.8	Nets	22-18
22.5.9	Regs	22-19
22.5.10	Variables, named event	22-20
22.5.11	Memory	22-21
22.5.12	Parameter, specparam	22-22
22.5.13	Primitive, prim term	22-23
22.5.14	UDP	22-24
22.5.15	Module path, timing check, intermodule path	22-25
22.5.16	Task and function call	22-26
22.5.17	Continuous assignment	22-27
22.5.18	Simple expressions	22-28
22.5.19	Expressions	22-29

22.5.20	Contribs	22-30
22.5.21	Process, block, statement, event statement	22-31
22.5.22	Assignment, delay control, event control, repeat control	22-32
22.5.23	While, repeat, wait, for, forever	22-33
22.5.24	If, if-else, case	22-34
22.5.25	Assign statement, deassign, force, release, disable	22-35
22.5.26	Callback, time queue	22-36

24 VPI routine definitions 24-1

24.1	vpi_chk_error()	24-2
24.2	vpi_compare_objects()	24-3
24.3	vpi_decl_deriv()	24-4
24.4	vpi_decl_discontinuity()	24-5
24.5	vpi_free_object()	24-6
24.6	vpi_get()	24-7
24.7	vpi_get_cb_info()	24-8
24.8	vpi_get_continuous_delta()	24-9
24.9	vpi_get_continuous_time()	24-10
24.10	vpi_get_delays()	24-11
24.11	vpi_get_str()	24-14
24.12	vpi_get_systf_info()	24-15
24.13	vpi_get_time()	24-16
24.14	vpi_get_value()	24-17
24.15	vpi_get_vlog_info()	24-22
24.16	vpi_get_real()	24-23
24.17	vpi_handle()	24-24
24.18	vpi_handle_by_index()	24-25
24.19	vpi_handle_by_name()	24-26
24.20	vpi_handle_multi()	24-27
24.21	vpi_iterate()	24-28
24.22	vpi_mcd_close()	24-29
24.23	vpi_mcd_name()	24-30
24.24	vpi_mcd_open()	24-31
24.25	vpi_mcd_printf()	24-32
24.26	vpi_printf()	24-33
24.27	vpi_put_delays()	24-34
24.28	vpi_put_deriv()	24-37
24.29	vpi_put_value()	24-38
24.30	vpi_register_cb()	24-40
24.30.1	Simulation-event-related callbacks	24-41
24.30.2	Simulation-time-related callbacks	24-42
24.30.3	Simulator action and feature related callbacks	24-43
24.31	vpi_register_acb()	24-45
24.31.1	Simulation-event-related callbacks	24-46

24.31.2	SSimulator action and feature related callbacks	24-46
24.32	vpi_register_systf()	24-48
24.32.1	System task and function callbacks	24-48
24.32.2	Initializing VPI system task/function callbacks	24-49
24.33	vpi_remove_cb()	24-51
24.34	vpi_scan()	24-52
A	Scheduling Semantics	A-1
A.1	Analog Simulation Cycle	A-1
A.1.1	Nodal Analysis	A-1
A.1.2	Transient Analysis	A-2
A.1.3	Convergence	A-3
A.2	Mixed-Signal Simualtion Cycle	A-4
A.2.1	Circuit Initialization	A-5
A.2.2	dc_init Flag	A-5
A.2.3	Transient Analysis & A/D Algorithm Synchronization	A-5
A.2.4	The Synchronization Loop	A-7
A.2.5	Assumptions about the Analog and Digital Algorithms	A-8
B	Open Issues	B-1
C	Analog Language Subset	C-1
C.1	Verilog-AMS Overview	C-1
C.2	Lexical Tokens	C-1
C.3	Data Types	C-2
C.4	Expressions	C-2
C.5	Signals	C-2
C.6	Analog Behavior	C-2
C.7	Mixed Signal	C-2
C.8	Hierarchical Structure	C-3
C.9	Scheduling Sematics	C-3
C.10	Open Issues	C-3
C.11	Syntax	C-3
C.12	Keywords	C-3
C.13	System Tasks and Functions	C-3
C.14	Compiler Directives	C-3
C.15	Standard Definitions	C-4
C.16	SPICE Compatability	C-4
C.17	Changes from Verilog-A LRM v1.0	C-4
C.17.1	New functions	C-4
C.17.2	Changes	C-5
C.18	Obsolete Functionality	C-5
C.18.1	Forever statement	C-5

C.18.2 NULL statement	C-5
C.18.3 Generate statement	C-5

D Syntax..... D-1

D.1 Source text	D-1
D.2 Natures	D-2
D.3 Disciplines	D-3
D.4 Declarations	D-3
D.5 Module instantiation	D-5
D.6 Connect statements	D-6
D.7 Behavioral statements	D-6
D.8 Analog Expressions	D-9
D.9 Expressions	D-9
D.10 General	D-12

E Keywords..... E-1

F System Tasks and Functions F-1

F.1 Environment parameter functions	F-1
F.2 \$random function	F-1
F.3 \$dist_ functions	F-2
F.4 Simulation control system tasks	F-3
F.4.1 \$finish	F-3
F.4.2 \$stop	F-3
F.5 File operation tasks	F-4
F.5.1 \$fopen	F-4
F.5.2 \$fclose	F-4
F.6 Displaying results	F-4
F.6.1 Escape sequences for special characters	F-5
F.6.2 Format specifications	F-5
F.6.3 Hierarchical name format	F-6
F.6.4 String format	F-6
F.7 Others - from Ian's writeup	F-6
F.7.1 System tasks and functions	F-7
F.7.2 Display tasks	F-7
F.7.3 File I/O tasks	F-7
F.7.4 Timescale tasks	F-8
F.7.5 Simulation control tasks	F-8
F.7.6 Timing check tasks	F-8
F.7.7 PLA modeling tasks	F-8
F.7.8 Stochastic analysis tasks	F-8
F.7.9 Simulation time functions	F-8
F.7.10 Conversion functions for reals	F-9

	F.7.11 Probabilistic distribution functions	F-9
	F.7.12 Environment Parameters (from 4.2.3 of A/MS LRM)	F-10
G	Compiler Directives	G-1
	G.1 `default_discipline	G-1
	G.2 `timescale	G-2
	G.3 `default_transition	G-3
	G.4 `define and `undef	G-4
	G.4.1 `define	G-4
	G.4.2 `undef	G-6
	G.5 `ifdef, `else, `endif	G-6
	G.6 `include	G-7
	G.7 `resetall	G-8
H	Standard Definitions	H-1
I	SPICE Compatibility	I-1
	I.1 Introduction	I-1
	I.1.1 Scope of Compatibility	I-1
	I.1.2 Degree of Incompatibility	I-1
	I.2 Accessing Spice Objects from Verilog	I-2
	I.2.1 Case Sensitivity	I-2
	I.2.2 Examples	I-2
	I.3 Preferred Primitive, Parameter, and Port Names	I-4
	I.3.1 Independent Sources	I-5
	I.3.2 Unsupported Components	I-6
	I.4 Other Issues	I-6
	I.4.1 Multiplicity Factor on Subcircuits	I-6
	I.4.2 Binning and Libraries	I-7
J	Glossary	J-1

Section 1

Verilog-AMS Overview

1.1 Overview

This Verilog-AMS Hardware Description Language (HDL) language reference manual defines a behavioral language for analog and mixed-signal systems. Verilog-AMS HDL is derived from the IEEE 1364 Verilog HDL specification. This document is intended to cover the definition and semantics of Verilog-AMS HDL as proposed by Open Verilog International (OVI).

The figure below shows the components and architecture of the Verilog-AMS HDL. The Verilog-AMS HDL consists of the complete IEEE 1364-1995 Verilog HDL specification (noted as Verilog-D in the figure), an analog equivalent for describing analog systems (noted as Verilog-A), and extensions to both for specifying the full Verilog AMS HDL (noted as MS Extensions).

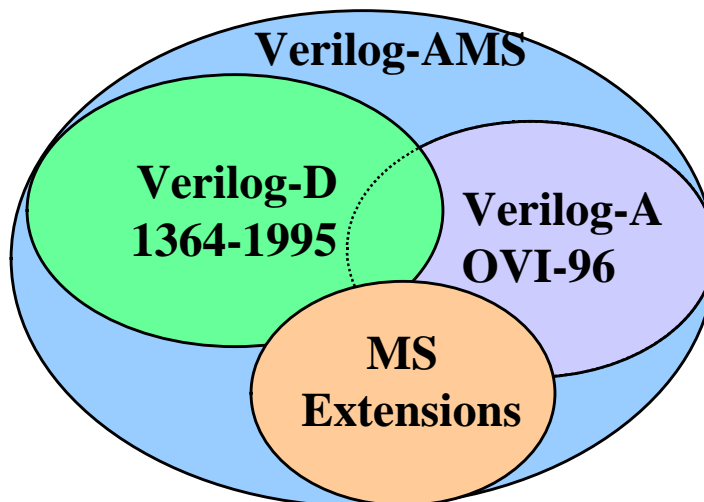


Figure 1-1: Verilog-AMS Architecture

The intent of Verilog-AMS HDL is to let designers of analog and mixed-signal systems and integrated circuits create and use modules that encapsulate high-level behavioral descriptions as well as structural descriptions of systems and components. The behavior of each module can be described mathematically in terms of its terminals and external parameters applied to the module. The structure of each component can be described in

terms of interconnected sub-components. These descriptions can be used in many disciplines such as electrical, mechanical, fluid dynamics, and thermodynamics.

Verilog-AMS HDL is defined to be applicable to both electrical and non-electrical systems description. It supports *conservative* and *signal-flow* descriptions by using the terminology for these descriptions using the concepts of *nodes*, *branches*, and *ports*. The solution of analog behaviors which obey the laws of conservation fall within the generalized form of Kirchhoff's Potential and Flow laws (KPL and KFL). Both of these are defined in terms of the quantities (e.g. voltage and current) associated with the analog behaviors.

1.2 Mixed-signal language features

The Verilog-AMS extends the features of the digital modeling language (IEEE 1364, Verilog Hardware Description Language, henceforth called Verilog-D) to provide a single unified language with both analog and digital semantics with backward compatibility. Below is a list of salient features of the resulting language:

- signals of both analog and digital types may be declared in the same module
- initial, always, and analog procedural blocks may appear in the same module
- both analog and digital signal values may be accessed (read operations) from any context (analog or digital) in the same module
- digital signal values may be set (write operations) from any context outside of an analog procedural block
- analog potentials and flows may only receive contributions (write operations) from inside an analog procedural block
- the semantics of the initial, always, and analog procedural blocks remain the same as in their respective languages
- the discipline declaration is extended to digital signals
- a new construct, connect statement, is added to facilitate auto-insertion of user defined connection modules between the analog and digital domains
- when hierarchical connections are of mixed type (i.e. analog signal connected to digital port or digital signal connected to analog port) then user defined connection modules are automatically inserted to perform signal value conversion

1.3 Systems

A *system* is considered to be a collection of interconnected *components* that are acted upon by a stimulus and produce a response. The components themselves might also be systems, in which case a hierarchical system is defined. If a component does not have any sub-components, then it is considered a primitive component. Each primitive component connects to one or more nodes. The behavior of each component is defined in terms of signal values at each node.

The components connect to nodes through ports to build hierarchy as shown in figure 1-2.

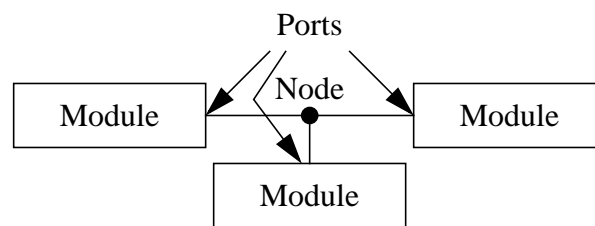


Figure 1-2: Components connect to nodes through ports.

In order to simulate systems, it is necessary to have a complete description of the system and all of its components. Descriptions of systems are usually given structurally. That is, the description of a system contains instances of components and how they are interconnected. Descriptions of components are given using behavior and or structure. A behavior is a mathematical description that relates the signals at the ports of the components.

1.3.1 Conservative systems

An important characteristic of conservative systems is that there are two values associated with every node (and hence every terminal) - the potential (also known as the across value, or the voltage in electrical systems) and the flow (the through value, or the current in electrical systems). The potential of the node is shared with all terminals connected to the node in such a way that all terminals see the same potential. The flow is shared such that flow from all terminals at a node must sum to zero. In this way, the node acts as an infinitesimal point of interconnection in which the potential is the same everywhere on the node and on which no flow can accumulate. Thus, the node embodies Kirchhoff's Potential and Flow Laws (KPL and KFL). When a component connects to a node through a conservative terminal, it may either affect, or be affected by, either the potential at the node, and/or the flow onto the node through the terminal.

With conservative systems it is also useful to define the concept of a branch. A branch is a path of flow between two nodes through a component. Every branch has an associated potential (the potential difference between the two nodes) and flow.

A behavioral description of a conservative component is constructed as a collection of interconnected branches. The constitutive equations of the component are formulated as to relate the branch potentials and flows. In the probe/source approach, the branch potential or flow is specified as a function of branch potentials and flows. If the branch potential and flow are left unspecified, not on the left-hand side of a contribution statement, then the branch acts as a probe. In this case, if the branch flow is used in an expression, the branch potential is forced to zero. Otherwise the branch flow is assumed to be zero and the branch potential is available for use in an expression. Using both the potential and flow of a 'probe' branch in an expression is not allowed. Nor is specifying both the branch potential and flow at the same time. (While these last two conditions are not really necessary, they do eliminate conditions that are useless and confusing.)

1.3.1.1 Reference nodes

The potential of a single node is given with respect to a reference node. The potential of the reference node, which is called **ground** in electrical systems, is always zero. **ground** is the global reference node in the circuit. It is compatible with all analog disciplines. It is used to bind a terminal of an instantiated module to the reference node.

1.3.1.2 Reference directions

The reference directions for a generic branch are as follows.

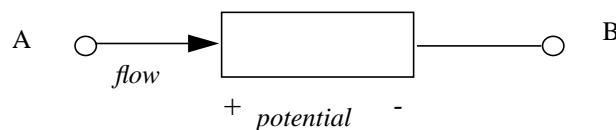


Figure 1-3: Reference directions

The reference direction for a potential is indicated by the plus and minus symbols near each terminal. Given the chosen reference direction, the branch potential is positive whenever the potential of the terminal marked with a plus sign (A) is larger than the potential of the terminal marked with a minus sign (B). Similarly, the flow is positive whenever it moves in the direction of the arrow (in this case from + to -).

Verilog-AMS HDL uses associated reference directions. A positive flow enters a branch through the terminal marked with the plus sign and exits the branch through the terminal marked with the minus sign.

1.3.2 Kirchhoff's laws

In formulating system equations, Verilog-AMS HDL uses two sets of relationships. The first are the constitutive relationships that describe the behavior of each component. Constitutive relationships can be kept inside the simulator as built-in primitives, or they can be provided by Verilog-AMS HDL module definitions.

The second set of relationships, interconnection relationships, describe the structure of the network. Interconnection relationships, which contain information on how the components are connected to each other, are only a function of the system topology. They are independent of the nature of the components.

The Verilog-AMS HDL simulator uses Kirchhoff's laws to define the relationships between the nodes and the branches. Kirchhoff's laws are typically associated with electrical circuits that relate voltages and currents. However, by generalizing the concepts of voltages and currents to potentials and flows, Kirchhoff's laws can be used to formulate interconnection relationships for any type of system.

Kirchhoff's laws provide the following properties relating the quantities present on nodes and branches.

- Kirchhoff's Flow Law (KFL)
The algebraic sum of all flows out of a node at any instant is zero.
- Kirchhoff's Potential Law (KPL)
The algebraic sum of all the branch potentials around a loop at any instant is zero.

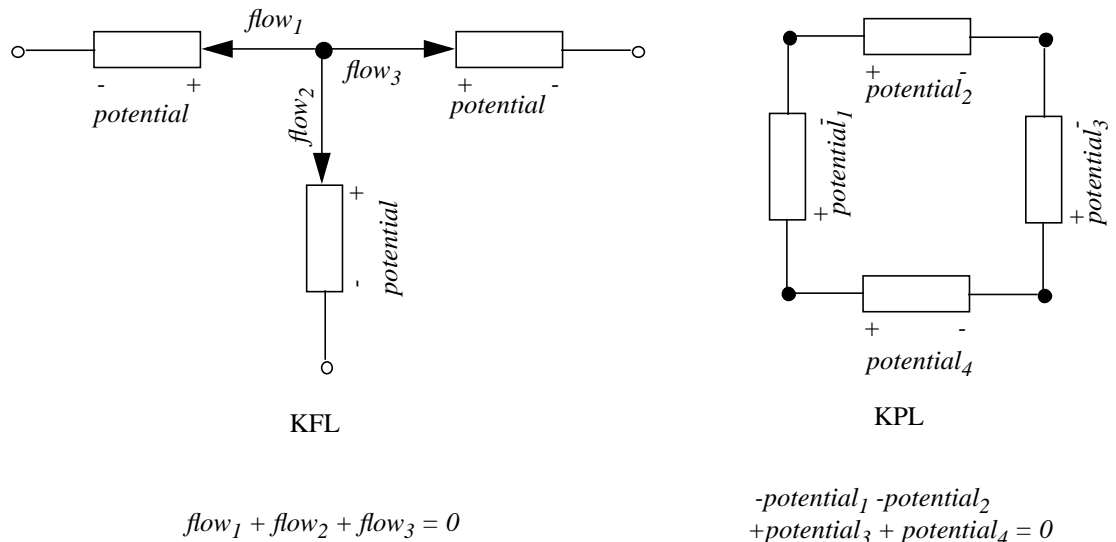


Figure 1-4: Kirchhoff's Flow Law (KFL) and Potential Law (KPL)

These laws imply that a node is infinitely small so that there is negligible difference in potential between any two points on the node and a negligible accumulation of flow.

1.3.3 Signal-flow systems

Unlike conservative systems, signal-flow systems only have one potential associated with every node. As a result, a signal-flow terminal must be unidirectional. It may either read the potential of the node, or it may specify it. Signal-flow terminals are either considered input ports if they pass the potential of the node into a component, or output ports if they specify the potential of a node.

Signal-flow terminals support a subset of the functionality of conservative terminals. As such, one can always use conservative semantics to represent signal-flow components. There are, however, two important benefits that result from allowing direct description of signal-flow components using signal-flow semantics. First, one only need declare the types of signals that one intends to use. Second, signal-flow semantics require a smaller number of equations and unknowns, and so results in a formulation that is more efficient to simulate.

There are some restrictions that are typically present in signal-flow formulations. For example,

- Typically, one cannot directly interface signal-flow and conservative components.
- Typically, signals are potential-like, making it difficult to represent flow-like signals.
- Typically, components descriptions can only be written in terms of **ground-**referred signals, making it difficult to write descriptions of components that use floating or differential signals.

1.3.4 Mixed conservative/signal flow systems

When practicing the top-down design style, it is extremely useful to mix conservative and signal-flow components in the same system. Users typically use signal-flow models early in the design cycle when the system is described in abstract terms, and gradually convert component models to conservative form as the design progresses. Thus, it is important to be able to initially describe a component using a signal-flow model, and later convert it to a conservative model, with minimum changes. It is also important to allow conservative and signal-flow components to be arbitrarily mixed in the same system.

The approach taken is to write component descriptions using conservative semantics, except that terminal and node declarations will only require types for those values that are actually used in the description. Thus, signal-flow terminals will only require the type of one potential to be specified (typically the potential, but could alternatively be the

flow), whereas conservative terminals would require types for both values (the potential and flow). For example, consider a differential voltage amplifier, a differential current amplifier, and a resistor. The amplifiers are written using signal-flow terminals and the resistor uses conservative terminals. These examples are meant to illustrate conceptual points only, and are not complete descriptions of the model.

```

module voltage_amplifier (out, in) ;
input in ;
output out ;
voltage      out ,          // Discipline voltage defined elsewhere
              in ;          // with access function V()
parameter real GAIN_V = 10.0 ;

analog
    V(out) <+ GAIN_V * V(in) ;

endmodule

```

In this case, only the voltage on the terminals are declared because only voltage is used in the body of the model.

```

module current_amplifier (out, in) ;
input in ;
output out ;
current      out ,          // Discipline current defined elsewhere
              in ;          // with access function I()
parameter real GAIN_I = 10.0 ;

analog
    I(out) <+ GAIN_I * I(in) ;

endmodule

```

Here, only current is used in the body of the model, so only current need be declared at the terminals.

```

module resistor (a, b) ;
inout a, b ;
electrical a, b ;           // access functions are V() and I()
parameter real R = 1.0 ;

analog
    V(a,b) <+ R * I(a,b) ;

endmodule

```

The description of the resistor relates both the voltage and current on the terminals. Both are defined in the conservative discipline electrical.

In summary, only those signals types declared on the terminals are accessible in the body of the model. Conversely, only those signals types used in the body need be declared.

This approach provides all of the power of the conservative formulation for both signal-flow and conservative terminals, without forcing types to be declared for unused signals on signal-flow nodes and terminals. In this way, the first benefit of the traditional signal-flow formulation is provided without the restrictions. The second benefit, that of a smaller, more efficient, set of equations to solve, is provided in a manner that is hidden from the user. The simulator begins by treating all terminals as being conservative, which will allow the connection of signal-flow and conservative terminals. This results in additional unnecessary equations for those nodes that only have signal-flow terminals. This situation can be recognized by the simulator and those equations eliminated.

Thus, this approach to allowing mixed conservative/signal-flow descriptions provides the following benefits:

- Conservative components and signal-flow components can be freely mixed. In addition, signal-flow components can be converted to conservative components, and vice versa, by modifying only the component behavioral description.
- Many of the capabilities of conservative terminals, such as the ability to access flow and the ability to access floating potentials, are available with signal-flow terminals.
- Signal-types only have to be given for potentials and flows if they are accessed in a behavioral description.
- If nodes and terminals are used only in a structural description (only in instance statements), then no signal-types need be specified.

1.3.5 Natures, disciplines and nodes

Verilog-AMS HDL allows definition of nodes based on disciplines. The disciplines associate potential and flow natures for conservative systems or only potential nature for signal-flow systems. The natures are a collection of attributes, including user defined attributes, that describes the units (meter, gram, newton, etc.), absolute tolerance for convergence, and the names of potential and flow access functions.

The disciplines and natures can be shared by many nodes. The compatibility rules help enforce the legal operations between nodes of different disciplines.

1.4 Conventions used in this document

This document is organized into sections, each of which focuses on some specific area of the language. There are subsections within each section to discuss with 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 description, followed by some examples and notes.

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

1. Lower case words, some containing embedded underscores, are used to denote syntactic categories, for example:

module declaration

2. Bold face words are used to denote reserved keywords, operators and punctuation marks as required part of the syntax. For example:

module = :

3. A vertical bar separates alternative items. For example:

attribute ::=
abstol | **units** | identifier

4. Square brackets enclose optional items. For example:

$$\text{input_declaration} ::= \mathbf{input} \text{ [range] list_of_ports ;}$$

5. Braces enclose a repeated item unless the braces appear in bold face, 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:

$$\text{list_of_port_def} ::= \text{port_def} \{ , \text{port_def} \}$$

```
list_of_port_def ::=
    port_def
    | list_of_port_def , port_def
```

6. 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_constant_expression* and *lsb_constant_expression* are equivalent to *constant_expression*, and *node_identifier* is an identifier that is used to identify (declare or reference) a node.

The main text uses *italicized* font when a term is being defined, and constant-width font for examples, file names, and while referring to constants.

1.5 Contents

This document contains the following chapters:

1. Verilog-AMS Overview
This section gives the overview of analog modeling, basic concepts, and describes Kirchhoff's Potential and Flow Laws.
2. Lexical Conventions
This section lexical tokens used in Verilog-AMS HDL.
3. Data Types
This section describes the data types - integer, real, parameter, nature, discipline, and node - as used in Verilog-AMS HDL descriptions.
4. Expressions
This section describes expressions, mathematical functions, and time domain functions used in Verilog-AMS HDL.
5. Signals
This section describes signals and branches, access to signals and branches, and various transformation functions.
6. Analog Behavior
This section describes the basic analog block and procedural language constructs available in Verilog-AMS HDL for behavioral modeling.
7. Mixed-Signal
This section describes the mixed-signal aspects of the Verilog-AMS HDL language.

8. Hierarchical Structures

This section describes how to build hierarchical descriptions using Verilog-AMS HDL.

A. Scheduling Semantics

This annex describes the basic simulation cycle as applicable to Verilog-AMS HDL.

B. Open Issues

This annex lists the open issues known to the working group.

C. Analog Language Subset

This annex describes the analog subset of Verilog-AMS HDL.

D. Syntax

This annex describes formal syntax for all Verilog-AMS HDL constructs in Backus-Naur Form (BNF).

E. Keywords

This annex lists all the words that are recognized in Verilog-AMS HDL as keywords.

F. System Tasks and Functions

This annex describes all system tasks and functions in Verilog-AMS HDL.

G. Compiler Directives

This annex describes all compiler directives in Verilog-AMS HDL.

H. Standard Definitions

This annex provides definitions of several natures, disciplines and constants useful writing models in Verilog-AMS HDL.

I. SPICE Compatibility

This annex describes SPICE compatibility with Verilog-AMS HDL.

J. Glossary

This annex describes various terms used in this document.

Section 2

Lexical Conventions

This section describes the lexical tokens used in Verilog-AMS HDL source text and their conventions. This section is based on Section 2, Lexical conventions, of IEEE 1364-1995. The changes specific to Verilog-AMS can be found in sections 2.5.3 and 2.6.2.

2.1 Lexical tokens

A Verilog-AMS HDL source text file is a stream of lexical tokens. A *lexical token* consists of one or more characters. The layout of tokens in a source file is free format—that is, spaces and newlines are not syntactically significant other than being token separators, except escaped identifiers (Section 2.6.1).

The types of lexical tokens in the language are as follows:

- white space
- comment
- operator
- number
- string
- identifier and keyword

2.2 White space

White space token type contains the characters for spaces, tabs, newlines, and formfeeds. These characters are ignored except when they serve to separate other lexical tokens.

2.3 Comments

The Verilog-AMS HDL has two forms to introduce comments. A *one-line comment* starts with the two characters `//` and ends with a newline. *Block comments* start with `/*` and ends with `*/`. Block comments can not be nested. The one-line comment token `//` does not have any special meaning in a block comment.

```
comment ::=
    short_comment
    | long_comment
short_comment ::=
    // { any_ASCII_characters_except_end_of_line } \n
long_comment ::=
    /* { any_ASCII_characters } */
```

Figure 2-1: Syntax for comments

2.4 Operators

Operators are single, double, or triple character sequences and are used in expressions. Section 4 discusses the use of operators in expressions.

Unary operators appear to the left of their operand. *Binary operators* appear between their operands. A *conditional operator* has two operator characters that separate three operands.

2.5 Numbers

Constant numbers can be specified as integer constants or real constants. The syntax for constants is as shown below:

```

number ::=
    decimal_number
    | real_number

decimal_number ::=
    [ sign ] unsigned_num

real_number ::=
    [ sign ] unsigned_num . unsigned_num
    | [ sign ] unsigned_num [ . unsigned_num ] e [ sign ] unsigned_num
    | [ sign ] unsigned_num [ . unsigned_num ] E [ sign ] unsigned_num
    | [ sign ] unsigned_num [ . unsigned_num ] scale_factor

sign ::=
    + | -

unsigned_num ::=
    decimal_digit { _ | decimal_digit }

decimal_digit ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

scale_factor ::=
    T | G | M | K | k | m | u | n | p | f | a

```

Figure 2-2: Syntax for integer and real constants

2.5.1 Integer constants

Integer constants are specified in decimal format as a sequence of digits 0 through 9, optionally starting with a plus or minus unary operator. The underscore character (`_`) is legal anywhere in a decimal number except as the first character. The underscore character is ignored. This feature can be used to break up long numbers for readability purposes.

Examples:

```

27_195_000    // same as 27195000
-659

```

2.5.2 Real constants

The *real constant numbers* are represented as described by IEEE STD-754-1985, an IEEE standard for double precision floating point numbers.

Real numbers can be specified in either decimal notation (for example, 14.72) or in scientific notation (for example, 39e8, which indicates 39 multiplied by 10 to the 8th power). Real numbers expressed with a decimal point must have at least one digit on each side of the decimal point. The underscore character is legal anywhere in a real

constant except as the first character of the constant or the first character after the decimal point. The underscore character is ignored.

Examples:

```
1.2
0.1
2394.26331
1.2E12      // the exponent symbol can be e or E
1.30e-2
0.1e-0
23E10
29E-2
236.123_763_e-12      // underscores are ignored
```

The following are invalid forms of real numbers because they do not have at least one digit on each side of the decimal point:

```
.12
9.
4.E3
.2e-7
```

2.5.3 Scale factors for real constants

The floating-point numbers can be specified with the following letter symbols for the scale factors indicated. Scale factors and scientific notation are not allowed to be used together in describing a real number.

	m = 10 ⁻³
T = 10 ¹²	u = 10 ⁻⁶
G = 10 ⁹	n = 10 ⁻⁹
M = 10 ⁶	p = 10 ⁻¹²
K = 10 ³ ; k = 10 ³	f = 10 ⁻¹⁵
	a = 10 ⁻¹⁸

Figure 2-3: Symbols used as multipliers to numbers

No space is permitted between the number and the symbol.
This form of floating-point number specification is provided in Verilog-AMS HDL in addition to the two methods for writing floating-point numbers described earlier.

Example:

Short form	Expanded form
1.3u	1.3e-6 or 0.0000013
5.46K	5460

2.6 Identifiers, keywords, and system names

An *identifier* is used to give an object a unique name so it can be referenced. An identifier can be any sequence of letters, digits, dollar signs (\$), and the underscore characters (_).

The first character of an identifier can not be a digit or \$; it can be a letter or an underscore. Identifiers are case sensitive.

Examples:

```
shiftreg_a
busa_index
error_condition
merge_ab
_bus3
n$657
```

2.6.1 Escaped identifiers

Escaped identifiers start with the backslash character (\) and end with white space (space, tab, newline). They provide a means of including any of the printable ASCII characters in an identifier (the decimal values 33 through 126, or 21 through 7E in hexadecimal).

Neither the leading back-slash character nor the terminating white space is considered to be part of the identifier. Therefore, an escaped identifier \cpu3 is treated the same as a non-escaped identifier cpu3.

Examples:

```
\busa+index
\clock
\***error-condition***
\net1/net2
\{a,b}
\a*(b+c)
```

2.6.2 Keywords

Keywords are predefined non-escaped identifiers that are used to define the language constructs. Preceding a Verilog-AMS keyword with an escape character causes it to be interpreted as an escaped identifier.

All keywords are defined in lowercase only. Annex E gives a list of all defined keywords.

2.6.2.1 Verilog-AMS Keywords

In addition to the keywords within Verilog-D HDL, the following are additional keywords used by Verilog-AMS HDL. These additional keywords are used in declaration of datatypes (Section 3) and in behavioral modeling (Section 6)

abstol	continuous	enddiscipline	generate	merged	units
access	ddt_nature	endnature	genvar	nature	using
analog	discipline	exclude	ground	potential	with
branch	discrete	flow	idt_nature	split	
connect	domain	from	inf	to	

Figure 2-4: List of additional keywords

2.6.2.2 Built-in math functions

The following are reserved keywords used by the math library (Section 4.2).

abs	asin	atan2	cos	floor	log	pow	sqrt
acos	asinh	atanh	cosh	lhypot	max	sin	tan
acosh	atan	ceil	exp	ln	min	sinh	tanh

Figure 2-5: List of built-in math functions

2.6.2.3 Built-in analog functions

The following are reserved keywords for all built-in analog functions which can be used in analog blocks (Section 4.4 and Section 6).

ac_stim	discontinuity	initial_step	last_crossing	white_noise
analysis	final_step	laplace_nd	limexp	zi_nd
bound_step	flicker_noise	laplace_np	noise_table	zi_np
ddt	idt	laplace_zd	slew	zi_zd
delay	idtmmod	laplace_zp	transition	zi_zp

Figure 2-6: List of built-in analog functions

2.6.2.4 Built-in analog and mixed-signal functions

The following are reserved keywords for all built-in mixed-signal functions (Section 6.7.5).

cross **timer**

Figure 2-7: List of built-in mixed-signal functions

2.6.2.5 Built-in driver access functions

The following are reserved keywords for all built-in driver access functions (Section 7.7).

driver_active	driver_local	driver_state
driver_count	driver_next_state	driver_strength
driver_delay	driver_next_strength	

Figure 2-8: List of built-in driver access functions

2.6.3 System tasks and functions

The \$ character introduces a language construct that enables development of user-defined tasks and functions. A name following the \$ is interpreted as a *system task* or a *system function*.

The syntax for a system task or function is as follows:

```
system_task_or_function ::=  
    $system_task_identifier [ ( list_of_arguments ) ] ;  
    | $system_function_identifier [ ( list_of_arguments ) ] ;  
list_of_arguments ::=  
    argument { , [ argument ] }  
argument ::=  
    expression
```

Figure 2-9: Syntax for system tasks and functions

Annex F lists all the system tasks for Verilog-AMS.

Any valid identifier, including keywords already in use in contexts other than this construct can be used as a system task or function name.

Examples:

```
$display ("display a message");  
$finish;
```

2.6.4 Compiler directives

The ``` character (the ASCII value 60, called open quote or accent grave) introduces a language construct used to implement compiler directives. The compiler behavior dictated by a compiler directive takes effect as soon as the compiler reads the directive. The directive remains in effect for the rest of the compilation unless a different compiler directive specifies otherwise. A compiler directive in one description file can therefore control compilation behavior in multiple description files.

Annex G lists all the compiler directives for Verilog-AMS.

Any valid identifier, including keywords already in use in contexts other than this construct can be used as a compiler directive name.

Example:

```
`define wordsize 8
```

Section 3

Data Types

Verilog-AMS HDL supports integer, real, and parameter data types as found in Verilog HDL. It also modifies the parameter data types and introduces array of real as an extension of real data type.

Verilog-AMS HDL introduces a new data type, called *node*, for representing analog signals. The nodes have *disciplines* that define the natures of potential and flow and associated attributes. A new datatype called *genvar* has been introduced for use with behavioral loops.

3.1 Integer and real datatypes

The syntax for declaring **integer** and **real** is as follows:

```
integer_declaration ::=
    integer list_of_identifiers ;
real_declaration ::=
    real list_of_identifiers ;
list_of_identifiers ::=
    var_name { , var_name }
var_name ::=
    variable_identifier
    | array_identifier range
range ::=
    [upper_limit_constant_expression : lower_limit_constant_expression]
```

Figure 3-1: Syntax for integer and real declarations

An *integer* declaration declares one or more variables of type integer. These variables can hold values ranging from -2^{31} to $2^{31}-1$. Arrays of integers can be declared using a range that defines the upper and lower indices of the array. Both indices must be constant expressions and must evaluate to a positive integer, a negative integer, or zero.

Arithmetic operations performed on integer variables produce 2's complement results.

A *real* declaration declares one or more variables of type real. The real variables are stored as 64 bit quantities, as described by IEEE STD-754-1985.

Arrays of real can be declared using a range that defines the upper and lower indices of the array. Both indices must be constant expressions and must evaluate to a positive integer, a negative integer, or zero.

Both integer and real variables are initialized to zero at the start of a simulation.

Examples:

```
integer a[1:64];           // an array of 64 integer values
real float ;              // a variable to store real value
real gain_factor[1:30] ;   // array of 30 gain multipliers
                           // with floating point values
```

3.2 Parameters

The syntax for parameter declarations is as follows:

```

parameter_declaration ::=
    parameter [opt_type] list_of_param_assignments ;

opt_type ::=
    real
    | integer

list_of_param_assignments ::=
    declarator_init
    | list_of_param_assignments , declarator_init

declarator_init ::=
    parameter_identifier = constant_expression { opt_value_range }
    | parameter_array_identifier = constant_param_arrayinit

parameter_identifier ::=
    identifier

parameter_array_identifier ::=
    identifier range

opt_value_range ::=
    from value_range_specifier
    | exclude value_range_specifier
    | exclude value_constant_expression

value_range_specifier ::=
    start_paren expression1 : expression2 end_paren

start_paren ::=
    [
    | (

end_paren ::=
    ]
    | )

expression1 ::=
    constant_expression | -inf

expression2 ::=
    constant_expression | inf

constant_param_arrayinit ::=
    { param_arrayinit_element_list }

param_arrayinit_element_list ::=
    param_arrayinit_element { , param_arrayinit_element }

param_arrayinit_element ::=
    constant_expression [ = value_range_specifier ]
    | constant_expression { constant_expression [ = value_range_specifier ] }

```

Figure 3-2: Syntax for parameter declaration

The list of parameter assignments must be a comma-separated list of assignments, where the right hand side of the assignment must be a constant expression, that is, an expression containing only constant numbers and previously defined parameters. For parameters that are defined as arrays, the initializer must be a `constant_param_arrayinit` expression which is a list of constant expressions containing only constant numbers and previously defined parameters within '{' and '}' delimiters.

Parameters represent constants, hence it is illegal to modify their value at runtime. However, parameters can be modified at compilation time to have values that are different from those specified in the declaration assignment. This allows customization of module instances. A parameter can be modified with the **defparam** statement, or in the module instance statement.

By nature, analog behavioral specifications are characterized more extensively in terms of parameters than their digital counterparts. There are three fundamental extensions to the parameter declarations defined in IEEE 1364:

- An optional type for the parameter can be specified in Verilog-AMS HDL. In IEEE 1364, the type of a parameter defaults to the type of the default expression.
- A range of permissible values can be defined for each parameter. In IEEE 1364, this check had to be done in user's model or was left as an implementation specific detail.
- Parameter arrays of basic integer and real data types.

3.2.1 Type Specification

The parameter declaration can contain an optional type specification. In this sense, the parameter keyword acts more as a type qualifier than a type specifier. A default value for the parameter must be specified.

The following examples illustrate this concept:

```
parameter real slew_rate = 1e-3 ;  
parameter integer size = 16 ;
```

If the type of a parameter is not specified, it is derived from the type of the value of the constant expression as in IEEE 1364.

If the type of the parameter is specified, and the value assigned to the parameter conflicts with the type of the parameter, the value is coerced to the type of the parameter. For example,

```
parameter real size = 10 ;
```

Here, size will be coerced to 10.0.

3.2.2 Value Range Specification

The parameter declaration can contain optional specifications of the permissible range of the values of a parameter. More than one range may be specified for inclusion or exclusion of values as legal values for the parameter.

The use of brackets, [and], indicate inclusion of the end points in the valid range. The use of parenthesis, (and), indicate exclusion of the end points from the valid range. It is possible to include one end point and not the other using [) and (]. The first expression in the range must be numerically smaller than the second expression in the range.

For example,

```
parameter real neg_rail = -15 from [-50:0) ;
parameter integer pos_rail = 15 from (0:50) ;
parameter real gain = 1 from [1:1000] ;
```

Here, the parameter `neg_rail` is given a default value of -15 and only allowed to acquire values within the range of $-50 \leq \text{neg_rail} < 0$. Similarly, for parameter `pos_rail`, the default value is 15 and it is only allowed to acquire values within the range of $0 < \text{pos_rail} < 50$. For parameter `gain`, the default value is 1 and it is allowed to acquire values within the range of $1 \leq \text{gain} \leq 1000$.

The keyword **inf** may be used to indicate infinity. If preceded by a negative sign, it indicates negative infinity. For example,

```
parameter real val3=0 from [0:inf) exclude (10:20) exclude (30:40];
```

A single value may be excluded from the possible valid values for a parameter. For example,

```
parameter real res = 1.0 exclude 0 ;
```

The value of a parameter is checked against the specified range.

3.2.3 Parameter Arrays

The Verilog-AMS HDL specification includes behavioral extensions that utilize arrays. It requires that these arrays be initialized in their definitions and allow overriding their value as with other parameter types. The declaration of arrays of parameters is in a similar manner to those of parameters and register arrays of reals and integers in Verilog-D HDL.

For example,

```
parameter real poles[0:3] = { 1.0, 3.198, 4.554, 2.00 };
```

3.3 Genvars

Genvars are integer-valued variables that compose static expressions for instantiating structure behaviorally such as accessing analog signals within behavioral looping constructs. The syntax for declaring genvar variables is as follows:

```
genvar_declaration ::=
    genvar list_of_genvar_identifiers ;

list_of_genvar_identifiers ::=
    genvar_identifier { , genvar_identifier }
```

Figure 3-3: Syntax for genvar declaration

The static nature of genvar variables is derived from the limitations upon the contexts in which their values can be assigned. For example:

```
genvar i;
analog begin
    ...
    for (i = 0; i < 8; i = i + 1) begin
        V(out[i]) <+ transition(value[i], td, tr);
    end
    ...
end
```

The genvar variable, *i*, can only be assigned to within the for-loop control. Assignments to the genvar variable *i* can consist only of expressions of static values, e.g., parameters, literals and other genvar variables.

3.4 Nodes

In addition to the data types supported by IEEE 1364, for continuous time simulation an additional data type, *node*, is introduced in Verilog-AMS. The fundamental characteristic of a node data type is that the values of a node are defined by simultaneous solution of equations defined by the instances connected to the *node* using Kirchhoff's conservation laws. In general, a node represents a point of physical connections between entities of continuous-time description, obeying conservation-law semantics.

A node is characterized by the *discipline* it follows. For example, all low-voltage nodes have certain common characteristics, all mechanical nodes have certain characteristics, etc. Therefore, a node is always declared as a type of discipline. In this sense, a discipline is a user defined type for declaring a node.

A discipline is characterized by the attributes defined in *natures* for potential and flow.

3.4.1 Natures

A **nature** is a collection of attributes. In Verilog-AMS HDL, there are several pre-defined attributes. In addition, user-defined attributes may be declared and assigned constant values in a nature.

The nature declarations are at the same level as discipline and module declarations in the source text. That is, natures are declared at the top level, and nature declarations do not nest inside other nature declarations, discipline declarations, or module declarations.

The syntax for defining a nature is as follows:

```
nature_declaration ::=
    nature          nature_name
    [ nature_descriptions ]
    endnature

nature_name ::=
    nature_identifier
    | nature_identifier : parent_identifier

parent_identifier ::=
    nature_identifier
    | discipline_identifier.flow
    | discipline_identifier.potential

nature_descriptions ::=
    nature_description { nature_description }

nature_description ::=
    attribute = constant_expression ;

attribute ::=
    abstol
    | access
    | ddt_nature
    | idt_nature
    | units
    | attribute_identifier
```

Figure 3-4: Syntax for nature declaration

A nature must be defined between the keywords **nature** and **endnature**. Each nature definition must have a unique identifier as the name of the nature, and must include all the required attributes as noted in 3.4.1.2.

For example,

```

nature current
    units = "A" ;
    access = I ;
    idt_nature = charge ;
    abstol = 1u ;
endnature

nature voltage
    units = "V" ;
    access = V;
    abstol = 1u ;
endnature

```

3.4.1.1 Derived Natures

A nature may be derived from an already declared nature. This allows the new nature to have the same attributes as the attributes for the already declared nature. The new nature is called a *derived nature*, and the existing nature is called a *parent nature*. If a nature is not derived from any other nature, then it is called a *base nature*.

In order to derive a new nature from an existing nature, the new nature name should be followed by a colon (:) and the name of the parent nature in the nature definition.

A derived nature may declare additional attributes, or override values of the attributes already declared in the parent nature, with certain restrictions (as outlined in section 3.4.1.2) for the predefined attributes.

The attributes of the derived nature are accessed in the same manner as accessing attributes of any other nature.

For example,

```

nature ttl_curr
    units = "A" ;
    access = I ;
    abstol = 1u ;
endnature

// An alias

nature ttl_node_curr : ttl_curr
endnature

nature new_curr : ttl_curr           // derived, but different
    abstol = 1m ;                   // modified for this nature
    max = 12.3 ;                   // new attribute for this nature
endnature

```

3.4.1.2 Attributes

Attributes define the value of certain quantities that characterize the nature. There are five predefined attributes — **abstol**, **access**, **idt_nature**, **ddt_nature**, and **units**. In addition, user defined attributes may be defined in a nature.

Attribute declaration assigns a constant expression to the attribute name.

abstol

The **abstol** attribute provides a tolerance measure (metric) for convergence of potential or flow calculation. It specifies the maximum negligible for signals associated with the nature.

This attribute is required for all base natures. It is legal for a derived nature to change the **abstol** attribute but if left unspecified it will inherit the **abstol** from its parent nature. The constant expression assigned to it must evaluate to a real value.

access

The **access** attribute identifies the name for the access function. When the nature is used to bind potential, the name is used as an access function for the potential; when the nature is used to bind flow, the name is used as an access function for the flow. The usage of access function is described further in section 4.3.

This attribute is required for all base natures. It is illegal for a derived nature to change the **access** attribute; the derived nature always inherits the **access** attribute of its parent nature. When specified, the constant expression assigned to it must be an identifier (name, not a string).

idt_nature

The **idt_nature** attribute provides a relationship between a nature and the nature that represents its time integral.

The **idt_nature** is used to reduce the need for users to specified tolerances on the **idt()** operator. If this operator is applied directly on nodes, then the tolerance can be taken from the signal eliminating the need to give a tolerance with the operator.

When specified, the constant expression assigned to an **idt_nature** attribute must be the name (not a string) of a nature that is defined elsewhere. It is possible for a nature to be self referring with respect to its **idt_nature** attribute. In other words, the value of the **idt_nature** attribute may be the nature that the attribute itself is associated with.

The **idt_nature** attribute is optional with its default value being the nature itself. While it is possible to override the parent's value of the **idt_nature** attribute using a derived nature, the nature specified must be related (share the same base nature) to the nature used for the **idt_nature** attribute by the parent.

ddt_nature

The **ddt_nature** attribute provides a relationship between a nature and the nature that represents its time derivative.

The **ddt_nature** is used to reduce the need for users to specified tolerances on the **iddt()** operator. If this operator is applied directly on nodes, then the tolerance can be taken from the signal eliminating the need to give a tolerance with the operator.

When specified, the constant expression assigned to a `ddt_nature` attribute must be the name (not a string) of a nature that is defined elsewhere. It is possible for a nature to be self referring with respect to its `ddt_nature` attribute. In other words, the value of the **`ddt_nature`** attribute may be the nature that the attribute itself is associated with.

The **`ddt_nature`** attribute is optional with its default value being the nature itself. While it is possible to override the parent's value of the `ddt_nature` attribute using a derived nature, the nature specified must be related (share the same base nature) to the nature used for the `ddt_nature` attribute by the parent.

units

The **`units`** attribute provides a binding between the value of the access function and the units for that value.

The units field is provided so that simulators can annotate the signals with their units and is also used in the node compatibility rule check.

This attribute is required for all base natures. It is illegal for a derived nature to define or change the units attribute; the derived nature always inherits the units attribute of its parent nature. When specified, the constant expression must be a string.

3.4.1.3 User Defined Attributes

In addition to the predefined attributes listed above, a nature can have other attributes that may be useful for analog modeling. Typical examples include certain maximum and minimum values to define valid range, etc.

A user defined attribute may be declared in the same manner as any of the predefined attributes. The name of the attribute must be unique in the nature being defined, and the value being assigned to the attribute must be constant.

3.4.2 Disciplines

A discipline description consists of binding natures to potential and flow.

The syntax for declaring a discipline is as follows:

```

discipline_declaration ::=
    discipline discipline_identifier
    [ discipline_descriptions ]
    enddiscipline

discipline_descriptions ::=
    discipline_description { discipline_description }

discipline_description ::=
    nature_binding
    | domain_binding
    | attr_override

nature_binding ::=
    pot_or_flow nature_identifier ;

domain_binding ::=
    domain continuous ;
    | domain discrete ;

attr_override ::=
    pot_or_flow . attribute_identifier = constant_expression ;

pot_or_flow ::=
    potential
    | flow

```

Figure 3-5: Syntax for discipline declaration

A discipline must be defined between the keywords **discipline** and **enddiscipline**. Each discipline must have a unique identifier as the name of the discipline.

The discipline declarations are at the same level as nature and module declarations in the source text. That is, disciplines are declared at the top level, and discipline declarations do not nest inside other discipline declarations, nature declarations, or module declarations.

3.4.2.1 Nature Binding

Each discipline can bind a nature to its potential and flow.

Only the name of the nature is specified in the discipline. The nature binding for potential is specified using the keyword **potential**. The nature binding for flow is specified using the keyword **flow**.

The access function defined in the nature bound to potential is used in the model to describe the signal-flow that obeys Kirchhoff's Potential Law (KPL). This access function is called the *potential access function*.

The access function defined in the nature bound to flow is used in the model to describe the signal-flow that obeys Kirchhoff's Flow Law (KFL). This access function is called the *flow access function*.

Disciplines with two natures are called conservative disciplines, and the nodes associated with conservative disciplines are called conservative nodes. Conservative disciplines must not have the same nature specified for both the potential and the flow. Disciplines with a single potential nature are called as signal-flow disciplines, and the nodes with signal-flow disciplines are called signal-flow nodes. Only the potential nature is allowed to be specified for a signal-flow discipline.

Example:

Conservative discipline:

```
discipline electrical
    potential Voltage ;
    flow Current ;
enddiscipline
```

Signal-flow disciplines:

```
discipline voltage
    potential Voltage ;
enddiscipline

discipline current
    potential Current;
enddiscipline
```

3.4.2.2 Domain Binding

Analog signal values are represented in continuous time whereas digital signal values are represented in discrete time. The domain attribute of the discipline stores this property of the signal.

It takes two possible values - **discrete** and **continuous**. Signals with continuous-time domain are real valued. Signals with discrete-time domain may either be binary (0, 1, X or Z), integer or real valued.

For example,

```
discipline electrical
    domain continuous;
    potential Voltage;
    flow Current;
enddiscipline

discipline logic
    domain discrete;
enddiscipline
```

This attribute is optional. The default value for domain is **continuous** for non-empty disciplines.

3.4.2.3 Empty Disciplines

It is possible to define a discipline with no nature bindings and it has no domain. These are known as empty disciplines, and may be used in structural descriptions when you wish to let the components connected to a node determine which natures are to be used for the node.

Example:

```
discipline neutral
enddiscipline

discipline interconnect
    domain continuous;
enddiscipline
```

3.4.2.4 Overriding Nature Attributes from Discipline

A discipline can override the value of the bound nature for the pre-defined attributes (except as restricted by section 3.4.1.2), as shown for the flow `ttl_curr` in the example below. To do so from a discipline declaration, the bound nature and attribute must be defined, as shown below for the `abstol` value within the discipline `ttl` in the following example. The general form is: the keyword **flow** or **potential**, then the hierarchical separator `..`, then the attribute name, and set all of this equal to (`=`) the new value (e.g., **flow.abstol** = 10u).

```
nature ttl_curr
    units = "A" ;
    access = I ;
    abstol = 1u ;
endnature

nature ttl_volt
    units = "V" ;
    access = V ;
    abstol = 100u ;
endnature

discipline ttl
    potential ttl_volt ;
    flow ttl_curr ;
    flow.abstol = 10u ;
enddiscipline
```

3.4.2.5 Deriving Natures from Disciplines

A nature may be derived from the nature bound to potential or flow in a discipline. This allows the new nature to have the same attributes as the attributes for the nature bound to the flow or the potential of the discipline.

If the nature binding to the flow or the potential of a discipline changes, the new nature will automatically inherit the attributes for the changed nature.

In order to derive a new nature from flow or potential of a discipline, the nature declaration shall also include the discipline name followed by the hierarchical separator . and the keyword **flow** or **potential**, as shown for `ttl_node_curr` in the example below.

A nature derived from the flow or potential of a discipline may declare additional attributes, or override values of the attributes already declared.

For example,

```
nature ttl_node_curr : ttl.flow // from the example in section 3.4.2.4
endnature                // abstol = 10u as modified in ttl

nature ttl_node_volt : ttl.potential // from the example in section 3.4.2.4
    abstol = 1m ;           // modified for this nature
    max = 12.3 ;           // new attribute for this nature
endnature
```

3.4.3 Node Declaration

Each node declaration is associated with an already declared discipline. The following syntax is used for declaring nodes:

```
node_declaration ::=
    discipline_identifier [range] list_of_nodes ;
range ::=
    [ msb_expression : lsb_expression ]
list_of_nodes ::=
    node_name
    | node_name , list_of_nodes
node_name ::=
    node_identifier
    | hierarchical_node_identifier
msb_expression ::=
    constant_expression
lsb_expression ::=
    constant_expression
```

Figure 3-6: Syntax for node declaration

If a range is specified for a node, the node is called a vector node; otherwise it is called a scalar node. A vector node is also called an analog bus.

Examples:

```

electrical [MSB:LSB] n1 ; // MSB and LSB are parameters
voltage [5:0] n2, n3 ;
magnetic inductor ;
logic [10:1] connector1 ;

```

Nodes represent the abstraction of information about signals. As terminals (ports of a module declared as nodes), nodes represent component interconnections. Nodes declared in the module interface define the terminals to the module (See section 8.3.4)

A node used for modeling a conservative system must have the discipline with both access functions (potential and flow) defined. For modeling a signal-flow system, the discipline of a node can have only one access function.

Nodes declared with an empty discipline do not have declared natures, so such nodes cannot be used in a behavioral description (because the access functions are not known). However, such nodes can be used in structural descriptions, where they inherit the natures from the ports of the instances of modules that connect to them.

3.4.4 Implicit Nodes

Nodes can be used in a structural descriptions without being declared. In this case, the node is implicitly declared to be a scalar node with the empty discipline. The ground node, as described in section 1.3.1.1, is a special implicit node which allows connection to the global reference. Implicit nodes cannot appear in behavioral descriptions. For example:

```

module top(i1, i2, o1, o2, o3);
  input i1, i2;
  output o1, o2, o3;
  electrical i1, i2, o1, o2, o3;

  // ab1, ab2, cb1, cb2 are implicit nodes, not declared
  blk_a a1( i1, ab1 );
  blk_a a2( i2, ab2 );
  blk_b b1( ab1, cb1 );
  blk_b b2( ab2, cb2 );
  blk_c c1( o1, o2, o3, cb1, cb2);

endmodule

```

3.5 Default Discipline

Verilog-AMS supports the 'default_discipline compiler directive. This directive specifies a default discipline to be applied to any signal that does not have an explicit discipline declaration.

It has the following syntax:

```
default_discipline_directive ::=
    'default_discipline' [discipline_identifier [qualifier] [scope]]

qualifier ::=
    integer | real | reg |
    wire | tri | wand | triand | wor | prior | trireg |
    tri0 | tri1 | supply0 | supply1

scope ::= module_identifier
```

Figure 3-7: Syntax for setting default discipline compiler directive

The scope of this directive is similar to the scope of the **'define** compiler directive. The default discipline is applied to all signals without a discipline declaration that appear in the text stream following the use of the **'default_discipline** directive until either the end of the text stream or until another **'default_discipline** directive with the same combination of qualifier and scope (if applicable) is found in the subsequent text. Therefore, more than one **'default_discipline** directives can be in force simultaneously, provide they differ in scope or qualifier or both.

If this directive is used without a discipline name, it turns off all currently active default disciplines without setting a new default discipline. The subsequent signals without a discipline will be associated with the empty discipline.

For example,

```
'default_discipline logic
module behavnand(in1, in2, out);
    input in1, in2;
    output out;
    reg out;

    always begin
        out = ~(in1 && in2);
    end

endmodule
```

This example illustrates the usage of the **'default_discipline** directive. The signals in1, in2 and out all have discipline logic by default.

There is a precedence of such compiler directives. The more specific directives have higher precedence over the general directives.

3.5.1 Discipline Precedence

While a net itself may be declared only in the module to which it belongs, the discipline of the net may be specified in a number of ways. The discipline name may appear in the declaration of the net. The discipline name may be used in a declaration which makes an out of context reference to the net from another module. The discipline name may be used in a **‘default discipline** compiler directive. Discipline conflicts may arise if more than one of these methods is applied to the same net. Discipline conflicts will be resolved using the following order of precedence:

1. A declaration from a module other than the module to which the net belongs using an out of module reference. e.g.

```
module example1;
    electrical example2.net;
endmodule
```
2. The local declaration of the net in the module to which it belongs. e.g.

```
module example2;
    electrical net;
endmodule
```
3. **‘default_discipline** with qualifier and scope e.g. **‘default_discipline** electrical trireg example1.instance5
4. **‘default_discipline** with scope only e.g. **‘default_discipline** electrical example1.instance5
5. **‘default_discipline** with qualifier only e.g. **‘default_discipline** electrical trireg
6. **‘default discipline** without qualifier or scope e.g. **‘default_discipline** electrical

It is not legal to have two different disciplines with the same level of precedence for the same net.

3.6 Node Compatibility

Certain operations can be done on nodes only if the two (or more) nodes are compatible. For example, if an access function has two nodes as arguments, they must be compatible. The nodes are considered compatible if their respective disciplines are compatible. The following rules apply in deciding whether two disciplines are compatible:

Self Rule: A discipline is compatible with itself.

Potential Compatibility Rule: If the natures of the two potential are compatible, and the natures of the two flows are not incompatible then the two disciplines are considered compatible.

Flow Compatibility Rule: If the natures of the two flows are compatible, and the natures of the two potentials are not incompatible then the two disciplines are considered compatible.

Nature Compatibility Rule: Two natures are compatible if they both exist and are derived from the same base nature.

Nature Incompatibility Rule: Two natures are not incompatible if they are compatible or if one or both do not exist.

Units Value Rule: All compatible natures must have the same value for the attribute **units**. Since a child nature cannot override a base nature's unit, this rule is always maintained.

Empty Discipline Rule: An empty discipline is compatible with all disciplines.

Discrete Domain Rule: Disciplines with discrete domain attribute that are of the same signal value (i.e. bit, real, integer) are compatible with each other.

Domain Incompatibility Rule: Disciplines with different domain attributes are incompatible with each other.

Node Connection Rule: It is an error to connect two ports with incompatible disciplines unless there is a connect statement (section 7.4) defined between these disciplines.

The following example illustrates these rules:

nature voltage access = V; units = "V"; abstol = 1u; endnature	nature position access = X; units = "m"; abstol = 1u; endnature
nature current access = I; units = "A"; abstol = 1p; endnature	nature force access = F; units = "N"; abstol = 1n; endnature
discipline electrical potential voltage; flow current; enddiscipline	discipline mechanical potential position; flow force; enddiscipline
discipline logic : electrical potential.abstol =1m; enddiscipline	discipline sig_flow_x potential position; enddiscipline
discipline sig_flow_v potential voltage; enddiscipline	discipline sig_flow_f flow force; enddiscipline
discipline sig_flow_i flow current; enddiscipline	discipline empty enddiscipline

The following compatibility observations can be made from the above example:

- *electrical* and *logic* are compatible disciplines because natures for both potential and flow exist and are derived from the same base natures.
- *electrical* and *sig_flow_v* are compatible disciplines because nature for potential is same for both disciplines and nature for flow does not exist in *sig_flow_v*.
- *electrical* and *sig_flow_i* are compatible disciplines because nature for flow is same for both disciplines and nature for potential does not exist in *sig_flow_i*.
- *electrical* and *mechanical* are incompatible disciplines because natures for both potential and flow are not derived from the same base natures.
- *electrical* and *sig_flow_x* are incompatible disciplines because nature for both potential are not derived from the same base nature.
- *sig_flow_v* and *sig_flow_i* are compatible disciplines as well as *sig_flow_v* and *sig_flow_f* are compatible disciplines because the natures do not conflict (the potential natures do not conflict because only *sig_flow_v* has a potential nature,

and the flow natures do not conflict because *sig_flow_v* does not have a flow nature)

- An empty discipline is compatible with all other disciplines because it has neither a potential nor a flow nature. Without natures, there can be no conflicting natures.

3.7 Branches

A branch is a path between two nodes. If both nodes are conservative, then the branch is a conservative branch and it defines a branch potential and a branch flow. If one node is a signal-flow node, then the branch is a signal-flow branch and it defines either a branch potential or a branch flow, but not both.

3.7.1 Branch Declaration

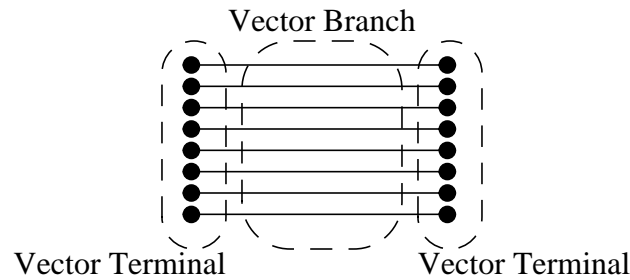
Each branch declaration is associated with two nodes from which it derives a discipline. These nodes are referred to as the branch terminals. Only one node need be specified, in which case the second is taken to be **ground** and the discipline for the branch is taken from the specified node. The disciplines for the nodes specified must be compatible (see section 3.6).

The following syntax is used for declaring branches:

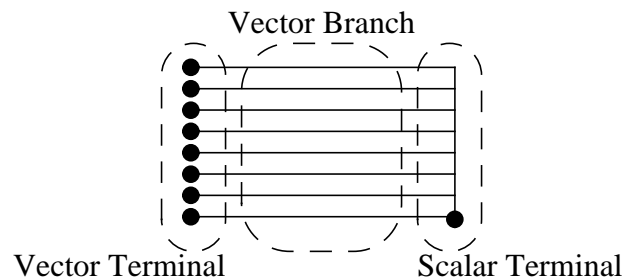
```
branch_declaration ::=
    branch list_of_branches ;
list_of_branches ::=
    terminals list_of_branch_identifiers
terminals ::=
    ( node_or_port_scalar_expression )
    | ( node_or_port_scalar_expression , node_or_port_scalar_expression )
list_of_branch_identifiers ::=
    branch_identifier [ range ]
    | branch_identifier [ range ], list_of_branch_identifiers
```

Figure 3-8: Syntax for branch declaration

If one of the terminals of a branch is a vector node, then the other terminal must either be a scalar or it must be a vector node of the same size. In this case, the branch is referred to as being a vector branch. When both terminals are vectors, the scalar branches that make up the vector branch connect between the corresponding scalar nodes that make up the vector terminals



When one terminal is a vector and the other is a scalar, there is one scalar branch connecting to each scalar node in the vector terminal, and the other terminal of each branch connects to the scalar terminal



3.7.2 Accessing Node and Branch Signals

Signals on nodes and branches can be accessed only by the access functions of the discipline associated with them. The name of the node or the branch must be specified as the argument to the access function.

For example,

```
electrical out, in ;           // as defined in Section 3.4.2.1
parameter real gm = 1 ;
```

```
analog
    I(out) <+ gm*V(in) ;
```

```
electrical p, n;
branch (p,n) res;
parameter real R = 50;
```

```
analog
    V(res) <+ R*I(res);
```

The formal syntax for referencing access functions is as follows:

```

access_function_reference ::=
    bvalue
    | pvalue
bvalue ::=
    access_identifier ( analog_signal_list )
analog_signal_list ::=
    branch_identifier
    | array_branch_identifier [ genvar_expression ]
    | node_or_port_scalar_expression
    | node_or_port_scalar_identifier , node_or_port_scalar_identifier
node_or_port_scalar_expression ::=
    node_or_port_identifier
    | array_node_or_port_identifier [ genvar_expression ]
    | buss_node_or_port_identifier [ genvar_expression ]
pvalue ::=
    flow_access_identifier ( < port_scalar_expression > )
port_scalar_expression ::=
    port_identifier
    | array_port_identifier [ genvar_expression ]
    | buss_port_identifier [ genvar_expression ]

```

Figure 3-9: Syntax for referencing access functions of a node

3.7.3 Accessing Attributes

The attributes are attached to the nature of potential or flow. Therefore, the attributes for a node or a branch can be accessed using the hierarchical referencing operator (.) to the potential or flow for the node or the branch.

For example,

```

electrical a, b, n1, n2;
branch (n1, n2) cap ;
parameter real c= 1p;

analog begin
    I(a,b) <+ c*ddt(V(a,b), a.potential.abstol);
    I(cap) <+ c*ddt(V(cap), n1.potential.abstol) ;
end

```

The formal syntax for referencing access attributes is as follows:

```

attribute_reference ::=
    node_identifier . pot_or_flow . attribute_identifier

```

Figure 3-10: Syntax for referencing attributes of a node

3.8 Namespace

3.8.1 Nature and Discipline

The natures and disciplines are defined at the same level of scope as that of modules. Thus, identifiers defined as natures or disciplines have the global scope, and allows declaration of nodes inside any module in the same manner as an instance of a module.

3.8.2 Access Functions

Each access function name, defined before a module is parsed, is automatically added to that module's name space unless there is another identifier defined with same name as the access function in that module's name space. Furthermore the access function of each base nature must be unique for all the base nature access functions.

3.8.3 Node

The scope rules for node identifiers are the same as the scope rules for any other identifier declarations with one exception - nodes may not be declared anywhere other than the port of a module or in the module itself. In other words, a node may not be declared inside any block (named or unnamed) other than a module; there is no local declaration for a node.

All access functions are always uniquely defined for each node based on the discipline of the node. Each access function is always used with the name of the node as its argument, and a node is always accessed only through its access functions.

The hierarchical reference character (.) may be used to reference a node across the module boundary using the rules specified in IEEE 1364.

3.8.4 Branch

The scope rules for branch identifiers are the same as the scope rules for node identifiers. In other words, branches are declared inside modules but may not be declared inside any block (named or unnamed) other than a module; there is no local declaration for a branch.

The access functions are always uniquely defined for each branch based on the discipline of the branch. The access function is always used with the name of the branch as its argument, and a branch is always accessed only through its access functions.

The hierarchical reference character (.) may be used to reference a node across the module boundary using the rules specified in IEEE 1364.

Section 4

Expressions

This section describes the operators and operands available in the Verilog-AMS HDL, and how to use them to form expressions.

An *expression* is a construct that combines *operands* with *operators* to produce a result that is a function of the values of the operands and the semantic meaning of the operator. Any legal operand, such as an integer or an indexed element from an array of real, without any operator is also considered an expression. Wherever a value is needed in a Verilog-AMS HDL statement, an expression can be used.

Some statement constructs require an expression to be a *constant expression*. The operands of a constant expression consists of constant numbers and parameter names, but can use any of the operators defined in Table 4-1.

4.1 Operators

The symbols for the Verilog-AMS HDL operators are similar to those in the C programming language. Table 4-1 lists these operators.

Table 4-1 : Operators

{}, {}	concatenation, replication
+ - * /	arithmetic
%	modulus
> >= < <=	relational
!	logical negation
&&	logical and
	logical or
==	logical equality
!=	logical inequality
~	bit-wise negation
&	bit-wise and
	bit-wise inclusive or
^	bit-wise exclusive or

Table 4-1 : Operators

<code>^~</code> or <code>~^</code>	bit-wise equivalence
<code><<</code>	left shift
<code>>></code>	right shift
<code>? :</code>	conditional
<code>or</code>	event or

4.1.1 Operators with real operands

The operators shown in Table 4-2 are legal when applied to real operands. All other operators are considered illegal when used with real operands.

Table 4-2 : Legal operators for use in real expressions

<code>{}, {{}}</code>	concatenation and replication operator
<code>unary +</code> <code>unary -</code>	unary operators
<code>+</code> <code>-</code> <code>*</code> <code>/</code>	arithmetic
<code>></code> <code>>=</code> <code><</code> <code><=</code>	relational
<code>!</code> <code>&&</code> <code> </code>	logical
<code>==</code> <code>!=</code>	logical equality
<code>?:</code>	conditional

The result of using logical or relational operators on real numbers is an integer value 0 (false) or 1 (true).

Table 4-2 lists operators that can not be used to operate on real numbers.

Table 4-3 : Operators not allowed for real expressions

<code>%</code>	modulus
<code><<</code> <code>>></code>	shift

4.1.1.1 Real To Integer Conversion

Real numbers are converted to integers by rounding the real number to the nearest integer, rather than by truncating it. Implicit conversion takes place when a real number is assigned to an integer. The ties are rounded away from zero.

Examples:

The real numbers 35.7 and 35.5 both become 36 when converted to an integer and 35.2 becomes 35.

Converting -1.5 to integer yields -2, converting 1.5 to integer yields 2.

4.1.1.2 Arithmetic Conversion

For operands, a common data type for each operand is determined before the operator is applied. If either operand is real, the other operand is converted to real. Implicit conversion takes place when a integer number is used with a real number in an operand.

Examples:

```
a = 3 + 5.0;
// The expression "3 + 5.0" is evaluated by "casting" the
// integer 3 to the real 3.0, and the result of the expression is 8.0.


b = 1 / 2;
// The above is integer division and the result is 0.

c = 8.0 + (1/2);
// (1/2) is treated as integer division, but the result is cast to a
// real (0.0) during the addition, and the result of the expression is 8.0.
```

4.1.2 Binary operator precedence

The precedence order of *binary operators* and the *conditional operator* (?:) is shown below in Table 4-4.

Table 4-4 : Precedence rules for operators

+ - ! ~ (unary)	highest precedence
* / %	
+ - (binary)	
<< >>	
< <= > >=	
== !=	
& ~&	
^ ^~ ~^	
~	
&&	
?: (conditional operator)	lowest precedence

Operators shown on the same row in Table 4-4 have the same precedence. Rows are arranged in order of decreasing precedence for the operators. For example, *, /, and % all have the same precedence, which is higher than that of the binary + and - operators.

All operators associate left to right with the exception of the conditional operator which associate right to left. Associativity refers to the order in which the operators having the

same precedence are evaluated. Thus, in the following example B is added to A and then C is subtracted from the result of A+B.

$$A + B - C$$

When operators differ in precedence, the operators with higher precedence associate first. In the following example, B is divided by C (division has higher precedence than addition) and then the result is added to A.

$$A + B / C$$

Parentheses can be used to change the operator precedence.

$$(A + B) / C \quad // \text{ not the same as } A + B / C$$

4.1.3 Expression evaluation order

The operators follow the associativity rules while evaluating an expression as described in section 4.1.2. However, if the final result of an expression can be determined early, the entire expression need not be evaluated. This is called *short-circuiting* an expression evaluation.

```
integer A, B, C, result ;
result = A & (B | C) ;
```

If A is known to be zero, the result of the expression can be determined as zero without evaluating the sub-expression B | C.

4.1.4 Arithmetic operators

The binary arithmetic operators are the following:

Table 4-5 : Arithmetic operators defined

a + b	a plus b
a – b	a minus b
a * b	a multiply by b
a / b	a divide by b
a % b	a modulo b

The integer division truncates any fractional part toward zero. The modulus operator, for example $y \% z$, gives the remainder when the first operand is divided by the second, and thus is zero when z divides y exactly. The result of a modulus operation takes the sign of the first operand.

For the case of the modulus operator in which either argument is real, the operation performed is:

$$a \% b = a - \text{floor}(a/b)*b;$$

The unary arithmetic operators take precedence over the binary operators. The unary operators are the following:

Table 4-6 : Unary operators defined

+m	unary plus m (same as m)
-m	unary minus m

Table 4-7 gives examples of modulus operations.

Table 4-7 : Examples of modulus operations

Modulus Expression	Result	Comments
10 % 3	1	10/3 yields a remainder of 1
11 % 3	2	11/3 yields a remainder of 2
12 % 3	0	12/3 yields no remainder
-10 % 3	-1	the result takes the sign of the first operand
11 % -3	2	the result takes the sign of the first operand
10 % 3.75	2.5	[10 - floor(10/3.75)*3.75] yields a remainder of 2.5

4.1.5 Relational operators

Table 4-8 lists and defines the relational operators

Table 4-8 : The relational operators defined

a < b	a less than b
a > b	a greater than b
a <= b	a less than or equal to b
a >= b	a greater than or equal to b

An expression using these *relational operators* yields the value 0 if the specified relation is *false*, or the value 1 if it is *true*.

All the relational operators have the same precedence. Relational operators have lower precedence than arithmetic operators.

The following examples illustrate the implications of this precedence rule:

```

a < foo - 1           // this expression is the same as
a < (foo - 1)         // this expression, but . . .
foo - (1 < a)         // this one is not the same as
foo - 1 < a           // this expression

```

When `foo - (1 < a)` evaluates, the relational expression evaluates first and then either zero or one is subtracted from `foo`. When `foo - 1 < a` evaluates, the value of `foo` operand is reduced by one and then compared with `a`.

4.1.6 Equality operators

The *equality operators* rank lower in precedence than the relational operators. Table 4-9 lists and defines the equality operators.

Table 4-9 : The equality operators defined

<code>a == b</code>	a equal to b,
<code>a != b</code>	a not equal to b,

Both equality operators have the same precedence. These operators compare the value of the operands. As with the relational operators, the result will be 0 if comparison fails, 1 if it succeeds.

4.1.7 Logical operators

The operators *logical and* (`&&`) and *logical or* (`||`) are logical connectives. The result of the evaluation of a logical comparison can be 1 (defined as *true*), or 0 (defined as *false*). The precedence of `&&` is greater than that of `||`, and both are lower than relational and equality operators.

A third logical operator is the unary *logical negation* operator `!`. The negation operator converts a non-zero or true operand into 0 and a zero or false operand into 1.

The following expression performs a logical and of three sub-expressions without needing any parentheses:

```
a < param1 && b != c && index != lastone
```

However, it is recommended for readability purposes that parentheses be used to show very clearly the precedence intended, as in the following rewrite of the above example:

```
(a < param1) && (b != c) && (index != lastone)
```

4.1.8 Bit-wise operators

The *bit-wise operators* perform bit-wise manipulations on the operands—that is, the operator combines a bit in one operand with its corresponding bit in the other operand to calculate one bit for the result. The logic tables below show the results for each possible calculation.

Table 4-10 : Bit-wise binary and operator

&	0	1
0	0	0
1	0	1

Table 4-11 : Bit-wise binary or operator

	0	1
0	0	1
1	1	1

Table 4-12 : Bit-wise binary exclusive or operator

^	0	1
0	0	1
1	1	0

Table 4-13 : Bit-wise binary exclusive nor operator

^~ ~^	0	1
0	1	0
1	0	1

Table 4-14 : Bit-wise unary negation operator

~	
0	1
1	0

4.1.9 Shift operators

The *shift operators*, << and >>, perform left and right shifts of their left operand by the number of bit positions given by the right operand. Both shift operators fill the vacated bit positions with zeroes. The right operand is always treated as an unsigned number.

```

integer start, result;
analog begin
    start = 1;
    result = (start << 2);
end

```

In this example, the register `result` is assigned the binary value 0100, which is 0001 shifted to the left two positions and zero filled.

4.1.10 Conditional operator

The *conditional operator*, also known as *ternary operator*, is right associative and must be constructed using three operands separated by two operators with the following syntax:

```

conditional_expression ::=
    expression1 ? expression2 : expression3

```

Figure 4-1: Syntax for conditional operator

The evaluation of a conditional operator begins with the evaluation of `expression1`. If `expression1` evaluates to false (0), then `expression3` is evaluated and used as the result of the conditional expression. If `expression1` evaluates to true (value other than 0), then `expression2` is evaluated and used as the result.

4.1.11 Event or

The event **or** operator performs an or of events. See section 6.7.2 for events and triggering of events.

4.1.12 Concatenations

A concatenation is used for joining scalar elements into compound elements (buses or arrays) for the built-in types of **integer** or **real** or elements declared of type `node`. The concatenation shall be expressed using the brace characters { and }, with commas separating the expressions within.

Example:

```

module x;
  parameter real p1[0:2] = { 1.0, 2.0, 3.0 };
  ...
endmodule

module y;
  parameter real pole1 = 0, pole2 = 0, pole3 = 0;
  x #(.p1({pole1, pole2, pole3}) x1;
  ...
endmodule

```

Module `x` defines a real-array parameter `p1`. Module `y` instantiates `x` and overrides the array value of the parameter `p1` of module `x` via the concatenation of the scalar parameters `pole1`, `pole2`, and `pole3`.

Concatenations can be expressed using a replication multiplier as shown in the following example:

```
{c, {2{a, b}}} // equivalent to: {c, a, b, a, b}
```

The replication multiplier must be a constant expression.

4.2 Built-In Mathematical Functions

Verilog-AMS HDL supports the following standard mathematical functions.

4.2.1 Standard Mathematical Functions

These are the standard mathematical functions supported by Verilog-AMS HDL. The operands must be numeric (integer or real). For **min()**, **max()**, and **abs()**, if either operand is real, both are converted to real, as is the result. All other arguments are converted to real.

Function	Description	Domain
ln (x)	Natural logarithm	$x > 0$
log (x)	Decimal logarithm	$x > 0$
exp (x)	Exponential	$x < 80$
sqrt (x)	Square root	$x \geq 0$
min (x , y)	Minimum	All x , all y
max (x , y)	Maximum	All x , all y
abs (x)	Absolute	All x

Function	Description	Domain
pow (x, y)	Power. x^y	if $x \geq 0$, all y ; if $x < 0$, int(y)
floor (x)	Floor	All x
ceil (x)	Ceiling	All x

The **min**(), **max**(), and **abs**() functions have discontinuous derivatives, and it is necessary to define the behavior of the derivative of these functions at the point of the discontinuity. In that context, these functions are defined such that

min(x, y) is equivalent to $(x < y) ? x : y$

max(x, y) is equivalent to $(x > y) ? x : y$

abs(x) is equivalent to $(x > 0) ? x : -x$

4.2.2 Transcendental Functions

These are the trigonometric and hyperbolic functions supported by Verilog-AMS HDL. All operands must be of the numeric type (**integer** or **real**) and are converted to real if necessary.

All arguments to the trigonometric and hyperbolic functions are specified in radians.

Function	Description	Domain
sin (x)	Sine	All x
cos (x)	Cosine	All x
tan (x)	Tangent	$x \neq n\left(\frac{\pi}{2}\right), n$ is odd
asin (x)	Arc-sine	$-1 \leq x \leq 1$
acos (x)	Arc-cosine	$-1 \leq x \leq 1$
atan (x)	Arc-tangent	All x
atan2 (x, y)	Arc-tangent of x/y	All x , All y
hypot (x, y)	$\sqrt{x^2 + y^2}$	All x , All y
sinh (x)	Hyperbolic sine	$x < 80$
cosh (x)	Hyperbolic cosine	$x < 80$
tanh (x)	Hyperbolic tangent	All x

Function	Description	Domain
asinh (x)	Arc-hyperbolic sine	All x
acosh (x)	Arc-hyperbolic cosine	$x \geq 1$
atanh (x)	Arc-hyperbolic tangent	$-1 \leq x \leq 1$

4.2.3 Error Handling

All math functions not defined for any input must report an error.

4.3 Signal Access Functions

Access functions are used to access signals on nodes, ports, and branches. There are two types of access functions - *signal access functions* and *port access functions*. The name of the access function for a signal is taken from the discipline of the node, port, or branch to which the signal or port is associated and utilizes the function "()" operator. A port access function also takes its name from the discipline of the port to which it is associated but utilizes the port access "<>" operator. If the signal or port access function is used in an expression, the access function returns the value of the signal. If the signal access function is being used on the left side of a branch assignment or contribution statement, it assigns a value to the signal. A port access function cannot be used on the left side of a branch assignment or contribution statement.

The following table shows how access functions can be applied to branches, nodes, and ports. In this table, *b1* refers to a branch, *n1* and *n2* represent either nodes or ports, and *p1* represents a port. These branches, nodes, and ports are assumed to belong to the electrical discipline where V is the name of the access function for the voltage (potential), and I is the name of the access function for the current (flow).

Example	Comments
V(b1)	Accesses the voltage across branch <i>b1</i>
V(n1)	Accesses the voltage of <i>n1</i> (a node or a port) relative to ground
V(n1,n2)	Accesses the voltage difference between <i>n1</i> and <i>n2</i> (nodes or ports)
V(p1,p1)	Error
I(b1)	Accesses the current on branch <i>b1</i>
I(n1)	Accesses the current flowing from <i>n1</i> (a node or port) to ground
I(n1, n2)	Accesses the current flowing between <i>n1</i> and <i>n2</i>

Example	Comments
I(p1, p1)	Error
I(<p1>)	Accesses the current flow into the module through port <i>p1</i>

The argument expression list for signal access functions must be a branch identifier, or a list of one or two node or terminal expressions. If two node expressions are given as arguments to an access function, they must not be the same expressions. The node identifiers must be scalar or resolve to a constant node of a composite node type (array or bus) accessed by a constant expression. The operands of an expression must be unique to define a valid branch. The access function name must match the discipline declaration for the nodes, ports, or branch given in the argument expression list. In this case, V and I were used as examples of access functions for electrical potential and flow.

For port access functions, the expression list is a single port of the module. The port identifier must be scalar or resolve to a constant node of a bus port accessed by a constant expression. The access function name must match the discipline declaration for the port identifier.

4.4 Analog Operators

Analog operators are functions that operate on more than just the current value of their arguments. Rather, they maintain internal state and their output is a function of both the input and the internal state.

Analog operators operate on an expression and return a value.

Analog operators are also referred to as filters. They include the time derivative, time integral, and delay operators from calculus. They also include the transition and slew filters, that are used to remove discontinuity from piecewise constant and piecewise continuous waveforms. Finally they include more traditional filters, such as those described with Laplace and Z-transform descriptions.

One special analog operator is the **limexp()** function, which is a version of the **exp()** function with built-in limits that improves convergence.

4.4.1 Restrictions on analog operators

Analog operators are subject to several important restrictions because they maintain internal state.

- Analog operators must not be used inside conditional statements (**if** and **case**) and looping (**for** and **generate**) unless the conditional expression that controls the statement consists of terms that cannot change their value during the course of an analysis. In particular, the conditional expression can only consist of literal

numerical constants, genvar variables, parameter values, and the **analysis()** function.

- Analog operators are not allowed in the **repeat** and **while** loop statements.
- Analog operators can only be used inside an **analog** block; can not be used inside an **initial** or an **always** block. They cannot be used inside a user defined function.
- It is illegal to specify a null argument in the argument list of an analog operator.

These restrictions are present to prevent use that would cause the internal state to be corrupted or become out-of-date, which results in anomalous behavior.

4.4.2 Vector or Array Arguments to Analog Operators

Certain analog operators require passing of arrays or vectors as parameters (Laplace and Z transform filters, and **noise_table**). The array can be passed as either:

- `array_identifier`
- `const_array_expression`

The *const_array_expression* allows the arrays to be passed within the argument list without explicit declaration of the array object.

The syntax is as follows:

```
constant_array_expression ::=
    { constant_arrayinit_element_list }
constant_arrayinit_element_list
    | constant_arrayinit_element { , constant_arrayinit_element }
constant_arrayinit_element ::=
    | constant_expression
    | integer_constant_expression { constant_expression }
```

Figure 4-2: Syntax for constant array expression

4.4.3 Analog Operators and Equations

Generally, simulators formulate the mathematical description of the system in terms of first-order differential equations and solve them numerically. There is no direct way to solve a set of nonlinear differential equations so iterative approaches are used. When using iterative approaches, one must have criteria used to determine when the algorithm is close enough to the solution to stop the iteration. Tolerances are used for this purpose. Thus, each equation, at minimum, must have a tolerance defined and associated with it.

Occasionally, analog operators require that new equations and new unknowns be introduced by the simulator to convert a module description into a set of first-order differential equations. In this case, the simulator will attempt to determine from context which tolerance should be associated with the new equation and new unknown. Alternatively, these operators allow tolerances to be specified.

Specifying natures directly enforces reusability and allows other signal attributes to be accessed by the simulator.

4.4.4 Time Derivative Operator

The **ddt** operator computes the time derivative of its argument.

Operator	Example	Comments
ddt	ddt (<i>x</i>)	Returns $\frac{d}{dt}x(t)$, the time-derivative of <i>x</i>
	ddt (<i>x</i> , <i>abstol</i>)	Same as above, except absolute tolerance is specified explicitly.
	ddt (<i>x</i> , <i>nature</i>)	Same as above, except nature is specified explicitly.

In DC analysis, **ddt**() returns zero. The optional parameter *abstol* is used as an absolute tolerance if needed. Whether an absolute tolerance is needed depends on the context in which **ddt** is used. See section 4.4.3 for more information on the application of tolerances to equations. The absolute tolerance, *abstol* or derived from *nature*, applies to the output of the **ddt** operator, and is the largest signal level that is considered negligible.

4.4.5 Time Integral Operator

The *idt* operator computes the time-integral of its argument.

Operator	Example	Comments
idt	idt (<i>x</i>)	Returns $\int_0^t x(\tau) d\tau$, the time-integral of <i>x</i> from 0 to <i>t</i> with the initial condition being computed in the DC analysis.
	idt (<i>x</i> , <i>ic</i>)	Returns $\int_0^t x(\tau) d\tau + ic$, the time-integral of <i>x</i> from 0 to <i>t</i> with initial condition <i>ic</i> . In DC analysis, <i>ic</i> is returned.

Operator	Example	Comments
	idt ($x, ic, assert$)	Returns $\int_{t_0}^t x(\tau) d\tau + ic$, the time-integral of x from t_0 to t with initial condition ic . <i>Assert</i> is a integer-valued expression. idt returns ic when <i>assert</i> is nonzero. t_0 is the time when <i>assert</i> last became 0.
	idt ($x, ic, assert, abstol$)	Same as above, except absolute tolerance is specified explicitly.
	idt ($x, ic, assert, nature$)	Same as above, except nature is specified explicitly.

When specified with initial conditions, the **idt**() operator returns the value of the initial condition in DC and IC analyses and whenever *assert* is given and is nonzero. Without initial conditions, **idt** multiplies its argument by infinity in DC analysis. Hence, without initial conditions, it must be used in a system with feedback that forces its argument to zero. The optional parameter *abstol* or *nature* is used to derive an absolute tolerance if needed. Whether an absolute tolerance is needed depends on the context in which **idt** is used. See section 4.4.3 for more information. The absolute tolerance applies to the input of the **idt** operator and is the largest signal level that is considered negligible.

4.4.6 Circular Integrator Operator

The **idtmod** operator, also called the *circular integrator*, converts an expression argument into its indefinitely integrated form similar to **idt** operator.

Operator	Example	Comments
idtmod	idtmod (x)	Returns $\int_0^t x(\tau) d\tau$, the time-integral of x from 0 to t with the initial condition being computed in the DC analysis.
	idtmod (x, ic)	Returns $\int_0^t x(\tau) d\tau + ic$, the time-integral of x from 0 to t with initial condition ic . In DC analysis, ic is returned.

Operator	Example	Comments
	idtmod (<i>x,ic,modulus</i>)	Returns k, where $0 \leq k < \text{modulus}$ and k is such that $\int_0^t x(\tau) d\tau + ic = n * \text{modulus} + k$, $n = \dots -3, -2, -1, 0, 1, 2, 3, \dots$
	idtmod (<i>x,ic,modulus,offset</i>)	Returns k, where $\text{offset} \leq k < \text{offset} + \text{modulus}$ and k is such that $\int_0^t x(\tau) d\tau + ic = n * \text{modulus} + k$
	idtmod (<i>x,ic,assert,abstol</i>)	Same as above, except absolute tolerance is specified explicitly.
	idtmod (<i>x,a,assert,nature</i>)	Same as above, except nature is specified explicitly.

The initial condition is optional. If the initial condition is not specified, it defaults to zero. If **idtmod** is used in a system with feedback configuration that forces expr to zero, the initial condition can be omitted without any unexpected behavior during simulation. For example, an operational amplifier alone needs an initial condition, but the same amplifier with the right external feedback circuitry does not need that forced DC solution.

The initial condition shall force the DC solution to the system.

The output of the **idtmod** function shall remain in the range

$$\text{offset} \leq \mathbf{idtmod} < \text{offset} + \text{modulus}$$

The modulus shall be an expression that evaluates to a positive value. If the modulus is not specified, then **idtmod** shall behave like **idt**, and perform no limiting on the output of the integrator.

The default for offset shall be zero.

The following relationship between **idt** and **idtmod** shall hold at all times.

Let

$$\begin{aligned} y &= \mathbf{idt}(\text{expr}, ic); \\ z &= \mathbf{idtmod}(\text{expr}, ic, \text{modulus}, \text{offset}); \end{aligned}$$

Then

$$y = n * \text{modulus} + z; \quad // n \text{ is an integer}$$

where

$$\text{offset} \leq z < \text{modulus} + \text{offset}$$

In this example, the circular integrator is useful in cases where the integral can get very large, such as a VCO. In a VCO we are only interested in the output values in the range $[0, 2\pi]$,

```
phase = idtmod(fc + gain*V(IN), 0, 1, 0);
```

```
V(OUT) <+ sin(2*M_PI*phase);
```

In the example above, the circular integrator returns a value in the range $[0, 1]$.

4.4.7 Delay Operator

delay implements transport delay for continuous waveforms (use the **transition** operator to delay discrete-valued waveforms). The general form is:

```
delay(input, td [, maxdelay])
```

input is delayed by the amount *td*. In all cases *td* must be a positive number. If the optional *maxdelay* is specified then *td* can vary, but it shall be an error if it becomes larger than *maxdelay*. If *maxdelay* is not specified, changes to *td* shall be ignored. If *maxdelay* is specified, changes to it are ignored and initial value of *maxdelay* is used.

In DC and operating point analyses, **delay()** returns the value of its *input*.

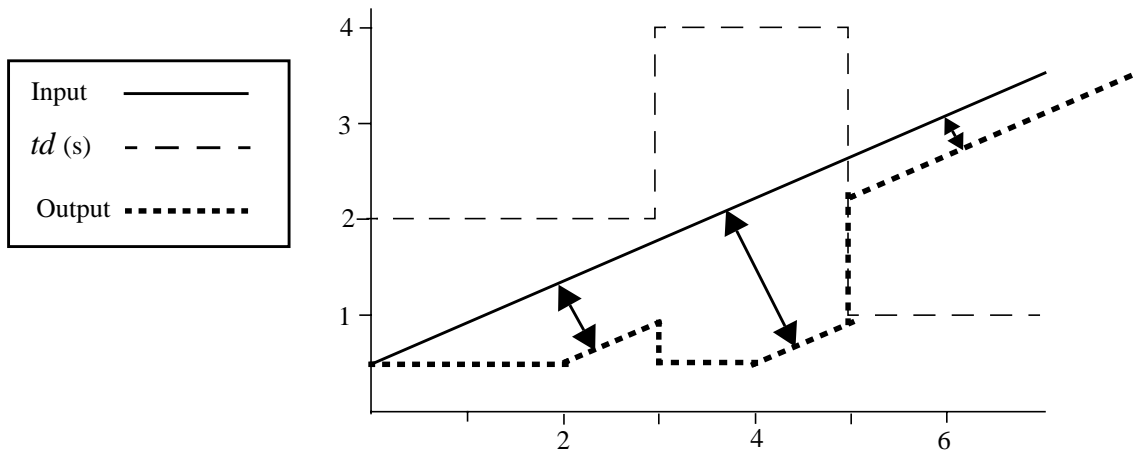
In AC and other small-signal analyses, the delay operator phase-shifts the input expression to the output of the delay operator according to the following:

$$Output(\omega) = Input(\omega) \cdot e^{-j\omega td}$$

In time-domain analyses, delay introduces a transport delay equal to the instantaneous value of *td* according to the following:

$$Output(t) = Input(\max(t - td, 0))$$

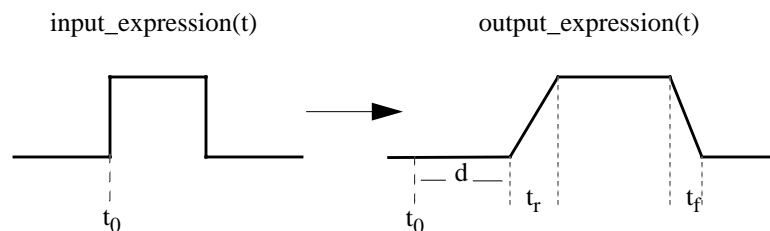
The transport delay, *td*, can be either constant (typical case) or vary as a function of time when *maxdelay* is defined. A time-dependent transport delay is illustrated below with a ramp input to the delay operator for both positive and negative changes in the transport delay *td* and a *maxdelay* of 5.



From time 0 until 2s, the output remains at $\text{input}(0)$. With a delay of 2s, the output then starts tracking the $\text{input}(t - 2)$. At 3s, the transport delay changes from 2s to 4s, switching the output back to $\text{input}(0)$ since $\text{input}(\max(t - td, 0))$ returns 0. The output remains at this level until 4s when it once again starts tracking the input from $t = 0$. At 5s the transport delay goes to 1s, and the output correspondingly jumps from its current value to the value defined by $\text{input}(t - 1)$.

4.4.8 Transition Filter

transition smooths out piece-wise constant waveforms. The transition filter is used to imitate transitions and delays on digital signals. (For non-piecewise-constant signals see **slew**). This function provides controlled transitions between discrete signal levels by setting the rise time and fall time of signal transitions. **transition** stretches instantaneous changes in signals over a finite amount of time, as shown below, and can delay the transitions



The general form is

transition(expression [, delay [, rise_time [, fall_time [, Timetol]]])

transition takes the following arguments (all real-valued expressions):

- The input expression

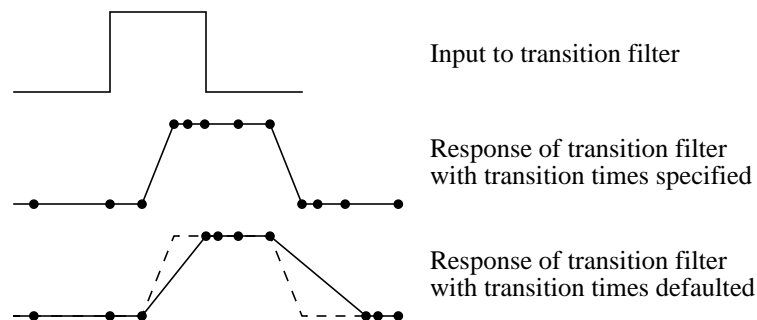
- The delay time (must be nonnegative)
- The rise time (must be greater than or equal to 0)
- The fall time (must be greater than or equal to 0)
- The Timetol (must be positive)

The input expression is expected to evaluate over time to a piecewise constant waveform. When applied, **transition** forces all positive transitions of *expression* to occur over *rise_time* and all negative transitions to occur in *fall_time*, after an initial delay of *delay*. Thus, *delay* models transport delay and *rise_time* and *fall_time* model inertial delay.

transition returns a *real* number that over time describes a piecewise linear function. If *Timetol* is not specified, the transition function causes the simulator to place time-points at both corners of a transition to assure that each transition is adequately resolved. If *Timetol* is specified, the transition function causes the simulator to place time-points at both corners of a transition.

delay, *rise_time*, *fall_time*, and *Timetol* are optional. If *delay* is not specified, it is taken to be zero. If only a positive *rise_time* value is specified, the simulator uses it for both rise and fall times. If neither rise nor fall time are specified or are equal to 0, and *Timetol* is specified, the rise and fall time are taken to be *Timetol*. If neither rise nor fall time are specified or are equal to 0, and *Timetol* is not specified, the rise and fall time are taken to be 1.

The rationale for this behavior is that the default behavior is chosen to approximate the ideal behavior of a zero duration transition. Forcing a zero duration transition is undesirable because it may cause convergence problems. Instead, a negligible, but nonzero, transition time is used. The small nonzero transition time allows the simulator to shrink the timestep small enough to experience a smooth transition if necessary to avoid convergence problems. The transition time compiler directive provides what is



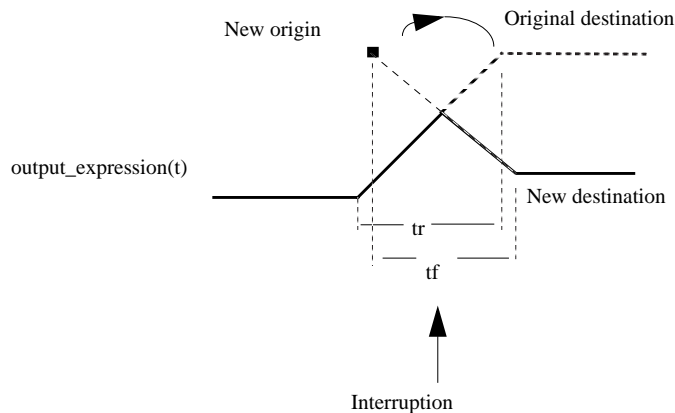
considered a negligible transition time. The simulator does not force a time point at the trailing corner of a transition to avoid causing the simulator to take very small time steps, which would result in poor performance.

In DC analysis, **transition** passes the value of the *expression* directly to its output. The **transition** filter is designed to smooth out piecewise constant waveforms. When applied to waveforms that vary smoothly, the simulation results are generally unsatisfactory. In addition, applying the transition function to a continuously varying waveform can cause the simulator to run slowly. Use **transition** for discrete signals and **slew** for continuous signals.

If interrupted on a rising transition, **transition** tries to complete the transition in the specified time.

- If the new final value level is below the value level at the point of the interruption (the current value), **transition** uses the old destination as the origin.
- If the new destination is above the current level, the first origin is retained.

In the following example, a rising transition is interrupted near its midpoint, and the new destination level of the value is below the current value. For the new origin and destination, **transition** computes the slope that completes the transition from the origin (not the current value) in the specified transition time. It then uses the computed slope to transition from the current value to the new destination.



With different delays, it is possible for a new transition to be specified before a previously specified transition starts. The transition function handles this by deleting any transitions that would follow a newly scheduled transition. A transition function can have an arbitrary number of transitions pending. A transition function can be used in this way to implement transport delay for discrete-valued signals.

Because the transition function cannot be linearized in general, it is not possible to accurately represent a transition function in AC analysis. The AC transfer function is approximately modeled as having unity transmission for all frequencies in all situations. Because the transition function is intended to handle discrete-valued signals, the small signals present in AC analysis rarely reach transition functions. As a result, the approximation used is generally sufficient.

4.4.8.1 QAM Modulator

In this example, the transition function is used to control the rate of change of the modulation signal in a QAM modulator.

```

module qam16(out, in) ;
parameter freq=1.0, ampl=1.0, dly=0, ttime=1.0/freq ;
input [0:4] in ;
output out ;
electrical [0:4] in;
electrical out ;
real x, y ;
integer row, col ;

analog begin
  row = 2*(V(in[3]) > thresh) + (V(in[2]) > thresh) ;
  col = 2*(V(in[1]) > thresh) + (V(in[0]) > thresh) ;
  x = transition(row - 1.5, dly, ttime) ;
  y = transition(col - 1.5, dly, ttime) ;
  V(out) <+ ampl*x*cos(2*M_PI*freq*$realttime)
    + ampl*y*sin(2*M_PI*freq*$realttime) ;
  bound_step(0.1/freq) ;
end
endmodule

```

4.4.8.2 A-D Converter

The following example, an analog behavioral N-bit analog to digital converter, demonstrates the ability of the transition function to handle vectors.

```

module adc(in, clk, out) ;
parameter bits = 8, fullscale = 1.0, dly = 0, ttime = 10n ;
input in, clk ;
output [0:bits-1] out ;
electrical in, clk;
electrical [0:bits-1] out;
real sample, thresh ;
integer result[0:bits-1];
genvar i;

analog begin
  @(cross(V(clk)-2.5, +1)) begin
    sample = V(in);
    thresh = full_scale/2.0;
    for (i = bits - 1; i >= 0; i = i - 1) begin
      if (sample > thresh) begin
        result[i] = 1.0;
        sample = sample - thresh ;
      end else begin
        result[i] = 0.0;
      end
      sample = 2.0*sample;
    end
  end

```

```

        end
    end
    for (i = 0; i < bits; i = i + 1) begin
        V(out) <+ transition(result[i], dly, ttime);
    end
end
endmodule

```

4.4.9 Slew Filter

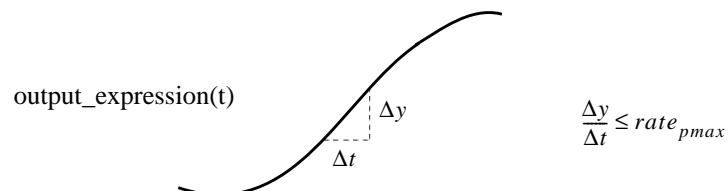
The **slew** analog operator bounds the rate of change (slope) of the waveform. A typical use for **slew** is generating continuous signals from piecewise continuous signals. (For discrete-valued signals, see **transition**.) The general form is

```
slew(expression [, max_pos_slew_rate [, max_neg_slew_rate ] ])
```

slew takes the following arguments (all *real* numbers):

- The input expression
- The maximum positive slew rate
- The maximum negative slew rate

When applied, **slew** forces all transitions of *expression* faster than *max_pos_slew_rate* to change at *max_pos_slew_rate* rate for positive transitions and limits negative transitions to *max_neg_slew_rate* rate



The two rate values are optional. *max_pos_slew_rate* must be greater than 0 and *max_neg_slew_rate* must be less than 0. If only one rate is specified, its absolute value is used for both rates. If no rates are specified, *slew* passes the signal through unchanged. If the rate of change of *expression* is less than the specified maximum slew rates, **slew** returns the value of *expression*. In DC analysis, **slew** simply passes the value of the destination to its output. In AC small-signal analyses, the **slew** function has unity transfer function except when slewing, in which case it has zero transmission through the function.

4.4.10 Last_Crossing Function

Related to the cross function, the **last_crossing** function returns a real value representing the simulation time when a signal expression last crossed 0.

The general form is

```
last_crossing( expression, direction ) ;
```

The *direction* flag is interpreted in the same way as with the **cross** function. The **last_crossing** function is subject to the same usage restrictions as the **cross** function.

The **last_crossing** function does not control the timestep to get accurate results, and uses linear interpolation to estimate the time of the last crossing. However, it can be used with the **cross** function for improved accuracy.

The following example measures the period of its input signal using cross and **last_crossing** functions.

```
module period(in) ;
  input in ;
  voltage in ;
  integer crossings ;
  real latest, previous ;

  analog begin
    @(initial_step) begin
      crossings = 0 ;
      previous = 0 ;
    end

    @(cross(V(in), +1)) begin
      crossings = crossings + 1 ;
      previous = latest ;
    end
    latest = last_crossing(V(in), +1) ;

    @(final_step) begin
      if (crossings < 2)
        $strobe("Could not measure period.") ;
      else
        $strobe("period = %g, crossings = %d",
          latest-previous, crossings) ;
      end
    end
  endmodule
```

Before the expression crosses zero for the first time, the **last_crossing** function returns a negative value.

4.4.11 Laplace Transform Filters

The Laplace transform filters implement lumped linear continuous-time filters. Each filter takes an optional parameter ϵ , which is a real number or a nature used for deriving

an absolute tolerance if needed. Whether an absolute tolerance is needed depends on the context in which the filter is used.

4.4.11.1 `laplace_zp`

laplace_zp implements the zero-pole form of the Laplace transform filter.

laplace_zp(expr, ζ , ρ [, ϵ])

where ζ (zeta) is a vector of M pairs of real numbers. Each pair represents a zero, the first number in the pair is the real part of the zero, and the second is the imaginary part.

Similarly, ρ (rho) is the vector of N real pairs, one for each pole. The poles are given in the same manner as the zeros. The transfer function is

$$H(s) = \frac{\prod_{k=0}^{M-1} \left(1 - \frac{s}{\zeta_k^r + j\zeta_k^i} \right)}{\prod_{k=0}^{N-1} \left(1 - \frac{s}{\rho_k^r + j\rho_k^i} \right)}$$

where ζ_k^r and ζ_k^i are the real and imaginary parts of the k^{th} zero, while ρ_k^r and ρ_k^i are the real and imaginary parts of the k^{th} pole. If a root (a pole or zero) is real, the imaginary part must be specified as 0. If a root is complex, its conjugate must also be present. If a root is zero, then the term associated with it is implemented as s rather than $(1 - s/r)$, where r is the root.

4.4.11.2 `laplace_zd`

laplace_zd implements the zero-denominator form of the Laplace transform filter.

laplace_zd(expr, ζ , d [, ϵ])

where ζ (zeta) is a vector of M pairs of real numbers. Each pair represents a zero, the first number in the pair is the real part of the zero, and the second is the imaginary part.

Similarly, d is the vector of N real numbers that contains the coefficients of the denominator. Its transfer function is

$$H(s) = \frac{\prod_{k=0}^{M-1} \left(1 - \frac{s}{\zeta_k^r + j\zeta_k^i} \right)}{\sum_{k=0}^{N-1} d_k s^k}$$

where ζ_k^r and ζ_k^i are the real and imaginary parts of the k^{th} zero, while d_k is the coefficient of the k^{th} power of s in the denominator. If a zero is real, the imaginary part must be specified as 0. If a zero is complex, its conjugate must also be present. If a zero is zero, then the term associated with it is implemented as s rather than $(1 - s/\zeta)$.

4.4.11.3 `laplace_np`

laplace_np implements the numerator-pole form of the Laplace transform filter.

laplace_np(expr, n, ρ [, ε])

where n is a vector of M real numbers that contains the coefficients of the numerator. Similarly, ρ (rho) is a vector of N pairs of real numbers. Each pair represents a pole, the first number in the pair is the real part of the pole, and the second is the imaginary part. The transfer function is

$$H(s) = \frac{\sum_{k=0}^{M-1} n_k s^k}{\prod_{k=0}^{N-1} \left(1 - \frac{s}{\rho_k^r + j\rho_k^i}\right)}$$

where n_k is the coefficient of the k^{th} power of s in the numerator, while ρ_k^r and ρ_k^i are the real and imaginary parts of the k^{th} pole. If a pole is real, the imaginary part must be specified as 0. If a pole is complex, its conjugate must also be present. If a pole is zero, then the term associated with it is implemented as s rather than $(1 - s/\rho)$.

4.4.11.4 `laplace_nd`

laplace_nd implements the numerator-denominator form of the Laplace transform filter.

laplace_nd(expr, n, d [, ε])

where n is a vector of M real numbers that contains the coefficients of the numerator, and d is a vector of N real numbers that contains the coefficients of the denominator. The transfer function is

$$H(s) = \frac{\sum_{k=0}^M n_k s^k}{\sum_{k=0}^N d_k s^k}$$

where n_k is the coefficient of the k^{th} power of s in the numerator, and d_k is the coefficient of the k^{th} power of s in the denominator.

4.4.11.5 Examples

`V(out) <+ laplace_zp(V(in), {-1,0}, {-1,-1,-1,1});`
implements

$$H(s) = \frac{1 + s}{\left(1 + \frac{s}{1+j}\right)\left(1 + \frac{s}{1-j}\right)}$$

and,

`V(out) <+ laplace_nd(V(in), {0,1}, {-1,0,1});`
implements

$$H(s) = \frac{s}{s^2 - 1}$$

Finally, this example

`V(out) <+ laplace_zp(white_noise(k), , {1,0,1,0,-1,0,-1,0});`
implements a band-limited white noise source as

$$\overline{v_{out}^2} = \frac{k}{|s^2 - 1|^2}$$

4.4.12 Z-Transform Filters

The Z-transform filters implement linear discrete-time filters. Each filter supports the a parameter T that specifies the sampling period of the filter. A filter with unity transfer function acts like a simple sample-and-hold that samples every T seconds and exhibits no delay.

All Z-transform filters share three common arguments, T , τ , and t_0 . T specifies the period of the filter, is mandatory, and it must be positive. τ specifies the transition time, is optional, and must be nonnegative. If the transition time is specified and is nonzero, the timestep is controlled to accurately resolve both the leading and trailing corner of the transition. If it is not specified, the transition time is taken to be one unit of time (as defined by the ‘**timescale**’ compiler directive) and the timestep is not controlled to resolve the trailing corner of the transition. If the transition time is specified as 0, then the output is abruptly discontinuous. It is not recommended that a Z-filter with 0 transition time be directly assigned to a branch. Finally t_0 specifies the time of the first transition, and is also optional. If not given, the first transition occurs at $t=0$.

4.4.12.1 zi_zp

zi_zp implements the zero-pole form of the Z transform filter.

zi_zp(expr, ζ , ρ , T [, τ [, t_0]])

where ζ (zeta) is a vector of M pairs of real numbers. Each pair represents a zero, the first number in the pair is the real part of the zero, and the second is the imaginary part. Similarly, ρ (rho) is the vector of N real pairs, one for each pole. The poles are given in the same manner as the zeros. The transfer function is

$$H(z) = \frac{\prod_{k=0}^{M-1} 1 - z^{-1}(\zeta_k^r + j\zeta_k^i)}{\prod_{k=0}^{N-1} 1 - z^{-1}(\rho_k^r + j\rho_k^i)}$$

where ζ_k^r and ζ_k^i are the real and imaginary parts of the k^{th} zero, while ρ_k^r and ρ_k^i are the real and imaginary parts of the k^{th} pole. If a root (a pole or zero) is real, the imaginary part must be specified as 0. If a root is complex, its conjugate must also be present. If a root is zero, then the term associated with it is implemented as z rather than $(1 - z/r)$, where r is the root.

4.4.12.2 zi_zd

zi_zd implements the zero-denominator form of the Z transform filter.

zi_zd(expr, ζ , d , T [, τ [, t_0]])

where ζ (zeta) is a vector of M pairs of real numbers. Each pair represents a zero, the first number in the pair is the real part of the zero, and the second is the imaginary part. Similarly, d is the vector of N real numbers that contains the coefficients of the denominator. Its transfer function is

$$H(z) = \frac{\prod_{k=0}^{M-1} 1 - z^{-1}(\zeta_k^r + j\zeta_k^i)}{\sum_{k=0}^{N-1} d_k z^{-k}}$$

where ζ_k^r and ζ_k^i are the real and imaginary parts of the k^{th} zero, while d_k is the coefficient of the k^{th} power of s in the denominator. If a zero is real, the imaginary part

must be specified as 0. If a zero is complex, its conjugate must also be present. If a zero is zero, then the term associated with it is implemented as z rather than $(1 - z/\zeta)$.

4.4.12.3 **zi_np**

zi_np implements the numerator-pole form of the Z transform filter.

zi_np(expr, n, ρ, T [, τ [, t₀]])

where n is a vector of M real numbers that contains the coefficients of the numerator. Similarly, ρ (rho) is a vector of N pairs of real numbers. Each pair represents a pole, the first number in the pair is the real part of the pole, and the second is the imaginary part. The transfer function is

$$H(z) = \frac{\sum_{k=0}^{M-1} n_k z^{-k}}{\prod_{k=0}^{N-1} 1 - z^{-1}(\rho_k^r + j\rho_k^i)}$$

where n_k is the coefficient of the k^{th} power of s in the numerator, while ρ_k^r and ρ_k^i are the real and imaginary parts of the k^{th} pole. If a pole is real, the imaginary part must be specified as 0. If a pole is complex, its conjugate must also be present. If a pole is zero, then the term associated with it is implemented as z rather than $(1 - z/\rho)$.

4.4.12.4 **zi_nd**

zi_nd implements the numerator-denominator form of the Z transform filter.

zi_nd(expr, n, d, T [, τ [, t₀]])

where n is a vector of M real numbers that contains the coefficients of the numerator, and d is a vector of N real numbers that contains the coefficients of the denominator. The transfer function is

$$H(z) = \frac{\sum_{k=0}^{M-1} n_k z^{-k}}{\sum_{k=0}^{N-1} d_k z^{-k}}$$

where n_k is the coefficient of the k^{th} power of s in the numerator, and d_k is the coefficient of the k^{th} power of s in the denominator.

4.4.13 Limited Exponential

The **limexp** function is an operator whose internal state contains information about the argument on previous iterations. It returns a real value that is the exponential of its single real argument, however it internally limits the change of its output from iteration to iteration in order to improve convergence. On any iteration where the change in the output of the **limexp** function is bounded, the simulator is prevented from terminating the iteration. Thus, the simulator can only converge when the output of **limexp** equals the exponential of the input. The general form is

limexp (expr)

The apparent behavior of **limexp** is not distinguishable from **exp**, except using **limexp** to model semiconductor junctions generally results in dramatically improved convergence. There are different ways of implementing limiting algorithms for the exponential¹.

4.4.14 Constant vs Dynamic Arguments

Some of the arguments to the analog operators described above and events described in section 6 expect dynamic expressions and some expect their arguments to be constant expressions. The dynamic expressions can be functions of circuit quantities and can change during an analysis. The constant expressions remain static through out an analysis.

Table 4-15 summarizes the arguments of the analog operators defined earlier.

Table 4-15 : Analog operator arguments

Operator	Constant expression arguments	Dynamic expression arguments
ddt	tol	expr
idt	tol	expr, ic, assert
idtmod	tol, modulus, offset	expr, ic
cross	abstol, timetol	expr, dir
last_crossing		expr, dir
delay	max_td	expr, td

1. Laurence W. Nagel, "SPICE2: A computer program to simulate semiconductor circuits," Memorandum No. ERL-M520, University of California, Berkeley, California, May 1975.

W. J. McCalla, *Fundamentals of Computer-Aided Circuit Simulation*. Kluwer Academic Publishers, 1988.

Table 4-15 : Analog operator arguments

Operator	Constant expression arguments	Dynamic expression arguments
transition		expr, td, tr, tf
slew		expr, sr, sf
zi_zp zi_zd zi_np zi_nd	zeros, poles, T, t0	expr, t
laplace_zp laplace_zd laplace_np laplace_nd	poles, tol, zero	expr
bound_step		expr
timer		tstop, period
limexp		expr

If a dynamic expression is passed as an argument that expects a constant expression, then the value of the dynamic expression at the start of the analysis is taken to be the constant value of the argument. Any further change in value of that expression is ignored during the iterative analysis.

4.5 Analysis Dependent Functions

This section describes the **analysis** function, which is used to determine which type of analysis is being performed. The remaining functions are used to implement small-signal sources. The small-signal source functions only affect the behavior of a module during small-signal analyses. The small-signal analyses provided by SPICE include the AC and noise analyses, but others are possible. When not active, the small-signal source functions return 0.

4.5.1 Analysis

The analysis function takes one or more string arguments and returns 1 if any argument matches the current analysis type. Otherwise it returns 0.

analysis(analysis_list)

There is no fixed set of analysis types. Each simulator can support its own set. However, simulators shall use the following types to represent analyses that are similar to those provided by SPICE.

Name	Analysis Description
"ac"	.AC analysis.
"dc"	.OP or .DC analysis.
"noise"	.NOISE analysis.
"tran"	.TRAN analysis.
"ic"	The initial-condition analysis that precedes a transient analysis.
"static"	Any equilibrium point calculation, including a DC analysis as well as those that precede another analysis, such as the DC analysis that precedes an AC or noise analysis, or the IC analysis that precedes a transient analysis.
"nodeset"	The phase during an equilibrium point calculation where nodesets are forced.

Any type names unsupported by a simulator are assumed to not be a match.

Table 4-16 describes the implementation of the analysis function. Each column shows the return value of the function. A status of 1 represents True and 0 represents False.

Table 4-16 Return Values for analysis functions

Analysis	Argument	Simulator Analysis Types					
		DC	TRAN			AC	
			OP	TRAN	UIC	OP	AC
First part of "static"	"nodeset"	1	1	0	?	1	0
Initial DC state	"static"	1	1	0	?	1	0
Initial condition	"ic"	0	1	0	?	0	0
Transfer function	"dc"	1	0	0	?	0	0
Transient	"tran"	0	1	1	?	0	0
Small-signal	"ac"	0	0	0	?	1	1
Noise	"noise"	0	0	0	?	0	0

Using the **analysis** function, it is possible to have a module behave differently depending on which analysis is being run. For example, it is possible to implement nodesets or initial conditions using the analysis function and switch branches.

```
if (analysis("ic"))
    V(cap) <+ initial_value;
else
    I(cap) <+ ddt(C*V(cap));
```

4.5.2 AC Stimulus

A small-signal analysis computes the steady-state response of a system that has been linearized about its operating point and is driven by a small sinusoid. The sinusoidal stimulus is provided using the **ac_stim** function.

```
ac_stim([analysis_name [, mag [, phase]])]
```

The AC stimulus function returns 0 during large-signal analyses (such as DC and transient) as well as on all small-signal analyses with names different from *analysis_name*. The name of a small-signal analysis is implementation dependent, though it is expected that the name of the equivalent of a SPICE AC analysis will be named “ac”, which is the default value of *analysis_name*. When the name of the small-signal analysis matches *analysis_name*, the source becomes active and models a source with magnitude *mag* and phase *phase*. The default magnitude is 1 and the default phase is 0. Phase is given in radians.

4.5.3 Noise

Several functions are provided to support noise modeling during small-signal analyses. To model large-signal noise during transient analyses, use the `$random()` system task. The noise functions are often referred to as noise sources. There are three noise functions, one models white noise processes, another models $1/f$ or flicker noise processes, and the last interpolates a vector to model a process where the spectral density of the noise varies as a piecewise linear function of frequency. The noise functions are only active in small-signal noise analyses, and return 0 otherwise.

4.5.3.1 white_noise

White noise processes are those whose current value is completely uncorrelated with any previous or future values. This implies that their spectral density does not depend on frequency. They are modeled using

```
white_noise(pwr [, name ])
```

where **white_noise** generates white noise with a power of *pwr*. For example, the thermal noise of a resistor could be modelled using

```
I(a,b) <+ V(a,b)/R +
    white_noise(4 * 'P_K * $temperature/R, "thermal");
```

The optional *name* argument acts as a label for the noise source that is used if the simulator outputs the individual contribution of each noise source to the total output noise. The contributions of noise sources with the same name from the same instance of a module are combined in the noise contribution summary.

4.5.3.2 flicker_noise

The **flicker_noise** function models flicker noise.

```
flicker_noise(pwr, exp [ , name ])
```

which generates pink noise with a power of *pwr* at 1Hz that varies in proportion to $1/f^{exp}$.

The optional *name* argument acts as a label for the noise source that is used if the simulator outputs the individual contribution of each noise source to the total output noise. The contributions of noise sources with the same name from the same instance of a module are combined in the noise contribution summary.

4.5.3.3 noise_table

The **noise_table** function interpolates a vector to model a process where the spectral density of the noise varies as a piecewise linear function of frequency.

```
noise_table(vector [ , name ])
```

where *vector* contains pairs of real numbers, the first number in each pair is the frequency in Hertz, and the second is the power. Noise pairs are specified in the order of ascending frequencies. **noise_table** performs piecewise linear interpolation to compute the power spectral density generated by the function at each frequency.

The optional *name* argument acts as a label for the noise source that is used if the simulator outputs the individual contribution of each noise source to the total output noise. The contributions of noise sources with the same name from the same instance of a module are combined in the noise contribution summary.

4.5.3.4 Noise model for diode

The noise of a junction diode could be modelled as follows:

```
I(a,c) <+ is*(exp(V(a,c) / (n * $vt)) - 1)
+ white_noise(2*P_Q*I(<a>))
+ flicker_noise(kf*pow(abs(I(<a>)), af), ef);
```

4.5.3.5 Correlated noise

Each noise function generates noise that is uncorrelated with the noise generated by other functions. Perfectly correlated noise is generated by using the output of one noise function for more than one noise source. Partially correlated noise is generated by combining the output of shared and unshared noise functions.

Consider the case where two noise voltages are perfectly correlated:

```

n = white_noise(pwr);
V(a,b) <+ c1*n;
V(c,d) <+ c2*n;

```

One can also model partially correlated noise sources:

```

n1 = white_noise(1-corr);
n2 = white_noise(1-corr);
n12 = white_noise(corr);
V(a,b) <+ Kv*(n1 + n12);
I(b,c) <+ Ki*(n2 + n12);

```

4.6 User defined functions

The purpose of a user defined function is to return a value that is to be used in an expression. All functions are defined within modules. Each function can be a digital function (as defined in *IEEE 1364-1995*) or an analog function.

4.6.1 Defining an analog function

The syntax for defining an analog function is as follows:

```

analog_function_declaration ::=
analog function [ type ] function_identifier ;
    function_item_declaration { function_item_declaration }
    statement
endfunction

type ::=
    integer
    | real

function_item_declaration ::=
    input_declaration
    | block_item_declaration

block_item_declaration ::=
    parameter_declaration
    | integer_declaration
    | real_declaration

```

Figure 4-3: Syntax for an analog function declaration

An analog function declaration shall begin with the keywords **analog function**, followed by the type of the return value from the function, followed by the name of the function and a semicolon, and shall end with the keyword **endfunction**.

type specifies the return value of the function; its use is optional. *type* can be a **real** or an **integer**; if unspecified, the default is **real**.

An analog function:

- can use any statements available for conditional execution (*see 6.1*);
- shall not use access functions,
- shall not use contribution statements or event control statements;
- shall have at least one input declared;
The block item declaration shall declare the type of the inputs as well as local variables used in the function.
- shall not use named blocks; and
- shall only reference locally-defined variables or variables passed as arguments.

The following example defines an analog function called `maxValue`, which returns potential of the node that is larger in magnitude.

```
analog function real maxValue;
input n1, n2 ;
real n1, n2 ;
begin
    // code to compare potential of two nodes
    maxValue = (n1 > n2) ? n1 : n2 ;
end
endfunction
```

4.6.2 Returning a value from an analog function

The analog function definition implicitly declares a variable, internal to the analog function, with the same name as the analog function. This variable has the same type as the type specified in the analog function declaration. The analog function definition initializes the return value from the analog function by assigning the analog function result to the internal variable with the same name as the analog function. This variable can be read and assigned within the flow; its last assigned value is passed back on the return call.

The following line (from the previous example) illustrates this concept:

```
maxValue = (n1 > n2) ? n1 : n2 ;
```

An analog function definition must include an assignment of the analog function result value to the internal variable that has the same name as the analog function name.

4.6.3 Calling an analog function

An analog function call is an operand within an expression. The analog function call has the following syntax:

```
analog_function_call ::=  
    function_identifier ( expression { , expression } )
```

Figure 4-4: Syntax for function call

The order of evaluation of the arguments to an analog function call is undefined.

An analog function:

- shall not call itself directly or indirectly, that is, recursive functions are not permitted;
- shall only be called within an analog block; and
- can be called outside of their immediate scope.

The following example uses `maxValue` function defined in section 4.6.1

```
V(out) <+ maxValue(val1, val2) ;
```

Section 5

Signals

5.1 Analog Signals

Analog signals are distinguished from digital signals in that an analog signal must have a *discipline* with continuous domain. Disciplines, nodes and branches are described in Section 3, and ports are described in Section 8.

This section describes signal access mechanisms and operators in Verilog-AMS HDL.

5.1.1 Access Functions

Flows and potentials on nodes, ports, and branches are accessed using *access functions*. The name of the access function is taken from the discipline of the node, port, or branch associated with the signal.

For example, consider a named electrical branch *b* where *electrical* is a discipline with *V* as the access function for the potential and *I* as the access function for the flow. The potential (voltage) would be accessed with:

$$V(b)$$

and the flow (current) is accessed with

$$I(b)$$

Unnamed branches are accessed in a similar manner, except that the access functions are applied to nodes or ports rather than branches (terminals of the branch). For example, if *n1* and *n2* are electrical nodes or ports, then

$$V(n1, n2)$$

creates an unnamed branch from *n1* to *n2* if it does not already exist, and then accesses the branch potential (or the potential difference between *n1* to *n2*), and

$$V(n1)$$

does the same from *n1* to **ground**. In other words, accessing the potential from a node or port to a node or port defines an unnamed branch. Accessing the potential on a single node or port defines an unnamed branch from that node or port to **ground**. There can only be one unnamed branch between any two nodes or ports.

An analogous access method is used for flows.

$$I(n1, n2)$$

creates an unnamed branch from *n1* to *n2* if it does not already exist and then accesses the branch flow, and

$I(n1)$

does the same from *n1* to **ground**.

Thus, accessing the flow from a node or port to a node or port defines an unnamed branch. Accessing the potential on a single node or port defines an unnamed branch from that node or port to **ground**.

It is also possible to access the flow passing through a port into a module. The name of the access function is derived from the flow nature of the discipline of the port. However, in this case " $\langle \rangle$ " is used to delimit the port name rather than "()". For example,

$I(\langle p1 \rangle)$

is used to access the current flow into the module through electrical port *p1*. This capability is discussed further in section 5.1.4.

5.1.2 Probes and Sources

It is possible to interpret the behavioral descriptions in Verilog-AMS HDL as a network of probes and controlled sources. While it is not necessary to do so, it is often helpful for two reasons,

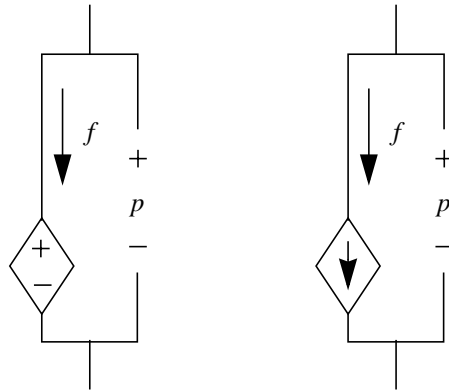
- Describe the component with a network of probes and controlled sources, and then use the simple rules presented here to map the network into a behavioral description.
- Often behavioral descriptions that are difficult to decipher can be more easily understood if it is first converted into a network of probes and controlled sources.

One additional benefit of the probe/source interpretation is that it provides an unambiguous way of defining the behavior for manipulating signals.

5.1.2.1 Sources

A branch, either named or unnamed, is a *source branch* if either the potential or the flow is assigned a value by a contribution statement anywhere in the module. It is a *potential source* if the branch potential is specified, and it is a *flow source* if the branch flow is specified. A branch cannot simultaneously be both a potential and a flow source, though it can switch between them, in which case it is referred to as being a *switch branch*.

Both the potential and the flow of a source branch are accessible in expressions anywhere in the module. The models for potential and flow sources are shown below:



f is a probe that measures the flow through the branch, and p is a probe that measures the potential across the branch.

Figure 5-1: Equivalent circuit models for source branches.

5.1.2.2 Probes

If no value is specified for either the potential or the flow, the branch is a *probe*. If the flow of the branch is used in an expression anywhere in the module, the branch is a *flow probe*, otherwise the branch is a *potential probe*. Using both the potential and the flow of a probe branch is considered illegal. The models for probe branches are shown below



Figure 5-2: Equivalent circuit models for probe branches.

The branch potential of a flow probe is zero. The branch flow of a potential probe is zero.

5.1.3 Examples

The following examples demonstrate how to formulate models and the correspondence between the behavioral description and the equivalent probe/source model.

For simplification, only the node or branch declarations and contribution statements are shown.

5.1.3.1 The Four Controlled Sources

The model for a voltage controlled voltage source is.

```
branch (ps,ns) in;
branch (p,n) out;
V(out) <+ A * V(in);
```

The model for a voltage controlled current source is.

```
branch (ps,ns) in;
branch (p,n) out;
I(out) <+ A * V(in);
```

The model for a current controlled voltage source is.

```
branch (ps,ns) in;
branch (p,n) out;
V(out) <+ A * I(in);
```

The model for a current controlled current source is.

```
branch (ps,ns) in;
branch (p,n) out;
I(out) <+ A * I(in);
```

5.1.3.2 Resistor and Conductor

The model for a linear conductor is

```
branch (p,n) cond;
I(cond) <+ G * V(cond);
```

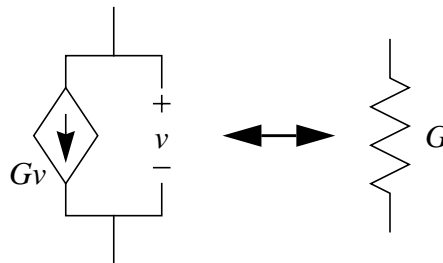


Figure 5-3: Linear conductor model

The assignment to `I(cond)` makes `cond` a current source branch and `V(cond)` simply accesses the optional potential probe built into the current source branch.

The model for a linear resistor is

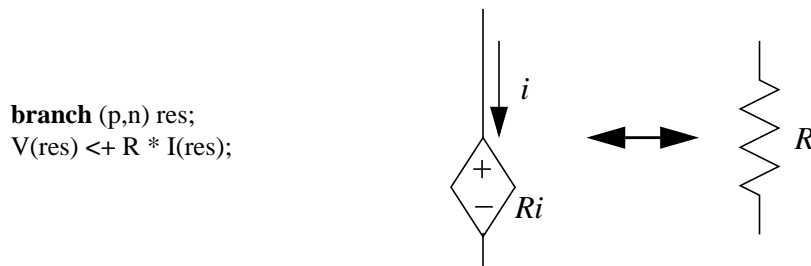


Figure 5-4: Linear resistor model

The assignment to V(res) makes res a potential source branch and I(res) simply accesses the optional flow probe built into the potential source branch.

5.1.3.3 RLC Circuits

A series RLC circuit is formulated by summing the voltage across the three components.

$$v(t) = Ri(t) + L \frac{d}{dt} i(t) + \frac{1}{C} \int_{-\infty}^t i(\tau) d\tau$$

It is described as

$$V(p, n) <+ R * I(p, n) + L * \text{ddt}(I(p, n)) + \text{idt}(I(p, n))/C;$$

A parallel RLC circuit is formulated by summing the currents through the three components.

$$i(t) = \frac{v(t)}{R} + C \frac{d}{dt} v(t) + \frac{1}{L} \int_{-\infty}^t v(\tau) d\tau$$

It is described as

$$I(p, n) <+ V(p, n)/R + C * \text{ddt}(V(p, n)) + \text{idt}(V(p, n))/L;$$

5.1.3.4 Simple Implicit Diode

Verilog-AMS HDL allows components to be described with implicit equations. In the following example, which is a simple diode with a series resistor, the model is implicit because the diode current I(a, c) appears on both sides of the contribution operator. The current of the diode branch is specified, making it a flow source branch. In addition, both the voltage and current of diode branch is used in the behavioral description.

$$I(a, c) <+ \text{is} * (\text{limexp}((V(a, c) - rs * I(a, c)) / \$vt) - 1);$$

5.1.4 Port Branches

The port access function is used to access the flow into a port of a module. The name of the access function is derived from the flow nature of the discipline of the port. However, in this case "<>" is used to delimit the port name. For example,

```
I(<a>)
```

accesses the current through module port *a*.

As one example of how this capability might be used, consider the *junction diode* re-written such that the total diode current is monitored and a message is issued if it exceeds a given value:

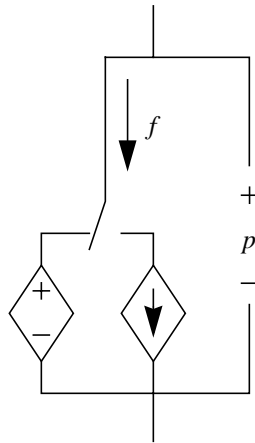
```
module diode (a, c);
  electrical a, c;
  branch (a, c) i_diode, junc_cap;
  parameter real is = 1e-14, tf = 0, cjo = 0, imax = 1, phi = 0.7 ;

  analog begin
    I(i_diode) <+ is*(limexp(V(i_diode)/$vt) - 1);
    I(junc_cap) <+ ddt(tf*I(i_diode) - 2*cjo*sqrt(phi*(phi*V(junc_cap))));
    if (I(<a>) > imax)
      $strobe( "Warning: diode is melting!" );
  end
endmodule
```

The expression V(<a>) is invalid for ports and nodes, where V is a potential access function. The port branch I(<a>) cannot be used on the left side of a contribution operator <+.

5.1.5 Switch Branches

Source branches have the ability to switch between being potential and flow sources. To switch a branch to being a potential source, assign to its potential. To switch a branch to being a flow source, assign to its flow. This type of branch is useful when modeling ideal switches and mechanical stops. The full circuit model for a branch is shown below



Position of the switch depends on whether a potential or flow is assigned to the branch.

Figure 5-5: Circuit model for a source branch.

An ideal relay (a controlled switch) can be implemented as

```

if (closed)
    V(p,n) <+ 0;
else
    I(p,n) <+ 0;

```

A discontinuity of order zero is assumed to occur when the branch switches and so it is not necessary to use the **discontinuity** function with switch branches.

5.1.6 Unassigned Sources

If a value is not assigned to a branch, the branch flow is set to zero.

Consider

```

if (closed)
    V(p,n) <+ 0;

```

This example is equivalent to

```

if (closed)
    V(p,n) <+ 0;
else
    I(p,n) <+ 0;

```

5.2 Signal Access for Vector Branches

Verilog-AMS HDL allows ports, nodes, and branches to be arranged as vectors, however the access functions can only be applied to scalars or individual elements of a vector. The scalar element of a vector is selected with an index. For example,

```
V(in[1])
```

accesses the voltage in[1].

The index must be a genvar expression. If the signal access occurs within the scope of a looping construct, then the index expression may also reference variables declared as genvars.

The following examples illustrate legal applications of access functions to elements of an analog signal vector or buss. In the N-bit DAC example, the access to the analog buss 'in' is done within via a genvar expression of the genvar variable 'i'. In the following fixed-width DAC example, literal values are used to access elements of the buss directly.

```
//
// N-bit DAC example.
//

module dac(out, in, clk);
    parameter integer width = 8 from [2:24];
    parameter real fullscale = 1.0, td = 1n, tt = 1n;
    output out;
    input [1:width] in;
    input clk;
    electrical out;
    electrical [1:width] in;
    electrical clk;

    real aout;
    genvar i;

    analog begin
        @(cross(V(clk) - 2.5, +1)) begin
            aout = 0;
            for (i = width - 1; i >= 0; i = i - 1) begin
                if (V(in[i]) > 2.5) begin
                    aout = aout + fullscale/pow(2, width - i);
                end
            end
        end
        V(out) <+ transition(aout, td, tt);
    end
endmodule

//
// 8-bit fixed-width DAC example.
//
```

```

module dac8(out, in, clk);
  parameter real fullscale = 1.0, td = 1n, tt = 1n;
  output out;
  input [1:8] in;
  input clk;
  electrical out;
  electrical [1:8] in;
  electrical clk;

  real aout;

  analog begin
    @(cross(V(clk) - 2.5, +1)) begin
      aout = 0;
      aout = aout + ((V(in[7]) > 2.5) ? fullscale/2.0 : 0.0);
      aout = aout + ((V(in[6]) > 2.5) ? fullscale/4.0 : 0.0);
      aout = aout + ((V(in[5]) > 2.5) ? fullscale/8.0 : 0.0);
      aout = aout + ((V(in[4]) > 2.5) ? fullscale/16.0 : 0.0);
      aout = aout + ((V(in[3]) > 2.5) ? fullscale/32.0 : 0.0);
      aout = aout + ((V(in[2]) > 2.5) ? fullscale/64.0 : 0.0);
      aout = aout + ((V(in[1]) > 2.5) ? fullscale/128.0 : 0.0);
      aout = aout + ((V(in[0]) > 2.5) ? fullscale/256.0 : 0.0);
    end

    V(out) <+ transition(aout, td, tt);
  end

endmodule

```

The syntax for analog signal access is as follows:

```

access_function ::=
    bvalue
    | pvalue
bvalue ::=
    access_identifier ( analog_signal_list )
analog_signal_list ::=
    branch_identifier
    | array_branch_identifier [ genvar_expression ]
    | node_or_port_scalar_expression
    | node_or_port_scalar_identifier , node_or_port_scalar_identifier
node_or_port_scalar_expression ::=
    node_or_port_identifier
    | array_node_or_port_identifier [ genvar_expression ]
    | vector_node_or_port_identifier [ genvar_expression ]
pvalue ::=
    flow_access_identifier < port_scalar_expression >
port_scalar_expression ::=
    port_identifier
    | array_port_identifier [ genvar_expression ]
    | vector_port_identifier [ genvar_expression ]

```

Figure 5-6: Syntax for scalar selection of vector signals

5.3 Contribution statements

Verilog-AMS HDL defines the *branch contribution operator* “<+” for the description of analog behavior. This operator is only valid within the *analog block*. Branch contribution statements are statements that use the branch contribution operators to describe behavior in terms of a mathematical mapping of input signals to output signals.

5.3.1 Branch Contribution Statements

In general, a branch contribution statement consists of two parts, a left-hand side, and a right-hand side separated by a branch contribution operator. The right-hand side can be any expression that evaluates or can be promoted to a real value. The left-hand side specifies the source branch signal that the right-hand side is to be assigned to. It must consist of a signal access function applied to a branch. Hence, analog behaviors can be described using:

```

V(n1, n2) <+ expression ;
or

```

```
I(n1, n2) <+ expression ;
```

where (n1, n2) represents an unnamed source branch, and V(n1,n2) refers to the potential on the branch while I(n1,n2) refers to the flow through the branch. The expression can be linear, nonlinear, or dynamic in nature. The left-hand side can not use a port access function.

This is illustrated in the following modules, which model a resistor and a capacitor.

```
module resistor(p, n);
    electrical p, n;
    parameter real r = 0;

    analog
        V(p,n) <+ r*I(p, n);

endmodule

module capacitor(p, n);
    electrical p, n;
    parameter real c = 0;

    analog
        I(p,n) <+ c*ddt(V(p, n));

endmodule
```

Branch contribution statements implicitly define source branch relations. The branch is directed from the first node of the access function to the second node. If the second node is not specified, **ground** is taken as the reference node.

A branch relation is a path of the flow between two nodes in a module. Each node has two signals associated with it—the potential of the node and the flow out of the node. In electrical circuits, the potential of a node is its voltage, whereas the flow out of the node is its current. Similarly, each branch has two signals associated with it—the potential across the branch and the flow through the branch.

For example, the following module models a simple single-ended amplifier.

```
module amp(out, in);

    input in;
    output out;
    electrical out, in;
    parameter real gain = 1;

    analog
        V(out) <+ gain*V(in);

endmodule
```

For source branch contributions, the statement is evaluated as follows:

1. The simulator evaluates the right-hand side.
2. The simulator adds the value of the right-hand side to any previously retained value for the branch for later assignment to the branch. If there are no previously retained values, the value of the right-hand side itself is retained.

3. At the end of the simulation cycle, the simulator assigns the retained value to the source branch.

Parasitics are added to the above amplifier by simply adding additional contribution statements to model the input admittance and output impedance.

```
module amp(out, in);
  input in;
  output out;
  electrical out, in;
  parameter real Gain = 1, Rin = 1, Cin = 1, Rout = 1, Lout = 1;

  analog begin
    // gain of amplifier
    V(out) <+ gain*V(in);

    // model input admittance
    I(in) <+ V(in)/Rin;
    I(in) <+ Cin*ddt(V(in));

    // model output impedance
    V(out) <+ Rout*I(out);
    V(out) <+ Lout*ddt(I(out));
  end
endmodule
```

Contributing a flow to a branch that already has a value retained for the potential results in the potential being discarded and the branch being converted to a flow source. Conversely, contributing a potential to a branch that already has a value retained for the flow results in the flow being discarded and the branch being converted into a potential source.

This is used to model switches, as shown in the following example:

```
module switch(p, n, cp, cn);
  electrical p, n, cp, cn;
  parameter real thresh = 0;

  analog begin
    // stop to resolve threshold crossings
    @(cross(V(cp,cn) - thresh, 0));

    if (V(cp,cn) > thresh)
      V(p,n) <+ 0;
    else
      I(p,n) <+ 0;
    end
  endmodule
```

The syntax for source contribution statement is shown below:

```
branch_contribution ::=
    bvalue <+ expression ;
```

Figure 5-7: Syntax for branch contribution

5.3.2 Indirect Branch Assignments

Verilog-AMS HDL allows descriptions that implicitly specify a branch voltage or current in fixed-point form. The branch voltage or current is assigned a value by an expression that uses the branch voltage or current. This occurred in the simple implicit diode model above where $I(a,c)$ appeared on both sides of the contribution operator.

Consider the model for an ideal opamp. In this model, the output is driven to the voltage that results in the input voltage being zero. The constitutive equation is

$$V(\text{in}) == 0$$

This can be formulated as

$$V(\text{out}) <+ V(\text{out}) + V(\text{in});$$

This statement defines the output of the opamp to be a controlled voltage source by assigning to $V(\text{out})$ and defines the input to be high impedance by only probing the input voltage. The desired behavior results because the description is formulated in such a way that it reduces to $V(\text{in}) = 0$. This approach does not result in the right tolerances being applied to the equation if out and in have different disciplines.

Verilog-AMS HDL includes a special syntax that is appropriate in this situation. The above branch contribution can be rewritten using an *indirect branch assignment*:

$$V(\text{out}): V(\text{in}) == 0;$$

which reads “find $V(\text{out})$ such that $V(\text{in}) == 0$ ”. It indicates that out should be driven with a voltage source and the source voltage should be such that the given equation is satisfied. Any branches referenced in the equation are only probed and not driven. In particular, $V(\text{in})$ acts as a voltage probe.

A complete description of an ideal opamp is shown below:

```
module opamp(out, pin, nin);
    electrical out, pin, nin;
    analog
        V(out):V(pin,nin) == 0;
endmodule
```

The syntax for the indirect assignment statement is

```

indirect_branch_assignmentment ::=
    target : equation ;

target ::=
    bvalue

equation ::=
    nexpr == expression

nexpr ::=
    bvalue
    | ddt ( bvalue )
    | idt ( bvalue )

```

Figure 5-8: Syntax for indirect branch assignment

If there are multiple indirect assignments statements, it is often the case that the targets can be paired with any equation. Consider the following ordinary differential equation,

$$\frac{dx}{dt} = f(x, y, z)$$

$$\frac{dy}{dt} = g(x, y, z)$$

$$\frac{dz}{dt} = h(x, y, z)$$

which can be written as

```

V(x): ddt(V(x)) == f(V(x), V(y), V(z));
V(y): ddt(V(y)) == g(V(x), V(y), V(z));
V(z): ddt(V(z)) == h(V(x), V(y), V(z));

```

or

```

V(y): ddt(V(x)) == f(V(x), V(y), V(z));
V(z): ddt(V(y)) == g(V(x), V(y), V(z));
V(x): ddt(V(z)) == h(V(x), V(y), V(z));

```

or

```

V(z): ddt(V(x)) == f(V(x), V(y), V(z));
V(x): ddt(V(y)) == g(V(x), V(y), V(z));
V(y): ddt(V(z)) == h(V(x), V(y), V(z));

```

without affecting the results.

5.3.2.1 Indirect Assignment and Contribution

Indirect assignment is incompatible with contribution. Once a value is indirectly assigned to a branch, it cannot be contributed to using the branch contribution operator (`<+`).

Section 6

Analog Behavior

The description of an analog behavior consists of setting up contributions (Section 5) for various nodes under certain procedural or timing control. This section describes an analog procedural block, procedural control statements and analog timing control functions.

6.1 Analog procedural block

Discrete time behavioral definitions within Verilog HDL are encapsulated within the **initial** and **always** procedural blocks. Every **initial** and **always** block starts a separate concurrent activity flow. For continuous time simulation, the behavioral description is encapsulated within the **analog** procedural block. Verilog-AMS HDL allows one analog procedural block in a module definition.

The **analog** procedural block defines the behavior as a procedural sequence of statements. The conditional and looping constructs are available for defining behaviors within the **analog** procedural block. Because the description is a continuous-time behavioral description, no blocking event control statements (such as blocking delays, events or waits) are supported.

The statements allowed within the analog block (Figure 6.1) are separated into two categories - *analog_statements* and *non_analog_statements*. The *analog_statements* are restricted to the **analog** block whereas the *non_analog_statements* can appear anywhere within the module scope, including an **analog** block. The distinction is based upon the visibility and usage of these behavioral constructs within a Verilog-AMS module definition.

The syntax for analog block is as follows:

```

analog_block ::=
    analog analog_statement

analog_statement ::=
    null_statement
    | analog_block_statement
    | analog_branch_contribution
    | analog_indirect_branch_assignment
    | analog_procedural_assignment
    | analog_conditional_statement
    | analog_for_statement
    | analog_case_statement
    | analog_event_controlled_statement
    | discontinuity_task
    | bound_step_task
    | system_task_enable
    | non_analog_statement

non_analog_statement ::=
    | block_statement
    | procedural_assignment
    | conditional_statement
    | loop_statement
    | case_statement

```

Figure 6-1: Syntax for analog procedural block

The statements within the analog block are used to define the continuous-time behavior of the module. The behavioral description is a mathematical mapping of input signals to output signals. The mapping is done with contribution statements of the form

```
signal <+ analog_expression ;
```

or indirect branch assignment. The *analog_expression* can be any combination of linear, nonlinear, or differential expressions of module signals, constants and parameters (see Section 5).

All analog blocks contained in various modules in a design are considered to be executing concurrent with respect to each other.

6.2 Block statements

The *block statements*, also referred to as *sequential blocks*, are a means of grouping two or more statements together so that they act syntactically like a single statement. The block statements are delimited by the keywords **begin** and **end**. The procedural statements in a block statement are executed sequentially in the given order.

The following is the formal syntax for sequential blocks:

```
block_statement ::=
    begin [ : block_identifier { block_item_declaration } ]
        { statement }
    end

analog_block_statement ::=
    begin [ : block_identifier { block_item_declaration } ]
        { analog_statement }
    end

block_item_declaration ::=
    parameter_declaration
    | integer_declaration
    | real_declaration
```

Figure 6-2: Syntax for the sequential blocks

An `analog_block_statement` is a `block_statement` that encapsulates one or more `analog_statements`.

6.2.1 Block names

A sequential block can be named by adding `: name_of_block` after the keyword **begin**. The naming of a block allows local variables to be declared for the block.

All local variables are static—that is, a unique location exists for all variables and leaving or entering blocks do not affect the values stored in them.

The block names give a means of uniquely identifying all variables at any simulation time.

6.3 Procedural assignments

In Verilog-AMS HDL, the branch contributions and indirect branch assignments are used for modifying signals. The procedural assignments are used for modifying integer and real variables. The syntax for procedural assignments are as follows:

```

procedural_assignment ::=
    lexpr = expression ;

analog_procedural_assignment ::=
    lexpr = analog_expression ;

lexpr ::=
    integer_identifier
    | real_identifier
    | array_element

array_element ::=
    integer_identifier [ expression ]
    | real_identifier [ expression ]

```

Figure 6-3: Syntax for procedural assignments

The left-hand side of a procedural assignment must be an integer or a real identifier or an element of an integer or real array. The right-hand side expression can be any arbitrary expression constituted from legal operands and operators as described in Section 4.

An `analog_procedural_assignment` is defined as a procedural assignment whose right-hand side expression is an `analog_expression` involving analog operators.

6.4 Conditional statement

The *conditional statement* (or *if-else* statement) is used to make a decision as to whether a statement is executed or not. The syntax of a conditional statement is as follows:

```

conditional_statement ::=
    if ( expression ) true_statement
    [ else false_statement ]

```

Figure 6-4: Syntax of conditional statement

If the expression evaluates to true (that is, has a non-zero value), the *true_statement* will be executed. If it evaluates to false (has a zero value), the *true_statement* will not be executed. If there is an **else** *false_statement* and expression is false, the *false_statement* will be executed.

Since the numeric value of the if expression is tested for being zero, certain shortcuts are possible. For example, the following two statements express the same logic:

```

if (expression)
if (expression != 0)

```

Because the **else** part of an **if-else** is optional, there can be confusion when an **else** is omitted from a nested **if** sequence. This is resolved by always associating the **else** with the closest previous **if** that lacks an **else**. In the example below, the **else** goes with the inner **if**, as shown by indentation.

```

if (index > 0)
  if (i > j)
    result = i;
  else           // else applies to preceding if
    result = j;

```

If that association is not desired, a *begin-end block statement* must be used to force the proper association, as shown below.

```

if (index > 0) begin
  if (i > j)
    result = i;
end
else result = j;

```

Nesting of **if** statements (known as an *if-else-if* construct) is the most general way of writing a multi-way decision. The expressions are evaluated in order; if any expression is true, the statement associated with it will be executed, and this will terminate the whole chain. Each statement is either a single statement or a sequential block of statements.

6.4.1 Analog Conditional Statements

Analog conditional statements are syntactically equivalent to conditional statements except that the true and/or false statement are *analog_statements*. The conditional expression must be a *genvar_expression*. See the discussion in section 4.4.1 regarding restrictions on the usage of analog operators.

```

analog_conditional_statement ::=
  if ( genvar_expression ) true_analog_statement
  [ else false_analog_statement ]

```

Figure 6-5: Syntax of analog conditional statement

6.5 Case statement

The *case statement* is a multi-way decision statement that tests whether an expression matches one of a number of other expressions, and branches accordingly. The case statement has the following syntax:

```

case_statement ::=
    case ( expression ) case_item { case_item } endcase

case_item ::=
    expression { , expression } : statement
    | default [ : ] statement

```

Figure 6-6: Syntax for case statement

The *default* statement is optional. Use of multiple default statements in one case statement is illegal.

The case expression and the case item expression can be computed at runtime; neither expression is required to be a constant expression.

The *case_item_expressions* are evaluated and compared in the exact order in which they are given. During the linear search, if one of the case item expressions matches the case expression given in parentheses, then the statement associated with that case item is executed. If all comparisons fail, and the default item is given, then the default item statement is executed. If the default statement is not given, and all of the comparisons fail, then none of the case item statements are executed.

6.5.1 Analog case statements

Analog case statements are syntactically equivalent to case statements except the case item statements can also be analog_statements. The conditional expression must be a genvar expression. See the discussion in section 4.4.1 regarding restrictions on the usage of analog operators.

```

analog_case_statement ::=
    case ( analog_expression ) case_item { case_item } endcase

case_item ::=
    analog_expression { , analog_expression } : analog_statement
    | default [ : ] analog_statement

```

Figure 6-7: Syntax for analog case statement

6.5.2 Constant expression in case statement

A constant expression can be used for case expression. The value of the constant expression shall be compared against case item expressions.

The following example demonstrates the usage by modeling a 3-bit priority encoder.

```
integer [2:0] encode ;

case (1)
  encode[2] : $display("Select Line 2") ;
  encode[1] : $display("Select Line 1") ;
  encode[0] : $display("Select Line 0") ;
  default $strobe("Error: One of the bits expected ON");
endcase
```

Note that the case expression is a constant expression (1). The case items are expressions (array elements), and are compared against the constant expression for a match.

6.6 Looping statements

There are four types of looping statements - **repeat**, **while**, **for** and **generate**. These statements provide a means of controlling the execution of a statement zero, one, or more times.

for and **generate** are the only looping statements that can be used to describe analog behaviors using analog operators.

```
looping_statement ::=
  repeat_statement
  | while_statement
  | for_statement
  | analog_for_statement
  | generate_statement
```

Figure 6-8: Syntax for the looping statements

6.6.1 Repeat and while statements

repeat executes a statement a fixed number of times. Evaluation of the expression decides how many times a statement is executed.

while executes a statement until an expression becomes false. If the expression starts out false, the statement is not executed at all.

The repeat and while expression must be evaluated once before the execution of any statement in order to determine the number of times, if any, the statements will be executed. The syntax for **repeat** and **while** statements is shown below:

```
repeat_statement ::=
    repeat ( expression ) statement

while_statement ::=
    while ( expression ) statement
```

Figure 6-9: Syntax for repeat and while statements

6.6.2 For statements

The **for** statement is a looping construct that controls execution of its associated statement(s) using an index variable. If the associated statement is an analog_statement, then the control mechanism must consist of genvar_assignments and genvar_expressions to adhere to the restrictions associated with the use of analog operators. If the associated statements are not analog_statements, the **for** statement may use procedural assignments and expressions, including genvar_expressions.

The **for** statement controls execution of its associated statement(s) by a three-step process, as follows:

1. executes an assignment normally used to initialize an integer that controls the number of loops executed
2. evaluates an expression—if the result is zero, the for-loop exits, and if it is not zero, the for-loop executes its associated statement(s) and then perform step 3.
3. executes an assignment normally used to modify the value of the loop-control variable, then repeats step 2 above.

The following shows the syntax for the two forms of the **for** statements:

```
for_statement ::=
    for ( procedural_assignment ; expression ;
        procedural_assignment ) statement

analog_for_statement ::=
    for ( genvar_assignment ; genvar_expression ;
        genvar_assignment ) analog_statement
```

Figure 6-10: Syntax for the for statements

Analog operators are not allowed in the **repeat**, **while** and **for** looping statements. They are allowed in *analog_for* and *generate* statements.

The *analog_for* statements are syntactically equivalent to the **for** statements except that associated statement is also an analog statement (which contains analog operations). The analog statement puts the additional restriction upon the procedural assignment and conditional expressions of the for loop such that they be statically evaluable. Verilog-AMS HDL provides genvar-derived expressions for this purpose.

Example:

```

module genvarexp(out, dt);
  parameter integer width = 1;
  output out;
  input dt[1:width];
  electrical out;
  electrical dt[1:width];
  integer i;
  genvar k;
  real tmp;

  analog begin
    tmp = 0.0;
    for (k = 1; k <= width; k = k + 1) begin
      tmp = tmp + V(dt[k]);
      V(out) <+ ddt(V(dt[k]));
    end
  end
endmodule

```

See the discussion in section 4.4.1 regarding additional information on restrictions on the usage of analog operators.

6.7 Events

The analog behavior of a component can be controlled using events. The events have the following characteristics:

1. events have no time duration
2. events can be triggered and detected in different parts of the behavioral model.
3. events do not block the execution of an analog block
4. events can be detected using @ *operator*
5. events do not hold any data

There are both digital and analog events. There are two types of analog events - *global events* (6.7.4) and *monitored events* (6.7.5). Null arguments are not allowed in analog events.

6.7.1 Event detection

Analog event detection consists of an event expression followed by a procedural statement. It takes the form:

```
event_controlled_statement ::=
    @ ( event_expression ) statement
event_expression ::=
    simple_event [ or event_expression ]
simple_event ::=
    global_event
    | event_function
```

Figure 6-11: Syntax for event detection

The procedural statement following the event expression is executed whenever the event described by the expression changes. The analog event detection is non-blocking, meaning that the execution of the procedural statement is skipped unless the analog event has occurred. The event expression consists of one or more signal names, global events, or monitored events separated by **or** operator.

The parenthesis around the event expression are required.

6.7.2 Event OR operator

The "OR-ing" of any number of events can be expressed such that the occurrence of any one of the events trigger the execution of the procedural statement that follows it. The keyword **or** is used as an event or operator.

For example,

```
analog begin
    @(initial_step or cross(V(smpl)-2.5,+1)) begin
        vout = (V(in) > 2.5);
    end
    V(out) <+ vout;
end
```

Here, **initial_step** is a global event and **cross()** returns a monitored event. The variable **vout** is set to 0 or 1 whenever one of the two events occur.

6.7.3 Event Triggered Statements

The following two restrictions apply to the statements evaluated as a result of an event being triggered.

- The statement can not have expressions that use analog operators. These statements can not maintain their internal state. This is because they are executed intermittently, only when the corresponding event is triggered.
- The statement can not be a contribution statement because it could generate discontinuity in analog signals.

6.7.4 Global events

The global events are generated by the simulator at various stages of the simulation. The user model can not generate these events. These events are detected by using the name of the global event in an event expression with the @ operator.

The global events are pre-defined in Verilog-AMS HDL. These events can not be redefined in a model.

The following are pre-defined global events:

```
global_event ::=
    initial_step [ ( analysis_list ) ]
    | final_step [ ( analysis_list ) ]
analysis_list ::=
    analysis_name { , analysis_name }
analysis_name ::=
    " analysis_identifier "
```

Figure 6-12: Global events

The **initial_step** and **final_step** generate global events on the first and the last point in an analysis respectively. They are useful when performing actions that should only occur at the beginning or the end of an analysis. Both global events can take optional arguments, consisting of an analysis list for which the global event is active. For example,

```
@(initial_step("ac", "dc"))           // active for dc and ac only
@(initial_step("tran"))               // active for transient only
```

Table 6-1 describes the return value of `initial_step` and `final_step` for standard analysis. Each column shows the return on event status. A status of 1 represents Yes and 0 represents No. A Verilog-AMS simulator can use any or all of these typical analysis types.

Additional analysis names can also be used as necessary for specific implementations. (See section 4.5.1 for further details.)

Table 6-1 Return Values for `inital_step` and `final_step`

Analysis ^a	DC	TRAN	AC	NOISE
	p1 p2 pN	OP p1 pN	OP p1 pN	OP p1 pN
<code>initial_step()</code>	1 0 0	1 0 0	1 0 0	1 0 0
<code>initial_step("ac")</code>	0 0 0	0 0 0	1 0 0	0 0 0
<code>initial_step("noise")</code>	0 0 0	0 0 0	0 0 0	1 0 0
<code>initial_step("tran")</code>	0 0 0	1 0 0	0 0 0	0 0 0
<code>initial_step("dc")</code>	1 0 0	0 0 0	0 0 0	0 0 0
<code>initial_step(unknown)</code>	0 0 0	0 0 0	0 0 0	0 0 0
<code>final_step()</code>	0 0 1	0 0 1	0 0 1	0 0 1
<code>final_step("ac")</code>	0 0 0	0 0 0	0 0 1	0 0 0
<code>final_step("noise")</code>	0 0 0	0 0 0	0 0 0	0 0 1
<code>final_step("tran")</code>	0 0 0	0 0 1	0 0 0	0 0 0
<code>final_step("dc")</code>	0 0 1	0 0 0	0 0 0	0 0 0
<code>final_step(unknown)</code>	0 0 0	0 0 0	0 0 0	0 0 0

a. pX designates analysis point X, X = 1 to N; OP designates the Operating Point.

The following example measures the bit-error rate of a signal and prints the result at the end of the simulation.

```

module bitErrorRate (in, ref) ;
  input in, ref ;
  electrical in, ref ;
  parameter real period=1, thresh=0.5 ;
  integer bits, errors ;

  analog begin
    @(initial_step) begin
      bits = 0 ;
      errors = 0 ;
    end

    @(timer(0, period)) begin
      if ((V(in) > thresh) != (V(ref) > thresh))
        errors = errors + 1 ;
      bits = bits + 1 ;
    end
  end

```

```

    @(final_step)
    $strobe("bit error rate = %f%%", 100.0 * errors / bits );
end
endmodule

```

The **initial_step** and **final_step** events take a list of quoted strings as optional arguments. The strings are compared to the name of the analysis being run. If any string matches the name of the current analysis name, then the simulator generates an event on the first point and the last point of that particular analysis, respectively.

If no analysis list is specified, then the global event is only active during a transient analysis. This is the default case. During a transient analysis the **initial_step** global event is active during the solution of the first timepoint (or initial DC analysis). Similarly the **final_step** global event is active during the solution of the last timepoint of a transient analysis.

6.7.5 Monitored events

The monitored events are detected using event functions with the @ operator. The triggering of the monitored event is implicit due to change in signals, simulation time, or other runtime conditions.

```

event_function ::=
    cross_function
    | timer_function

```

Figure 6-13: Monitored events

6.7.5.1 Cross Function

The **cross** function is used for generating a monitored analog event to detect threshold crossings in analog signals. The **cross** function generates events when the expression crosses zero in the specified direction. In addition, **cross** controls the timestep to accurately resolve the crossing.

The general form is

```
cross ( expression [ , direction [ , time_tol [ , expression_tol ] ] ] );
```

where *expression* is required, and *direction*, *time_tol*, and *expression_tol* are optional. All arguments are real expressions, except *direction* (which is an integer expression). If the tolerances are not defined, then the tool (e.g., the simulator) sets them. If either or both tolerances are defined, then the direction shall also be defined.

The *direction* indicator can only evaluate to +1, -1, or 0. If it is set to 0 or is not specified, the event and timestep control occur on both positive and negative crossings of the signal. If *direction* is +1 (or -1), the event and timestep control only occurs on rising edge

(falling edge) transitions of the signal. For any other transitions of the signal, the cross function does not generate an event.

The definition of *expression_tol* and *time_tol* are shown in Figure 6-14. They represent the maximum allowable error between the estimated crossing point and the true crossing point.

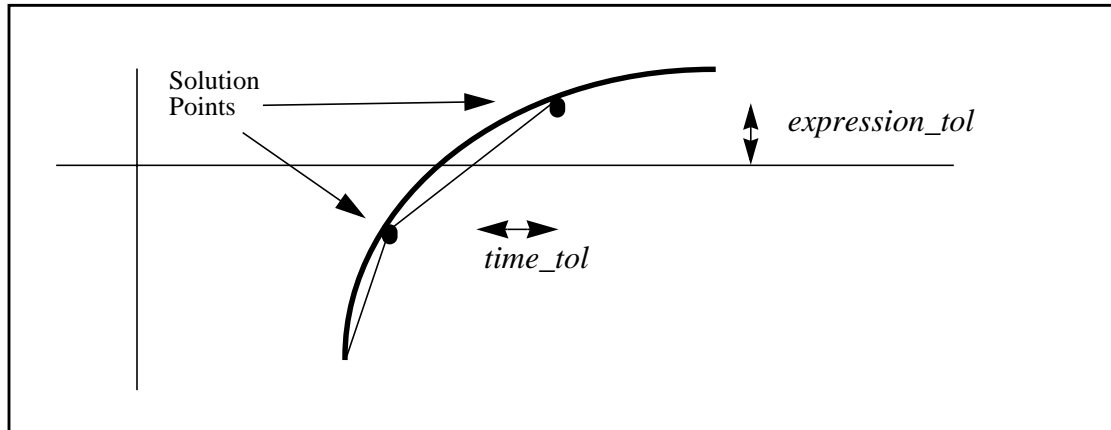


Figure 6-14: Relationship between time tolerance and expression tolerance

If *expression_tol* is defined, the *time_tol* must also be defined and both tolerances must be satisfied at the crossing.

The following description of a sample-and-hold illustrates how the *cross* function might be used.

```

module sh (in, out, smpl) ;
    output out ;
    input in, smpl ;
    electrical in, out, smpl ;
    real state ;

    analog begin
        @(cross(V(smpl) - 2.5, +1))
            state = V(in) ;
        V(out) <+ transition(state, 0, 10n) ;
    end
endmodule

```

The cross function maintains internal state and has the same restrictions as analog operators. In particular, it must not be used inside a conditional statement (**if** and **case**) unless the conditional expression is a genvar expression. In addition, **cross** is not allowed in the repeat, while, and while iteration statements. It is allowed in the *analog_for* statements

6.7.5.2 Timer Function

The **timer** function is used to generate analog events to detect specific points in time.

The general form is

```
timer ( start_time [ , period [ , Timetol ] ] ) ;
```

where *start_time* is required; *period* and *Timetol* are optional arguments. All arguments are real expressions. *Timetol* is set by the simulator to provide adequate resolution.

The **timer** function schedules an event that occurs at an absolute time (as specified by *start_time*). The analog simulator places a time point within *Timetol* of an event. At that time point, the event evaluates to True.

If *Timetol* is not specified, the default time point is at, or just beyond, the time of the event. If *period* is specified as greater than 0, then the timer function schedules subsequent events at multiples of *period*.

A pseudo-random bit stream generator is an example how the timer function might be used.

```
module bitStream (out) ;
  output out ;
  electrical out ;
  parameter period = 1.0 ;
  integer x ;

  analog begin
    @(timer(0, period))
      x = $random + 0.5 ;
      V(out) <+ transition( x, 0.0, period/100.0 ) ;
  end
endmodule
```

6.8 Announcing Discontinuity

The **discontinuity** function is used to give hints to the simulator about the behavior of the module so that it can control the simulation algorithms to get accurate results in exceptional situations. It does not directly specify the behavior of the module. The **discontinuity** function should be executed whenever the analog behavior changes discontinuously.

The general form is

```
discontinuity();
```

Because discontinuous behavior can cause convergence problems, discontinuity should be avoided whenever possible.

The filter functions (**transition**, **slew**, **laplace**, etc.) are provided to smooth discontinuous behavior. However, in some cases it is not possible to implement the desired functionality using these filters. In this case, **discontinuity** function should be executed when the signal behavior changes abruptly.

Discontinuity created by switch branches and built-in system functions, such as **transition** and **slew** do not need to be announced.

The following example uses the discontinuity function to model a relay.

```

module relay (c1, c2, pin, nin) ;
    inout c1, c2 ;
    input pin, nin ;
    electrical c1, c2, pin, nin ;
    parameter real r=1 ;

    analog begin
        @(cross(V(pin,nin))) discontinuity() ;
        if (V(pin,nin) >= 0)
            I(c1,c2) <+ V(c1,c2)/r;
        else
            I(c1,c2) <+ 0 ;
        end
    endmodule

```

In this example, **cross** function controls the time step so that the time when the relay changes position is accurately resolved. It also triggers the discontinuity function that causes the simulator to react properly to the discontinuity. This would have been handled automatically if the type of the branch (c1,c2) had been switched between voltage and current.

Another example is a source that generates a triangular wave. In this case, neither the model nor the waveforms generated by the model are discontinuous. Rather, the waveform generated is piecewise linear with discontinuous slope. If the simulator is aware of the abrupt change in slope, it can adapt to eliminate problems that result from the discontinuous slope (typically changing to a first order integration method).

```

module triangle(out);
    output out;
    voltage out;
    parameter real period = 10.0, amplitude = 1.0;
    integer slope;
    real offset;

    analog begin
        @(timer(0, period)) begin
            slope = +1;
            offset = $realtime;
            discontinuity();
        end
        @(timer(period/2, period)) begin
            slope = -1 ;
            offset = $realtime;
            discontinuity();
        end
        V(out) <+ amplitude*slope*
            (4*($realtime - offset)/period - 1);
    end
endmodule

```

Finally, here is a case where timer function is used without using a **discontinuity** function. In this case, the event generated by the **timer** function indicates that a measurement should be printed, but that neither the model nor the waveforms contain discontinuity.

```

module sampler (in) ;
    input in ;
    voltage in ;
    parameter real period = 10.0 ;

    analog @(timer(0, period))
        $strobe("%g\t%g", $realtime, V(in)) ;
endmodule

```

6.9 Time related functions

There are two functions, **bound_step** and **last_crossing**, related to simulation time.

6.9.1 Bounding the time step

The **bound_step** function puts a bound on the next time step. It does not specify exactly what the next time step should be, but it bounds how far the next time point can be from the present time point. The function takes the maximum time step as an argument. It does not return a value.

The general form is

```
bound_step (expression) ;
```

where *expression* is a required argument and represents the maximum timestep the simulator can advance.

The example below implements a sinusoidal voltage source and uses the **bound_step()** function to assure that the simulator faithfully follows the output signal (it is forcing 20 points per cycle).

```

module vsine(out);
    output out;
    voltage out;
    parameter real freq=1.0, ampl=1.0, offset=0.0;

    analog begin
        V(out) <+ ampl*sin(2.0*M_PI*freq*$realtime) + offset;
        bound_step(0.05/freq);
    end
endmodule

```


Section 7

Mixed-Signal

This Section is a work-in-progress!

7.1 Fundamentals

7.1.1 Domains

7.1.2 Contexts

7.1.3 Analog and Digital Disciplines

7.1.4 Nets, Nodes, and Signals

7.2 Discipline Resolution and Connection Module Insertion

7.2.1 Discipline Resolution

In some cases incompatible disciplines may appear at the high and low connection of a port but both may be in the same domain. That is they may both be continuous-time disciplines, or both discrete-time disciplines. In this case no connection module may be needed, but we may wish the two segments to be treated as a single segment with only one discipline. In this case the following form of the connect statement is used:

```
connect discipline_list using discipline ;
```

where the disciplines in the discipline list are the disciplines which may be consolidated and the final discipline is the discipline to which they resolve.

7.2.2 Resolution of Discrete-time Disciplines

Signals and ports of discrete-time disciplines must obey the rules imposed by Verilog-D on such connections.

In addition the real-valued nets cannot be connected to scalar or vector bit-valued nets without a connection module.

7.3 Behavioral Interaction

- Verilog-AMS has a separate block for defining analog behavior inside a module.
 - q Analog behavior can only be described inside of the analog block.
 - q Analog functions can be created but only used inside of the analog block
 - q There can be only one analog block per module.
- In general digital behavioral is defined in the initial/always blocks and analog in the analog block.
 - q All three types of blocks can appear in the same module.
- Read operations of continuous-time and discrete-time signals are allowed from any context
- Write operations of:
 - q continuous-time signals are only allowed from inside an analog construct.
 - q discrete-time signals are allowed from any context outside of an analog construct.

Verilog-AMS provides ways to :

- Access analog signals from a digital block
- Access digital signals from an analog block
- Allow a analog event to effect a digital block
- Allow a digital event to effect a analog block
- Use the same variable in both the analog and digital blocks

Analog Signal Appearing in an Digital Expression

```
...
reg clock;
real r;
electrical x;

always @(posedge clock) begin
r = V(x);
end
```

Digital Signal Appearing in an Analog Expression

```

reg d;
electrical x;

analog begin
  if (d == 0)
    V(x) <+ 0.0;
  else
    V(x) <+ 3.0;
end

```

Analog Event Appearing in an Digital Event Control

```

electrical x;
reg d;
integer i;
always @(cross(V(x) - 4.5, 1)) begin
  i = d;
end

```

Digital Event Appearing in an Analog Event Control

```

real r;
reg d;
electrical x, y;
analog begin
  @(posedge d or cross(V(y), 1))
    r = V(x);
end

```

Common variables in both the analog and digital blocks

- Variables can be read from any context

- Variables can only be written to by one context
 - This context defines the owner or what type of variable it is, analog or digital

```

real vth;
integer cm;
always @(cross(V(in) - vth, 1 ))
  cm=1b1;
always @(cross(V(in) - vth, -1))
  cm=1b0;

analog begin
  vth = (V(vcc) - V(vee)) / 2  + V(vee);
  v(out) <+ transition( ((cm==1) ? 5.0 : 0.0 ), 10n, 5n, 5n) ;
end

```

7.3.1 Synchronous

7.3.1.1 Events and Event Controls

7.3.2 Asynchronous

7.4 Connect Statement and Connection Module Semantics

The connect statement performs the following functions.

- defines rules for the auto-insertion of connection modules between incompatible disciplines (section 7.5).
- supports manual insertion of a connection module.
- defines the rules for incompatible discipline resolution (section 7.2.1).
- supports back-annotation of parasitics(section 7.6).

The connect statement has the following syntax.

```

connect_statement ::=
    connect discipline_identifier to discipline_identifier
    module_identifier attributes ;
    | connect discipline_identifier with
    discipline_identifier module_identifier attributes ;
    | connect module_identifier ;
    | connect discipline_list using discipline_identifier
    module_identifier attributes ;
    | connect module_identifier ( port_list ) signal_path ;
attributes ::=
    /* empty */
    | #( attribute_list )
attribute_list ::=
    attribute
    | attribute_list , attribute
attribute ::=
    .parameter_identifier ( parameter_value )
    | .connect_mode ( conn_mode )
conn_mode ::=
    split | merged

```

Figure 7-1: Syntax for connect statement

The first two forms of the connect statement listed above deal with automatic insertion of connection modules between incompatible disciplines. The third form is used to manually insert a connection module. The fourth form is used for discipline resolution and the last deals with back annotation of parasitic information.

For convenience we will refer to the various forms of the connect statement by the following names:

UNIDIRECTIONAL CONNECT STATEMENT : **connect** *discipline_identifier1* **to** *discipline_identifier2* *module_identifier* *attributes* ;

In this form *discipline_identifier1* is referred to as the **source** and *discipline_identifier2* is referred to as the **sink**.

BIDIRECTIONAL CONNECT STATEMENT : **connect** *discipline_identifier* **with** *discipline_identifier* *module_identifier* *attributes* ;

CONNECT MODULE DECLARATION : **connect** *module_identifier* ;

RESOLUTION CONNECT STATEMENT : **connect** *discipline_list* **using** *discipline_identifier* *module_identifier* *attributes* ;

PARASITIC CONNECT STATEMENT : **connect** *module_identifier* (*port_list*) *signal_path* ;

7.5 Automatic Insertion of Connection Modules

Automatic insertion of connection modules is performed when signals and ports with discrete time domain and continuous time domain disciplines are connected. The connection module defines the conversion between these different disciplines.

An instance of the connection module will be inserted across any port that matches the rule specified by a connect statement. Rules for matching connect statements with ports (stated in detail later) take into account the port direction, and the disciplines of the signals connected to the port.

For example,

```

module dig_inv(in, out);
  input in;
  output out;
  logic in, out;

  always begin
    out = #10 ~in;
  end
endmodule


module analog_inv(in, out);
  input in;
  output out;
  electrical in, out;
  parameter real vth =2.5;

  analog begin
    if (V(in) > vth)
      outval = 0;
    else
      outval = 5 ;
      V(out) <+ transition(outval);
    end
endmodule


module ring;

  dig_inv d1 (n1, n2);
  dig_inv d2 (n2, n3);
  analog_inv a3 (n3, n1);

endmodule


module elect_to_logic(el,cm);
  input el;
  reg cm;
  electrical el;
  logic cm;

```

```

    always
        @(cross(V(el) - 2.5, 1)
           cm = 1;

    always
        @(cross(V(el) - 2.5, -1)
           cm = 0;

endmodule

module logic_to_elect(cm,el);
    input cm;
    output el;
    logic cm;
    electrical el;

    analog
        V(el) <+ transition((cm == 1) ? 5.0 : 0.0);

endmodule

connect electrical to logic elect_to_logic;
connect logic to electrical logic_to_elect;

```

Each connect statement designates a module to be a connection module. In the example above two modules, `elect_to_logic` and `logic_to_elect` are specified as the connection modules to be automatically inserted whenever a signal and a module port of disciplines electrical and logic are connected.

For example, module `elect_to_logic` will convert signals on port out of instance a3 to port in of instance d1. The module `logic_to_elect` will convert the signal on port out of instance d2 to port in of instance a3.

7.5.1 Connection Module Selection and Insertion

The selection of a connection module depends upon the disciplines of all the ports and signals connected together. It is, therefore, a post elaboration operation. This is because the signal connected to a port is only known when the module in which the port is declared has been instantiated.

7.5.1.1 Signal Segmentation

After a connection module has been selected it cannot be inserted until we determine whether there should be one connection module per port, or one connection module for all the ports on a signal that match a given connect statement. Inserting multiple copies of the same connection module on one signal (i.e. between the signal and the multiple ports) will have the effect of creating distinct segments of the signal which are of the same discipline.

This segmentation of the signal that connects ports is only performed in the case of digital ports (i.e. ports with discrete-time domain or digital discipline). It is assumed that for analog (or continuous-time domain) disciplines it is never desirable to segment the signal between the ports. That is, there should never be more than one analog node representing a signal.

However it may be desirable for the simulator's internal representation of the signal to consist of various separate digital segments each with its own connection module. For example, this is useful to model the loading effect of each individual digital port on the analog signal or node.

Insertion of Connection Instances Creates Distinct Segments in a Signal

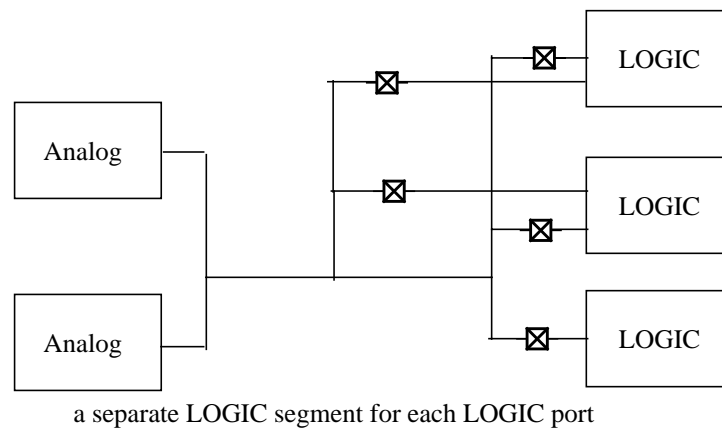
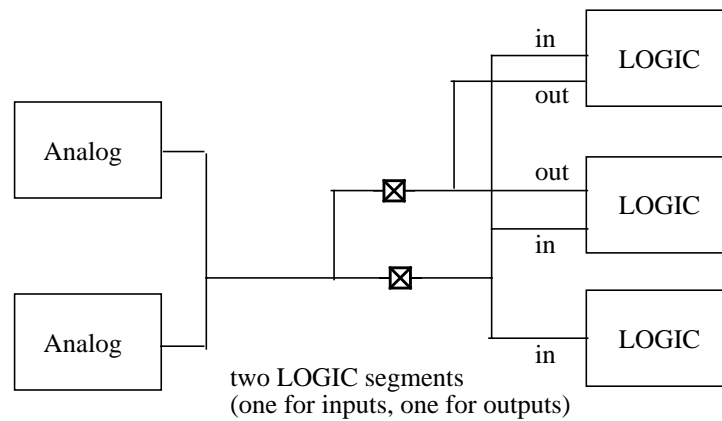
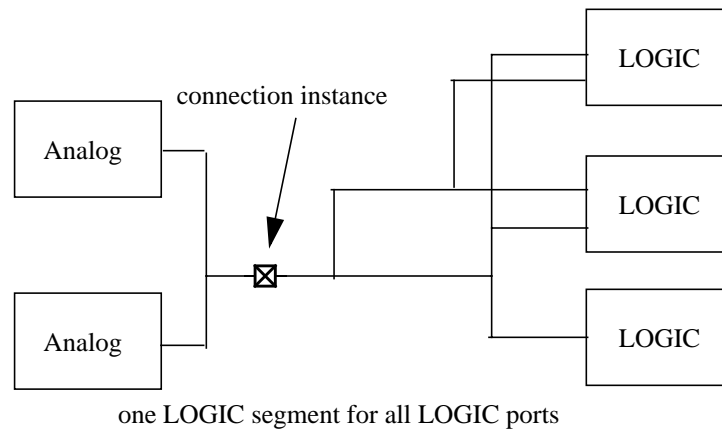


Figure 7-2: Signal segmentation by connection modules

7.5.1.2 Connect_mode Attribute

An attribute is provided for the connect statement to direct the segmentation of the signal which may occur while inserting a connection module. If segmentation is desired then it may be specified in the connect statement using the attribute **connect_mode**. This attribute can take one of three predefined values -- **split** or **merged**. It has a default value of **merged**.

This attribute applies when there is more than one port on a signal for which the connect statement applies, and when those ports have a digital discipline. The keyword **connect_mode** indicates how input, output or inout ports of the given discipline should be combined for the purpose of inserting connection modules.

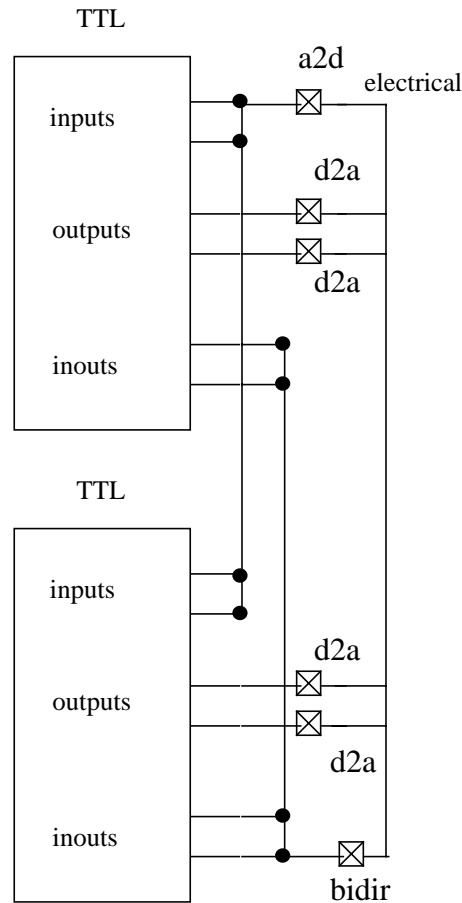
For example,

```
connect electrical to logic elect_to_logic #(.connect_mode(split));
```

This connect statement specifies a module, `elect_to_logic`, will be inserted across a module port

- if an input port has logic discipline and the signal connecting to the port has electrical discipline.
- if an output port has electrical discipline and the signal connecting to the port has logic discipline.

If there is more than one such input port connected at a node, then setting `connect_mode` attribute to `split` requires that there be one connection module for each port, that converts between signal discipline and the port discipline. In this way the signal connecting to the ports is segmented by the insertion of one connection module for each port.



```

connect ttl to electrical d2a #(.connect_mode(split));
connect electrical to ttl a2d #(.connect_mode(merged));
connect electrical with ttl bidir #(.connect_mode(merged));
  
```

Figure 7-3: Connect module insertion with Signal Segmentation

In figure 7-3 the connections of an electrical signal to ttl output ports results in a distinct instance of the d2a connection module being inserted for each output port. This is mandated by the connect_mode attribute set to split.

Connection of the electrical signal to ttl input ports results in a single instance of the a2d connection module being inserted between the electrical signal and all the ttl input ports. This is mandated by the connect_mode attribute set to merged. This behavior is also seen for ttl inout ports which has a connect_mode attribute set to merged.

For example:

```
connect electrical to cmos02u cmosA2d #(.r(30k),
    .connect_mode(split));
```

performs three functions:

1. Connect an instance of cmosA2d module between a signal with electrical discipline and the input port with cmos02u discipline, or an output port with electrical discipline and the signal with cmos02u discipline.
2. Set the value of the parameter r to 30k.
3. Use one module instance for each input port.

If there are many output ports for which this rule applies, then by definition there will be no segmentation of the signal between these ports, since the ports have discipline electrical (an analog discipline).

Another example:

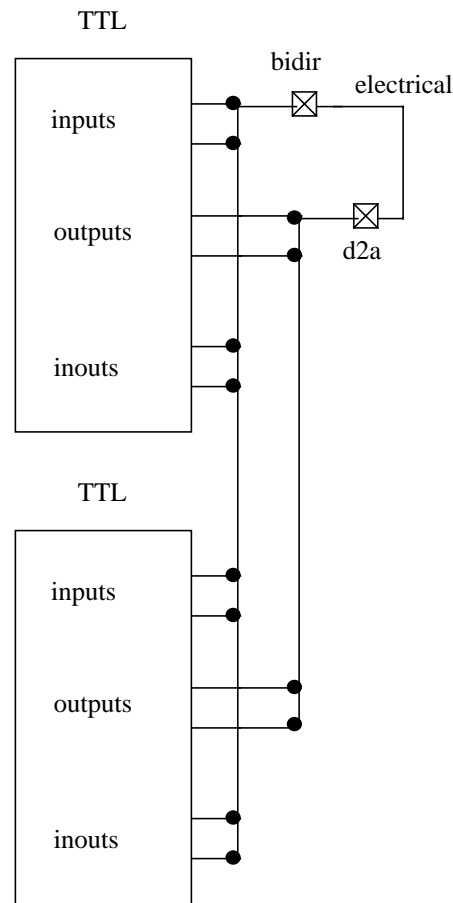
```
connect electrical to cmos04u cmosA2d #(.r(15k),
    .connect_mode(merged));
```

1. Connect an instance of cmosA2d module between a signal with electrical discipline and an input port with cmos04u discipline, or an output port with electrical discipline and a signal with cmos4u discipline.
2. Set the value of the parameter r to 15k.
3. Use one module instance regardless of the number of ports.

7.5.1.3 Attribute Merged

The other possible value for the `connect_mode` attribute is `merged`. This value for the attribute instructs the simulator to try to group all ports (whether they are input, output or inout) and to have just one connector module for all, provided that the module is the same for all.

The example which follows illustrates the effect of the `merged` attribute. Connection of the electrical signal to ttl inout ports and ttl input ports results in a single connector module, `bidir`, inserted between the ports and the electrical signal. The ttl output ports are merged, but with a different connection module which means that there is one connector module inserted between the electrical signal and all of the ttl output ports.



```

connect ttl to electrical d2a #(.connect_mode(merged));
connect electrical to ttl bidir#(.connect_mode(merged));
connect ttl with electrical bidir #(.connect_mode(merged));

```

Figure 7-4: Connector insertion with connect_mode attribute merged

7.5.2 Internal Representation, Driver Receiver Segregation

If the hierarchical segments of a signal are all digital, or all analog then the signal is not a mixed signal and the internal representation of the signal will not differ from that of a purely digital or an analog signal.

If, on the other hand, the signal has both analog and digital segments in its hierarchy, then it is a mixed signal. In this case appropriate conversion elements will be inserted, either manually or automatically.

- All the analog segments of a mixed signal are representations of a single analog node.
- Each of the noncontiguous digital segments of a signal will be represented internally as a separate digital signal, with its own state.

7.5.2.1 Driver-Receiver Segregation

In the digital domain signals may have drivers and receivers. A driver makes a contribution to the state of the signal. A receiver accesses, or reads, the state of the signal. In a pure digital net, i.e. one without an analog segment, the simulation kernel resolves the values of the drivers of a signal, and when there is a change in state it propagates the new value to the receivers by means of an event.

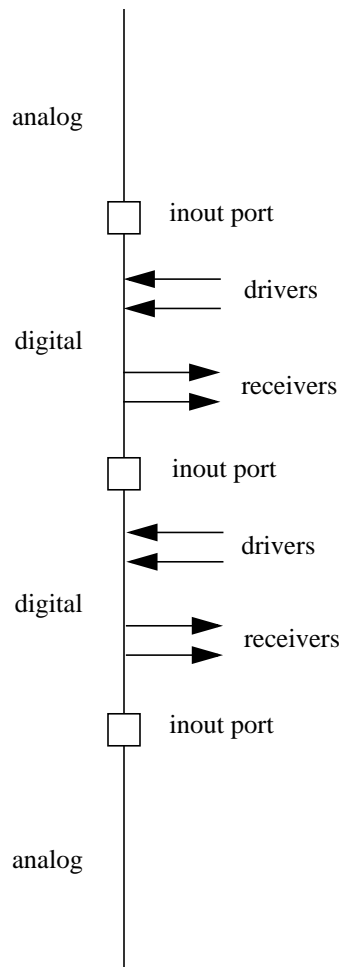
In the case of a mixed net, that is one with digital segments and an analog segment, we may not want the digital simulation kernel to propagate new values directly from drivers to receivers, but, to propagate the change to the analog simulation kernel which can then detect a threshold crossing and then propagate the change in state back to the digital kernel. This, among other things, allows the simulation to account for rise and fall times caused by analog parasitics.

Within digital segments of a mixed-signal net, drivers and receivers of ordinary modules may be segregated, so that transitions are not propagated directly from drivers to receivers, but propagate through the analog domain.

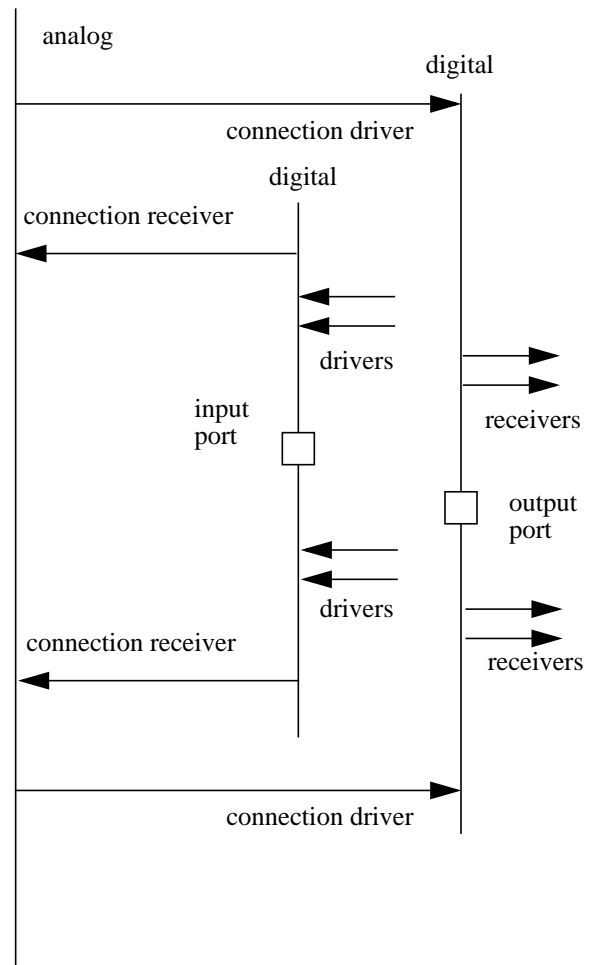
The drivers and receivers of connection modules will be oppositely segregated. That is, the connection module drivers will be grouped with the ordinary module receivers and the ordinary module drivers will be grouped with the connection module receivers.

Thus digital transitions are propagated from drivers to receivers by way of analog, through the connection module instances.

Hierarchical Definition



Internal Representation

**Figure 7-5: Driver-Receiver Segregation in Modules with Bidirectional ports**

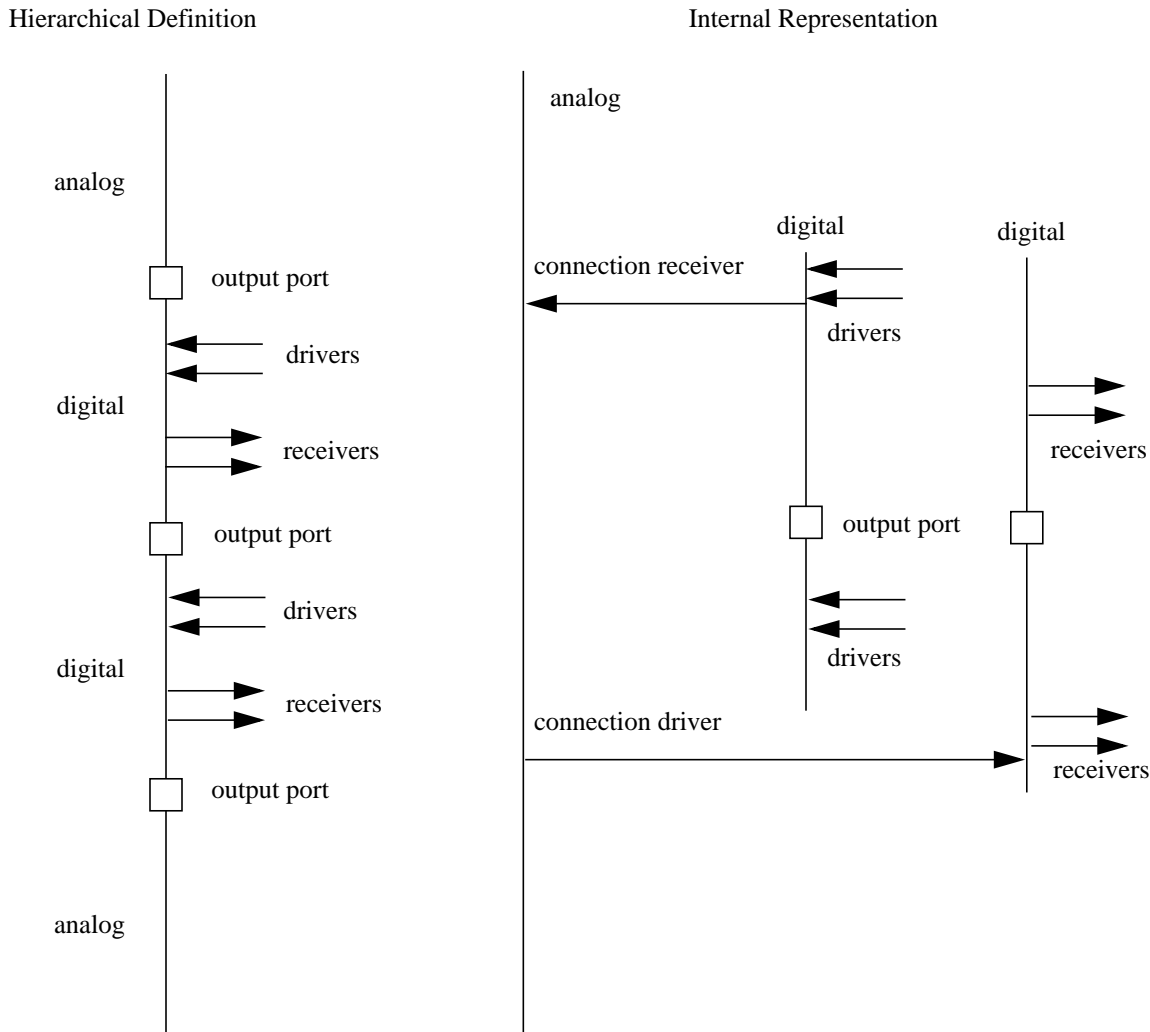


Figure 7-6: Driver-Receiver Segregation in modules with Unidirectional ports

7.5.3 Rules for Driver/Receiver Segregation and Connection Module Selection and Insertion

Driver/receiver segregation and connection module insertion is a post elaboration operation. It depends on a complete hierarchical examination of each signal in the design, that is, an examination of the signal in all the contexts through which it passes. If the complete hierarchy of a signal is digital, that is, the signal has a digital discipline in all contexts through which it passes, then it is a digital signal rather than a mixed signal. Similarly, if the complete hierarchy of a signal is analog, then it is an analog signal rather than a mixed signal. Rules for driver/receiver segregation and connection module insertion apply only to mixed signals, that is, to signals which have an analog

discipline in one or more of the contexts through which they pass, and a digital discipline in one or more of the contexts. In this case context refers to the appearance of a signal in a particular module instance. For a particular signal we will refer to a module instance as a digital context if the signal has a digital discipline in that module or an analog context if the signal has an analog discipline. We refer to the appearance of a signal in a particular context as a segment of the signal. In general a signal in a fully elaborated design consists of various segments some of which may be analog and some of which may be digital. A port represents a connection between two segments of a signal the context of one of the segments is an instantiated module and the context of the other is the module which instantiates it. We refer to the segment in the instantiated module as the lower or formal connection and the segment in the instantiating module as the upper or actual connection. A connection element is selected for each port to which one connection is analog and the other digital.

The following rules govern driver/receiver segregation and connection module selection. These rules apply only to mixed signals.

1. A mixed signal is represented in the analog domain by a single node, regardless of how its analog contexts are distributed hierarchically.
2. Digital drivers of mixed signals are segregated from receivers so that the digital drivers contribute to the analog state of the signal and the analog state, in turn, determines the value seen by the receivers.
3. A connection will be selected for a port only if one of the connections to the port is digital and the other is analog. If this is the case then the port must match one (and only one) connection statement. The module named in the connection statement is the one which will be selected for the port.
4. Input ports will match unidirectional connection statements. An input port matches a unidirectional connection statement if the upper connection discipline of the port matches the source discipline in the connect statement and the lower connection discipline of the port matches the sink discipline in the connect statement.
5. Output ports will match unidirectional connection statements. An output port matches a unidirectional connection statement if the upper connection discipline of the port matches the sink discipline in the connect statement and the lower connection discipline of the port matches the source discipline in the connect statement.
6. Inout ports will match bidirectional connection statements. The connection statement will match the port if the two disciplines in the connection statement are the same as the disciplines of the connections to the port.

Once connection modules have been selected, they will be inserted according to the `connect_mode` parameters in the pertinent connect statements. These rules apply to connection module insertion:

1. The connect mode of a port for which a connection module has been selected will be determined by the value of the `connect_mode` parameter of the connect statement which was used to select the connection module.
2. The connection module for a port will be instantiated in the context of the ports upper connection.
3. All ports connecting to the same signal (upper connection) and having the same connection module and having a `connect_mode` parameter of merged will share a single instance of the selected connection module.
4. All other ports will have an instance of the selected connection module, that is one connection module instance per port.

7.5.4 Instance Names for Auto-Inserted Instances

Parameters of auto-inserted connection instances may be set on an instance by instance basis with the use of the **defparam** statement. This necessitates predictable instance names for the auto-inserted modules.

For the case of auto-inserted instances the following naming scheme is employed to unambiguously distinguish the connector modules. Depending on the `connect_mode` attribute the following name identifies the connector module.

1. Merged
In the merged case one or more ports have a given discipline at their bottom connection, call it BottomDiscipline, and a common signal, call it SigName, of another discipline, call it TopDiscipline, at their top connection. A single connection module is placed between the top signal and the bottom signals. In this case the instance name of the connection module is derived from the signal name and the bottom discipline,

`<SigName><BottomDiscipline>`

2. Split
In the split case one or more ports have a given discipline at their bottom connection and a common signal, of another discipline, call it TopDiscipline, at their top connection. One module instance is instantiated for each such port. In this case the instance name of the connection module is,

`<SigName><InstName><PortName>`

where `InstName` and `PortName` are the local instance name of the port and its instance respectively.

7.6 Back Annotation of Parasitics

The following form of the connect statement is for back annotation of parasitics such as SPF data extracted from the output of physical layout tools:

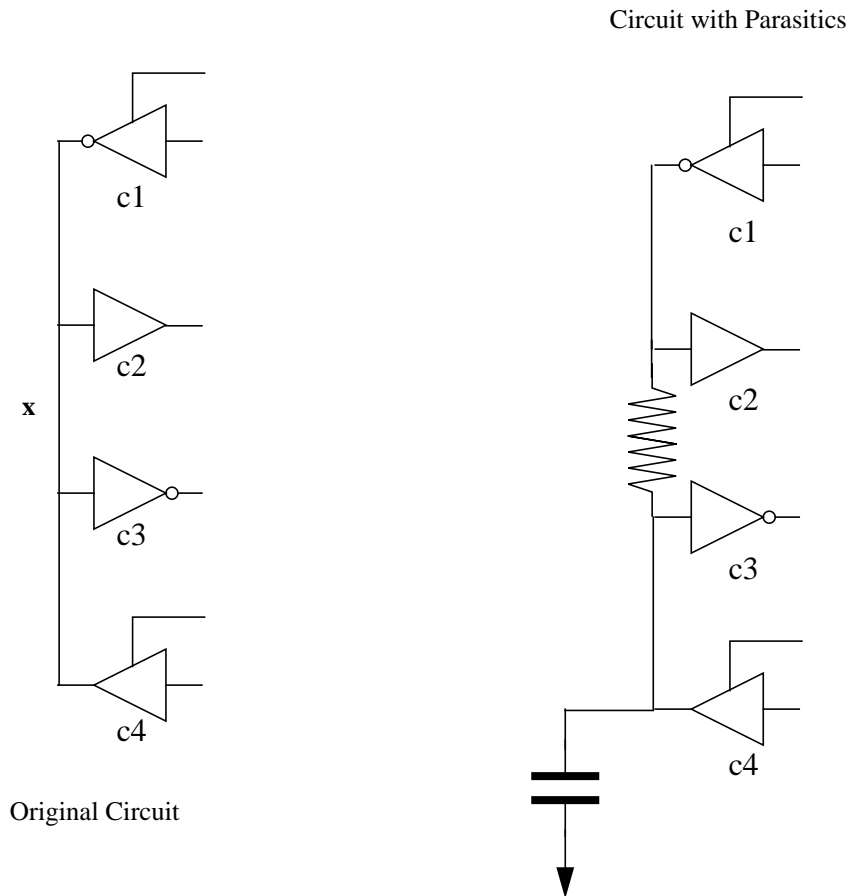
```
connect module_identifier ( port_list ) signal_path ;
```

The signal whose path is given is removed. The module whose name is given is instantiated with the same path name as the signal which was removed. The port_list must contain all the ports to which the signal had been connected. This includes ports which make out of context reference to the signal.

Connect statements of this form will not be executed until after the rest of the design has been elaborated. The named signal will be removed from each port in the port list regardless of whether it was at the upper or lower connection of the port. The named module will be instantiated with same name as the removed signal and each of the removed connections will be replaced by the signal at the lower connection of the

corresponding port on the new instance.

Example



If the original circuit in the diagram above is expressed as follows:

```

module top();
    logic inn, outb, outn, inb, x;

    notif1 c1(x,inn,cn);
    buf c2(outb,x);
    not c3(outn,x);
    bufif1 c4(x,inb,cb);
endmodule

```

Then the following module could be used to backannotate parasitics:

```

module backan(x1,x2,x3,x4)
  inout x1,x2,x3,x4;
  electrical x1,x2,x3,x4;

  resistor #(100) r1(x2,x3);
  voltage #(0.0) v1(x1,x2);
  voltage #(0.0) v2(x3,x4);
  capacitor #(0.000001)(x4,gnd);
endmodule

```

This would be done with the following connect statement:

```
connect backan (top.c1.out,top.c2.in,top.c3.in,top.c4.out) top.x;
```

The result is that the wire top.x is removed and replaced with the module backan.

7.6.1 Port Names for Verilog Built-in Primitives

In the cases of instances of modules and instances of UDPs port names are well defined. In these cases the port name is the name of the signal at the lower connection of the port. in the case of built in primitives, however, Verilog-D does not define port names. It is, thus necessary to define port names for the ports of built in primitives in Verilog-MS.

The following conventions will be used for naming Verilog Ports.

1. For N-input gates (and, nand, nor, or, xnor, xor) the output will be named out, and the inputs reading from left to right will be in1, in2, in3, etc.
2. For N-output gates (buf, not) The input will be named in, and the outputs reading from left to right will be named out1, out2, out3, etc.
3. For 3 port MOS switches (nmos, pmos, rnmos, rpmos) the ports reading from left to right will be named source, drain, gate.
4. For 4 port MOS switches (cmos, rcmos) the ports reading from left to right will be named source, drain, ngate, pgate.
5. For bidirectional pass switches (tran, tranif1, tranif0, rtran, rtranif1, rtranif0) the ports reading from left to right will be named source, drain, gate.
6. For single port primitives (pullup, pulldown) the port will be named out.

7.7 Driver Access Functions

Access to individual drivers is necessary for accurate implementation of connection modules (section 7.5). A driver of a signal is a process which assigns a value to the signal, or a connection of the signal to an output port of a module instance or simulation

primitive. The driver access functions described here apply only to drivers found in ordinary modules and not to those found in connection modules.

A signal may have a number of drivers and each driver may have a current value and a pending value. The current value is the current contribution of the driver to the resolved state of the signal and the pending value is the next scheduled contribution, if any, of the driver to the resolved state of the signal.

7.7.1 driver_update event

The status of drivers for a given signal can be monitored with a new event detection keyword **driver_update**. It can be used in conjunction with the event detection operator **@** to detect updates to any of the drivers of the signal. For example:

```
always @(driver_update clock)
    statement;
```

will cause statement to execute any time a driver of the signal clock is updated. Here, an update is defined as the addition of a new pending value to the driver. This is true whether or not there is a change in the resolved value of the signal.

The functions described below can be used to access the information about the drivers of a signal.

7.7.2 driver_count function

The *driver_count function* returns an integer representing the number of drivers associated with the signal in question. The syntax is as follows:

```
driver_count_function ::=
    driver_count ( signal_name )
```

Figure 7-7: Syntax for driver_count function

The drivers are arbitrarily numbered from 0 to N-1, where N is the total number of drivers contributing to the signal value. For example, if this function returns a value 5 then the signal has 5 drivers numbered from 0 to 4.

7.7.3 driver_active function

The *driver_active function* returns an integer index for each driver of the signal which is currently active. The syntax is as follows:

```
driver_active_function ::=  
    driver_active ( signal_name )
```

Figure 7-8: Syntax for driver_active function

Each call to this function returns the index of the driver that is currently active. Repeated calls of this function will return the index of the active drivers in increasing order. When the indices for all the active drivers at the current time have been returned, the next call to this function will return -1. If this function is called after it has returned -1, it will cycle through the active drivers again.

The returned value (driver index) may be used as the driver argument in any of the following driver access tasks described below. The drivers are arbitrarily numbered 0 to N-1, where N is the total number of drivers contributing to the signal value.

For example, if a the signal has 10 drivers, of which drivers numbered 3, 5, and 8 are active currently then 8 successive calls to this function will return, in order, 3, 5, 8, -1, 3, 5, 8, -1.

7.7.4 driver_local function

The *driver_local function* returns an integer value that represents the index of the driver if the calling process has a driver for the signal (active or inactive). The syntax is as follows:

```
driver_local_function ::=  
    driver_local ( signal_name )
```

Figure 7-9: Syntax for driver_local function

If there is no driver for the signal in the local process, this function returns -1.

7.7.5 driver_state function

The *driver_state function* returns the current value contribution of a specific driver to the state of the signal. The syntax is as follows:

```
driver_state_function ::=  
    driver_state ( signal_name, driver_index )
```

Figure 7-10: Syntax for driver_state function

The `driver_index` value is an integer value between 0 and N-1, where N is the total number of drivers contributing to the signal value. The state value is returned as 0, 1, x, or z.

7.7.6 driver_strength function

The *driver_strength function* returns the current strength contribution of a specific driver to the strength of the signal. The syntax is as follows:

```
driver_strength_function ::=  
    driver_strength ( signal_name, driver_index )
```

Figure 7-11: Syntax for driver_strength function

The `driver_index` value is an integer value between 0 and N-1, where N is the total number of drivers contributing to the signal value. The strength value is returned as an integer between 0 and 7.

7.7.7 driver_delay function

The *driver_delay function* returns the delay, from current simulation time, after which the pending state or strength becomes active. If there is no pending value on a signal it will return zero. The syntax is as follows:

```
driver_delay_function ::=  
    driver_delay ( signal_name, driver_index )
```

Figure 7-12: Syntax for driver_delay function

The `driver_index` value is an integer value between 0 and N-1, where N is the total number of drivers contributing to the signal value.

The returned delay value is an integer.

7.7.8 **driver_next_state function**

The *driver_next_state function* returns the pending state of the driver, if there is one. If there is no pending state it returns the current state.

```
driver_next_state_function ::=  
    driver_next_state ( signal_name, driver_index )
```

Figure 7-13: Syntax for driver_next_state function

The *driver_index* value is an integer value between 0 and N-1, where N is the total number of drivers contributing to the signal value. The pending state value is returned as 0, 1, x, or z.

7.7.9 **driver_next_strength function**

The *driver_next_strength function* returns the strength associated with the pending state of the driver, if there is one. If there is no pending state it returns the current strength.

```
driver_next_strength_function ::=  
    driver_next_strength ( signal_name, driver_index )
```

Figure 7-14: Syntax for driver_next_strength function

The *driver_index* value is an integer value between 0 and N-1, where N is the total number of drivers contributing to the signal value. The pending strength value is returned as an integer between 0 and 7.

Section 8

Hierarchical Structures

Verilog-AMS HDL supports a hierarchical hardware description by allowing modules to be embedded within other modules. Higher-level modules create instances of lower-level modules and communicate with them through input, output, and bidirectional ports.

To describe a hierarchy of modules, the user provides textual definitions of various modules. Each module definition stands alone; the definitions are not nested. Statements within the module definitions create instances of other modules, thus describing the hierarchy.

Verilog-AMS provides a `connect` statement to define the rules for automatic insertion of user defined modules to connect ports of incompatible disciplines.

8.1 Modules

A module definition is enclosed between the keywords **module** and **endmodule**. The identifier following the keyword **module** is the name of the module being defined. The optional list of ports specify an ordered list of the module's ports. The order used can be significant when instantiating the module (section 8.1.2). The identifiers in this list must be declared in input, output, and inout declaration statements within the module definition. The module items define what constitutes a module, and include many different types of declarations and definitions. A module definition can have at most one analog block.

The keyword **macromodule** can be used interchangeably with the keyword **module** to define a module. An implementation can choose to treat module definitions beginning with the **macromodule** keyword differently.

```

module_declaration ::=
    module_keyword module_identifier [ list_of_ports ] ;
    [ module_items ]
    endmodule

module_keyword ::=
    module
    | macromodule

list_of_ports ::=
    ( port { , port } )

port ::=
    port_expression
    | .port_identifier ( [ port_expression ] )

port_expression ::=
    port_identifier
    | port_identifier [ constant_expression ]
    | port_identifier [ constant_range ]

constant_range ::=
    msb_constant_expression : lsb_constant_expression

module_items ::=
    { module_item }
    | analog_block

module_item ::=
    module_item_declaration
    | parameter_override
    | module_instantiation

module_item_declaration ::=
    parameter_declaration
    | input_declaration
    | output_declaration
    | inout_declaration
    | integer_declaration
    | node_declaration
    | real_declaration

parameter_override ::=
    defparam list_of_param_assignments ;

```

Figure 8-1: Syntax for module

8.1.1 Top-level modules

Top-level modules are modules that are included in the source text but are not instantiated, as described in section 8.1.2.

8.1.2 Module instantiation

Instantiation allows one module to incorporate a copy of another module into itself. Module definitions do not nest. That is, one module definition does not contain the text

of another module definition within its **module-endmodule** keyword pair. A module definition nests another module by *instantiating* it. The *module instantiation statement* creates one or more named *instances* of a defined module.

The following is the syntax for specifying instantiations of modules:

```

module_instantiation ::=
    module_identifier [ parameter_value_assignment ]
    instance_list
parameter_value_assignment ::=
    # ( ordered_param_override_list )
    | # ( named_param_override_list )
ordered_param_override_list ::=
    expression { , expression }
named_param_override_list ::=
    named_param_override { , named_param_override }
named_param_override ::=
    . parameter_identifier ( constant_expression )
instance_list ::=
    module_instance { , module_instance } ;
module_instance ::=
    name_of_instance ( [ list_of_module_connections ] )
name_of_instance ::=
    module_instance_identifier [ range ]
list_of_module_connections ::=
    ordered_port_connection { , ordered_port_connection }
    | named_port_connection { , named_port_connection }
ordered_port_connection ::=
    [ expression ]
named_port_connection ::=
    . port_identifier ( [ expression ] )
range ::=
    [ constant_expression : constant_expression ]

```

Figure 8-2: : Syntax for module instantiation

The instantiations of modules can contain a range specification. This allows an array of instances to be created.

One or more module instances (identical copies of a module definition) can be specified in a single module instantiation statement.

The list of module connections can be provided only for modules defined with ports. The parentheses, however, are always required. When a list of module connections is given, the first element in the list connects to the first port, the second to the second port, and so on. See section 8.3 for a more detailed discussion of ports and port connection rules.

A connection can be a simple reference to a node identifier or a sub-range of a vector node. The example below illustrates a comparator and an integrator (lower-level modules) which are instantiated in sigma-delta A/D converter module (the higher-level module).

```

module comparator(cout, inp, inm);
output cout;
input inp, inm;
    electrical cout, inp, inm;
parameter real td = 1n, tr = 1n, tf = 1n;

analog begin
    @cross(V(inp) - V(inm), 0)
        V(cout) <+ transition((V(inp) > V(inm)) ? 1 : 0, td, tr, tf);
end
endmodule

```

```

module integrator(out, in);
output out;
input in;
    electrical in, out;
parameter real gain = 1.0;
parameter real ic = 0.0;

analog begin
    V(out) <+ gain*idt(V(in), ic);
end
endmodule

```

```

module sigmadelta(out, ref, in);
output out;
input ref, in;

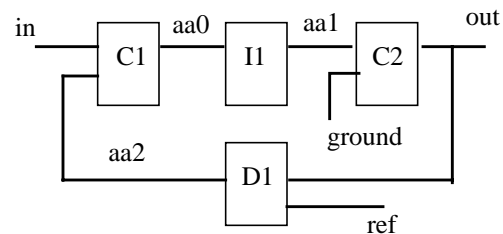
    comparator C1(.cout(aa0), .inp(in), .inm(aa2));
    integrator #(1.0) I1(.out(aa1), .in(aa0));
    comparator C2(out, aa1, ground);
    d2a #(.width(1)) D1(aa2, ref, out);

```

```

endmodule

```



// A D/A converter

The comparator instance C1 and the integrator instance I1 use named port connections, whereas the comparator instance C2 and the d2a (not described here) instance D1 uses ordered port connection.

The integrator instance I1 overrides gain parameter positionally, whereas the d2a instance D1 overrides width parameter by named association.

8.2 Overriding module parameter values

When one module instantiates another module, it can alter the values of any parameters declared within the instantiated module. There are three ways to alter parameter values: the *defparam statement*, which allows assignment to parameters using their hierarchical names, *module instance parameter value assignment by order*, which allows values to be assigned in-line during module instantiation in the order of their declaration, and *module instance parameter value assignment by name*, which allows values to be assigned in-line during module instantiation by explicitly associating parameter names with the overriding values.

8.2.1 Defparam statement

Using the *defparam statement*, parameter values can be changed in any module instance throughout the design using the hierarchical name of the parameter. See section 8.4 for hierarchical names.

The expression on the right hand side of the *defparam* assignments must be a constant expression involving only constant numbers and references to parameters. The referenced parameters (on the right hand side of the *defparam*) must be declared in the same module as the *defparam* statement.

The *defparam* statement is particularly useful for grouping all of the parameter value override assignments together in one module.

```

module tgate;
electrical io1,io2,control,control_bar;
mosn m1 (io1, io2, control);
mosp m2 (io1, io2, control_bar);
endmodule

module mosp (source,drain,gate);
  parameter gate_length = 0.3e-6,
             gate_width = 4.0e-6;

  spice_pmos #(.L(gate_length),.W(gate_width)) p(gate,source,drain);
endmodule

module mosn (source,drain,gate);
  parameter gate_length = 0.3e-6,
             gate_width = 4.0e-6;

  spice_nmos #(.L(gate_length),.W(gate_width)) n(gate,source,drain);
endmodule

module annotate;
defparam
  tgate.m1.gate_width = 5e-6,
  tgate.m2.gate_width = 10e-6;
endmodule

```

8.2.2 Module instance parameter value assignment by order

An alternative method for assigning values to parameters within module instances supplies values for particular instances of a module to any parameters that have been specified in the definition of that module.

The order of the assignments in module instance parameter value assignment must follow the order of declaration of the parameters within the module. It is not necessary to assign values to all of the parameters within a module when using this method. However, it is not possible to skip over a parameter assignment. Therefore, to assign values to a subset of the parameters declared within a module, the declarations of the parameters that make up this subset must precede the declarations of the remaining (optional) parameters. An alternative is to assign values to all of the parameters, but use the default value (the same value assigned in the declaration of the parameter within the module definition) for those parameters that do not need new values.

Consider the following example, where the parameters within module instance `mod_a` are changed during instantiation.

```

module m;
voltage clk;
electrical out_a, in_a;
electrical out_b, in_b;

// create an instance and set parameters
mosp #(2e-6,1e-6) weakp(out_a, in_a, clk);
// create an instance leaving default values
mosp plainp(out_b, in_b, clk);
endmodule

```

8.2.3 Module instance parameter value assignment by name

The third method of overriding parameters for a module instance is an explicit association between the name of the parameter and the new value being assigned to that parameter. The name of the parameter must be preceded by a period (.) and must be the name of a parameter in the definition of the module being instantiated. The overriding value for each parameter must be a constant expression and must be enclosed in parenthesis (). Only those parameters whose value is being overridden need specification.

In the following example of instantiating a voltage-controlled oscillator, the parameters are specified on a named-association basis much they are for ports.

```
vco #(.centerFreq(5000), .convGain(1000)) vco1(lo_out, rf_in);
```

Here, the name of the instantiated vco module is *vco1*. The *centerFreq* parameter is passed a value of 5000, and the *convGain* parameter is passed a value of 1000. The positional assignment mechanism for ports assigns *lo_out* as the first node, and *rf_in* as the second node of *vco1*.

8.2.4 Parameter override precedence

If the value of a parameter is overridden using defparam statement as well as module instance parameter value assignments (see section 8.2.2 and section 8.2.3), the value assignment specified by the defparam statement is retained and the other value assignments are ignored.

If the value of a parameter is overridden using one of the three forms at different levels of module hierarchy, the value assignment done in the hierarchically highest level of module is retained and the other value assignments are ignored.

If the hierarchical relationship between the modules containing defparam statements cannot be determined, it must be reported as an error.

8.2.5 Parameter dependence

A parameter (for example, `gate_cap`) can be defined with an expression containing another parameter (for example, `gate_width` or `gate_length`). Since `gate_cap` depends on the value of `gate_width` and `gate_length`, a modification of `gate_width` or `gate_length` changes the value of `gate_cap`. For example, in the following parameter declaration, an update of `gate_width`, whether by `defparam` statement or in an instantiation statement for the module that defined these parameters, automatically updates `gate_cap`.

```
parameter
    gate_width = 0.3e-6,
    gate_length = 4.0e-6,
    gate_cap = gate_length * gate_width * 'COX;
```

8.3 Ports

Ports provide a means of interconnecting instances of modules. For example, if a module A instantiates module B, the ports of module B are associated with either the ports or the internal nodes of module A. The top-level module does not have ports, so every port is eventually associated with a node.

8.3.1 Port association

The syntax for a port association is given below. It is the completion of the syntax presented in section 8.1.

```
port ::=
    port_expression
    | .port_identifier ( [ port_expression ] )
port_expression ::=
    port_identifier
    | port_identifier [ constant_expression ]
    | port_identifier [ constant_range ]
constant_range ::=
    msb_constant_expression : lsb_constant_expression
```

Figure 8-3: Syntax for port

The port expression in the port definition can be one of the following:

- a simple node identifier
- a scalar member of a vector node or port declared within the module
- a sub-range of a vector node or port declared within the module

The two types of module port definitions cannot be mixed; the ports of a particular module definition must all be defined by order or all by name. The port expression is optional because ports can be defined that do not connect to anything internal to the module.

8.3.2 Port declarations

The type and direction of each port listed in the module definition's list of ports are declared in the body of the module.

8.3.2.1 Port type

The type of a port is declared by giving its discipline. If the type of a port is not declared, the port can only be used in a structural description (it can be passed to instances of modules, but cannot be accessed in a behavioral description).

```
node_declaration ::=  
    discipline_identifier [ range ] list_node_identifiers;  
list_node_identifiers ::=  
    node_identifier { , node_identifier }
```

Figure 8-4: Syntax for port type declarations

8.3.2.2 Port direction

The direction of a port can be specified as **input**, **output**, or **inout** (bidirectional). If the direction is specified as being an input port, then the module will only monitor the signals at the port, and not modify them. That is, within the module the port can only be passed into other modules as input ports and the signals on the ports can only be used in expressions, they cannot be used on the left side of a contribution statement. If the direction is specified as being an output port, then the module will only affect the signals at the port, but not be affected by them. Thus, the port can be passed to instances of other modules as output ports and the signals on the ports cannot be used in expressions but can be used on the left side of a contribution statement. Finally, ports that are declared as being bidirectional are not subject to these restrictions. If the direction of the port is not specified, it is taken to be bidirectional. The syntax for port declarations is as follows:

```
input_declaration ::= input [ range ] list_of_port_identifiers ;  
output_declaration ::= output [ range ] list_of_port_identifiers ;  
inout_declaration ::= inout [ range ] list_of_port_identifiers ;
```

Figure 8-5: Syntax for port direction declarations

A port can be declared in both a port type declaration and a port direction declaration. If a port is declared as a vector, the range specification between the two declarations of a port must be identical.

Note: Implementations may limit maximum number of ports in a module definition, but will at least be 256.

8.3.3 Real valued ports

Verilog-AMS supports ports that are declared to be real valued and have a discrete-time discipline.

```
module sum(in1, in2, out);  
input in1, in2;  
output out;  
real in1, in2, out;  
logic in1, in2;  
  
always begin  
  out = in1 + in2;  
end  
endmodule
```

In a module instantiation a real valued port can only be bound to a real variable whose value may be assigned only from the digital context, or to another real valued port. The result of binding a real variable to a port is to make the variable visible in any context in which the port is visible. Real valued ports are, therefore, subject to the same rules of usage as real variables. They also exhibit the same behavior. For example, assignment of a value to a real port will overwrite the existing value. Thus, if two processes attempt to assign different values to a real variable (either directly or through a port bound to the variable) at the same time the result is a race condition. There is no resolution of multiple drivers as with other signal types. One of the assignments will win, but, it can not be predetermined. It is an error to modify the value of a real port which is declared as an input.

As with other discrete time ports (digital), a real valued port may be assigned a discipline for purposes of connection element insertion using connect statements. In this case, the connection module being inserted will have a real valued port of the same discipline, and a discrete time port of some other discipline. Since it is illegal to connect the real valued port to anything other than a real variable or port, it does not make sense to apply a discipline which has been used to declare real ports or variables, to any other type of object (wire, reg, etc.).

8.3.4 Connecting module ports by ordered list

One method of making the connection between the ports listed in a module instantiation and the ports defined by the instantiated module is the ordered list—that is, the ports listed for the module instance must be in the same order as the ports listed in the module definition.

```

module adc4 (out, rem, in);
output [3:0] out ;           output rem;
input in;
electrical [3:0] out;
electrical in, rem, rem_chain;

adc2 hi2 (out[3:2], rem_chain, in) ;
adc2 lo2 (out[1:0], rem, rem_chain) ;
endmodule

module adc2 (out, remainder, in);
output [1:0] out ;           output remainder;
input in;
electrical [1:0] out ;
electrical in, remainder, r;

adc hi1 (out[1], r, in) ;
adc lo1 (out[0], remainder, r) ;
endmodule

module adc (out, remainder, in);
output out, remainder;
input in;
electrical out, in, remainder;
integer d;

    analog begin
        d = (V(in) > 0.5) ;
        V(out) <+ transition(d) ;
        V(remainder) <+ 2.0 * V(in) ;
        if (d)
            V(remainder) <+ -1.0 ;
    end
endmodule

```

8.3.5 Connecting module ports by name

The second way to connect module ports consists of explicitly linking the two names for each side of the connection—the name used in the module definition, followed by the name used in the instantiating module. This compound name is then placed in the list of

module connections. The name of port must be the name specified in the module definition. The name of port cannot be a bit select or a part select.

The port expression must be the name used by the instantiating module and can be one of the following:

- a simple node identifier
- a scalar member of a vector node or port declared within the module
- a sub-range of a vector node or port declared within the module
- a vector node formed as a result of the concatenation operator

The port expression is optional so that the instantiating module can document the existence of the port without connecting it to anything. The parentheses are required.

The two types of module port connections can not be mixed; connections to the ports of a particular module instance must be all by order or all by name.

```

module adc4 (out, rem, in);
input in;
output [3:0] out;           output rem;
electrical [3:0] out;
electrical in, rem, rem_chain;

adc2 hi (.in(in), .out(out[3:2]), .remainder(rem_chain)) ;
adc2 lo (.in(rem_chain), .out(out[1:0]), .remainder(rem)) ;
endmodule

module adc2 (out, in, remainder);
output [1:0] out;           output remainder;
input in;
electrical [1:0] out;
electrical in, remainder, r;

adc hi1 (out[1], r, in) ; // adc is same as defined in section 8.3.4
adc lo1 (out[0], remainder, r) ;
endmodule

```

Since these connections were made by port name, the order in which the connections appear is irrelevant.

8.3.6 Port connection rules

The following rules govern the way module ports are declared and the way they are interconnected.

8.3.6.1 Compatible discipline rule

All ports connected to a node must be compatible with each other as well as to the discipline of the node. For discussion on compatible disciplines, see section 3.6.

Ports of any discipline are compatible when connected to a ground node.

8.3.6.2 Matching size rule

A scalar port can be connected to a scalar node, and a vector port can be connected to a vector node or concatenated node expression of the matching width. In other words, sizes of the ports and nodes must match.

8.3.6.3 Resolving Discipline of Undeclared Interconnect Signal

Verilog-AMS supports undeclared interconnect between module instances when describing hierarchical structures. That is, a signal appearing in the connection list of a module instantiation need not appear in any port declaration or discipline declaration.

- an undeclared net segment (signal) that connects to one or more ports that are declared with a discrete domain discipline resolves (inherits) to that discrete discipline.
- If the ports are of different discrete domain disciplines then the resulting discipline is undetermined unless there is a **connect** (section 7.4) statement to specify the resulting discipline.
- If some or all of the ports are declared with continuous domain disciplines then the undeclared interconnect signal resolves to a continuous domain discipline type.

8.3.7 Inheriting Port Natures

If a node is missing a nature, it will inherit that nature from any port that connects to it. Typically such a situation occurs when

- a node is either implicitly or explicitly declared with an empty discipline.
- a conservative port connects to a node that is declared as a signal flow discipline.
- a signal-flow port with a potential nature connects to a signal-flow node declared with a flow nature, or visa versa.

As additional ports connect to the same node, it is possible for conflicts to develop. For example, connecting either an electrical or a mechanical port to a node with empty discipline results in no conflicts, but connecting both to the same node defined with an empty discipline does result in a conflict.

At each node there may be many different values of the absolute tolerance **abstol**. This may be because various ports connecting to the node have different, yet compatible, natures for either the potential, the flow, or both. Even if the natures are identical, the value of **abstol** may be overridden in the discipline of one or more of the ports. In such cases, all of the absolute tolerances must be satisfied at the node. This leads to applying the smallest tolerance value for all calculations involving such nodes.

8.3.8 Multi-disciplinary example

The example below shows how an application that spans multiple disciplines can be modeled in Verilog-AMS. The example models a DC-motor driven by a voltage source.

```

module motorckt();
  parameter real freq=100;

  electrical drive;
  mechanical shaft;

  motor m1 (drive, ground, shaft);
  vsource #(.freq(freq), .ampl(1.0)) v1 (drive, ground);

endmodule

// vp:          positive terminal [V,A]          vn:          negative terminal [V,A]
// shaft: motor shaft [rad,Nm]
//
// INSTANCE parameters
// Km = motor constant [Vs/rad]          Kf = flux constant [Nm/A]
// j = inertia factor [Nms^2/rad] D= drag (friction) [Nms/rad]
// Rm = motor resistance [Ohms]          Lm = motor inductance [H]
//
// A model of a DC motor driving a shaft

module motor(vp, vn, shaft);
  inout vp, vn, shaft;
  electrical vp, vn ;
  mechanical shaft ;

  parameter real Km = 4.5, Kf = 6.2;
  parameter real j = .004, D = 0.1;
  parameter real Rm = 5.0, Lm = .02;

  analog begin
    V(vp, vn) <+ Km*W(shaft) + Rm*I(vp, vn) + ddt(Lm*I(vp, vn));
    T(shaft) <+ Kf*I(vp, vn) - D*W(shaft) - ddt(j*W(shaft));
  end
endmodule

```

8.4 Hierarchical names

Every identifier in Verilog-AMS HDL description has a unique *hierarchical path name*. The hierarchy of modules and the definition of items such as named blocks within the modules define these names. The hierarchy of names can be viewed as a tree structure,

where each module instance or a named begin-end block defines a new hierarchical level, or scope, in a particular branch of the tree.

At the top of the name hierarchy are the names of modules of which no instances have been created. It is the *root* of the hierarchy. Inside any module, each module instance, and named begin-end block define a new branch of the hierarchy. Named blocks within named blocks also create new branches.

Each node in the hierarchical name tree is treated as a separate scope with respect to identifiers. A particular identifier can be declared at most once in any scope.

Any named object can be referenced uniquely in its full form by concatenating the names of the module instance or named blocks that contain it. The period character (.) is used to separate each of the names in the hierarchy. The complete path name to any object starts at a top-level module. This path name can be used from any level in the description. The first name in a path name can also be the top of a hierarchy that starts at the level where the path is being used.

```
module samplehold (in, cntrl, out);
input in, cntrl;
output out;
electrical in, cntrl, out;
electrical store, sample;
parameter real vthresh = 0.0;
parameter real cap = 10e-9;
```

```
amp op1 (in, sample, sample);
amp op2(store, out, out);
```

```
analog begin
  I(store) <+ cap * ddt(V(store));
  if (V(cntrl) > vthresh)
    V(store, sample) <+ 0;
  else
    I(store, sample) <+ 0;
end
endmodule
```

```
module amp(inp, inm, out);
input inp, inm;
output out;
electrical inp, inm, out;
parameter real gain=1e5;

analog begin
  V(out) <+ gain*V(inp,inm);
end
endmodule
```

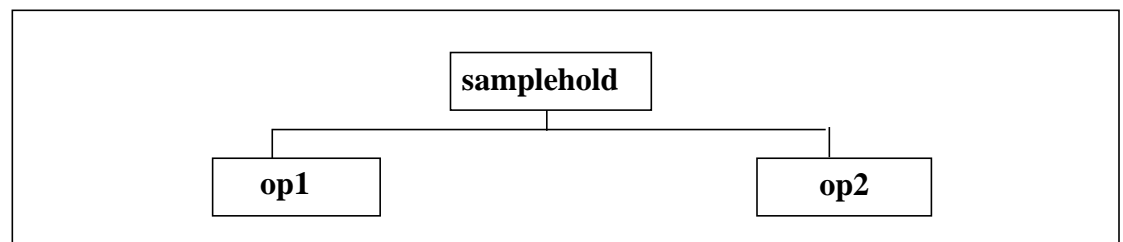


Figure 8-6: : Hierarchy in a model

samplehold	in, cntrl, out, sample, store, vthresh, cap
op1	op1.inp, op1.inm, op1.out, op1.gain
op2	op2.inp, op2.inm, op2.out, op2.gain

Figure 8-7: : Hierarchical path names in a model

From within an analog block, it is possible to use hierarchical name referencing to access signals on an external branch, but not external variables or parameters. When accessing external branches, a branch signal (its potential or flow) can be monitored (probed), or with source branches, contributions can be made to the output signal. However, contributing to an external switch branch is considered illegal.

It is illegal to indirectly assign to an external branch or contribute to an external branch that has indirect branch assignment.

8.5 Scope rules

The following two elements define a new scope in Verilog-AMS HDL:

modules
named blocks

An identifier can be used to declare only one item within a scope. This rule means it is illegal to declare two or more variables that have the same name, or to give an instance the same name as the name of the node connected to its output.

If an identifier is referenced directly (without a hierarchical path) within a named block, it must be declared either locally within the named block, or within a module, or named block that is higher in the same branch of the name tree that contains the named block. If it is declared locally, then the local item must be used; if not, the search continue upward until an item by that name is found or until a module boundary is encountered. The search can cross named block boundaries, but not module boundaries.

Because of the upward searching, path names that are not strictly on a downward path can be used.

Section 22

Using VPI routines

Sections 22 and 23 specify the Verilog Procedural Interface (VPI) for the Verilog HDL. This section describes how the VPI routines are used, and Section 23 defines each of the routines in alphabetical order.

22.1 The VPI interface

The VPI interface provides routines that allow Verilog product users to access information contained in a Verilog design, and that allow facilities to interact dynamically with a software product. Applications of the VPI interface can include delay calculators and annotators, connecting a Verilog simulator with other simulation and CAE systems, and customized debugging tasks.

The functions of the VPI interface can be grouped into two main areas:

- Dynamic software product interaction using VPI callbacks
- Access to Verilog HDL objects and simulation specific objects

22.1.1 VPI callbacks

Dynamic software product interaction shall be accomplished with a registered callback mechanism. VPI callbacks shall allow a user to request that a Verilog HDL software product, such as a logic simulator, call a user-defined application when a specific activity occurs. For example, the user can request that the user application `my_monitor()` be called when a particular net changes value, or that `my_cleanup()` be called when the software product execution has completed.

The VPI callback facility shall provide the user with the means to interact dynamically with a software product, detecting the occurrence of value changes, advancement of time, end of simulation, etc. This feature allows applications such as integration with other simulation systems, specialized timing checks, complex debugging features, etc.

The reasons for which callbacks shall be provided can be separated into four categories:

- *Simulation event* (e.g., a value change on a net or a behavioral statement execution)
- *Simulation time* (e.g., the end of a time queue or after certain amount of time)
- *Simulator action/feature* (e.g., the end of compile, end of simulation, restart, or enter interactive mode)
- *User-defined system task or function execution*

VPI callbacks shall be registered by the user with the functions `vpi_register_cb()` and `vpi_register_systf()`. These routines indicate the specific reason for the callback, the application to be called, and what system and user data shall be passed to the callback application when the callback occurs. A facility is also provided to call the callback functions when a Verilog HDL product is first invoked. A primary use of this facility shall be for registration of user-defined system tasks and functions.

22.1.2 VPI access to Verilog HDL objects and simulation objects

Accessible Verilog HDL objects and simulation objects and their relationships and properties are described using data model diagrams. These diagrams are presented in 22.5. The data diagrams indicate the routines and constants that are required to access and manipulate objects within an application environment. An associated set of routines to access these objects is defined in Section 23.

The VPI interface also includes a set of utility routines for functions such as handle comparison, file handling, and redirected printing, which are described in 23.12.

VPI routines provide access to objects in an *instantiated* Verilog design. An instantiated design is one where each instance of an object is uniquely accessible. For instance, if a module *m* contains wire *w* and is instantiated twice as *m1* and *m2*, then *m1.w* and *m2.w* are two distinct objects, each with its own set of related objects and properties.

The VPI interface is designed as a *simulation* interface, with access to both Verilog HDL objects and specific simulation objects. This simulation interface is different from a hierarchical language interface, which would provide access to HDL information but would not provide information about simulation objects.

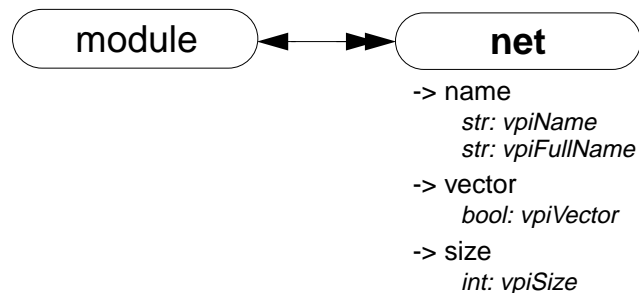
22.1.3 Error handling

To determine if an error occurred, the routine **vpi_chk_error()** shall be provided. The **vpi_chk_error()** routine shall return a nonzero value if an error occurred in the previously called VPI routine. Callbacks can be set up for when an error occurs as well. The **vpi_chk_error()** routine can provide detailed information about the error.

22.2 VPI object classifications

VPI objects are classified with data model diagrams. These diagrams provide a graphical representation of those objects within a Verilog design to which the VPI routines shall provide access. The diagrams shall show the relationships between objects and the properties of each object. Objects with sufficient commonality are placed in groups. Group relationships and properties apply to all the objects in the group.

As an example, this simplified diagram shows that there is a *one-to-many relationships* from objects of type **module** to objects of type **net**, and a *one-to-one relationship* from objects of type **net** to objects of type **module**. Objects of type **net** have properties **vpiName**, **vpiVector**, and **vpiSize**, with C data types string, Boolean, and integer respectively.



The VPI object data diagrams are presented in 22.5.

22.2.1 Accessing object relationships and properties

The VPI interface defines the C data type of **vpiHandle**. All objects are manipulated via a **vpiHandle** variable. Object handles can be accessed from a relationship with another object, or from a hierarchical name, as the following example demonstrates:

```

vpiHandle net;
net = vpi_handle_by_name("top.m1.w1", NULL);

```

This example call retrieves a handle to wire `top.m1.w1` and assigns it to the **vpiHandle** variable `net`. The `NULL` second argument directs the routine to search for the name from the top level of the design.

The VPI interface provides generic functions for tasks, such as traversing relationships and determining property values. One-to-one relationships are traversed with routine **vpi_handle()**. In the following example, the module that contains `net` is derived from a handle to that net:

```

vpiHandle net, mod;
net = vpi_handle_by_name("top.m1.w1", NULL);
mod = vpi_handle(vpiModule, net);

```

The call to **vpi_handle()** in the above example shall return a handle to module `top.m1`.

Properties of objects shall be derived with routines in the `vpi_get` family. The routine **vpi_get()** returns integer and Boolean properties. The routine **vpi_get_str()** accesses string properties. To retrieve a pointer to the full hierarchical name of the object referenced by handle `mod`, the following call would be made:

```

char *name = vpi_get_str(vpiFullName, mod);

```

In the above example, character pointer `name` shall now point to the string `"top.m1"`.

One-to-many relationships are traversed with an iteration mechanism. The routine **vpi_iterate()** creates an object of type **vpiIterator**, which is then passed to the routine **vpi_scan()** to traverse the desired objects. In the following example, each net in module `top.m1` is displayed:

```

vpiHandle itr;
itr = vpi_iterate(vpiNet, mod);
while (net = vpi_scan(itr) )
    vpi_printf("\t%s\n", vpi_get_str(vpiFullName, net) );

```

As the above examples illustrate, the routine naming convention is a *'vpi'* prefix with *'_'* word delimiters (with the exception of callback-related defined values, which use the *'cb'* prefix). Macro-defined types and properties have the *'vpi'* prefix, and they use capitalization for word delimiters.

The routines for traversing Verilog HDL structures and accessing objects are described in Section 23.

22.2.2 Delays and values

Properties are of type integer, boolean, real or string. Delay and logic value properties, however, are more complex and require specialized routines and associated structures. The routines **vpi_get_delays()** and **vpi_put_delays()** use structure pointers, where the structure contains the pertinent information about delays. Similarly, simulation values are also handled with the routines **vpi_get_value()** and **vpi_put_value()**, along with an associated set of structures. The derivatives are handled with the routines **vpi_decl_deriv()** and **vpi_put_deriv()**.

The routines and C structures for handling delays, derivatives and logic values are presented in Section 23.

22.3 List of VPI routines by functional category

The VPI routines can be divided into groups based on primary functionality.

- VPI routines for simulation-related callbacks
- VPI routines for system task/function callbacks
- VPI routines for traversing Verilog HDL hierarchy
- VPI routines for accessing properties of objects
- VPI routines for accessing objects from properties

- VPI routines for delay processing
- VPI routines for logic and strength value processing
- VPI routines for task parameters derivatives processing
- VPI routines for analysis and simulation time processing
- VPI routines for miscellaneous utilities

Tables 22-1 through 22-9 list the VPI routines by major category. Section 23 defines each of the VPI routines, listed in alphabetical order.

Table 22-1—VPI routines for simulation related callbacks

To	Use
Register a simulation-related callback	vpi_register_cb()
Remove a simulation-related callback	vpi_remove_cb()
Get information about a simulation-related callback	vpi_get_cb_info()

Table 22-2—VPI routines for system task/function callbacks

To	Use
Register a system task/function callback	vpi_register_systf()
Get information about a system task/function callback	vpi_get_systf_info()

Table 22-3—VPI routines for traversing Verilog HDL hierarchy

To	Use
Obtain a handle for an object with a one-to-one relationship	vpi_handle()
Obtain handles for objects in a one-to-many relationship	vpi_iterate() vpi_scan()
Obtain a handles for an object in a many-to-one relationship	vpi_handle_multi()

Table 22-4—VPI routines for accessing properties of objects

To	Use
Get the value of objects with types of int or bool	vpi_get()
Get the value of objects with types of string	vpi_get_str()
Get the value of objects with types of real	vpi_get_real()

Table 22-5—VPI routines for accessing objects from properties

To	Use
Obtain a handle for a named object	vpi_handle_by_name()
Obtain a handle for an indexed object	vpi_handle_by_index()

Table 22-6—VPI routines for delay processing

To	Use
Retrieve delays or timing limits of an object	vpi_get_delays()
Write delays or timing limits to an object	vpi_put_delays()

Table 22-7—VPI routines for logic, real and strength value processing

To	Use
Retrieve logic value or strength value of an object	vpi_get_value()
Write logic value or strength value to an object	vpi_put_value()
Retrieve real value of an object	vpi_get_real()

Table 22-8—VPI routines for task/function parameters derivatives processing

To	Use
Declare a partial derivative between two task/function parameters	vpi_decl_deriv()
Write a partial derivative value	vpi_put_deriv()

Table 22-9—VPI routines for analysis and simulation time processing

To	Use
Find the current simulation time or the scheduled time of future events	vpi_get_time()
Find the current simulation time value in the continuous domain.	vpi_get_continuous_time()
Find the current simulation time delta value in continuous domain.	vpi_get_continuous_delta()
Declare a discontinuity order.	vpi_decl_discontinuity()


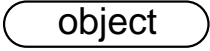



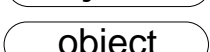
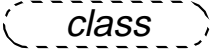
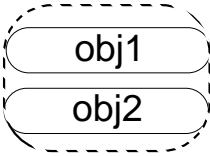
Table 22-10—VPI routines for miscellaneous utilities

To	Use
Write to stdout and the current log file	vpi_printf()
Open a file for writing	vpi_mcd_open()
Close one or more files	vpi_mcd_close()
Write to one or more files	vpi_mcd_printf()
Retrieve the name of an open file	vpi_mcd_name()
Retrieve data about product invocation options	vpi_get_vlog_info()
See if two handles refer to the same object	vpi_compare_objects()
Obtain error status and error information about the previous call to a VPI routine	vpi_chk_error()
Free memory allocated by VPI routines	vpi_free_object()

22.4 Key to object model diagrams

This clause contains the keys to the symbols used in the object model diagrams. Keys are provided for objects and classes, traversing relationships, and accessing properties.

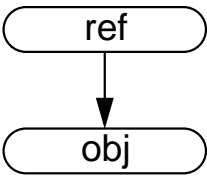
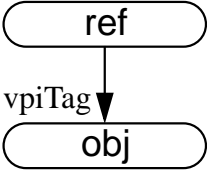
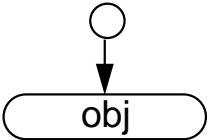
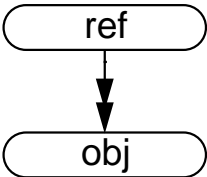
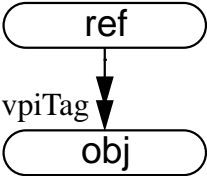
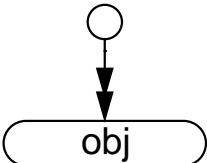
22.4.1 Diagram key for objects and classes

	<p>Object Definition:</p> <p>Bold letters in a solid enclosure indicate an object definition. The properties of the object are defined in this location.</p>
	<p>Object Reference:</p> <p>Normal letters in a solid enclosure indicate an object reference.</p>
   	<p>Class Definition:</p> <p><i>Bold italic</i> letters in a dotted enclosure indicate a class definition, where the class groups other objects and classes. Properties of the class are defined in this location. The class definition can contain an object definition.</p>
	<p>Class Reference:</p> <p><i>Italic</i> letters in a dotted enclosure indicate a class reference.</p>
	<p>Unnamed Class:</p> <p>A dotted enclosure with no name is an unnamed class. It is sometimes convenient to group objects although they shall not be referenced as a group elsewhere, so a name is not indicated.</p>

22.4.2 Diagram key for accessing properties

<p>obj</p> <p>-> vector <i>bool: vpiVector</i></p> <p>-> size <i>int: vpiSize</i></p>	<p>Integer and Boolean properties are accessed with the routine vpi_get().</p> <p>Example: Given a vpiHandle obj_h to an object of type vpiObj, get the size of the object.</p> <pre>bool vect_flag = vpi_get(vpivector, obj_h); int size = vpi_get_size(vpiSize, obj_h);</pre>
<p>obj</p> <p>-> name <i>str: vpiName</i> <i>str: vpiFullName</i></p>	<p>String properties are accessed with routine vpi_get_str().</p> <p>Example:</p> <pre>char name[nameSize]; vpi_get_str(vpiName, obj_h);</pre>
<p>object</p> <p>-> complex <i>func1()</i> <i>func2()</i></p>	<p>Complex properties for time and logic value are accessed with the indicated routines. See the descriptions of the routines for usage.</p>

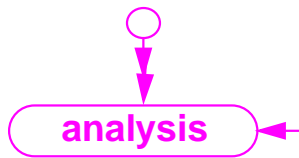
22.4.3 Diagram key for traversing relationships

	<p>A single arrow indicates a <i>one-to-one</i> relationship accessed with the routine vpi_handle().</p> <p>Example: Given vpiHandle variable ref_h of type ref, access obj_h of type vpiObj:</p> <pre>obj_h = vpi_handle(vpiObj, ref_h);</pre>
	<p>A tagged <i>one-to-one</i> relationship is traversed similarly, using vpiTag instead of vpiObj:</p> <p>Example:</p> <pre>obj_h = vpi_handle(vpiTag, ref_h);</pre>
	<p>A top-level <i>one-to-one</i> relationship is traversed similarly, using NULL instead of ref_h:</p> <p>Example:</p> <pre>obj_h = vpi_handle(vpiObj, NULL);</pre>
	<p>A double arrow indicates a <i>one-to-many</i> relationship accessed with the routine vpi_scan().</p> <p>Example: Given vpiHandle variable ref_h of type ref, scan objects of type vpiObj:</p> <pre>itr = vpi_iterate(vpiObj, ref_h); while (obj_h = vpi_scan(itr)) /* process 'obj_h' */</pre>
	<p>A tagged <i>one-to-many</i> relationship is traversed similarly, using vpiTag instead of vpiObj:</p> <p>Example:</p> <pre>itr = vpi_iterate(vpiTag, ref_h); while (obj_h = vpi_scan(itr)) /* process 'obj_h' */</pre>
	<p>A top-level <i>one-to-many</i> relationship is traversed similarly, using NULL instead of ref_h:</p> <p>Example:</p> <pre>itr = vpi_iterate(vpiObj, NULL); while (obj_h = vpi_scan(itr)) /* process 'obj_h' */</pre>

22.5 Object data model diagrams

Subclauses 22.5.2 through 22.5.26 contain the data model diagrams that define the accessible objects and groups of objects, along with their relationships and properties.

22.5.1 Analysis

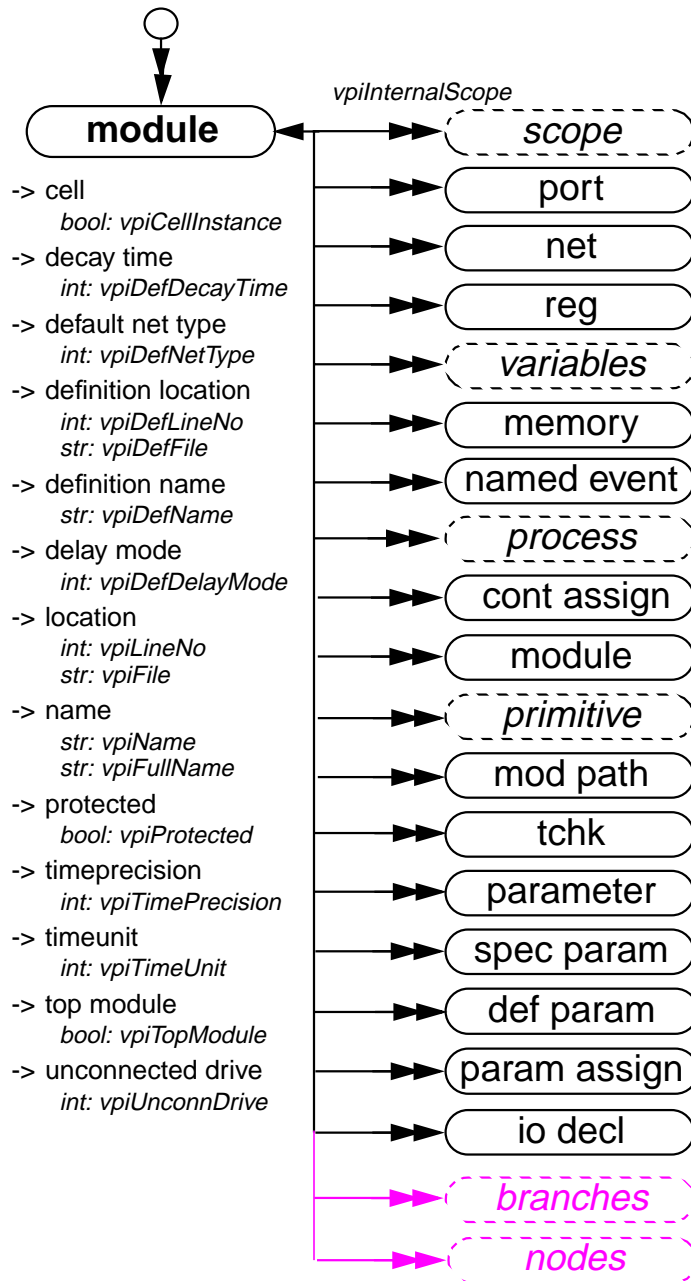


- > dc analysis
bool: vpiTransientAnalysis
- > transient analysis
bool: vpiTransientAnalysis
- > start time
real: vpiContinuousStartTime
- > end time
real: vpiContinuousEndTime
- > maximum time step
real: vpiDefNetType
- > ac analysis
bool: vpiAcAnalysis
- > start frequency
real: vpiContinuousStartTime
- > end frequency
real: vpiContinuousEndTime
- > maximum time step
real: vpiContinuousMaxTimeStep

NOTES

- 1—Top-level modules shall be accessed using **vpi_iterate()** with a NULL reference object.
- 2—Passing a NULL handle to **vpi_get()** with types **vpiTimePrecision** or **vpiTimeUnit** shall return the smallest time precision of all modules in the instantiated design.

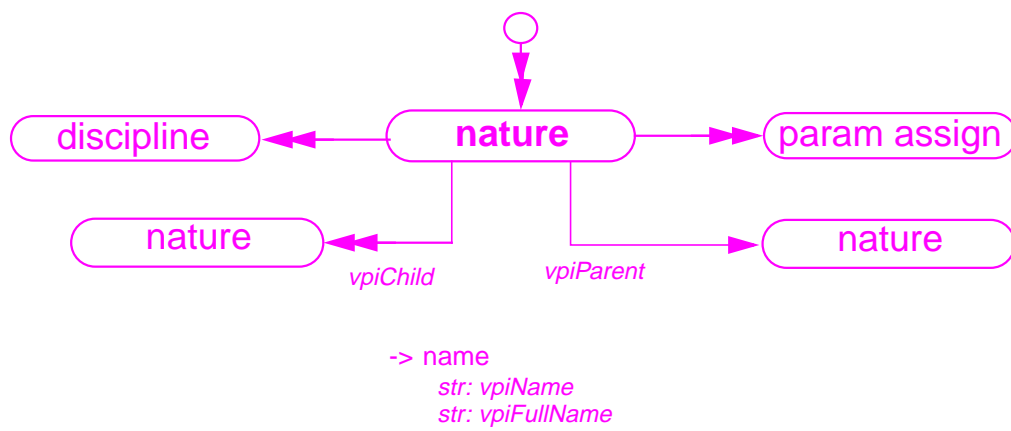
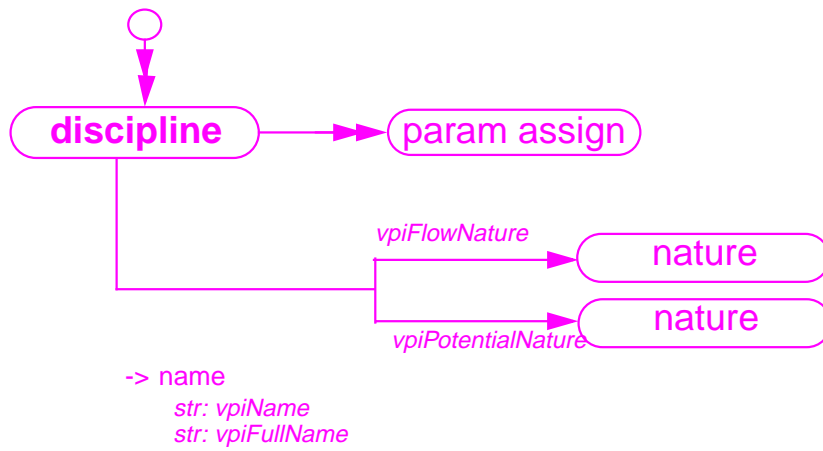
22.5.2 Module



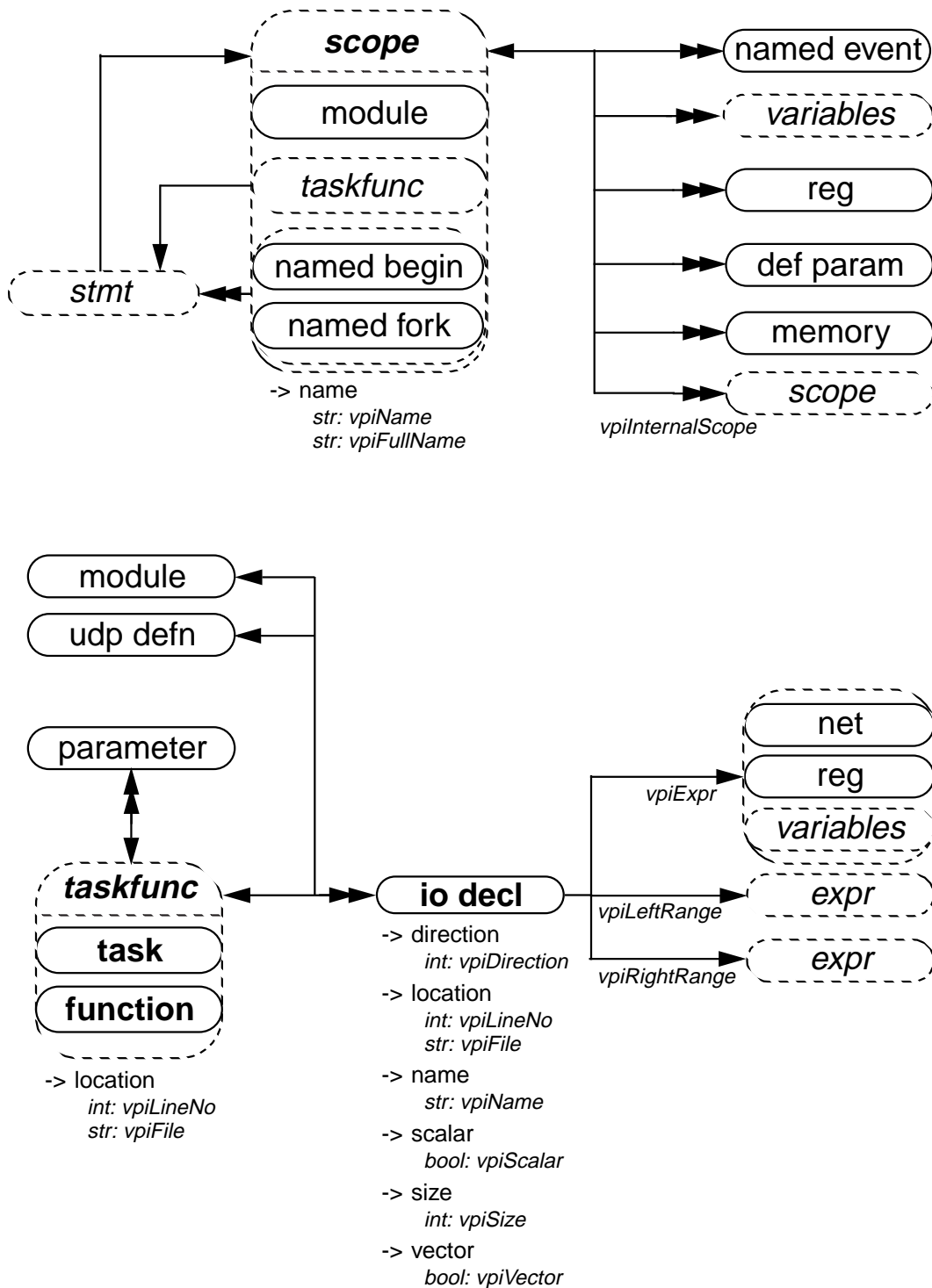
NOTES

- 1—Top-level modules shall be accessed using **vpi_iterate()** with a NULL reference object.
- 2—Passing a NULL handle to **vpi_get()** with types **vpiTimePrecision** or **vpiTimeUnit** shall return the smallest time precision of all modules in the instantiated design.

22.5.3 Nature, Discipline

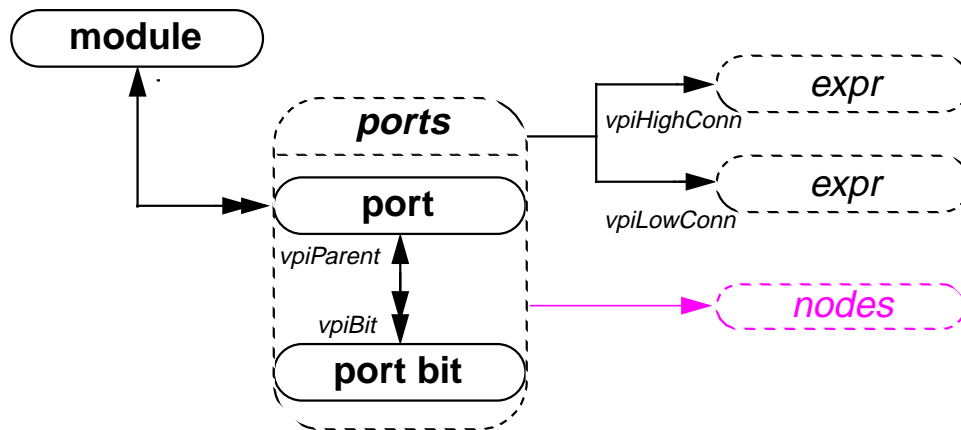


22.5.4 Scope, task, function, IO declaration



NOTE—A Verilog HDL function shall contain an object with the same name, size, and type as the function.

22.5.5 Ports

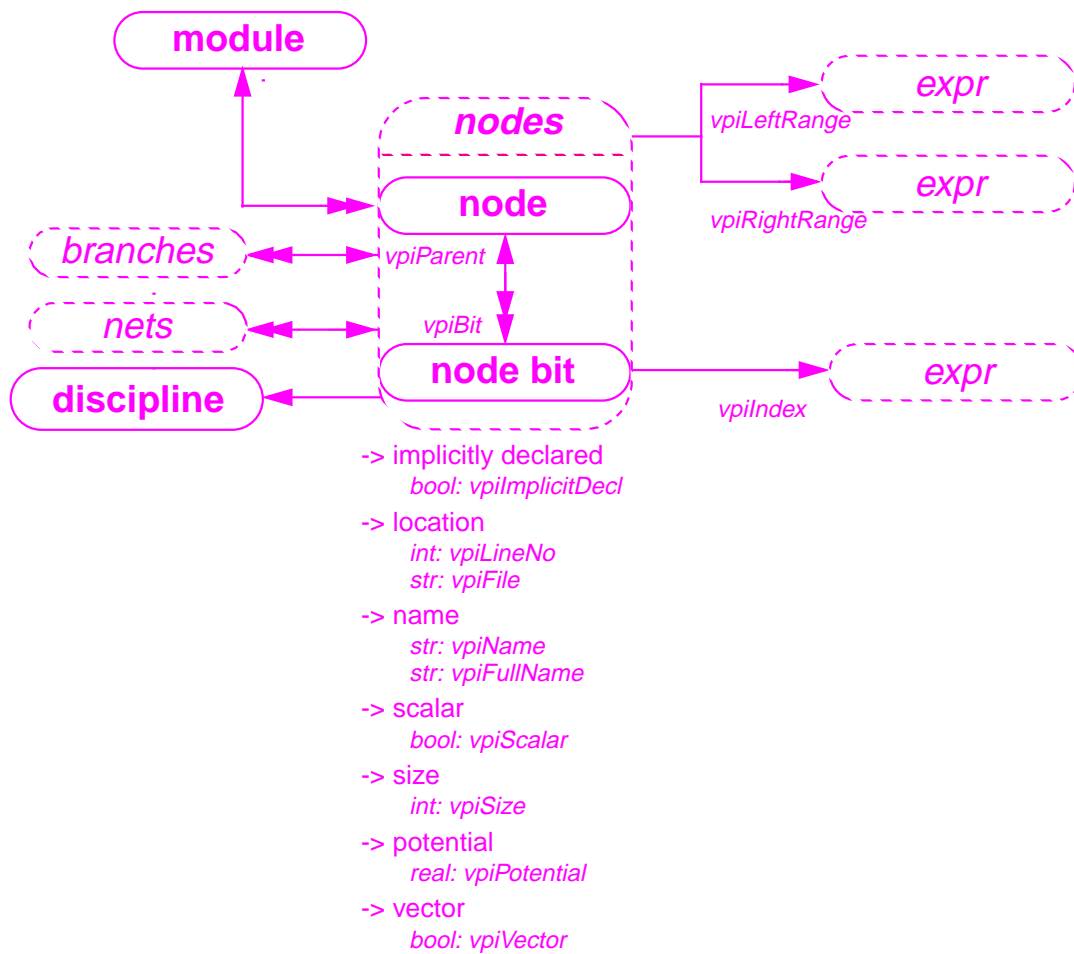


- > connected by name
bool: vpiConnByName
- > delay (mipd)
vpi_get_delays()
vpi_put_delays()
- > direction
int: vpiDirection
- > explicitly named
bool: vpiExplicitName
- > index
int: vpiPortIndex
- > location
int: vpiLineNo
str: vpiFile
- > name
str: vpiName
str: vpiFullName
- > scalar
bool: vpiScalar
- > size
int: vpiSize
- > vector
bool: vpiVector

NOTES

- 1—**vpiHighConn** shall indicate the hierarchically higher (closer to the top module) port connection.
- 2—**vpiLowConn** shall indicate the lower (further from the top module) port connection.
- 3—Properties *scalar* and *vector* shall indicate if the port is 1 bit or more than 1 bit. They shall not indicate anything about what is connected to the port.
- 4—Properties *index* and *name* shall not apply for port bits.
- 5—If a port is explicitly named, then the explicit name shall be returned. If not, and a name exists, then that name shall be returned. Otherwise, NULL shall be returned.
- 6—**vpiPortIndex** can be used to determine the port order.

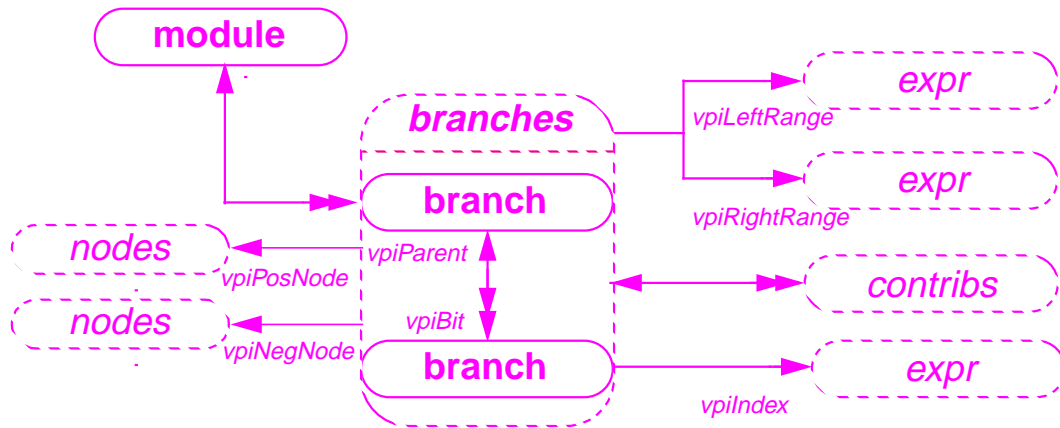
22.5.6 Nodes



NOTES

- 1—Properties *scalar* and *vector* shall indicate if the node is 1 bit or more than 1 bit.
- 2—Property *potential* shall indicate the node to ground potential

22.5.7 Branches



-> implicitly declared
bool: *vpiImplicitDecl*

-> location
int: *vpiLineNo*
str: *vpiFile*

-> name
str: *vpiName*
str: *vpiFullName*

-> scalar
bool: *vpiScalar*

-> size
int: *vpiSize*

-> discipline
int: *vpiDiscipline*

-> potential
real: *vpiPotential*

-> flow
real: *vpiFlow*

-> vector
bool: *vpiVector*

-> flow source
bool: *vpiZFlowSrc*

-> potential source
bool: *vpiPotentialSrc*

-> equation target
bool: *vpiEqnTarget*

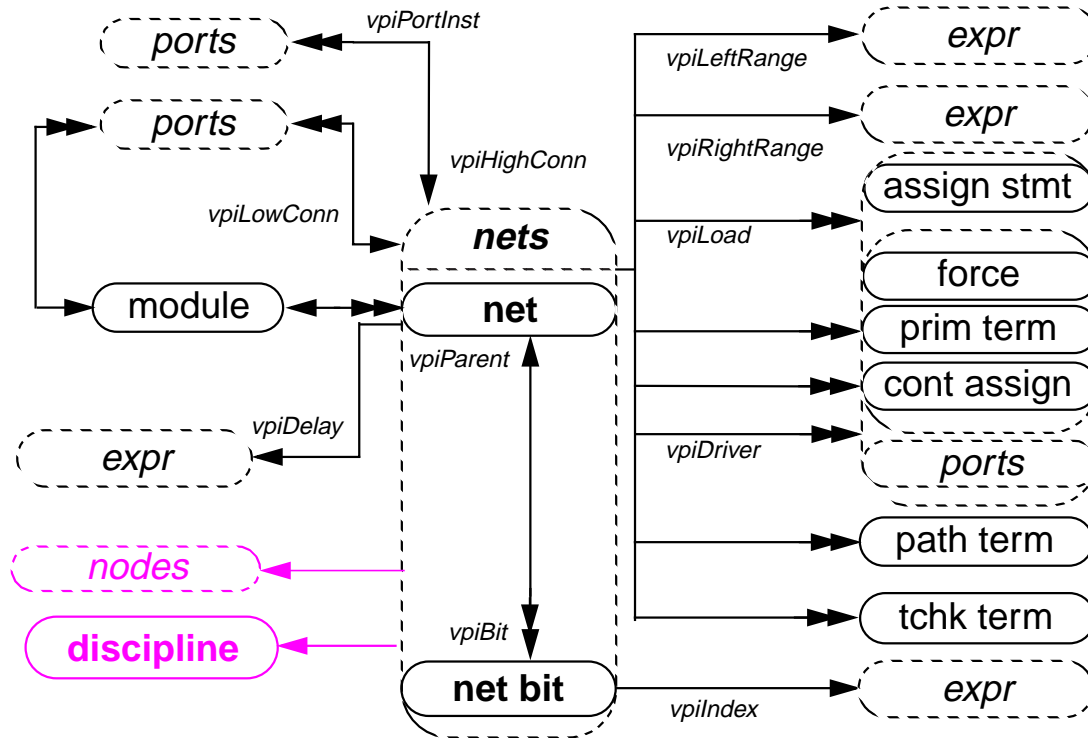
NOTES

1—Properties *scalar* and *vector* shall indicate if the node is 1 bit or more than 1 bit.

2—Property *potential* shall indicate the potential of **vpiPosNode** with respect to **vpiNegNode**

3—Property *flow* shall indicate the flow through the branch the reference (positive sign) direction is positive to negative

22.5.8 Nets



-> delay
 vpi_get_delays()
-> expanded
 bool: vpiExpanded
-> implicitly declared
 bool: vpiImplicitDecl
-> location
 int: vpiLineNo
 str: vpiFile
-> name
 str: vpiName
 str: vpiFullName

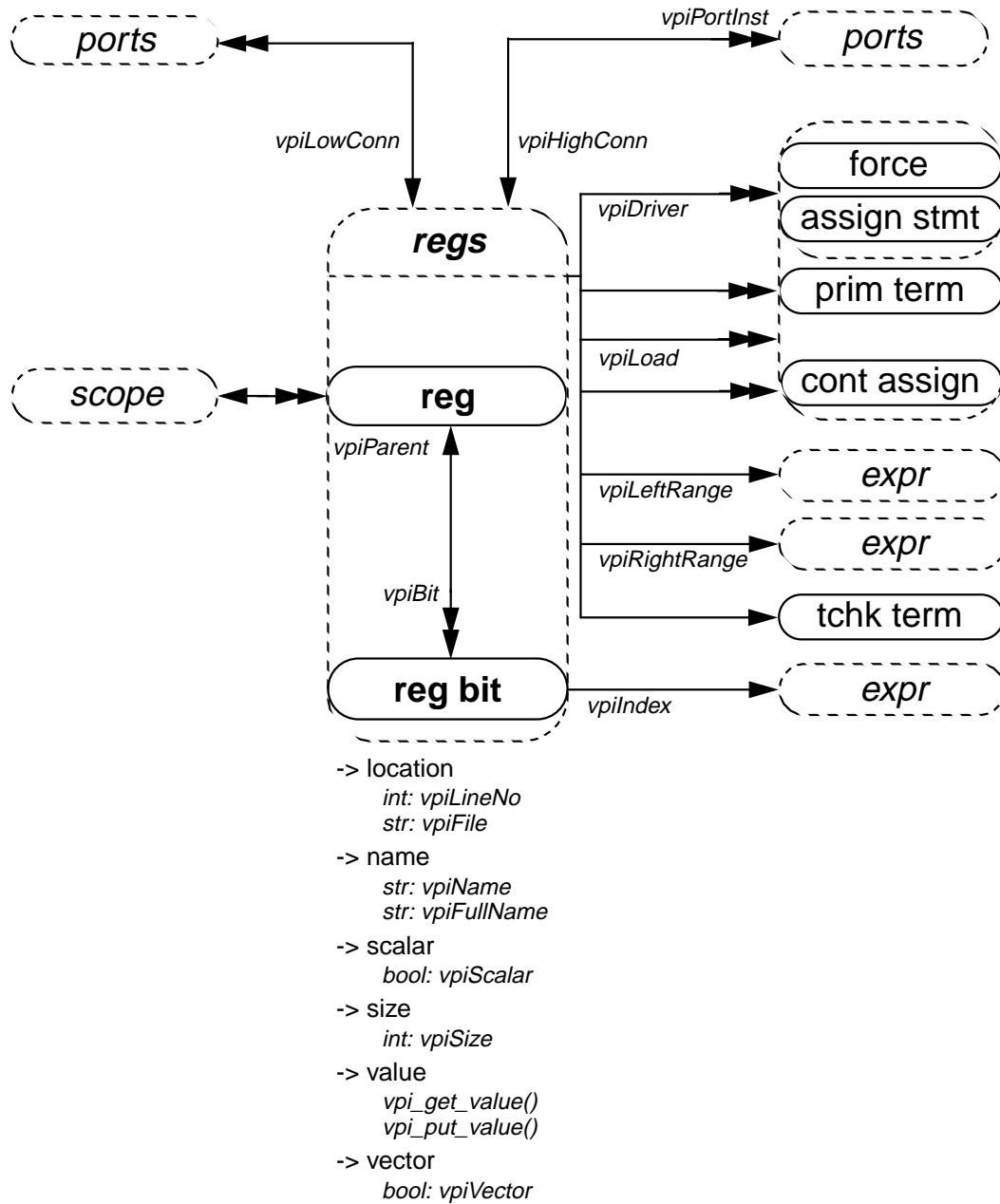
-> net decl assign
 bool: vpiNetDeclAssign
-> net type
 int: vpiNetType
-> scalar
 bool: vpiScalar
-> scaled declaration
 bool: vpiExplicitScaled
-> size
 int: vpiSize
-> domain
 int vpiDomain

-> strength
 int: vpiStrength0
 int: vpiStrength1
 int: vpiChargeStrength
-> value
 vpi_get_value()
 vpi_put_value()
-> vector
 bool: vpiVector
-> vectored declaration
 bool: vpiExplicitVectored

NOTES

- 1—For vectors, net bits shall be available regardless of vector expansion.
- 2—Continuous assignments and primitive terminals shall be accessed regardless of hierarchical boundaries.
- 3—Continuous assignments and primitive terminals shall only be accessed from scalar nets or bit selects.
- 4—For **vpiPortInst** and **vpiPort**, if the reference handle is a bit or the entire vector, the relationships shall return a handle to either a port bit or the entire port, respectively.
- 5—For implicit nets, **vpiLineNo** shall return 0, and **vpiFile** shall return the filename where the implicit net is first referenced.
- 6—Only active forces and assign statements shall be returned for **vpiLoad**.
- 7—Only active forces shall be returned for **vpiDriver**.
- 8—**vpiDriver** shall also return ports that are driven by objects other than nets and net bits.

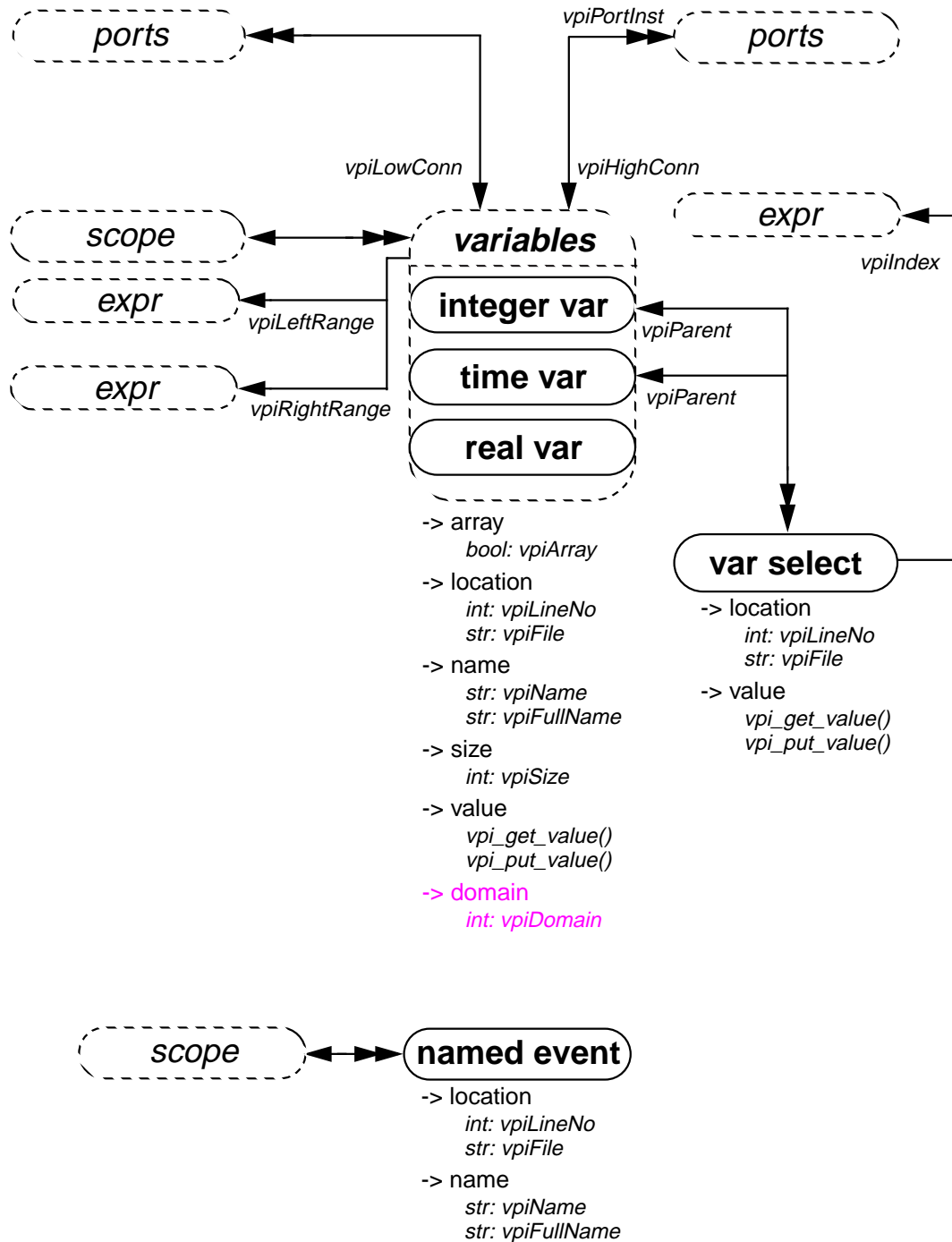
22.5.9 Regs



NOTES

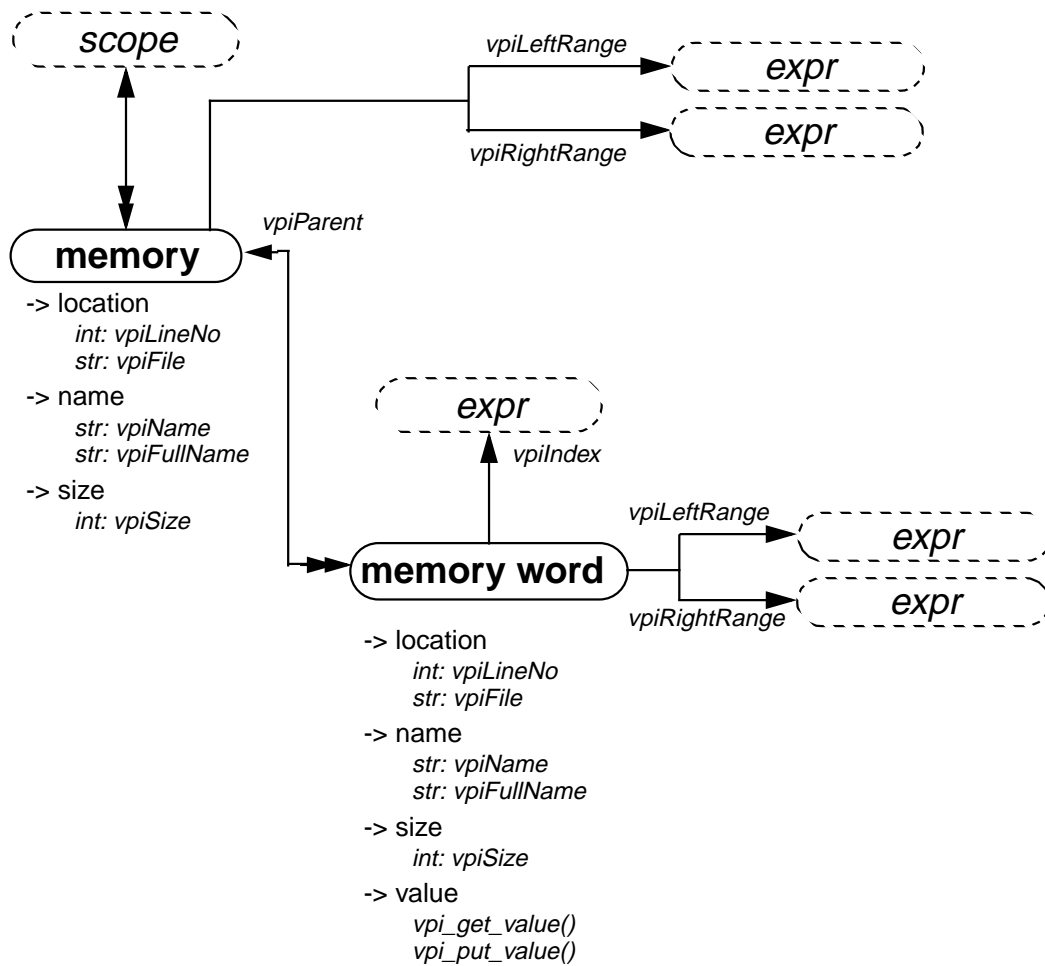
- 1—Continuous assignments and primitive terminals shall be accessed regardless of hierarchical boundaries.
- 2—Continuous assignments and primitive terminals shall only be accessed from scalar regs and bit selects.
- 3—Only active forces and assign statements shall be returned for **vpiLoad** and **vpiDriver**.

22.5.10 Variables, named event



NOTE—**vpiLeftRange** and **vpiRightRange** shall be invalid for reals, since there cannot be arrays of reals.

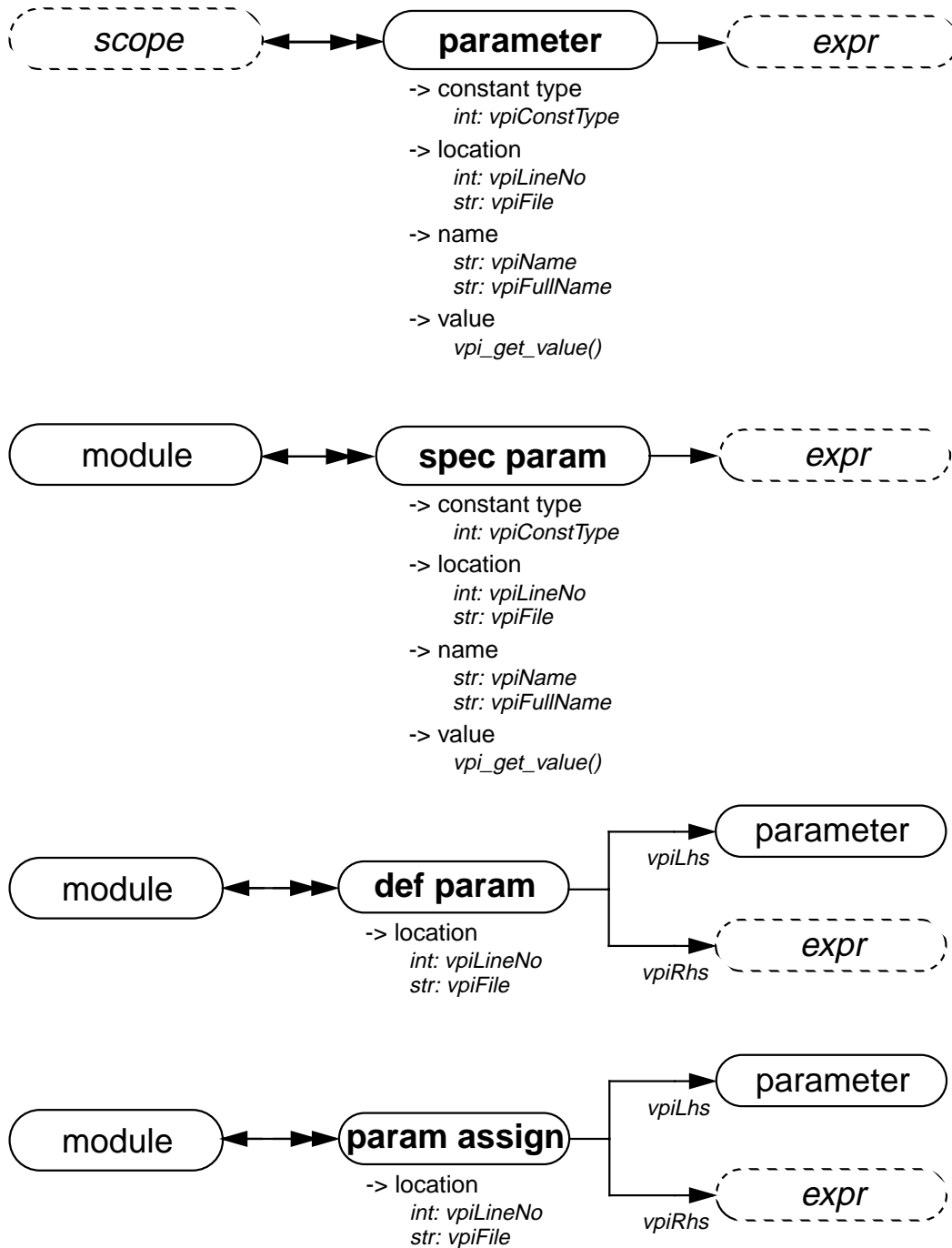
22.5.11 Memory



NOTES

- 1—**vpiSize** for a memory shall return the number of words in the memory.
- 2—**vpiSize** for a memory word shall return the number of bits in the word.

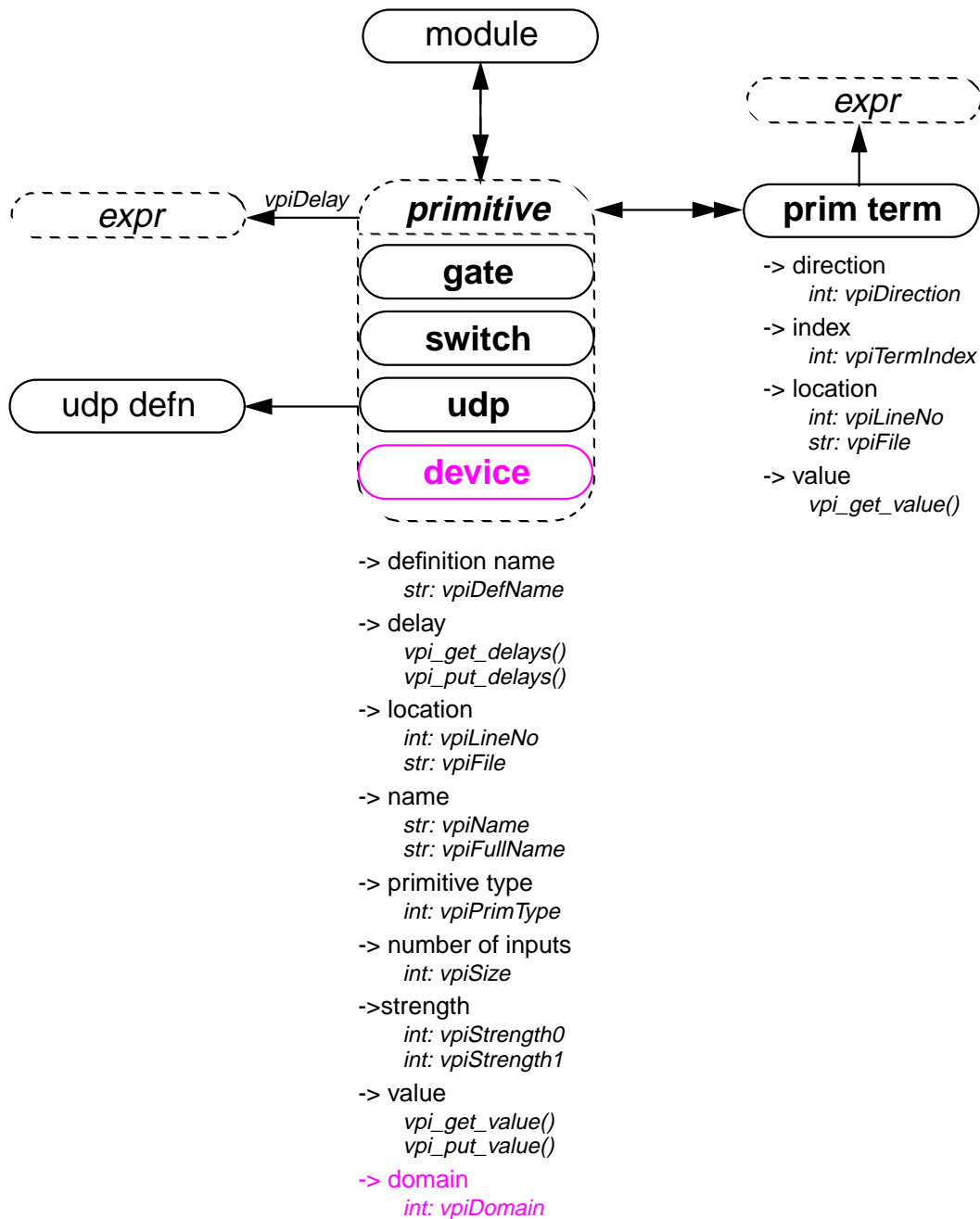
22.5.12 Parameter, specparam



NOTES

- 1—Obtaining the value from the object **parameter** shall return the final value of the parameter after all module instantiation overrides and defparams have been resolved.
- 2—**vpiLhs** from a param assign object shall return a handle to the overridden parameter.

22.5.13 Primitive, prim term

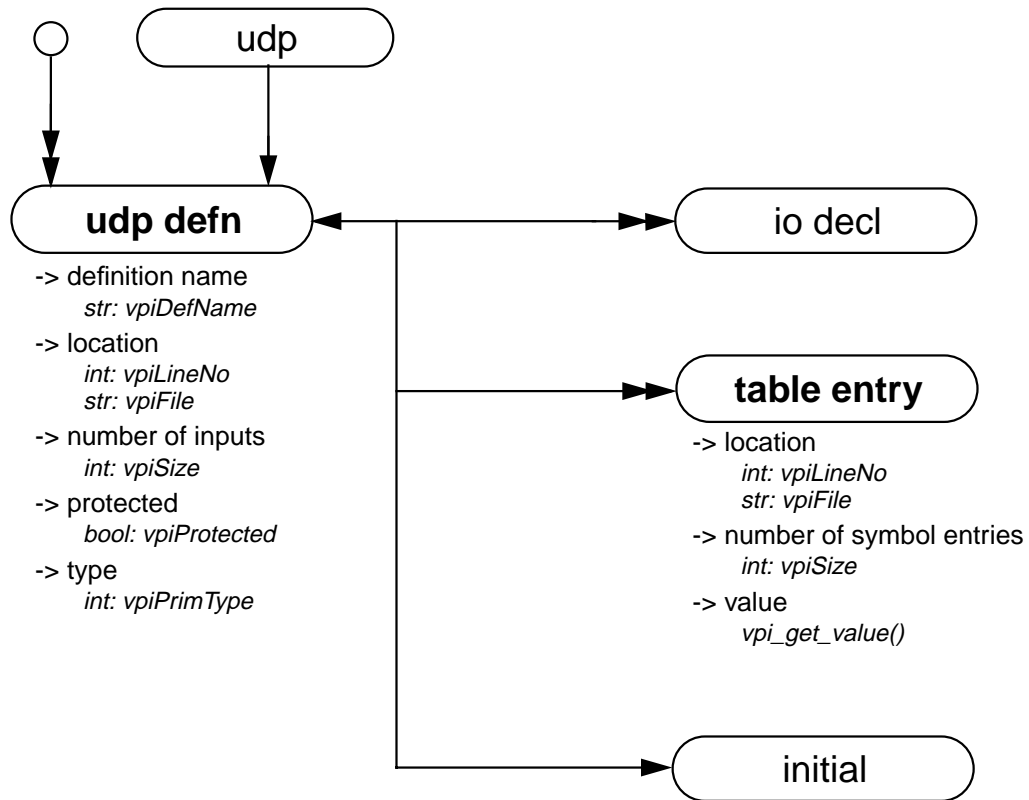


NOTES

1—**vpiSize** shall return the number of inputs.

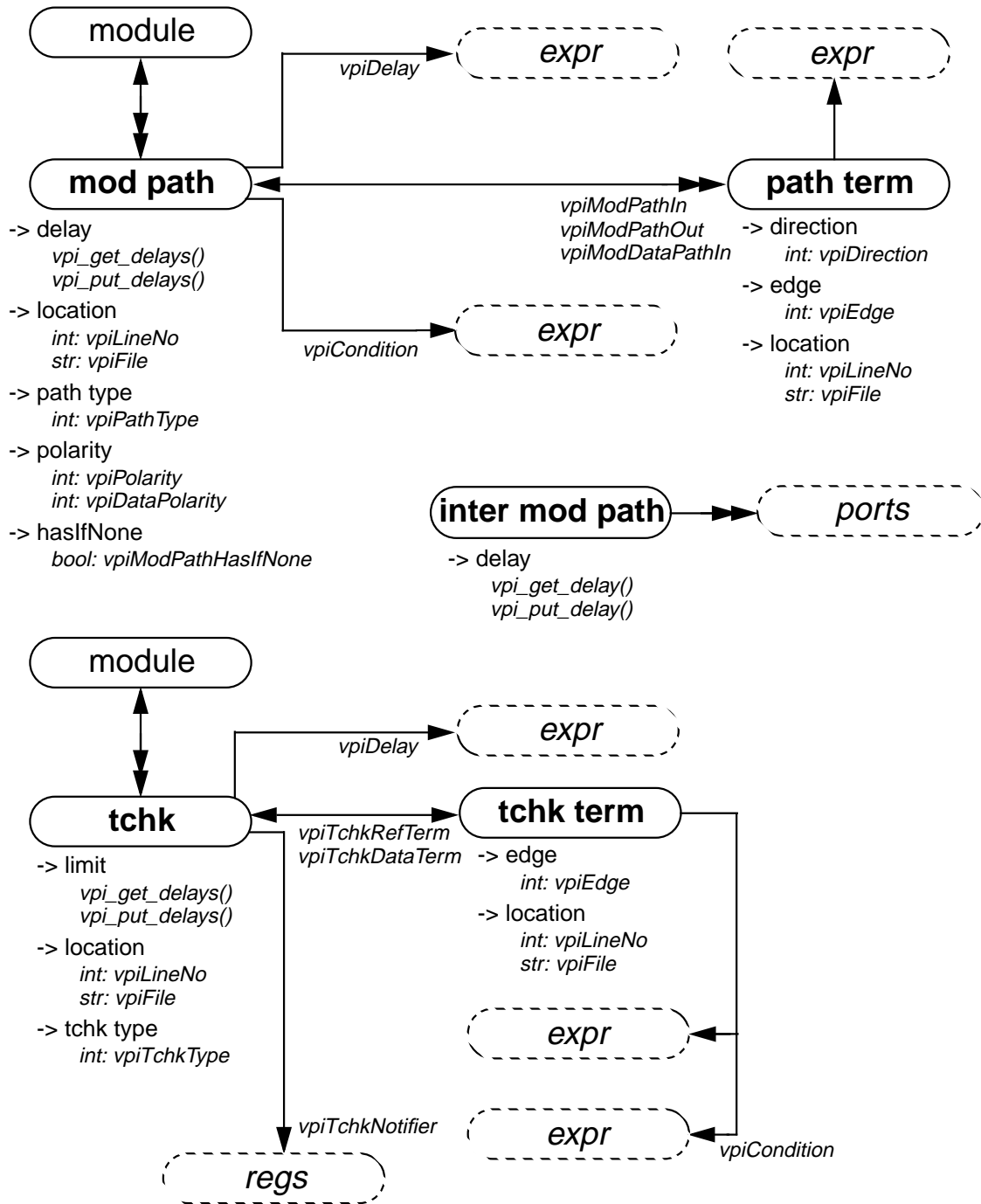
2—For primitives, **vpi_put_value()** shall only be used with sequential UDP primitives.

22.5.14 UDP



NOTE—Only string (decompilation) and vector (ASCII values) shall be obtained for table entry objects using **vpi_get_value()**. Refer to the definition of **vpi_get_value()** for additional details.

22.5.15 Module path, timing check, intermodule path

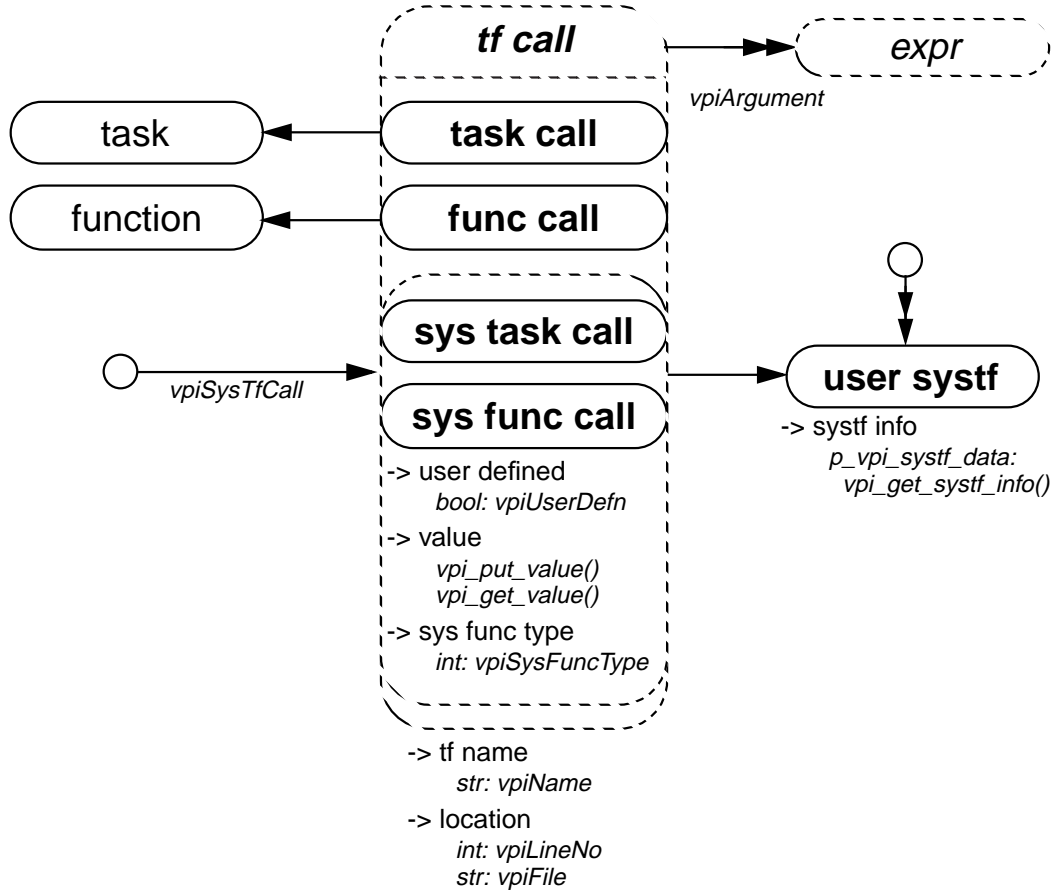


NOTES

1—The **vpiTchkRefTerm** is the first terminal for all tchks except **\$setup**, where **vpiTchkDataTerm** is the first terminal and **vpiTchkRefTerm** is the second terminal.

2—To get to an intermodule path, **vpi_handle_multi(vpiInterModPath, port1, port2)** can be used.

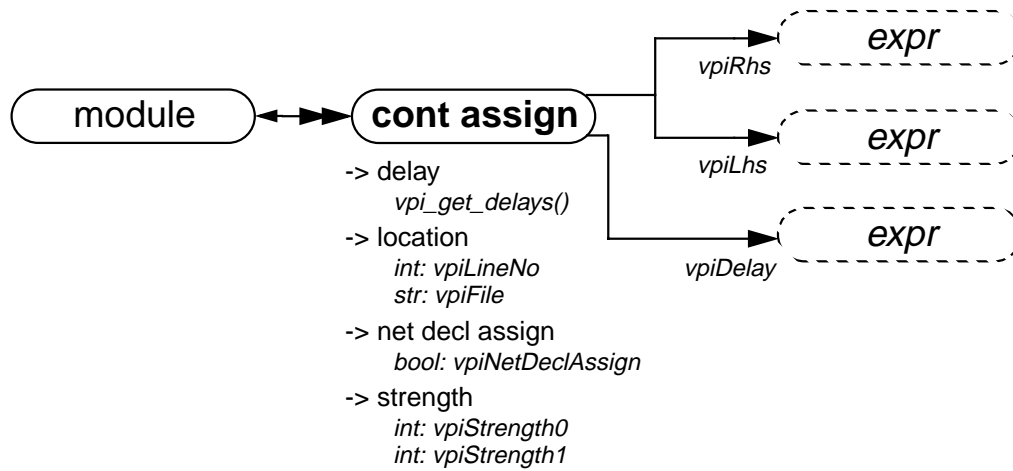
22.5.16 Task and function call



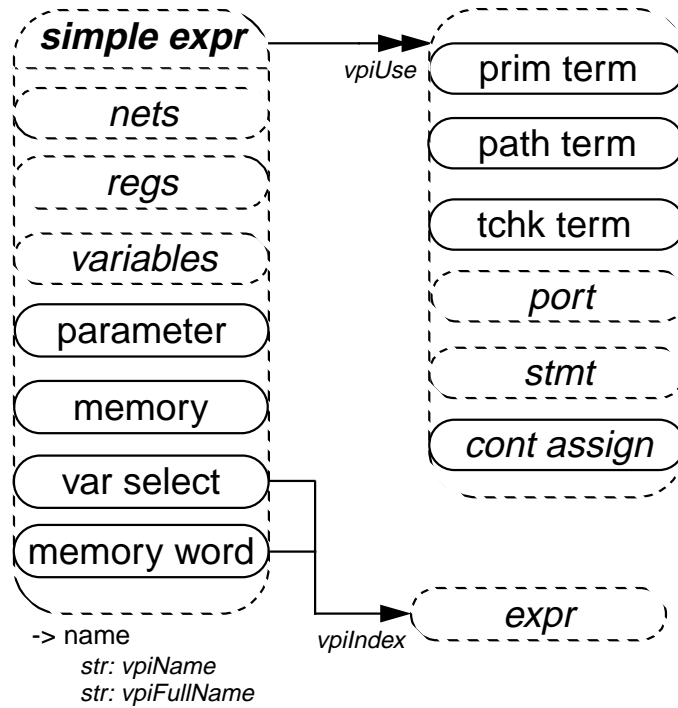
NOTES

- 1—The system task or function that invoked an application shall be accessed with **vpi_handle(vpiSysTfCall, NULL)**
- 2—**vpi_get_value()** shall return the current value of the system function.
- 3—If the **vpiUserDefn** property of a system task or function call is true, then the properties of the corresponding systf object shall be obtained via **vpi_get_systf_info()**.
- 4—All user-defined system tasks or functions shall be retrieved using **vpi_iterate()**, with **vpiUserSystf** as the type argument, and a NULL reference argument.

22.5.17 Continuous assignment



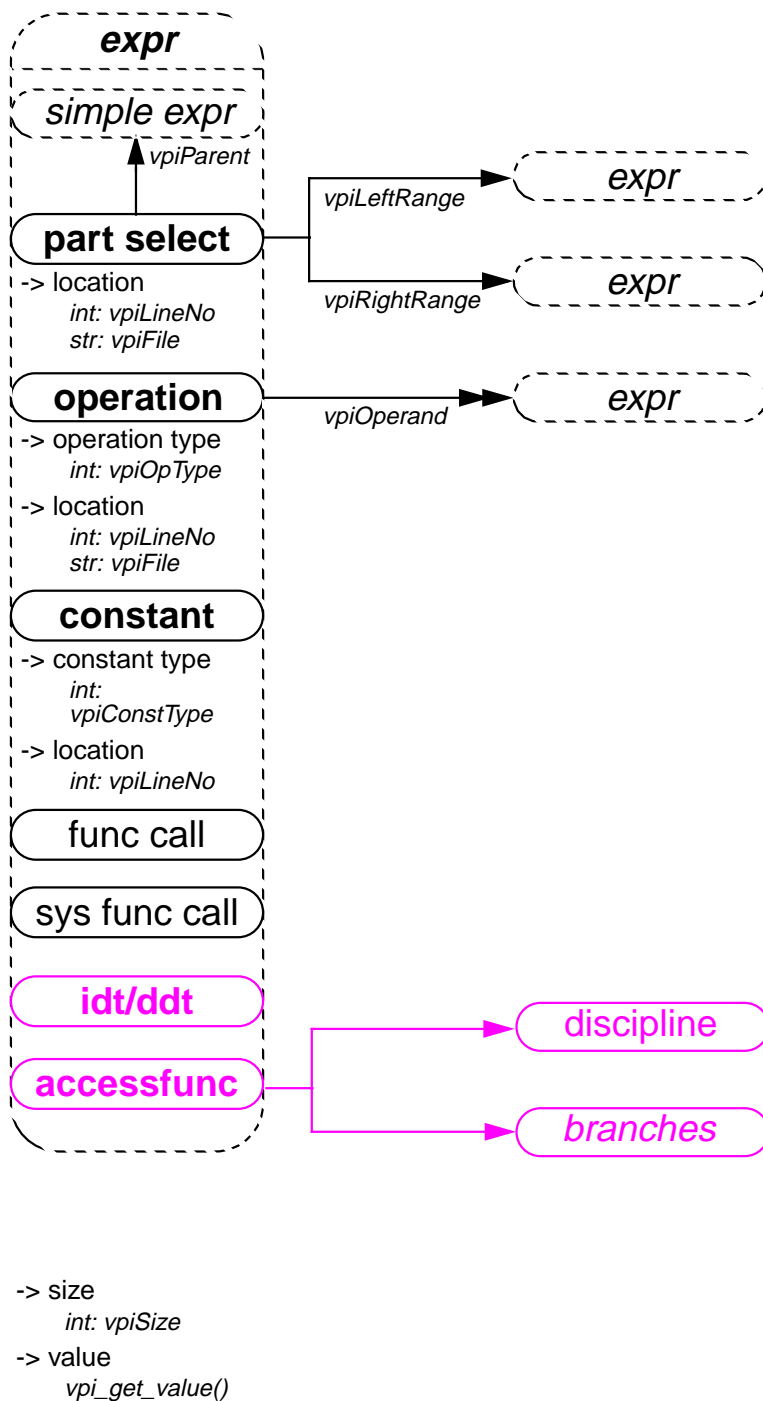
22.5.18 Simple expressions



NOTES

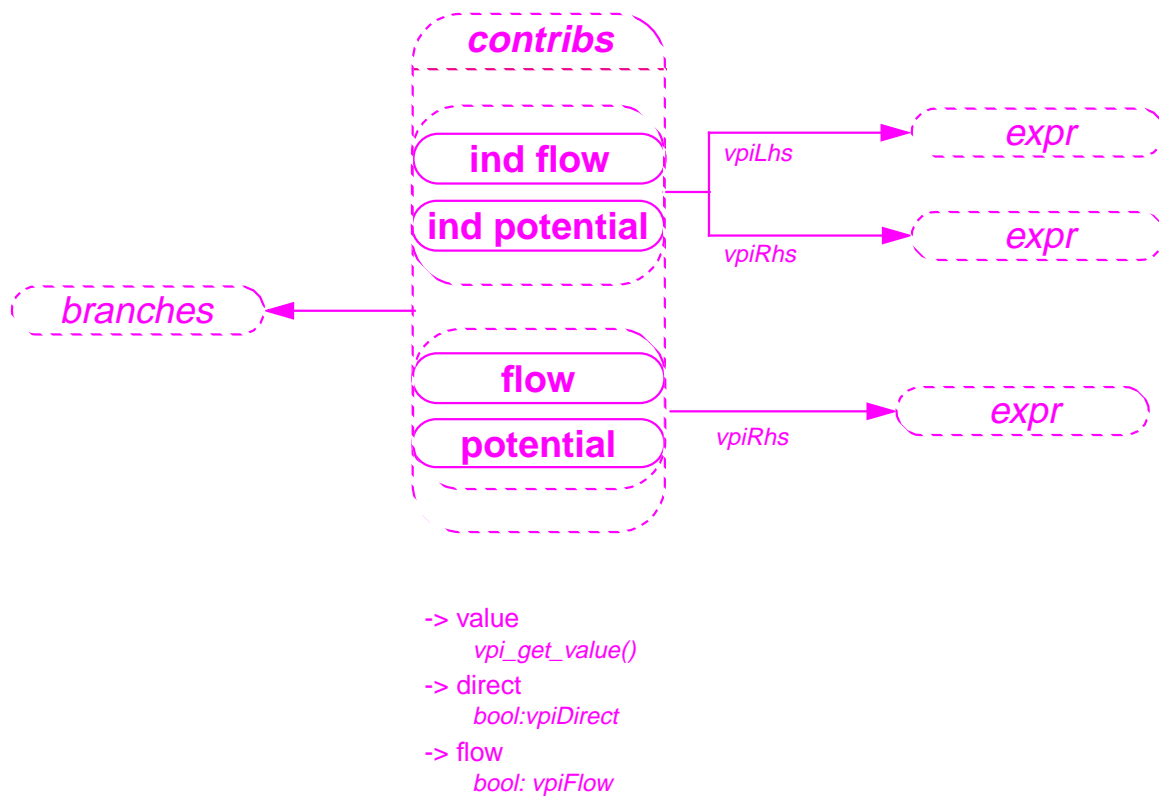
- 1—For vectors, the **vpiUse** relationship shall access any use of the vector or part-selects or bit-selects thereof.
- 2—For bit-selects, the **vpiUse** relationship shall access any specific use of that bit, any use of the parent vector, and any part-select that contains that bit.

22.5.19 Expressions

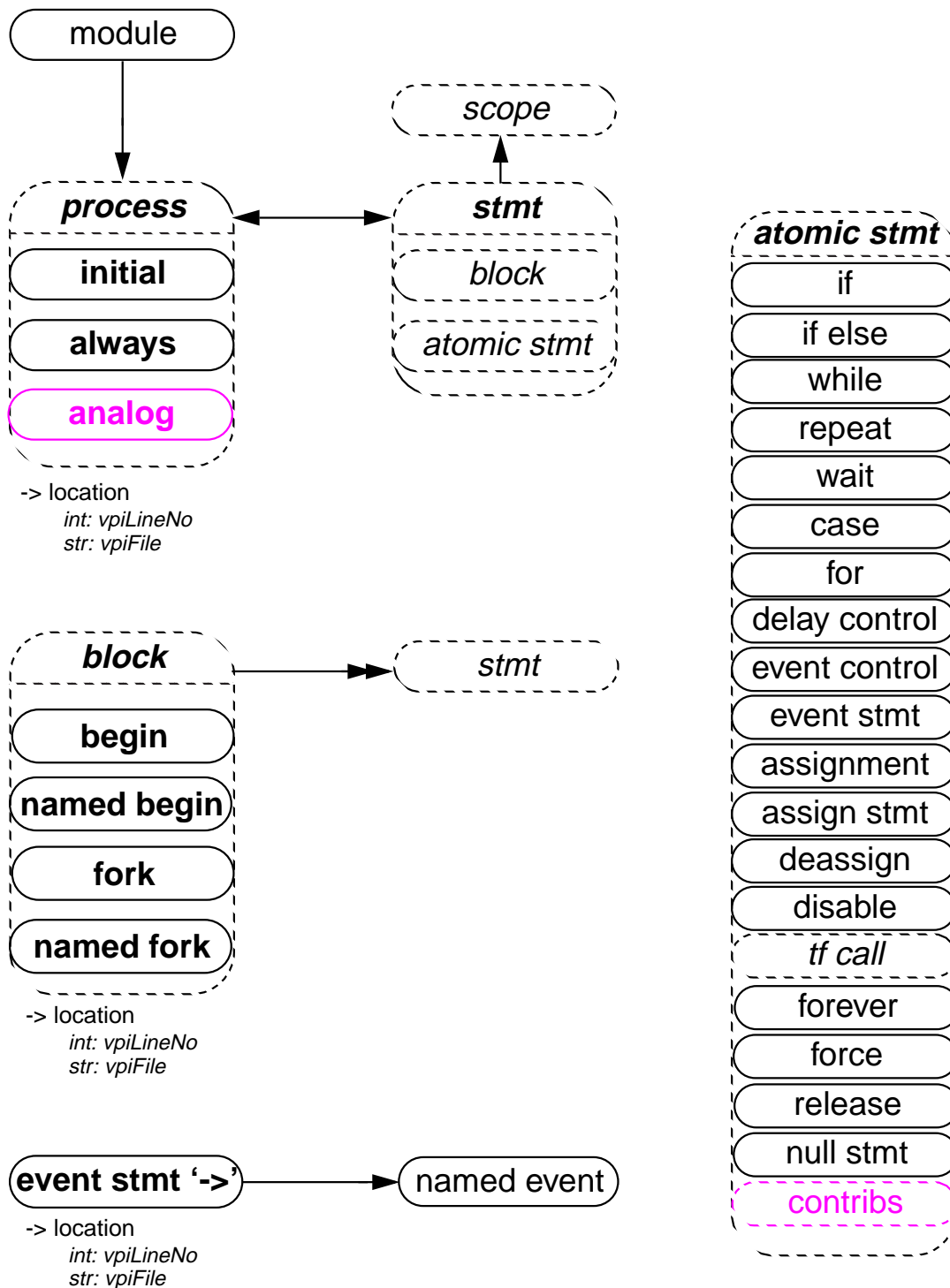


NOTE—For an operator whose type is **vpiMultiConcat**, the first operand shall be the multiplier expression.

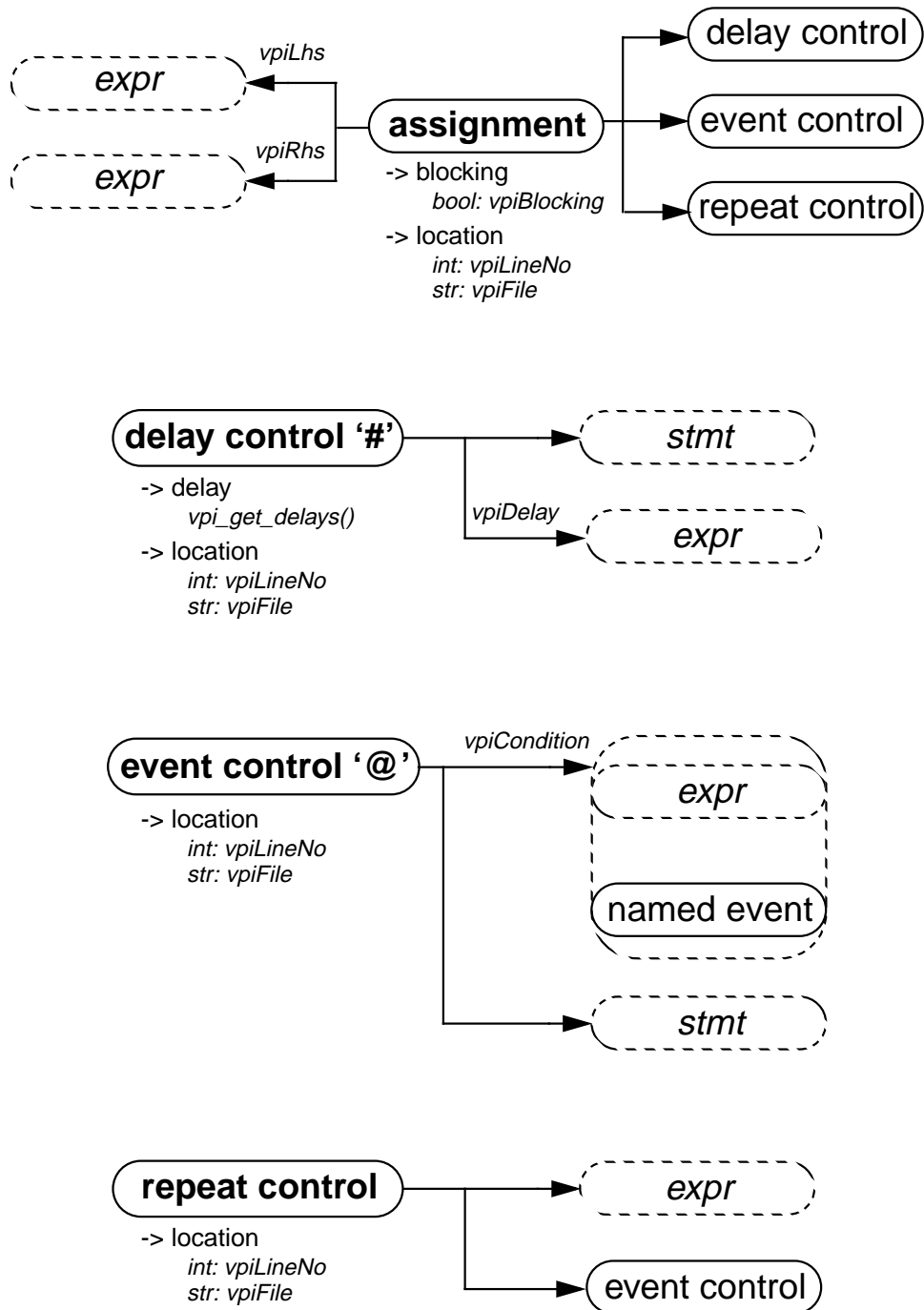
22.5.20 Contribs



22.5.21 Process, block, statement, event statement

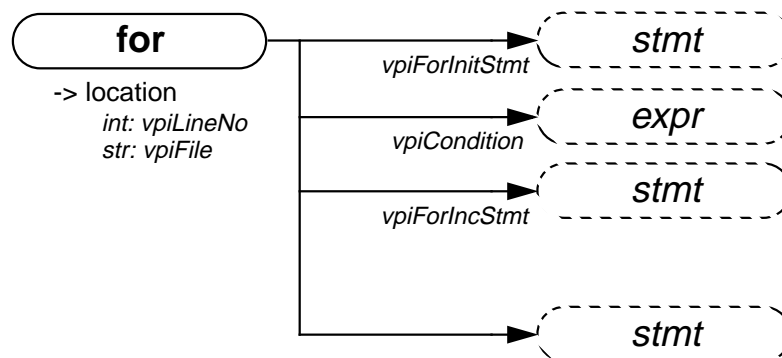
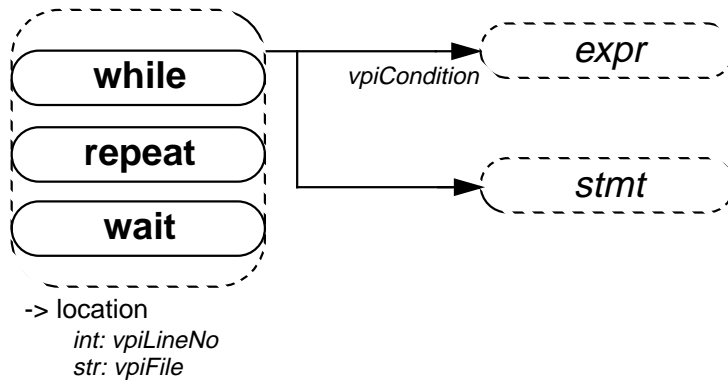


22.5.22 Assignment, delay control, event control, repeat control

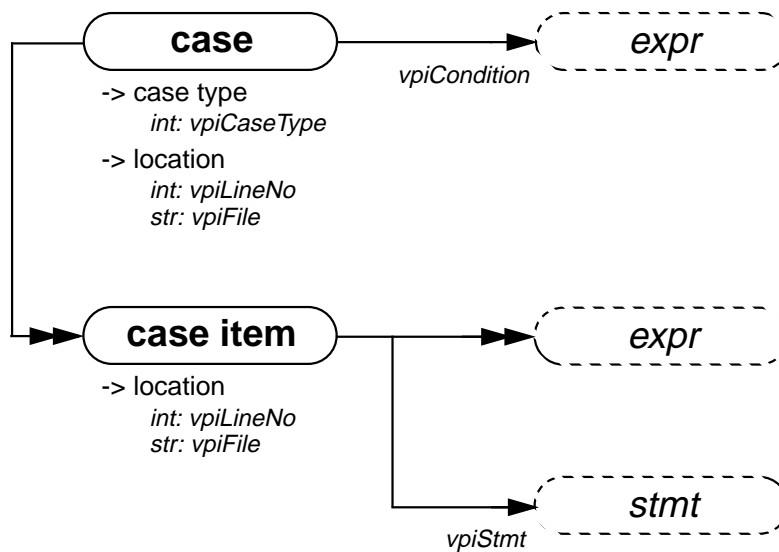
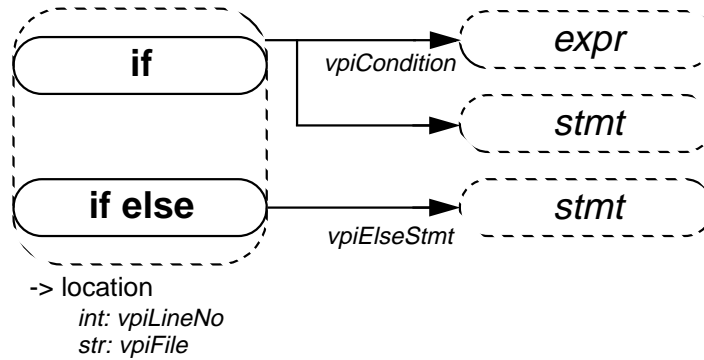


NOTE—For delay control and event control associated with assignment, the statement shall always be NULL.

22.5.23 While, repeat, wait, for, forever

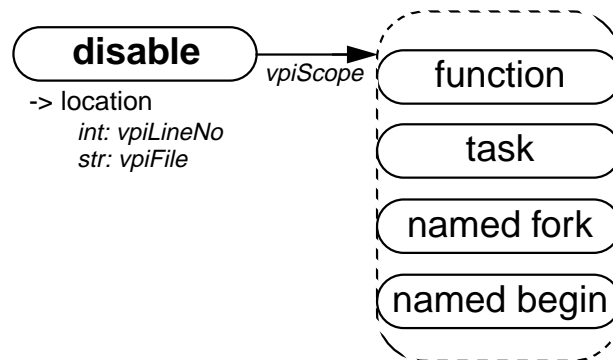
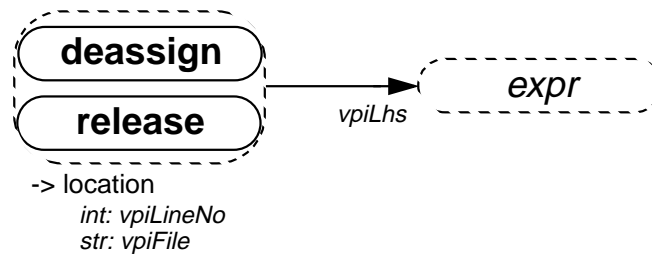
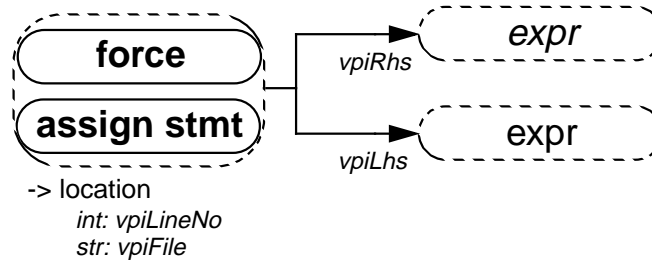


22.5.24 If, if-else, case

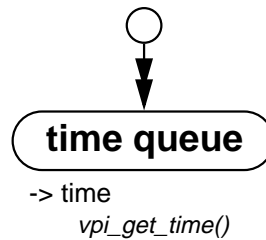
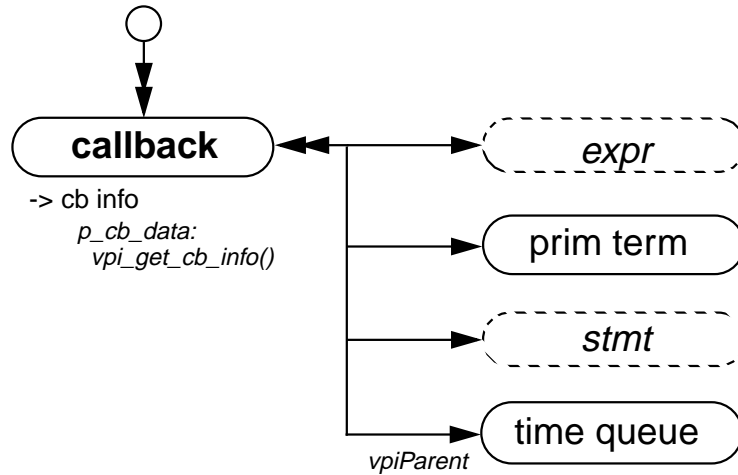


NOTES

- 1—The *case item* shall group all case conditions that branch to the same statement.
- 2—**vpi_iterate()** shall return NULL for the default case item since there is no expression with the default case.

22.5.25 Assign statement, deassign, force, release, disable

22.5.26 Callback, time queue



NOTES

- 1—To get information about the callback object, the routine **vpi_get_cb_info()** can be used.
- 2—To get callback objects not related to the above objects, the second argument to **vpi_iterate()** shall be NULL.
- 3—The time queue objects shall be returned in increasing order of simulation time.
- 4—**vpi_iterate()** shall return NULL if there is nothing left in the simulation queue.
- 5—If any events after read only sync remain in the current queue, then it shall not be returned as part of the iteration.

Section 24

VPI routine definitions

This section describes the Verilog Procedural Interface (VPI) routines, explaining their function, syntax, and usage. The routines are listed in alphabetical order. See Section 20 for the conventions used in the definitions of the PLI routines.

24.1 vpi_chk_error()

vpi_chk_error()			
Synopsis:	Retrieve information about VPI routine errors.		
Syntax:	vpi_chk_error(error_info_p)		
Returns:	Type	Description	
	int	returns the error severity level if the previous VPI routine call resulted in an error and FALSE if no error occurred	
Arguments:	Type	Name	Description
	p_vpi_error_info	error_info_p	Pointer to a structure containing error information

PTF-147

PTF-147

The VPI routine **vpi_chk_error()** shall return an integer constant representing an error severity level if the previous call to a VPI routine resulted in an error. The error constants are shown in Table 24-1. If the previous call to a VPI routine did not result in an error, then **vpi_chk_error()** shall return FALSE. The error status shall be reset by any VPI routine call except **vpi_chk_error()**. Calling **vpi_chk_error()** shall have no effect on the error status.

Table 24-1—Return error constants for vpi_chk_error()

Error Constant	Severity Level
vpiNotice	<div>lowest severity</div> <div>↓</div> <div>highest severity</div>
vpiWarning	
vpiError	
vpiSystem	
vpiInternal	

If an error occurred, the s_vpi_error_info structure shall contain information about the error. If the error information is not needed, a NULL can be passed to the routine. The s_vpi_error_info structure used by **vpi_chk_error()** is defined in vpi_user.h and is listed in Figure 24-1.

```
typedef struct t_vpi_error_info {
    int state;          /* vpi[Compile,PLI,Run] */
    int level;          /* vpi[Notice, Warning, Error, System, Internal] */
    char *message;
    char *product;
    char *code;
    char *file;
    int line;
} s_vpi_error_info, *p_vpi_error_info;
```

Figure 24-1—The s_vpi_error_info structure definition

24.2 vpi_compare_objects()

vpi_compare_objects()			
Synopsis:	Compare two handles to determine if they reference the same object.		
Syntax:	vpi_compare_objects(obj1, obj2)		
Returns:	Type	Description	
	bool	true if the two handles refer to the same object. Otherwise, false	
Arguments:	Type	Name	Description
	vpiHandle	obj1	Handle to an object
	vpiHandle	obj2	Handle to an object

The VPI routine **vpi_compare_objects()** shall return true if the two handles refer to the same object. Otherwise, false shall be returned. Handle equivalence cannot be determined with a C '==' comparison.

24.3 vpi_decl_deriv()

vpi_decl_deriv()			
Synopsis:	Declare a partial derivative of one argument or return value with respect to another.		
Syntax:	vpi_decl_deriv(var,wrt)		
	Type	Description	
Returns:	bool	true on success and false on failure	
	Type	Name	Description
Arguments:	int	var	argument for which partial derivative is to be given
	int	wrt	argument with respect to which derivative is taken

The VPI routine **vpi_decl_deriv()** shall be used in the compile_tf callback to pre-allocate space for a partial derivative. This function is available to analog tasks and functions only. The purpose of this function is declarative only, it does not assign any value to the derivative being declared. Having declared a partial derivative using this function in the compile_tf callback, values may then be contributed to the derivative using the vpi_put_deriv function in the call_tf call back.

The value returned by a function is usually a function of one or more of the arguments and because a function or task may modify the values of its arguments any argument may be a function of one or more other arguments. Thus it is possible (though not necessary) that there will be a partial derivative of the returned value with respect to any or all of the arguments and that there will be a partial derivative of any particular argument with respect to any or all of the other arguments.

The values passed to **vpi_decl_deriv()** are integers. The first indicates the value for which a partial derivative is to be declared. Zero indicates the returned value, one represents the first argument, two the second argument, and so on. The second indicates the value with respect to which the derivative being declared will be calculated. For example **vpi_decl_deriv(0,3)** would indicate the partial derivative of the returned value with respect to the third argument.

24.4 vpi_decl_discontinuity()

vpi_decl_discontinuity()			
Synopsis:	Announce discontinuity in a continuously varying quantity to the simulator		
Syntax:	vpi_decl_discontinuity()		
	Type	Description	
Returns:	void		
	Type	Name	Description
Arguments:	int	order	the order of the discontinuity

The VPI routine **vpi_decl_discontinuity()** shall be used to announce discontinuity in values associated with variables which affects simulation. An abrupt change in the value of a variable or in the value of any of its derivatives are examples of discontinuity. If the analog simulation algorithm uses special techniques for dealing with discontinuities, a call to this function may serve as a signal to employ them.

This function is available to analog tasks and functions only.

24.5 vpi_free_object()

vpi_free_object()			
Synopsis:	Free memory allocated by VPI routines.		
Syntax:	vpi_free_object(obj)		
	Type	Description	
Returns:	bool	true on success and false on failure	
	Type	Name	Description
Arguments:	vpiHandle	obj	Handle of an object

The VPI routine **vpi_free_object()** shall free memory allocated for objects. It shall generally be used to free memory created for iterator objects. The iterator object shall automatically be freed when **vpi_scan()** returns NULL either because it has completed an object traversal or encountered an error condition. If neither of these conditions occur (which can happen if the code breaks out of an iteration loop before it has scanned every object), **vpi_free_object()** should be called to free any memory allocated for the iterator. This routine can also optionally be used for implementations that have to allocate memory for objects. The routine shall return true on success and false on failure.

24.6 vpi_get()

vpi_get()			
Synopsis:	Get the value of an integer or Boolean property of an object.		
Syntax:	vpi_get(prop, obj)		
Returns:	Type	Description	
	int	Value of an integer or Boolean property	
Arguments:	Type	Name	Description
	int	prop	An integer constant representing the property of an object for which to obtain a value
	vpiHandle	obj	Handle to an object
Related routines:	Use vpi_get_str() to get string properties		

The VPI routine **vpi_get()** shall return the value of object properties, for properties of type *int* and *bool* (*bool* shall be defined to *int*). Object properties of type *bool* shall return **1** for true and **0** for false. For object properties of type *int* such as **vpiSize**, any integer shall be returned. For object properties of type *int* that return a defined value, refer to Annex C for the value that shall be returned. Note for object property **vpiTimeUnit** or **vpiTimePrecision**, if the object is NULL, then the simulation time unit shall be returned. Should an error occur, **vpi_get()** shall return **vpiUndefined**.

PTF-148

PTF-093

24.7 vpi_get_cb_info()

vpi_get_cb_info()			
Synopsis:	Retrieve information about a simulation-related callback.		
Syntax:	vpi_get_cb_info(obj, cb_data_p)		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments:	vpiHandle	obj	Handle to a simulation-related callback
	p_cb_data	cb_data_p	Pointer to a structure containing callback information
Related routines:	Use vpi_get_systf_info() to retrieve information about a system task/function callback		

The VPI routine **vpi_get_cb_info()** shall return information about a simulation-related callback in an `s_cb_data` structure. The memory for this structure shall be allocated by the user.

The `s_cb_data` structure used by **vpi_get_cb_info()** is defined in `vpi_user.h` and is listed in Figure 24-2.

```
typedef struct t_cb_data {
    int reason;
    int (*cb_rtn)();
    vpiHandle obj;
    p_vpi_time time;          /* structure with simulation time info */
    p_vpi_value value;        /* structure with simulation value info */
    char *user_data;          /* user data to be passed to callback function */
} s_cb_data, *p_cb_data;
```

Figure 24-2—The `s_cb_data` structure definition

24.8 vpi_get_continuous_delta()

vpi_get_continuous_delta()			
Synopsis:	Get the time elapsed since the previous solution..		
Syntax:	vpi_get_continuous_delta()		
		Type	true on success and false on failureDescription
Returns:	double	time elapsed between the solution being calculated and the last converged solution	
		Type	Name
Arguments:	NONE		this function accepts no arguments

The VPI routine **vpi_get_continuous_delta()** shall be used to determine the size of the analog time step being attempted. It returns the elapsed time between the latest converged and accepted solution and the solution being calculated. The function will return zero during DC or the time zero transient solution.

24.9 vpi_get_continuous_time()

vpi_get_continuous_time()			
Synopsis:	Get the time of the current solution..		
Syntax:	vpi_get_continuous_time()		
	Type	true on success and false on failure	Description
Returns:	double	time associated with the current solution	
	Type	Name	Description
Arguments:	NONE		this function accepts no arguments

The VPI routine **vpi_get_continuous_time()** shall be used to determine the time of the solution attempted during an attempt, or of the latest converged and accepted solution otherwise. The function will return zero during DC or the time zero transient solution.

24.10 vpi_get_delays()

vpi_get_delays()			
Synopsis:	Retrieve the delays or pulse limits of an object.		
Syntax:	vpi_get_delays(obj, delay_p)		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	vpiHandle	obj	Handle to an object
	p_vpi_delay	delay_p	Pointer to a structure containing delay information
Related routines:	Use vpi_put_delays() to set the delays or timing limits of an object		

The VPI routine **vpi_get_delays()** shall retrieve the delays or pulse limits of an object and place them in an `s_vpi_delay` structure that has been allocated by the user. The format of the delay information shall be controlled by the *time_type* flag in the `s_vpi_delay` structure. This routine shall ignore the value of the *type* flag in the `s_vpi_time` structure.

The `s_vpi_delay` and `s_vpi_time` structures used by both **vpi_get_delays()** and **vpi_put_delays()** are defined in `vpi_user.h` and are listed in Figures 24-3 and 24-4.

```
typedef struct t_vpi_delay {
    struct t_vpi_time *da;           /* ptr to user allocated array of delay
                                     values */
    int no_of_delays;               /* number of delays */
    int time_type;                  /* [vpiScaledRealTime, vpiSimTime] */
    bool mtm_flag;                  /* true for mtm */
    bool append_flag;               /* true for append, false for replace */
    bool pulser_flag;               /* true for pulser values */
} s_vpi_delay, *p_vpi_delay;
```

Figure 24-3—The s_vpi_delay structure definition

```
typedef struct t_vpi_time
{
    int type;                       /* [vpiScaledRealTime, vpiSimTime] */
    unsigned int high, low; /* for vpiSimTime */
    double real;                  /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;
```

Figure 24-4—The s_vpi_time structure definition

The *da* field of the `s_vpi_delay` structure shall be a user-allocated array of `s_vpi_time` structures. This array shall store delay values returned by **vpi_get_delays()**. The number of elements in this array shall be determined by

- The number of delays to be retrieved

- The **mtm_flag** setting
- The **pulsere_flag** setting

The number of delays to be retrieved shall be set in the *no_of_delays* field of the *s_vpi_delay* structure. Legal values for the number of delays shall be determined by the type of object.

- For primitive objects, the *no_of_delays* value shall be 2 or 3.
- For path delay objects, the *no_of_delays* value shall be 1, 2, 3, 6, or 12.
- For timing check objects, the *no_of_delays* value shall match the number of limits existing in the timing check.
- For intermodule path objects, the *no_of_delays* value shall be 2 or 3.

PTF-002

The user allocated *s_vpi_delay* array shall contain delays in the same order in which they occur in the Verilog HDL description. The number of elements for each delay shall be determined by the flags **mtm_flag** and **pulsere_flag**, as shown in Table 24-2.

Table 24-2—Size of the *s_vpi_delay->da* array

Flag values	Number of <i>s_vpi_time</i> array elements required for <i>s_vpi_delay->da</i>	Order in which delay elements shall be filled
mtm_flag = false pulsere_flag = false	<i>no_of_delays</i>	1st delay: da[0] -> 1st delay 2nd delay: da[1] -> 2nd delay ...
mtm_flag = true pulsere_flag = false	3 * <i>no_of_delays</i>	1st delay: da[0] -> min delay da[1] -> typ delay da[2] -> max delay 2nd delay: ...
mtm_flag = false pulsere_flag = true	3 * <i>no_of_delays</i>	1st delay: da[0] -> delay da[1] -> reject limit da[2] -> error limit 2nd delay element: ...
mtm_flag = true pulsere_flag = true	9 * <i>no_of_delays</i>	1st delay: da[0] -> min delay da[1] -> typ delay da[2] -> max delay da[3] -> min reject da[4] -> typ reject da[5] -> max reject da[6] -> min error da[7] -> typ error da[8] -> max error 2nd delay: ...

The delay structure has to be allocated before passing a pointer to **vpi_get_delays()**. In the following example, a static structure, **prim_da**, is allocated for use by each call to the **vpi_get_delays()** function.

```
display_prim_delays(prim)
vpiHandle prim;t2

{
    static s_vpi_time prim_da[3];
    static s_vpi_delay delay_s = {NULL, 3, vpiScaledRealTime};
    static p_vpi_delay delay_p = &delay_s;

    delay_s.da = &prim_da;
    vpi_get_delays(prim, delay_p);
}
```

```
    vpi_printf("Delays for primitive %s: %6.2f %6.2f %6.2f\n",vpi_get_str(vpiFullName, prim)  
              delay_p->da[0].real, delay_p->da[1].real, delay_p->da[2].real);  
}
```

24.11 vpi_get_str()

vpi_get_str()			
Synopsis:	Get the value of a string property of an object.		
Syntax:	vpi_get_str(prop, obj)		
Returns:	Type	Description	
	char *	Pointer to a character string containing the property value	
Arguments:	Type	Name	Description
	int	prop	An integer constant representing the property of an object for which to obtain a value
	vpiHandle	obj	Handle to an object
Related routines:	Use vpi_get() to get integer and Boolean properties		

The VPI routine **vpi_get_str()** shall return string property values. The string shall be placed in a temporary buffer that shall be used by every call to this routine. If the string is to be used after a subsequent call, the string should be copied to another location. Note that a different string buffer shall be used for string values returned through the s_vpi_value structure.

The following example illustrates the usage of **vpi_get_str()**.

```
char *str;
vpiHandle mod = vpi_handle_by_name("top.mod1",NULL);
vpi_printf ("Module top.mod1 is an instance of %s\n",
            vpi_get_str(vpiDefName, mod));
```

24.12 vpi_get_systf_info()

vpi_get_systf_info()			
Synopsis:	Retrieve information about a user-defined system task/function-related callback.		
Syntax:	vpi_get_systf_info(obj, systf_data_p)		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments:	vpiHandle	obj	Handle to a system task/function-related callback
	p_vpi_systf_data	systf_data_p	Pointer to a structure containing callback information
Related routines:	Use vpi_get_cb_info() to retrieve information about a simulation-related callback		

The VPI routine **vpi_get_systf_info()** shall return information about a user-defined system task or function callback in an s_vpi_systf_data structure. The memory for this structure shall be allocated by the user.

The s_vpi_systf_data structure used by **vpi_get_systf_info()** is defined in vpi_user.h and is listed in Figure 24-5.

PTF-088

```
typedef struct t_vpi_systf_data {
    int type;                /* vpiSys[Task,Function] */
    int sysfunctype;         /* vpi[IntFunc,RealFunc,TimeFunc,SizedFunc] */
    char *tfname;            /* first character must be "$" */
    int (*calltf)();
    int (*compiletf)();
    int (*sizetf)();         /* for vpiSizedFunc system functions only */
    char *user_data;
} s_vpi_systf_data, *p_vpi_systf_data;
```

Figure 24-5—The s_vpi_systf_data structure definition

24.13 vpi_get_time()

vpi_get_time()			
Synopsis:	Retrieve the current simulation.		
Syntax:	vpi_get_time(obj, time_p)		
Returns:	Type		Description
	void		
Arguments:	Type	Name	Description
	vpiHandle	obj	Handle to an object
	p_vpi_time	time_p	Pointer to a structure containing time information
Related routines:			

The VPI routine **vpi_get_time()** shall retrieve the current simulation time, using the time scale of the object. If *obj* is NULL, the simulation time is retrieved using the simulation time unit. The *time_p->type* field shall be set to indicate if scaled real, continuous, or simulation time is desired. The memory for the *time_p* structure shall be allocated by the user.

The s_vpi_time structure used by **vpi_get_time()** is defined in vpi_user.h and is listed in Figure 24-6 [this is the same time structure as used by **vpi_put_value()**].

```
typedef struct t_vpi_time {
    int type;                /* for vpiScaledRealTime, vpiSimTime, vpiContinuousTime */
    unsigned int high, low; /* for vpiSimTime */
    double real;             /* for vpiScaledRealTime or vpiContinuousTime */
} s_vpi_time, *p_vpi_time;
```

Figure 24-6—The s_vpi_time structure definition

24.14 vpi_get_value()

vpi_get_value()			
Synopsis:	Retrieve the simulation value of an object.		
Syntax:	vpi_get_value(obj, value_p)		
Returns:	Type	Description	
	void		
Arguments:	Type	Name	Description
	vpiHandle	obj	Handle to an expression
	p_vpi_value	value_p	Pointer to a structure containing value information
Related routines:	Use vpi_put_value() to set the value of an object		

The VPI routine **vpi_get_value()** shall retrieve the simulation value of VPI objects. The value shall be placed in an `s_vpi_value` structure, which has been allocated by the user. The format of the value shall be set by the *format* field of the structure.

When the *format* field is **vpiObjTypeVal**, the routine shall fill in the value and change the *format* field based on the object type, as follows:

- For an integer, **vpiIntVal**
- For a real, **vpiRealVal**
- For a scalar, either **vpiScalar** or **vpiStrength**
- For a time variable, **vpiTimeVal** with **vpiSimTime**
- For a vector, **vpiVectorVal**

The buffer this routine uses for string values shall be different from the buffer that **vpi_get_str()** shall use. The string buffer used by `vpi_get_value()` is overwritten with each call. If the value is needed, it should be saved by the application.

PTF-094

The `s_vpi_value`, `s_vpi_vecval` and `s_vpi_strengthval` structures used by `vpi_get_value()` are defined in `vpi_user.h` and are listed in Figures 24-7, 24-8, and 24-9.

```
typedef struct t_vpi_value {
    int format;          /* vpi[[Bin,Oct,Dec,Hex]Str,Scalar,Int,Real,String,
                        Time,Vector,Strength,ObjType]Val*/
    union {
        char *str;
        int scalar; /* vpi[0,1,X,Z] */
        int integer;
        double real;
        struct t_vpi_time *time;
        struct t_vpi_vecval *vector;
        struct t_vpi_strengthval *strength;
        char *misc;
    } value;
} s_vpi_value, *p_vpi_value;
```

Figure 24-7—The `s_vpi_value` structure definition

```
typedef struct t_vpi_vecval {
    int aval, bval; /* bit encoding: ab: 00=0, 10=1, 11=X, 01=Z */
} s_vpi_vecval, *p_vpi_vecval;
```

Figure 24-8—The `s_vpi_vecval` structure definition

```
typedef struct t_vpi_strengthval {
    int logic;          /* vpi[0,1,X,Z] */
    int s0, s1;         /* refer to strength coding in the LRM */
} s_vpi_strengthval, *p_vpi_strengthval;
```

Figure 24-9—The `s_vpi_strengthval` structure definition

For vectors, the `p_vpi_vecval` field shall point to an array of `s_vpi_vecval` structures. The size of this array shall be determined by the size of the vector, where $array_size = ((vector_size-1)/32 + 1)$. The lsb of the vector shall be represented by the lsb of the 0-indexed element of `s_vpi_vecval` array. The 33rd bit of the vector shall be represented by the lsb of the 1-indexed element of the array, and so on. The memory for the union members *str*, *time*, *vector*, *strength*, and *misc* of the value union in the `s_vpi_value` structure shall be provided by the routine `vpi_get_value()`. This memory shall only be valid until the next call to `vpi_get_value()`. [Note that the user must provide the memory for these members when calling `vpi_put_value()`]. When a value change callback occurs for a value type of

vpiVectorVal, the system shall create the associated memory (an array of `s_vpi_vecval` structures) and free the memory upon the return of the callback.

Table 24-3—Return value field of the `s_vpi_value` structure union

Format	Union member	Return description
vpiBinStrVal	str	String of binary char(s) [1 , 0 , x , z]
vpiOctStrVal	str	String of octal char(s) [0–7 , x , X , z , Z] x When all the bits are x X When some of the bits are x z When all the bits are z Z When some of the bits are z
vpiDecStrVal	str	String of decimal char(s) [0–9]
vpiHexStrVal	str	String of hex char(s) [0–f , x , X , z , Z] x When all the bits are x X When some of the bits are x z When all the bits are z Z When some of the bits are z
vpiScalarVal	scalar	vpi1, vpi0, vpiX, vpiZ, vpiH, vpiL
vpiIntVal	integer	Integer value of the handle. Any bits x or z in the value of the object are mapped to a 0
vpiRealVal	real	Value of the handle as a double
vpiStringVal	str	A string where each 8-bit group of the value of the object is assumed to represent an ASCII character
vpiTimeVal	time	Integer value of the handle using two integers
vpiVectorVal	vector	<i>aval/bval</i> representation of the value of the object
vpiStrengthVal	strength	Value plus strength information of a scalar object only
vpiObjectVal	—	Return a value in the closest format of the object

NOTE—If the object has a real value, it shall be converted to an integer using the rounding defined by the Verilog HDL before being returned in a format other than **vpiRealVal**.

PTF-037

To get the ASCII values of UDP table entries (as explained in Section 8.1.6, Table 8-1), the *p_vpi_vecval* field shall point to an array of `s_vpi_vecval` structures. The size of this array shall be determined by the size of the table entry (no. of symbols per table entry), where $array_size = ((table_entry_size - 1) / 4 + 1)$. Each symbol shall require a byte; the ordering of the symbols within `s_vpi_vecval` shall be the most significant byte of *abit* first, then the least significant byte of *abit*, then the most significant byte of *bbit* and then the least significant byte of *bbit*. Each symbol can be either one or two characters; when it is a single character, the second half of the byte shall be an ASCII “\0”.

PTF-059

The *misc* field in the `s_vpi_value` structure shall provide for alternative value types, which can be implementation specific. If this field is utilized, one or more corresponding format types shall also be provided.

In the following example, the binary value of each net that is contained in a particular module and whose name begins with a particular string is displayed. [This function makes use of the `strcmp()` facility normally declared in a `string.h` C library.]

```
void display_certain_net_values(mod, target)
vpiHandle mod;
```

```

char *target;
{
    static s_vpi_value value_s = {vpiBinStrVal};
    static p_vpi_value value_p = &value_s;
    vpiHandle net, itr;

    itr = vpi_iterate(vpiNet, mod);
    while (net = vpi_scan(itr))
    {
        char *net_name = vpi_get_str(vpiName, net);
        if (strcmp(target, net_name) == 0)
        {
            vpi_get_value(net, value_p);
            vpi_printf("Value of net %s: %s\n",
                vpi_get_str(vpiFullName, net), value_p->value.str);
        }
    }
}

```

PTF-130

The following example illustrates the use of `vpi_get_value()` to access UDP table entries. Two sample outputs from this example are provided after the example.

```

/*
 * hUDP must be a handle to a UDP definition
 */
static void dumpUDPTTableEntries(vpiHandle hUDP)
{
    vpiHandle hEntry, hEntryIter;
    s_vpi_value value;
    int numb;
    int udpType;
    int item;
    int entryVal;
    int *abItem;
    int cnt, cnt2;
    numb = vpi_get(vpiSize, hUDP);
    udpType = vpi_get(vpiPrimType, hUDP);
    if (udpType == vpiSeqPrim)
        numb++; /* There is one more table entry for state */
    numb++; /* There is a table entry for the output */
    hEntryIter = vpi_iterate(vpiTableEntry, hUDP);
    if (!hEntryIter)
        return;
    value.format = vpiVectorVal;
    while(hEntry = vpi_scan(hEntryIter))
    {
        vpi_printf("\n");
        /* Show the entry as a string */
        value.format = vpiStringVal;
        vpi_get_value(hEntry, &value);
        vpi_printf("%s\n", value.value.str);
        /* Decode the vector value format */
        value.format = vpiVectorVal;
        vpi_get_value(hEntry, &value);
        abItem = (int *)value.value.vector;
        for(cnt=((numb-1)/2+1);cnt>0;cnt--)

```

```

    {
        entryVal = *abItem;
        abItem++;
        /* Rip out 4 characters */
        for (cnt2=0;cnt2<4;cnt2++)
        {
            item = entryVal&0xff;
            if (item)
                vpi_printf("%c", item);
            else
                vpi_printf("_");
            entryVal = entryVal>>8;
        }
    }
    vpi_printf("\n");
}

```

For a UDP table of:

```

1  0  :?:1;
0  (01) :?:-;
(10) 0  :0:1;

```

The output from the preceding example would be:

```

10:1
_0_1__1
01:0
_1_0__0
00:1
_0_0__1

```

For a UDP table entry of:

```

1  0  :?:1;
0  (01) :?:-;
(10) 0  :0:1;

```

The output from the preceding example would be:

```

10:?:1
_0_1_1_?
0(01):?:-
10_0_-_?
(10)0:0:1
_001_1_0

```

24.15 vpi_get_vlog_info()

vpi_get_vlog_info()			
Synopsis:	Retrieve information about Verilog simulation execution.		
Syntax:	vpi_get_vlog_info(vlog_info_p)		
Returns:	Type	Description	
	bool	true on success and false on failure	
Arguments:	Type	Name	Description
	p_vpi_vlog_info	vlog_info_p	Pointer to a structure containing simulation information

The VPI routine **vpi_get_vlog_info()** shall obtain the following information about Verilog product execution:

- The number of invocation options (*argc*)
- Invocation option values (*argv*)
- Product and version strings

The information shall be contained in an `s_vpi_vlog_info` structure. The routine shall return true on success and false on failure.

The `s_vpi_vlog_info` structure used by **vpi_get_vlog_info()** is defined in `vpi_user.h` and is listed in Figure 24-10.

```
typedef struct t_vpi_vlog_info {
    int argc;
    char **argv;
    char *product;
    char *version;
} s_vpi_vlog_info, *p_vpi_vlog_info;
```

Figure 24-10—The `s_vpi_vlog_info` structure definition

24.16 vpi_get_real()

vpi_get_real()			
Synopsis:	Fetch a real property value associated with an object..		
Syntax:	vpi_get_real(prop,obj)		
Type		Description	
Returns:	double	value of a real property	
Type		Name	Description
Arguments:	int	prop	An integer constant representing the property of an object for which to obtain a value
	vpiHandle	obj	Handle to an object

The VPI routine **vpi_get_real()** shall be used to access node voltages, branch currents and other real valued properties from design objects.

This function is available to analog tasks and functions only.

24.17 vpi_handle()

vpi_handle()			
Synopsis:	Obtain a handle to an object with a one-to-one relationship.		
Syntax:	vpi_handle(type, ref)		
Returns:	Type	Description	
	vpiHandle	Handle to an object	
Arguments:	Type	Name	Description
	int	type	An integer constant representing the type of object for which to obtain a handle
	vpiHandle	ref	Handle to a reference object
Related routines:	Use vpi_iterate() and vpi_scan() to obtain handles to objects with a one-to-many relationship Use vpi_handle_multi() to obtain a handle to an object with a many-to-one relationship		

The VPI routine **vpi_handle()** shall return the object of type *type* associated with object *ref*. The one-to-one relationships that are traversed with this routine are indicated as single arrows in the data model diagrams.

PTF-144

The following example application displays each primitive that an input net drives.

```
void display_driven_primitives(net)
vpiHandle net;
{
    vpiHandle load, prim, itr;
    vpi_printf("Net %s drives terminals of the primitives: \n",
        vpi_get_str(vpiFullName, net));
    itr = vpi_iterate(vpiLoad, net);
    if (!itr)
        return;
    while (load = vpi_scan(itr))
    {
        switch(vpi_get(vpiType, load))
        {
            case vpiGate:
            case vpiSwitch:
            case vpiUdp:
                prim = vpi_handle(vpiPrimitive, load);
                vpi_printf("\t%s\n", vpi_get_str(vpiFullName, prim));
            }
        }
}
```

24.18 vpi_handle_by_index()

vpi_handle_by_index()			
Synopsis:	Get a handle to an object using its index number within a parent object.		
Syntax:	vpi_handle_by_index(obj, index)		
Returns:	Type	Description	
	vpiHandle	Handle to an object	
Arguments:	Type	Name	Description
	vpiHandle	obj	Handle to an object
	int	index	Index number of the object for which to obtain a handle

The VPI routine **vpi_handle_by_index()** shall return a handle to an object based on the index number of the object within a parent object. This function can be used to access all objects that can access an expression using **vpiIndex**. Argument *obj* shall represent the parent of the indexed object. For example, to access a net-bit, *obj* would be the associated net, while for a memory word, *obj* would be the associated memory.

24.19 vpi_handle_by_name()

vpi_handle_by_name()			
Synopsis:	Get a handle to an object with a specific name.		
Syntax:	vpi_handle_by_name(name, scope)		
Returns:	Type	Description	
	vpiHandle	Handle to an object	
Arguments:	Type	Name	Description
	char *	name	A character string or pointer to a string containing the name of an object
	vpiHandle	scope	Handle to a Verilog HDL scope

The VPI routine **vpi_handle_by_name()** shall return a handle to an object with a specific name. This function can be applied to all objects with a *fullname* property. The *name* can be hierarchical or simple. If *scope* is NULL, then *name* shall be searched for from the top level of hierarchy. Otherwise, *name* shall be searched for from *scope* using the scope search rules defined by the Verilog HDL.

24.20 vpi_handle_multi()

vpi_handle_multi()			
Synopsis:	Obtain a handle to intermodule paths with a many-to-one relationship.		
Syntax:	vpi_handle_multi(type, ref1, ref2, ...)		
Returns:	Type	Description	
	vpiHandle	Handle to an object	
Arguments:	Type	Name	Description
	int	type	An integer constant representing the type of object for which to obtain a handle
	vpiHandle	ref1, ref2, ...	Handles to two or more reference objects
Related routines:	Use vpi_iterate() and vpi_scan() to obtain handles to objects with a one-to-many relationship Use vpi_handle() to obtain handles to objects with a one-to-one relationship		

The VPI routine **vpi_handle_multi()** shall return a handle to objects of type **vpiInterModPath** associated with a list of *output port* and *input port* reference objects. The ports shall be of the same size and can be at different levels of the hierarchy. This routine performs a *many-to-one* operation instead of the usual one-to-one or one-to-many.

24.21 vpi_iterate()

vpi_iterate()			
Synopsis:	Obtain an iterator handle to objects with a one-to-many relationship.		
Syntax:	vpi_iterate(type, ref)		
Type		Description	
Returns:	vpiHandle	Handle to an iterator for an object	
Type		Name	Description
Arguments:	int	type	An integer constant representing the type of object for which to obtain iterator handles
	vpiHandle	ref	Handle to a reference object
Related routines:	Use vpi_scan() to traverse the HDL hierarchy using the iterator handle returned from vpi_iterate() Use vpi_handle() to obtain handles to object with a one-to-one relationship Use vpi_handle_multi() to obtain a handle to an object with a many-to-one relationship		

PTF-144

The VPI routine **vpi_iterate()** shall be used to traverse one-to-many relationships, which are indicated as double arrows in the data model diagrams. The **vpi_iterate()** routine shall return a handle to an iterator, whose type shall be **vpi_iterator**, which can be used by **vpi_scan()** to traverse all objects of type *type* associated with object *ref*. To get the reference object from the iterator object use **vpi_handle(vpiUse, iterator_handle)**. If there are no objects of type *type* associated with the reference handle *ref*, then the **vpi_iterate()** routine shall return NULL.

The following example application uses **vpi_iterate()** and **vpi_scan()** to display each net (including the size for vectors) declared in the module. The example assumes it shall be passed a valid module handle.

```

void display_nets(mod)
vpiHandle mod;
{
    vpiHandle net;
    vpiHandle itr;

    vpi_printf("Nets declared in module %s\n",
    vpi_get_str(vpiFullName, mod));

    itr = vpi_iterate(vpiNet, mod);
    while (net = vpi_scan(itr))
    {
        vpi_printf("\t%s", vpi_get_str(vpiName, net));
        if (vpi_get(vpiVector, net))
        {
            vpi_printf(" of size %d\n", vpi_get(vpiSize, net));
        }
        else vpi_printf("\n");
    }
}

```

24.22 vpi_mcd_close()

vpi_mcd_close()			
Synopsis:	Close one or more files opened by vpi_mcd_open().		
Syntax:	vpi_mcd_close(mcd)		
		Type	Description
Returns:	unsigned int	0 if successful, the mcd of unclosed channels if unsuccessful	
		Type	Name
Arguments:	unsigned int	mcd	A multichannel descriptor representing the files to close
Related routines:	Use vpi_mcd_open() to open a file Use vpi_mcd_printf() to write to an opened file Use vpi_mcd_name() to get the name of a file represented by a channel descriptor		

The VPI routine **vpi_mcd_close()** shall close the file(s) specified by a multichannel descriptor, *mcd*. Several channels can be closed simultaneously, since channels are represented by discrete bits in the integer *mcd*. On success this routine returns a 0; on error it returns the *mcd* value of the unclosed channels.

PTF-096

The following descriptors are predefined, and cannot be closed using vpi_mcd_close():

- descriptor 1 is *stdout*
- descriptor 2 is *stderr*
- descriptor 3 is the current log file

24.23 vpi_mcd_name()

vpi_mcd_name()			
Synopsis:	Get the name of a file represented by a channel descriptor.		
Syntax:	vpi_mcd_name(cd)		
Returns:	Type	Description	
	char *	Pointer to a character string containing the name of a file	
Arguments:	Type	Name	Description
	unsigned int	cd	A single-channel descriptor representing a file
Related routines:	Use vpi_mcd_open() to open a file Use vpi_mcd_close() to close files Use vpi_mcd_printf() to write to an opened file		

The VPI routine **vpi_mcd_name()** shall return the name of a file represented by a single-channel descriptor, *cd*. On error, the routine shall return NULL. This routine shall overwrite the returned value on subsequent calls. If the application needs to retain the string, it should copy it.

PTF-097

24.24 vpi_mcd_open()

vpi_mcd_open()			
Synopsis:	Open a file for writing.		
Syntax:	vpi_mcd_open(file)		
	Type	Description	
Returns:	unsigned int	A multichannel descriptor representing the file that was opened	
	Type	Name	Description
Arguments:	char *	file	A character string or pointer to a string containing the file name to be opened
Related routines:	Use vpi_mcd_close() to close a file Use vpi_mcd_printf() to write to an opened file Use vpi_mcd_name() to get the name of a file represented by a channel descriptor		

The VPI routine **vpi_mcd_open()** shall open a file for writing and return a corresponding multichannel descriptor number (*mcd*). The following channel descriptors are predefined and shall be automatically opened by the system:

- Descriptor 1 is *stdout*
- Descriptor 2 is *stderr*
- Descriptor 3 is the current log file

The **vpi_mcd_open()** routine shall return a **0** on error. If the file is already opened, **vpi_mcd_open()** shall return the descriptor number.

24.25 vpi_mcd_printf()

PTF-098

vpi_mcd_printf()			
Synopsis:	Write to one or more files opened with vpi_mcd_open().		
Syntax:	vpi_mcd_printf(mcd, format, ...)		
Type		Description	
Returns:	int	The number of characters written	
Arguments:	Type	Name	Description
	unsigned int	mcd	A multichannel descriptor representing the files to which to write
	char *	format	A format string using the C fprintf() format
Related routines:	Use vpi_mcd_open() to open a file Use vpi_mcd_close() to close a file Use vpi_mcd_name() to get the name of a file represented by a channel descriptor		

The VPI routine **vpi_mcd_printf()** shall write to one or more channels (up to 32) determined by the *mcd*. An *mcd* of 1 (bit 0 set) corresponds to Channel 1, a *mcd* of 2 (bit 1 set) corresponds to Channel 2, a *mcd* of 4 (bit 2 set) corresponds to Channel 3, and so on. Channel 1 is *stdout*, channel 2 is *stderr*, and channel 3 is the current log file. Several channels can be written to simultaneously, since channels are represented by discrete bits in the integer *mcd*. The format strings shall use the same format as the C fprintf() routine. The routine shall return the number of characters printed, or EOF if an error occurred.

PTF-098

PTF-099

24.26 vpi_printf()

vpi_printf()		
Synopsis:	Write to stdout and the current product log file.	
Syntax:	vpi_printf(format, ...)	
	Type	Description
Returns:	int	The number of characters written
	Type	Name Description
Arguments:	char *	format A format string using the C printf() format
Related routines:	Use vpi_mcd_printf() to write to an opened file	

PTF-100

PTF-100

The VPI routine **vpi_printf()** shall write to both *stdout* and the current product log file. The format string shall use the same format as the C printf() routine. The routine shall return the number of characters printed, or EOF if an error occurred.

PTF-101

24.27 vpi_put_delays()

vpi_put_delays()			
Synopsis:	Set the delays or timing limits of an object.		
Syntax:	vpi_put_delays(obj, delay_p)		
Type		Description	
Returns:	void		
Type		Name	Description
Arguments:	vpiHandle	obj	Handle to an object
	p_vpi_delay	delay_p	Pointer to a structure containing delay information
Related routines:	Use vpi_get_delays() to retrieve delays or timing limits of an object		

The VPI routine **vpi_put_delays()** shall set the delays or timing limits of an object as indicated in the *delay_p* structure. The same ordering of delays shall be used as described in the **vpi_get_delays()** function. If only the delay changes, and not the pulse limits, the pulse limits shall retain the values they had before the delays were altered.

The *s_vpi_delay* and *s_vpi_time* structures used by both **vpi_get_delays()** and **vpi_put_delays()** are defined in *vpi_user.h* and are listed in Figures 24-11 and 24-12.

```
typedef struct t_vpi_delay {
    struct t_vpi_time *da;           /* ptr to user allocated array of delay
                                     values */
    int no_of_delays;               /* number of delays */
    int time_type;                  /* [vpiScaledRealTime, vpiSimTime] */
    bool mtm_flag;                  /* true for mtm */
    bool append_flag;               /* true for append, false for replace */
    bool pulseres_flag;             /* true for pulseres values */
} s_vpi_delay, *p_vpi_delay;
```

Figure 24-11—The s_vpi_delay structure definition

```
typedef struct t_vpi_time
{
    int type;                       /* [vpiScaledRealTime, vpiSimTime] */
    unsigned int high, low; /* for vpiSimTime */
    double real;                   /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;
```

Figure 24-12—The s_vpi_time structure definition

The *da* field of the *s_vpi_delay* structure shall be a user-allocated array of *s_vpi_time* structures. This array shall store the delay values to be written by **vpi_put_delays()**. The number of elements in this array shall be determined by:

- The number of delays to be retrieved

- The **mtm_flag** setting
- The **pulsere_flag** setting

The number of delays to be retrieved shall be set in the *no_of_delays* field of the *s_vpi_delay* structure. Legal values for the number of delays shall be determined by the type of object.

- For primitive objects, the *no_of_delays* value shall be 2 or 3.
- For path delay objects, the *no_of_delays* value shall be 1, 2, 3, 6, or 12.
- For timing check objects, the *no_of_delays* value shall match the number of limits existing in the timing check.
- For intermodule path objects, the *no_of_delays* value shall be 2 or 3.

The user allocated *s_vpi_delay* array shall contain delays in the same order in which they occur in the Verilog HDL description. The number of elements for each delay shall be determined by the flags **mtm_flag** and **pulsere_flag**, as shown in Table 24-2.

Table 24-4—Size of the *s_vpi_delay->da* array

Flag values	Number of <i>s_vpi_time</i> array elements required for <i>s_vpi_delay->da</i>	Order in which delay elements shall be filled
mtm_flag = false pulsere_flag = false	<i>no_of_delays</i>	1st delay: da[0] -> 1st delay 2nd delay: da[1] -> 2nd delay ...
mtm_flag = true pulsere_flag = false	3 * <i>no_of_delays</i>	1st delay: da[0] -> min delay da[1] -> typ delay da[2] -> max delay 2nd delay: ...
mtm_flag = false pulsere_flag = true	3 * <i>no_of_delays</i>	1st delay: da[0] -> delay da[1] -> reject limit da[2] -> error limit 2nd delay element: ...
mtm_flag = true pulsere_flag = true	9 * <i>no_of_delays</i>	1st delay: da[0] -> min delay da[1] -> typ delay da[2] -> max delay da[3] -> min reject da[4] -> typ reject da[5] -> max reject da[6] -> min error da[7] -> typ error da[8] -> max error 2nd delay: ...

The following example application accepts a module path handle, rise and fall delays, and replaces the delays of the indicated path.

```
void set_path_rise_fall_delays(path, rise, fall)
vpiHandle path;
double rise, fall;
{
    static s_vpi_time path_da[2];
    static s_vpi_delay delay_s = {NULL, 2, vpiScaledRealTime};
    static p_vpi_delay delay_p = &delay_s;

    delay_s.da = &path_da;
```

```
    path_da[0].real = rise;  
    path_da[1].real = fall;  
  
    vpi_put_delays(path, delay_p);  
}
```

24.28 vpi_put_deriv()

vpi_put_deriv()			
Synopsis:	Set the value of a partial derivative of one argument or return value with respect to another.		
Syntax:	vpi_put_deriv(var,wrt,value)		
Returns:	Type		Description
Arguments:	Type	Name	Description
	int	var	
	int	wrt	
	double	value	

The VPI routine **vpi_put_deriv()** shall be used to add a value to a partial derivative which has been declared using **vpi_declare_deriv()** in the compile_tf call back. This function should be called from the call_tf callback only, and may be called only to contribute to partial derivatives which have been previously declared. Calls for derivatives which have not been declared will be ignored. The **vpi_put_deriv()** should be used to assign a value to all derivatives which have been declared.

This function is available to analog tasks and functions only.

24.29 vpi_put_value()

vpi_put_value()			
Synopsis:	Set a value on an object.		
Syntax:	vpi_put_value(obj, value_p, time_p, flags)		
Returns:	Type		Description
	vpiHandle		Handle to the scheduled event caused by vpi_put_value()
Arguments:	Type	Name	Description
	vpiHandle	obj	Handle to an object
	p_vpi_value	value_p	Pointer to a structure with value information
	p_vpi_time	time_p	Pointer to a structure with delay information
	int	flags	Integer constants that set the delay mode
Related routines:	Use vpi_get_value() to retrieve the value of an expression		

The VPI routine **vpi_put_value()** shall set simulation logic values on an object. The value to be set shall be stored in an `s_vpi_value` structure that has been allocated. The delay time before the value is set shall be stored in an `s_vpi_time` structure that has been allocated. The routine can be applied to nets, regs, variables, memory words, system function calls, sequential UDPs, and schedule events. The *flags* argument shall be used to direct the routine to use one of the following delay modes:

vpiInertialDelay	All scheduled events on the object shall be removed before this event is scheduled.
vpiTransportDelay	All events on the object scheduled for times later than this event shall be removed (modified transport delay).
vpiPureTransportDelay	No events on the object shall be removed (transport delay).
vpiNoDelay	The object shall be set to the passed value with no delay. Argument <i>time_p</i> shall be ignored and can be set to NULL.
vpiForceFlag	The object shall be forced to the passed value with no delay (same as the Verilog HDL procedural force). Argument <i>time_p</i> shall be ignored and can be set to NULL.
vpiReleaseFlag	The object shall be released from a forced value (same as the Verilog HDL procedural release). Argument <i>time_p</i> shall be ignored and can be set to NULL. The <i>value_p</i> shall contain the current value of the object.
vpiCancelEvent	A previously scheduled event shall be cancelled. The object passed to vpi_put_value() shall be a handle to an object of type vpiSchedEvent .

If the *flags* argument also has the bit mask **vpiReturnEvent**, **vpi_put_value()** shall return a handle of type **vpiSchedEvent** to the newly scheduled event, provided there is some form of a delay and an event is scheduled. If the bit mask is not used, or if no delay is used, or if an event is not scheduled, the return value shall be NULL.

The handle to the event can be cancelled by calling **vpi_put_value()** with the flag set to **vpiCancelEvent**. It shall not be an error to cancel an event that has already occurred. The scheduled event can be tested by calling **vpi_get()** with the flag **vpiScheduled**. If an event is cancelled, it shall simply be removed from the event queue. Any effects that were caused by scheduling the event shall remain in effect (e.g., events that were cancelled due to inertial delay).

Calling **vpi_free_object()** on the handle shall free the handle but shall not effect the event.

Sequential UDPs shall be set to the indicated value with no delay regardless of any delay on the primitive instance.

PTF-131

NOTE—**vpi_put_value()** shall only return a function value in a calltf application, when the call to the function is active. The action of **vpi_put_value()** to a function shall be ignored when the function is not active.

The **s_vpi_value** and **s_vpi_time** structures used by **vpi_put_value()** are defined in **vpi_user.h** and are listed in Figures 24-13 and 24-14.

```
typedef struct t_vpi_value {
    int format;          /* vpi[[Bin,Oct,Dec,Hex]Str,Scalar,Int,Real,String,
                        Time,Vector,Strength,ObjType]Val*/
    union {
        char *str;
        int scalar; /* vpi[0,1,X,Z] */
        int integer;
        double real;
        struct t_vpi_time *time;
        struct t_vpi_vecval *vector;
        struct t_vpi_strengthval *strength;
        char *misc;
    } value;
} s_vpi_value, *p_vpi_value;
```

Figure 24-13—The s_vpi_value structure definition

```
typedef struct t_vpi_time {
    int type;             /* for vpiScaledRealTime, vpiSimTime */
    unsigned int high, low; /* for vpiSimTime */
    double real;          /* for vpiScaledRealTime */
} s_vpi_time, *p_vpi_time;
```

Figure 24-14—The s_vpi_time structure definition

For **vpiScaledRealTime**, the indicated time shall be in the timescale associated with the object.

24.30 vpi_register_cb()

vpi_register_cb()			
Synopsis:	Register simulation-related callbacks.		
Syntax:	vpi_register_cb(cb_data_p)		
Type		Description	
Returns:	vpiHandle	Handle to the callback object	
Type		Name	Description
Arguments:	p_cb_data	cb_data_p	Pointer to a structure with data about when callbacks should occur and the data to be passed
Related routines:	Use vpi_register_systf() to register callbacks for user-defined system tasks and functions Use vpi_remove_cb() to remove callbacks registered with vpi_register_cb()		

The VPI routine **vpi_register_cb()** is used for registration of simulation-related callbacks to a user-provided application for a variety of reasons during a simulation. The reasons for which a callback can occur are divided into three categories:

- Simulation event
- Simulation time
- Simulation action or feature

How callbacks are registered for each of these categories is explained in the following paragraphs.

The *cb_data_p* argument shall point to a *s_cb_data* structure, which is defined in *vpi_user.h* and given in Figure 24-15.

```
typedef struct t_cb_data {
    int reason;
    int (*cb_rtn)();
    vpiHandle obj;
    p_vpi_time time;           /* structure defined in vpi_user.h */
    p_vpi_value value;         /* structure defined in vpi_user.h */
    int index;                 /* index of memory word or var select which changed */
    char *user_data;           /* user data to be passed to callback function */
} s_cb_data, *p_cb_data;
```

Figure 24-15—The s_cb_data structure definition

For all callbacks, the *reason* field of the *s_cb_data* structure shall be set to a predefined constant, such as **cbValueChange**, **cbAtStartOfSimTime**, **cbEndOfCompile**, etc. The reason constant shall determine when the user application shall be called back. Refer to the *vpi_user.h* file listing in Annex C for a list of all callback reason constants.

The *cb_rtn* field of the *s_cb_data* structure shall be set to the application routine name, which shall be invoked when the simulator executes the callback. The use of the remaining fields are detailed in the following subclauses.

24.30.1 Simulation-event-related callbacks

The **vpi_register_cb()** callback mechanism can be registered for callbacks to occur for simulation events, such as value changes on an expression or terminal, or the execution of a behavioral statement. When the *cb_data_p->reason* field is set to one of the following, the callback shall occur as described below:

cbValueChange	After value change on an expression or terminal
cbStmt	Before execution of a behavioral statement
cbForce/cbRelease	After a force or release has occurred
cbAssign/cbDeassign	After a procedural assign or deassign statement has been executed
cbDisable	After a named block or task containing a system task or function has been disabled

The following fields shall need to be initialized before passing the *s_cb_data* structure to **vpi_register_cb()**:

<i>cb_data_p->obj</i>	This field shall be assigned a handle to an expression, terminal, or statement for which the callback shall occur. For force and release callbacks, if this is set to NULL, every force and release shall generate a callback.
<i>cb_data_p->time->type</i>	This field shall be set to either vpiScaledRealTime or vpiSimTime , depending on what time information the user application requires during the callback. If simulation time information is not needed during the callback, this field can be set to vpiSuppressTime .
<i>cb_data_p->value->format</i>	This field shall be set to one of the value formats indicated in Table 24-5. If value information is not needed during the callback, this field can be set to vpiSuppressVal . For cbStmt callbacks, value information is not passed to the callback routine, so this field shall be ignored.

Table 24-5—Value format field of *cb_data_p->value->format*

Format	Registers a callback to return
vpiBinStrVal	String of binary char(s) [1, 0, x, z]
vpiOctStrVal	String of octal char(s) [0–7, x, X, z, Z]
vpiDecStrVal	String of decimal char(s) [0–9]
vpiHexStrVal	String of hex char(s) [0–f, x, X, z, Z]
vpiScalarVal	vpi1, vpi0, vpiX, vpiZ, vpiH, vpiL
vpiIntVal	Integer value of the handle
vpiRealVal	Value of the handle as a double
vpiStringVal	An ASCII string
vpiTimeVal	Integer value of the handle using two integers
vpiVectorVal	<i>aval/bval</i> representation of the value of the object
vpiStrengthVal	Value plus strength information of a scalar object only
vpiObjectVal	Return a value in the closest format of the object

When a simulation event callback occurs, the user application shall be passed a single argument, which is a pointer to an *s_cb_data* structure [this is not a pointer to the same structure that was passed to **vpi_register_cb()**]. The *time* and

value information shall be set as directed by the time *type* and *value* format fields in the call to **vpi_register_cb()**. The *user_data* field shall be equivalent to the *user_data* field passed to **vpi_register_cb()**. The user application can use the information in the passed structure and information retrieved from other VPI interface routines to perform the desired callback processing.

For a **cbValueChange** callback, if the *obj* is a memory word or a variable array, the *value* in the *s_cb_data* structure shall be the value of the memory word or variable select that changed value. The *index* field shall contain the index of the memory word or variable select that changed value.

For **cbForce**, **cbRelease**, **cbAssign** and **cbDeassign** callbacks, the object returned in the *obj* field shall be a handle to the force, release, assign or deassign statement. The *value* field shall contain the resultant value of the LHS expression. In the case of a release, the *value* field shall contain the value after the release has occurred.

The following example shows an implementation of a simple monitor functionality for scalar nets, using a simulation-event-related callback.

```

setup_monitor(net)
vpiHandle net;
{
    static s_vpi_time time_s = {vpiScaledRealTime};
    static s_vpi_value value_s = {vpiBinStrVal};
    static s_cb_data cb_data_s =
        {cbValueChange, my_monitor, NULL, &time_s, &value_s};
    char *net_name = vpi_get_str(vpiFullName, net);
    cb_data_s.obj = net;
    cb_data_s.user_data = malloc(strlen(net_name)+1);
    strcpy(cb_data_s.user_data, net_name);
    vpi_register_cb(&cb_data_s);
}

my_monitor(cb_data_p)
p_cb_data cb_data_p; {
    vpi_printf("%d %d: %s = %s\n",
        cb_data_p->time->high, cb_data_p->time->low,
        cb_data_p->user_data,
        cb_data_p->value->value.str);
}

```

24.30.2 Simulation-time-related callbacks

The **vpi_register_cb()** can register callbacks to occur for simulation time reasons, include callbacks at the beginning or end of the execution of a particular time queue. The following time-related callback reasons are defined:

cbAtStartOfSimTime	Callback shall occur before execution of events in a specified time queue. A callback can be set for any time, even if no event is present.
cbReadWriteSynch	Callback shall occur after execution of events for a specified time.
cbReadOnlySynch	Same as cbReadWriteSynch , except that writing values or scheduling events before the next scheduled event is not allowed.
cbNextSimTime	Callback shall occur before execution of events in the next event queue.
cbAfterDelay	Callback shall occur after a specified amount of time, before execution of events in a specified time queue. A callback can be set for anytime, even if no event is present.

The following fields shall need to be set before passing the *s_cb_data* structure to **vpi_register_cb()**:

cb_data_p->time->type This field shall be set to either **vpiScaledRealTime** or **vpiSimTime**, depending on what time information the user application requires during the callback.

cb_data_p->[time->low,time->high,time->real]

These fields shall contain the requested time of the callback or the delay before the callback.

The *value* fields are ignored for all reasons with simulation-time-related callbacks.

When the *cb_data_p->time->type* is set to **vpiScaledRealTime**, the *cb_data_p->obj* field shall be used as the object for determining the time scaling.

PTF-103

PTF-104

For reason **cbNextSimTime**, the time structure is ignored.

When a simulation-time-related callback occurs, the user callback application shall be passed a single argument, which is a pointer to an *s_cb_data* structure [this is not a pointer to the same structure that was passed to **vpi_register_cb()**]. The *time* structure shall contain the current simulation time. The *user_data* field shall be equivalent to the *user_data* field passed to **vpi_register_cb()**.

The callback application can use the information in the passed structure and information retrieved from other interface routines to perform the desired callback processing.

24.30.3 Simulator action and feature related callbacks

The **vpi_register_cb()** can register callbacks to occur for simulator action reasons or simulator feature reasons. *Simulator action reasons* are callbacks such as the end of compilation or end of simulation. *Simulator feature reasons* are software-product-specific features, such as restarting from a saved simulation state or entering an interactive mode. Actions are differentiated from features in that actions shall occur in all VPI-compliant products, whereas features might not exist in all VPI-compliant products.

The following action-related callbacks shall be defined:

cbEndOfCompile	End of simulation data structure compilation or build
cbStartOfSimulation	Start of simulation (beginning of time 0 simulation cycle)
cbEndOfSimulation	End of simulation (e.g., \$finish system task executed)
cbError	Simulation run-time error occurred
cbPLIError	Simulation run-time error occurred in a PLI function call
cbTchkViolation	Timing check error occurred

PTF-152

should this
be "VPI
routine
call"?

Examples of possible feature related callbacks are

cbStartOfSave	Simulation save state command invoked
cbEndOfSave	Simulation save state command completed
cbStartOfRestart	Simulation restart from saved state command invoked
cbEndOfRestart	Simulation restart command completed
cbEnterInteractive	Simulation entering interactive debug mode (e.g., \$stop system task executed)
cbExitInteractive	Simulation exiting interactive mode
cbInteractiveScopeChange	Simulation command to change interactive scope executed
cbUnresolvedSystf	Unknown user-defined system task or function encountered

The only fields in the *s_cb_data* structure that shall need to be setup for simulation action/feature callbacks are the *reason*, *cb_rtn*, and *user_data* (if desired) fields.

When a simulation action/feature callback occurs, the user routine shall be passed a pointer to an `s_cb_data` structure. The *reason* field shall contain the reason for the callback. For **cbTchkViolation** callbacks, the *obj* field shall be a handle to the timing check. For **cbInteractiveScopeChange**, *obj* shall be a handle to the new scope. For **cbUnresolvedSysf**, *user_data* shall point to the name of the unresolved task or function. On a **cbError** callback, the routine **vpi_chk_error()** can be called to retrieve error information.

The following example shows a callback application that reports cpu usage at the end of a simulation. If the user routine `setup_report_cpu()` is placed in the `vlog_startup_routines` list, it shall be called just after the simulator is invoked.

```
static int initial_cputime_g;

void report_cpu()
{
    int total = get_current_cputime() - initial_cputime_g;
    vpi_printf("Simulation complete. CPU time used: %d\n", total);
}

void setup_report_cpu()
{
    static s_cb_data cb_data_s = {cbEndOfSimulation, report_cpu};
    initial_cputime_g = get_current_cputime();
    vpi_register_cb(&cb_data_s);
}
```

24.31 vpi_register_acb()

vpi_register_acb()			
Synopsis:	Register analog simulation-related callbacks.		
Syntax:	vpi_register_acb(acb_data_p)		
Type		Description	
Returns:	vpiHandle	Handle to the callback object	
Type		Name	Description
Arguments:	p_acb_data	acb_data_p	Pointer to a structure with data about when callbacks should occur and the data to be passed
Related routines:	Use vpi_register_systf() to register callbacks for user-defined system tasks and functions Use vpi_remove_acb() to remove callbacks registered with vpi_register_acb()		

The VPI routine **vpi_register_acb()** is used for registration of analog simulation-related callbacks to a user-provided application for a variety of reasons during a simulation. The reasons for which a callback can occur are divided into three categories:

- Threshold crossing of a potential or flow
- Simulation time
- Simulation action or feature

How callbacks are registered for each of these categories is explained in the following paragraphs.

The *acb_data_p* argument shall point to a *s_acb_data* structure, which is defined in *vpi_user.h* and given in Figure 24-15.

```
typedef struct t_acb_data {
    int reason;                /* acbAbsTime, acbElapsedTime, acbThreshold, ... */
    int (*acb_rtn)();          /* function to be called */
    vpiHandle obj;             /* handle of branch, node, analog variable */
    int property;              /* e.g. flow or potential, if obj is a branch */
    double time;               /* absolute or elapsed simulation time */
    double value;              /* threshold value */
    double delta;              /* time tolerance */
    double epsilon;            /* value tolerance */
    int sign;                  /* crossing direction 1,0,-1 */
    char *user_data;           /* user data to be passed to callback function */
} s_acb_data, *p_acb_data;
```

Figure 24-16—The s_acb_data structure definition

For all callbacks, the *reason* field of the *s_cb_data* structure shall be set to a predefined constant, such as **cbThreshold**, **cbAtStartOfSimTime**, **cbEndOfCompile**, etc. The reason constant shall determine when the user application shall be called back. Refer to the *vpi_user.h* file listing in Annex C for a list of all callback reason constants. Some reasons are not valid for analog simulation related callbacks.

The *acb_rtn* field of the *s_acb_data* structure shall be set to the application routine name, which shall be invoked when the simulator executes the callback. The use of the remaining fields are detailed in the following subclauses.

24.31.1 Simulation-event-related callbacks

The **vpi_register_acb()** callback mechanism can be registered for callbacks to occur for simulation events, such as threshold crossing of a flow or potential, or acceptance of the initial or final analog solution. When the *acb_data_p->reason* field is set to one of the following, the callback shall occur as described below:

acbInitialStep	Upon acceptance of the first analog solution
acbFinalStep	Upon acceptance of the last analog solution
acbAbsTime	Upon acceptance of the analog solution for the given time (this callback will force a solution at that time)
acbElapsedTime	Upon acceptance of the solution advanced from the current solution by the given interval (this callback will force a solution at that time)
acbThreshold	Upon threshold crossing of a variable, flow, or potential

The following field shall need to be initialized before passing the *s_cb_data* structure to **vpi_register_cb()**:

<i>acb_data_p->obj</i>	This field shall be assigned a handle to node, branch, or analog variable.
<i>acb_data_p->property</i>	In the case of a branch of node this field shall be assigned vpiFlow or vpiPotential .
<i>acb_data_p->user_data</i>	This field shall be assigned a handle to memory which may be used by the users application..

For a **acbAbsTime** or **acbElapsedTime** callback:

<i>acb_data_p->time</i>	This field shall be assigned the value of absolute or elapsed time when the callback should occur.
----------------------------	--

For a **acbThreshold** callback:

<i>acb_data_p->value</i>	This field shall be assigned the threshold value whose crossing will cause the callback.
<i>acb_data_p->delta</i>	The callback will occur within this tolerance of the actual crossing time.
<i>acb_data_p->epsilon</i>	The callback will occur within this tolerance of the actual crossing value.
<i>acb_data_p->sign</i>	1 = ascending crossing only, -1 = descending crossing only, 0 = either direction.

When a simulation event callback occurs, the user application shall be passed a single argument, which is a pointer to an *s_acb_data* structure [this is not a pointer to the same structure that was passed to **vpi_register_cb()**]. The *time* and *value* information shall be set to the values for the current analog solution. The *user_data* field shall be equivalent to the *user_data* field passed to **vpi_register_cb()**. The user application can use the information in the passed structure and information retrieved from other VPI interface routines to perform the desired callback processing.

24.31.2 Simulator action and feature related callbacks

The **vpi_register_cb()** can register callbacks to occur for simulator action reasons or simulator feature reasons. *Simulator action reasons* are callbacks such as the end of compilation or end of simulation. *Simulator feature reasons* are software-product-specific features, such as restarting from a saved simulation state or entering an interactive mode. Actions are differentiated from features in that actions shall occur in all VPI-compliant products, whereas features might not exist in all VPI-compliant products.

The following action-related callbacks shall be defined:

cbEndOfCompile	End of simulation data structure compilation or build
cbStartOfSimulation	Start of simulation (beginning of time 0 simulation cycle)
cbEndOfSimulation	End of simulation (e.g., \$finish system task executed)
cbError	Simulation run-time error occurred
cbPLIError	Simulation run-time error occurred in a PLI function call
cbFailConverge	Simulation terminated because of failure to converge

Examples of possible feature related callbacks are

cbStartOfSave	Simulation save state command invoked
cbEndOfSave	Simulation save state command completed
cbStartOfRestart	Simulation restart from saved state command invoked
cbEndOfRestart	Simulation restart command completed
cbEnterInteractive	Simulation entering interactive debug mode (e.g., \$stop system task executed)
cbExitInteractive	Simulation exiting interactive mode
cbInteractiveScopeChange	Simulation command to change interactive scope executed
cbUnresolvedSysf	Unknown user-defined system task or function encountered

The only fields in the `s_scb_data` structure that shall need to be setup for simulation action/feature callbacks are the *reason*, *cb_rtn*, and *user_data* (if desired) fields.

When a simulation action/feature callback occurs, the user routine shall be passed a pointer to an `s_acb_data` structure. The *reason* field shall contain the reason for the callback. For **cbInteractiveScopeChange**, *obj* shall be a handle to the new scope. For **cbUnresolvedSysf**, *user_data* shall point to the name of the unresolved task or function. On a **cbError** callback, the routine `vpi_chk_error()` can be called to retrieve error information.

PTF-152
should this
be "VPI
routine
call"?

24.32 vpi_register_systf()

vpi_register_systf()			
Synopsis:	Register user-defined system task/function-related callbacks.		
Syntax:	vpi_register_systf(systf_data_p)		
Type		Description	
Returns:	vpiHandle	Handle to the callback object	
Type		Name	Description
Arguments:	p_vpi_systf_data	systf_data_p	Pointer to a structure with data about when callbacks should occur and the data to be passed
Related routines:	Use vpi_register_cb() to register callbacks for simulation-related events		

The VPI routine **vpi_register_systf()** shall register callbacks for user-defined system tasks or functions. Callbacks can be registered to occur when a user-defined system task or function is encountered during compilation or execution of Verilog HDL source code. Tasks or functions may be registered with either the analog or digital domain. The domain with which the task or function is registered will determine the context or contexts from which the task or function may be invoked and how and when the call backs associated with the function will be called. The task or function name must be unique in the domain in which it is registered. That is, the same name may be shared by two sets of callbacks, provided that one set is registered in the digital domain and the other is registered in the analog.

The *systf_data_p* argument shall point to a *s_vpi_systf_data* structure, which is defined in *vpi_user.h* and listed in Figure 24-17.

PTF-088

```
typedef struct t_vpi_systf_data {
    int type;                /* vpiSys[Task,TaskA,Function,FunctionA] */
    int systf_type;          /* vpi[IntFunc,RealFunc,TimeFunc,SizedFunc] */
    char *tfname;            /* first character must be "$" */
    int (*calltf)();
    int (*compiletf)();
    int (*sizetf)();         /* for vpiSizedFunc system functions only */
    char *user_data;
} s_vpi_systf_data, *p_vpi_systf_data;
```

Figure 24-17—The s_vpi_systf_data structure definition

24.32.1 System task and function callbacks

User-defined Verilog system tasks and functions that use VPI routines can be registered with **vpi_register_systf()**. The following system task/function-related callbacks are defined.

The *type* field of the *s_vpi_systf_data* structure shall register the user application to be a system task or a system function. The *type* field value shall be an integer constant of **vpiSysTask**, **vpiSysTaskA**, **vpiSysFunction** or **vpiSysFunctionA**. **vpiSysTask** will register a task with the digital domain. **vpiSysTaskA** will register a task with the analog domain. **vpiSysFunction** will register a function with the digital domain. **vpiSysFunctionA** will register a function with the analog domain.

The *sysfunctype* field of the *s_vpi_systf_data* structure shall define the type of value that a system function shall return. The *sysfunctype* field shall be an integer constant of **vpiIntFunc**, **vpiRealFunc**, **vpiTimeFunc**, or **vpiSizedFunc**. This field shall only be used when the *type* field is set to **vpiSysFunction**.

PTF-088

The *compiletf*, *calltf*, and *sizetf* fields of the *s_vpi_systf_data* structure shall be pointers to the user-provided applications that are to be invoked by the system task/function callback mechanism. One or more of the *compiletf*, *calltf*, and *sizetf* fields can be set to NULL if they are not needed. Callbacks to the applications pointed to by the *compiletf* and *sizetf* fields shall occur when the simulation data structure is compiled or built (or for the first invocation if the system task or function is invoked from an interactive mode). Callbacks to the application pointed to by the *calltf* routine shall occur each time the system task or function is invoked during simulation execution.

PTF-107

The *sizetf* application shall only called if the PLI application type is **vpiSysFunction** and the *sysfunctype* is **vpiSizedFunc**. If no *sizetf* is provided, a user-defined system function of **vpiSizedFunc** shall return 32-bits.

PTF-191

The *user_data* field of the *s_vpi_systf_data* structure shall specify a user-defined value, which shall be passed back to the *compiletf*, *sizetf*, and *calltf* applications when a callback occurs.

The following example application demonstrates dynamic linking of a VPI system task. The example uses an imaginary routine, *dlink()*, which accepts a file name and a function name and then links that function dynamically. This routine derives the target file and function names from the target *systf* name.

```
link_systf(target)
char *target;
{
    char task_name[strSize];
    char file_name[strSize];
    char compiletf_name[strSize];
    char calltf_name[strSize];
    static s_vpi_systf_data task_data_s = {vpiSysTask};
    static p_vpi_systf_data task_data_p = &task_data_s;

    sprintf(task_name, "$%s", target);
    sprintf(file_name, "%s.o", target);
    sprintf(compiletf_name, "%s_compiletf", target);
    sprintf(calltf_name, "%s_calltf", target);

    task_data_p->tfname = task_name;
    task_data_p->compiletf = (int (*)(void)) dlink(file_name,
        compiletf_name);
    task_data_p->calltf = (int (*)(void)) dlink(file_name, calltf_name);
    vpi_register_systf(task_data_p);
}
```

24.32.2 Initializing VPI system task/function callbacks

A means of initializing system task/function callbacks and performing any other desired task just after the simulator is invoked shall be provided by placing routines in a NULL-terminated static array, **vlog_startup_routines**. A C function using the array definition shall be provided as follows:

```
void (*vlog_startup_routines[]) ();
```

This C function shall be provided with a VPI-compliant product. Entries in the array shall be added by the user. The location of **vlog_startup_routines** and the procedure for linking **vlog_startup_routines** with a software product shall be defined by the product vendor. (Note that callbacks can also be registered or removed at any time during an application routine, not just at startup time).

PTF-108

This array of C functions shall be for registering system tasks and functions. User tasks and functions that appear in a compiled description shall generally be registered by a routine in this array.

The following example uses **vlog_startup_routines** to register system tasks and functions and to run a user initialization routine.

```
/*In a vendor product file which contains vlog_startup_routines ...*/
extern void register_my_systfs();
extern void my_init();
void (*vlog_startup_routines[])() =
{
    setup_report_cpu,          /* user routine example in 23.24.3 */register_my_systfs,/* user
routine listed below */
    0                          /* must be last entry in list */
}
```

```
/* In a user provided file... */
void regiser_my_systfs()
{
    static s_vpi_systf_data systf_data_list[] = {
        {vpiSysTask, 0 "$my_task", my_task_calltf, my_task_compiletf},
        {vpiSysFunc, vpiIntFunc,"$my_func", my_func_calltf, my_func_compiletf},
        {vpiSysFunc, vpiRealFunc, "$my_real_func", my_rfunc_calltf, my_rfunc_compiletf},
        {0}
    };
    p_vpi_systf_data systf_data_p = &(systf_data_list[0]);
    while (systf_data_p->type)
        vpi_register_systf(systf_data_p++);
}
```

PTF-088

24.33 vpi_remove_cb()

vpi_remove_cb()		
Synopsis:	Remove a simulation callback registered with <code>vpi_register_cb()</code> .	
Syntax:	<code>vpi_remove_cb(cb_obj)</code>	
Returns:	Type	Description
	bool	1 (true) if successful; 0 (false) on a failure
Arguments:	Type	Name Description
	vpiHandle	cb_obj Handle to the callback object
Related routines:	Use <code>vpi_register_cb()</code> to register callbacks for simulation-related events	

The VPI routine **vpi_remove_cb()** shall remove callbacks that were registered with *vpi_register_cb()*. The argument to this routine shall be a handle to the callback object. The routine shall return a **1** (true) if successful, and a **0** (false) on a failure. After **vpi_remove_cb()** is called with a handle to the callback, the handle is no longer valid.

PTF-151

24.34 vpi_scan()

vpi_scan()			
Synopsis:	Scan the Verilog HDL hierarchy for objects with a one-to-many relationship.		
Syntax:	vpi_scan(itr)		
	Type	Description	
Returns:	vpiHandle	Handle to an object	
	Type	Name	Description
Arguments:	vpiHandle	itr	Handle to an iterator object returned from vpi_iterate()
Related routines:	Use vpi_iterate() to obtain an iterator handle Use vpi_handle() to obtain handles to an object with a one-to-one relationship Use vpi_handle_multi() to obtain a handle to an object with a many-to-one relationship		

The VPI routine **vpi_scan()** shall traverse the instantiated Verilog HDL hierarchy and return handles to objects as directed by the iterator *itr*. The iterator handle shall be obtained by calling **vpi_iterate()** for a specific object type. Once **vpi_scan()** returns NULL, the iterator handle is no longer valid and cannot be used again.

PTF-187

The following example application uses **vpi_iterate()** and **vpi_scan()** to display each net (including the size for vectors) declared in the module. The example assumes it shall be passed a valid module handle.

```

void display_nets(mod)
vpiHandle mod;
{
    vpiHandle net;
    vpiHandle itr;

    vpi_printf("Nets declared in module %s\n",
vpi_get_str(vpiFullName, mod));

    itr = vpi_iterate(vpiNet, mod);
    while (net = vpi_scan(itr))
    {
        vpi_printf("\t%s", vpi_get_str(vpiName, net));
        if (vpi_get(vpiVector, net))
        {
            vpi_printf(" of size %d\n", vpi_get(vpiSize, net));
        }
        else vpi_printf("\n");
    }
}

```

Annex A

Scheduling Semantics

This annex presents semantics of simulation cycle for analog simulation as well as mixed A/D simulation cycle.

A.1 Analog Simulation Cycle

Simulation of a network, or system, starts with an analysis of each node to develop equations that define the complete set of values and flows in a network. Through transient analysis, the value and flow equations are solved incrementally with respect to time. At each time increment, equations for each signal are iteratively solved until they converge on a final solution.

A.1.1 Nodal Analysis

To describe a network, simulators combine constitutive relationships with Kirchhoff's laws in *nodal analysis* to form a system of differential-algebraic equations of the form

$$f(v, t) = \frac{dq(v, t)}{dt} + i(v, t) = 0$$

$$v(0) = v_0$$

These equations are a restatement of Kirchhoff's Flow Law.

v is a vector containing all node values

t is time

q and i are the dynamic and static portions of the flow

$f()$ is a vector containing the total flow out of each node

v_0 is the vector of initial conditions

This equation was formulated by treating all nodes as being conservative (even signal flow nodes). In this way, signal-flow and conservative terminals can be connected naturally. However, this results in unnecessary KFL equations for those nodes with only signal-flow terminals attached. This situation is easily recognized and those unnecessary equations are eliminated along with the associated flow unknowns, which must be by definition zero.

A.1.2 Transient Analysis

The equation describing the network is differential and nonlinear, which makes it impossible to solve directly. There are a number of different approaches to solving this problem numerically. However, all approaches discretize time and solve the nonlinear equations iteratively.

The simulator replaces the time derivative operator (dq/dt) with a discrete-time finite difference approximation. The simulation time interval is discretized and solved at individual time points along the interval. The simulator controls the interval between the time points to ensure the accuracy of the finite difference approximation. At each time point, a system of nonlinear algebraic equations is solved iteratively. Most circuit simulators use the NR method to solve this system.

)

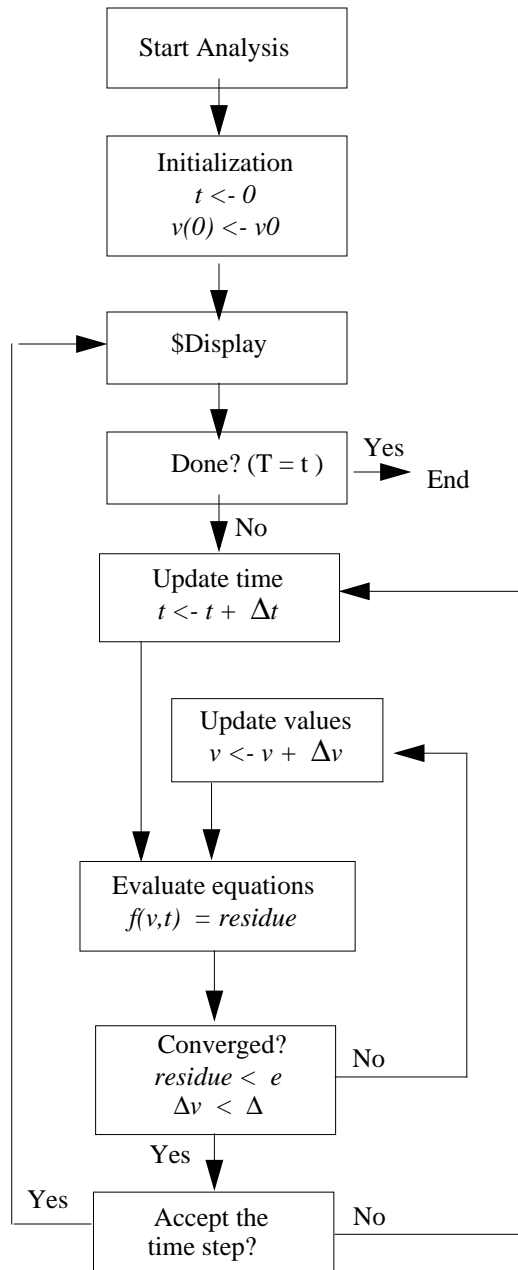


Figure A-1: Simulation Flowchart (Transient Analysis)

A.1.3 Convergence

In the analog kernel, the behavioral description is evaluated iteratively until the NR method converges. On the first iteration, the signal values used in expressions are approximate and do not satisfy Kirchhoff's laws.

In fact, the initial values might not be reasonable, so you must write models that do something reasonable even when given unreasonable signal values.

For example, if you compute the log or square root of a signal value, some signal values cause the arguments to these functions to become negative, even though a real-world system never exhibits negative values.

As the iteration progresses, the signal values approach the solution. Iteration continues until two convergence criteria are satisfied. The first criterion is that the proposed solution on this iteration, $v_n^{(j)}(t)$, must be close to the proposed solution on the previous iteration, $v_n^{(j-1)}(t)$, and

$$|v_n^{(j)} - v_n^{(j-1)}| < reltol (\max(|v_n^{(j)}|, |v_n^{(j-1)}|)) + abstol$$

where *reltol* is the relative tolerance and *abstol* is the absolute tolerance.

reltol is set as a simulator option and typically has a value of 0.001. There can be many absolute tolerances, and which one is used depends on the quantity the signal represents (volts, amps, and so on). The absolute tolerance is important when v_n is converging to zero. Without *abstol*, the iteration never converges.

The second criterion ensures that Kirchhoff's flow law is satisfied:

$$\left| \sum_n f_n(v^{(j)}) \right| < reltol (\max(|f_n^i(v^{(j)})|)) + abstol$$

where $f_n^i(v^{(j)})$ is the flow exiting node n from branch i .

Both of these criteria specify the absolute tolerance to ensure that convergence is not precluded when v_n or $f_n(v)$ go to zero. While you can set the relative tolerance once in an options statement to work effectively on any node in the circuit, the absolute tolerance must be scaled appropriately for its associated signal. The absolute tolerance should be the largest signal value that is considered negligible on all the signals with which it is associated.

The simulator uses absolute tolerance to get an idea of the scale of signals. Absolute tolerances are typically 1,000 to 1,000,000 times smaller than the largest typical value for signals of a particular quantity. For example, in a typical integrated circuit, the largest potential is about 5 volts, so the default absolute tolerance for voltage is 1μV. The largest current is about 1mA, so the default absolute tolerance for current is 1pA.

A.2 Mixed-Signal Simulation Cycle

This section describes the semantics of the initialization and time-sweep phases of a transient analysis in mixed-signal simulation cycle.

A.2.1 Circuit Initialization

The initialization phase of a transient analysis is the process of initializing the circuit state before advancing time.

A.2.2 `dc_init` Flag

The `dc_init` global signal assumes a value of 1 at the beginning of the initialization process and transitions to zero when a stable state has been reached.

This allows for a Verilog-AMS module to be customized for the initialization and time-sweep portions of a transient analysis.

```

module dcNand(out,a,b);
    output out;
    input a, b;
    reg out;

    always begin
        if (!dc_init)
            out = #5 ~(a && b);
        else
            out = ~(a && b);
    end
endmodule

```

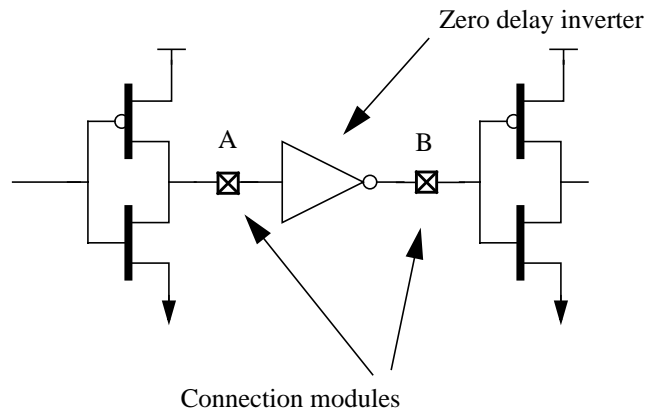
A.2.3 Transient Analysis & A/D Algorithm Synchronization

In the analog kernel time is a floating point value. In the digital kernel time is an integer value. Hence A2D events will in general not occur exactly at digital integer clock ticks.

For the purpose of reporting results and scheduling delayed future events the digital kernel truncates A2D events down to the earlier tick.

Any events that are scheduled with zero delay, as a result of the A2D, are not snapped down. Instead they are processed immediately.

Consequently an A2D event that results in a D2A event being scheduled with 0 delay, should have its effect propagated back to the analog kernel with zero delay.

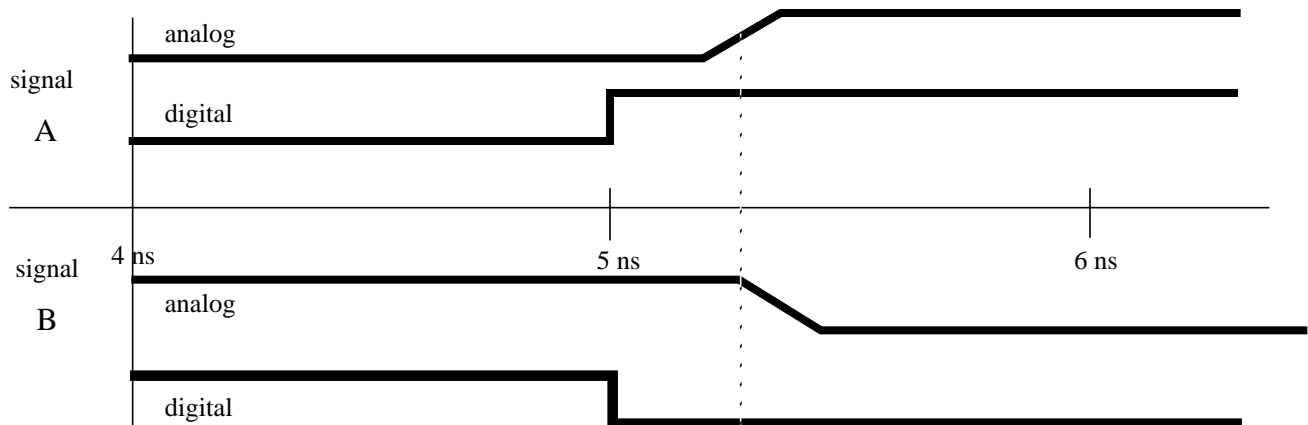
Example:

If this circuit is being simulated with a digital time resolution of $1\text{e-}9$ (one nanosecond) then all digital events will be reported by the digital kernel as having occurred at an integer multiple of $1\text{e-}9$.

If connector A detects a positive threshold crossing the resulting falling edge at connector B should be reported to the analog kernel with no further advance of analog time.

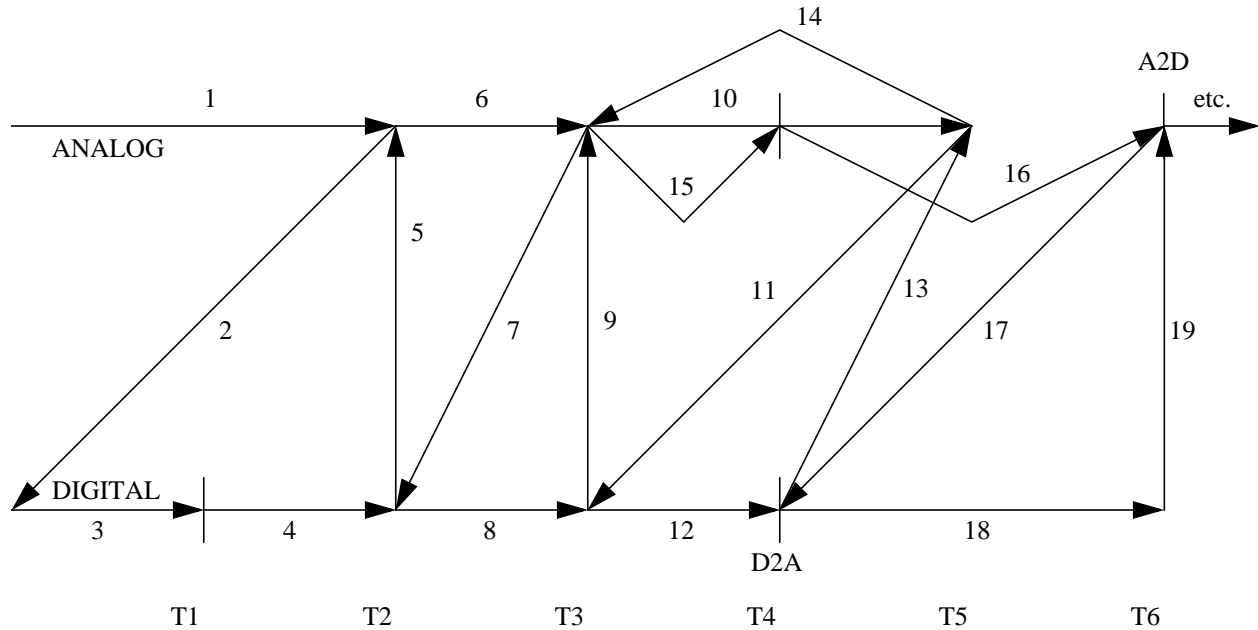
However the digital kernel may need to round the time of these events to the nearest nanosecond. Thus:

If A detects a positive crossing as a result of a transient solution at time $5.27\text{e-}9$ then the digital kernel will report a rising edge at A at time $5.0\text{e-}9$ and falling edge at B at time $5.0\text{e-}9$, but the analog kernel will see the transition at B begin at time $5.27\text{e-}9$



A.2.4 The Synchronization Loop

The digital and analog kernels will be synchronized in such a way that neither will compute results which the other is ineligible to accept. The synchronization algorithm may exploit characteristics of the analog and digital kernels described in the next section. A sample run is shown here:



1. Analog engine begins transient analysis and sends state information to the Digital engine (1,2)
2. Digital engine begins to run using its own time steps (3); however, if there is no D2A event, the Analog engine is not notified and the digital engine continues to simulate until it cannot advance its time without surpassing the time of the analog solution (4). Control of the simulation is then returned to the analog engine (5). The process is repeated (7,8,9,10, and 11).
3. If the Digital engine produces a D2A event (12), control of the simulation is returned to the Analog engine (13). The analog engine returns to the point at which the digital engine last surrendered control (14). The Analog engine recalculates the analog solution up to the time when the D2A event occurred (15). The Analog engine then takes the next time step (16).

4. If the analog engine produces an A2D event it returns control to the Digital engine (17), which simulates up to the time of the A2D event and then surrenders control (18 and 19).
5. This process continues until transient analysis is complete.

A.2.5 Assumptions about the Analog and Digital Algorithms

1. Advance of time in digital algorithm
 - The digital simulation has some minimum time granularity and all digital events occur at a time that is some integer multiple of that granularity
 - The digital simulator can always accept events for a given simulation time provided only that it has not yet executed events for a later time. Once it executes events for a given time it cannot accept events for an earlier time.
 - The digital simulator can always report the time of the most recently executed event, and the time of the next pending event.
2. Advance of time in analog algorithm
 - The analog simulator advances time by calculating a sequence of solutions. Each solution has an associated time which, unlike the digital time, is not constrained to a particular minimum granularity.
 - The analog simulator cannot tell for certain the time at which the next solution will converge. Thus, it can tell the time of the most recently calculated solution but not the time of the next solution.
 - In general the analog solution is a function of one or more previous solutions. Having calculated the solution for a given time the analog simulator can either accept or reject that solution. It cannot calculate a solution for a future time until it has accepted the solution for the current time.
3. Analog to Digital events
 - Analog to digital events are generated by conversion elements (which are analog/digital behavioral models) when evaluated by the analog simulator.
 - Analog events (e.g. cross, initial_step, final_step) cause an analog solution of the time at which they occur.
 - Thus, any analog to digital event is generated as the result of a particular transient solution. This means that until the events are passed to the digital simulator they

can stay associated with the solution which produced them, and be rejected along with the solution if it is rejected.

4. Digital to Analog events

- Digital to Analog events will cause an analog solution of the time in which they occur.

Annex B

Open Issues

This annex contains the list of all open issues known to the working group at this time:

- initial conditions for ddt operators
- real valued ports (Section 8.3.3) need more/better explanation
- idtmod function should only be used after careful analysis and understanding of its behavior.
- Connecting wires of different natures (analog-analog connection)
- Semantics of discipline resolutions
- Connect statement and matching different discipline connections
- Syntax and Semantics for backannotation (hierarchical references, SDF)
- Need for locally scoped parameters and variables
- VCD format for postprocessing tools

Annex C

Analog Language Subset

Prior to the release of Verilog-AMS the OVI board approved an analog only specification called Verilog-A v1.0. With the release of Verilog-AMS, the "official" Verilog-A LRM is no longer supported as it is included as part of the Verilog-AMS specification. The purpose of this Annex is to help developers define a working subset of Verilog-AMS HDL for analog only products.

C.1 Verilog-AMS Overview

The overview in Section 1 is applicable to both Verilog-AMS and Verilog-A with the following two additions:

1. Verilog-A overview: This Verilog-A Hardware Description Language (HDL) language annex defines a behavioral language for analog systems. Verilog-A HDL is derived from the IEEE 1364-1995 Verilog HDL specification and is a subset of the Verilog-AMS language specification. This annex is intended to cover the definition of Verilog-A HDL as proposed by Open Verilog International (OVI).
2. Verilog-A language features: The Verilog-A is a subset of Verilog-AMS containing only the analog semantics required for compatibility. Below is a list of salient features of the resulting language:

Build this list!

C.2 Lexical Tokens

With the exception of certain keywords required for Verilog-AMS the chapter 2 is applicable to both Verilog-A and Verilog-AMS. All Verilog-AMS keywords must be supported by Verilog-A as reserved words, but Verilog-D and Verilog-AMS specific keywords are not used. The following Verilog-AMS keywords are not required to be supported for a fully compliant Verilog-A subset:

1. From section 2.6.2.1, Verilog-AMS Keywords: The following are the keywords not used by Verilog-A HDL.

split	with	to
merged	using	connect

- From section 2.6.2.4, Built-in driver access functions: The following are reserved keywords for all built-in driver access functions and are not used by Verilog-A.

driver_active	driver_local	driver_state
driver_count	driver_next_state	driver_strength
driver_delay	driver_next_strength	

C.3 Data Types

The data types of Chapter 3 are applicable to both Verilog-AMS and Verilog-A with the following two exceptions.

- From section 3.4.2.2, Domain Binding: The domain binding type of discrete shall be an error in Verilog-A
- From section 3.5, Default Discipline: The default_discipline_directive compiler directive is not supported in Verilog-A. All Verilog-A modules must have a discipline defined within the module.

Note: This feature is provided to allow the use of digital modules in Verilog-AMS without editing them to add a discipline.

C.4 Expressions

The expressions defined in Chapter 4 are applicable to Verilog-AMS and Verilog-A.

C.5 Signals

The signals defined in Chapter 5 are applicable to Verilog-AMS and Verilog-A

C.6 Analog Behavior

The analog behavior defined in Chapter 6 are applicable to Verilog-AMS and Verilog-A

C.7 Mixed Signal

The Mixed-Signal section only applies to Verilog-AMS!!!!!!!!!!!!!!!!!!!!!!.

C.8 Hierarchical Structure

The hierarchical structure defined in Chapter 8 is applicable to Verilog-AMS and Verilog-A, except support for real value ports is only applicable to Verilog-AMS and Verilog-D (see from section 8.3.3, Real valued ports).

C.9 Scheduling Semantics

The analog simulation cycle is applicable to both Verilog-AMS and Verilog-A. The mixed signal simulation cycle from section A.2 is only applicable to Verilog-AMS.

C.10 Open Issues

Issues in annex B as they are addressed need to also be reviewed for their impact on this section.

C.11 Syntax

This annex, defines the differences between Verilog-AMS and Verilog-A. Annex D defines the BNF for Verilog-AMS.

C.12 Keywords

The keywords in this annex are the complete set of Verilog-AMS keywords including those from Verilog-D. Please refer to the above section on lexical tokens to define the list of Verilog-A keywords.

Note: All keywords of Verilog-AMS are reserved words for even Verilog-A.

C.13 System Tasks and Functions

The system tasks and functions in annex F are applicable to both Verilog-AMS and Verilog-A.

C.14 Compiler Directives

The compiler directives of annex G are applicable to both Verilog-AMS and Verilog-A.

C.15 Standard Definitions

The definitions of annex H are applicable to both Verilog-AMS and Verilog-A.

C.16 SPICE Compatability

Annex I defines the SPICE compatilibity for Verilog-A and Verilog-AMS..

C.17 Changes from Verilog-A LRM v1.0

As part of the Verilog-AMS development some changes have occured to the current Veriog-A. Most of these changes resulted in additional capabiliity; but some new compability issues now exist. This section highlights these differences.

C.17.1 New functions

- ceil
- floor
- idtmod

C.17.2 Changes

Expression	v1.0 Syntax	v1.4 Syntax
port branch	I< <i>a</i> >	I(< <i>a</i> >)
discontinuity	discontinuity(<i>x</i>)	discontinuity()
limexp	\$limexp(<i>expression</i>)	limexp(<i>expression</i>)
user-defined function	function	analog function
bound_step	bound_step(<i>const_expression</i>)	bound_step(<i>expression</i>)
domains	n/a	domain continuous
modulus operator	integers only	now supports integer and reals
k scalar (10^3)	n/a	now supported
genevar	n/a	genevar <i>list_of_genvar_identifiers</i>
initial_step	default = TRAN	default = ALL
final_step	default = TRAN	default = ALL

C.18 Obsolete Functionality

The following statements are not supported in the current version of Verilog-AMS; they are only noted for backward compability.

C.18.1 Forever statement

This statement is no longer supported.

C.18.2 NULL statement

This statement is no longer supported.

C.18.3 Generate statement

The *generate statement* is a looping construct that is unrolled at elaboration time. The generate statement can be used only in the analog block.

The syntax of generate statement is as follows:

```

generate_statement ::=
    generate genvar_identifier ( start_expr, end_expr [, incr_expr ] )
        statement

start_expr ::=
    genvar_expression

end_expr ::=
    genvar_expression

incr_expr ::=
    genvar_expression

```

Figure C-1: Syntax for generate statement

The index must not be assigned or modified inside the loop.

The start_expr, end_expr, and incr_expr are genvar expressions - expressions containing constants and variables declared as *genvar*. The genvar_identifier cannot be assigned to within the generate statement.

If the start_expr is less than the end_expr and the incr_expr is negative, or if the start_expr is greater than the end_expr and the incr_expr is positive, then the generate statement does not execute.

If the start_expr equals the end_expr, the incr_expr is ignored and the statement is executed once. If the incr_expr is not given, it defaults to +1 if the start_expr is less than the end_expr, and -1 if the start_expr is greater than the end_expr.

The statement, which can be a sequential block, is replicated with all occurrences of genvar_identifier in the statement replaced by its value. In the first instance of the statement, the genvar_identifier is replaced with the start_expr value. In the second, it is replaced by the value of the start_expr plus the incr_expr. In the third, it is replaced by the value of the start_expr plus two times the incr_expr. This pattern is repeated until the start_expr plus a multiple of the incr_expr is greater than the end_expr if incr_expr is positive or is less than the end_expr if incr_expr is negative.


Example:

This module implements a continuously running (not clocked) analog-to-digital converter.

```

module adc(in,out) ;
    parameter bits=8, fullscale=1.0, dly=0.0, ttime=10n;
    input in;
    output [0:bits-1] out;
    electrical in;
    electrical [0:bits-1] out;
    real sample, thresh;
    genvar i;

```



```
analog begin
  thresh = fullscale/2.0;
  generate i (bits-1,0) begin
    V(out[i]) <+ transition(sample > thresh, dly, ttime);
    if (sample > thresh) sample = sample - thresh;
    sample = 2.0*sample;
  end
end
endmodule
```


Annex D

Syntax

This annex contains the formal syntax definition of Verilog-AMS HDL. The conventions used are described in Section 1, Overview. Any category whose name begins with the italicized word *digital_* should be interpreted by its definition in the grammar given in ieee 1364 Annex A, and not by the local definition given herein. When such a category is defined herein (e.g. *digital_primary* ::=) then that definition should be taken to superceed the definition in ieee 1364 when used for Verilog-AMS.

D.1 Source text

```

source_text ::=
    {description}

description ::=
    module_declaration
    | discipline_definition
    | nature_definition
    | connect_statement
    | digital_udp_declaration

module_declaration ::=
    module module_identifier [ digital_list_of_ports ] ;
    [ module_items ]
    endmodule

module_items ::=
    { module_item }
    | analog_block

module_item ::=
    module_item_declaration
    | parameter_override
    | module_instantiation
    | digital_continuous_assignment
    | digital_gate_instantiation
    | digital_udp_instantiation
    | digital_specify_block
    | digital_initial_construct
    | digital_always_construct

module_item_declaration ::=
    parameter_declaration
    | digital_input_declaration

```

```

| digital_output_declaration
| digital_inout_declaration
| digital_integer_declaration
| digital_real_declaration
| node_declaration
| genvar_declaration
| branch_declaration
| function_declaration
| digital_net_declaration
| digital_reg_declaration
| digital_time_declaration
| digital_realtime_declaration
| digital_event_declaration
| digital_task_declaration

```

```

parameter_override ::=
    defparam list_of_param_assignments ;

```

D.2 Natures

```

nature_definition ::=
    nature nature_name
    [ nature_descriptions ]
    endnature

nature_name ::=
    nature_identifier
    | nature_identifier : parent_identifier

parent_identifier ::=
    nature_identifier
    | discipline_identifier.flow
    | discipline_identifier.potential

nature_descriptions ::=
    nature_description { nature_description }

nature_description ::=
    attribute = constant_expression ;

attribute ::=
    abstol
    | access
    | ddt_nature
    | idt_nature
    | units
    | attribute_identifier

```

D.3 Disciplines

```

discipline_definition ::=
    discipline discipline_identifier
    [ discipline_descriptions ]
    enddiscipline

discipline_descriptions ::=
    discipline_description { discipline_description }

discipline_description ::=
    nature_binding
    | attr_override
    | domain_binding

nature_binding ::=
    pot_or_flow nature_identifier ;

attr_override ::=
    pot_or_flow . attribute_identifier = constant_expression ;

pot_or_flow ::=
    potential
    | flow

domain_binding ::=
    domain discrete
    | domain continuous

```

D.4 Declarations

```

parameter_declaration ::=
    parameter [opt_type] list_of_param_assignments ;

opt_type ::=
    real
    | integer

list_of_param_assignments ::=
    declarator_init
    | list_of_param_assignments , declarator_init

declarator_init ::=
    parameter_identifier = constant_expression [ {opt_value_range} ]
    | identifier range = constant_param_arrayinit

opt_value_range ::=
    from value_range_specifier
    | exclude value_range_specifier
    | exclude constant_expression

```

```

value_range_specifier ::=
    start_range_spec expression1 : expression2 end_range_spec

start_range_spec ::=
    [
    | (

end_range_spec ::=
    ]
    | )

expression1 ::=
    constant_expression
    | -inf

expression2 ::=
    constant_expression
    | inf

constant_param_arrayinit ::=
    { param_arrayinit_element_list }

param_arrayinit_element_list
    param_arrayinit_element { , param_arrayinit_element }

param_arrayinit_element ::=
    | constant_expression [ = value_range_specifier ]
    | constant_expression { constant_expression [ =
    value_range_specifier ] }

node_declaration ::=
    discipline_identifier [range] list_of_nodes ;

list_of_nodes ::=
    node_identifier
    | hierarchical_node_identifier
    | node_identifier , list_of_nodes

branch_declaration ::=
    branch list_of_branches ;

list_of_branches ::=
    terminals list_of_branch_identifiers

terminals ::=
    ( node_or_port_scalar_expression )
    | ( node_or_port_scalar_expression , node_or_port_scalar_expression )

list_of_branch_identifiers ::=
    branch_identifier [ range ]
    | branch_identifier [ range ], list_of_branch_identifiers

genvar_declaration ::=
    genvar list_of_genvar_identifiers ;

list_of_genvar_identifiers ::=
    genvar_identifier { , genvar_identifier }

```

```

function_declaration ::=
    analog function [ type ] function_identifier ;
    function_item_declaration { function_item_declaration }
    statement
    endfunction
|
    function [ digital_range_or_type ] function_identifier ;
    function_item_declaration { function_item_declaration }
    digital_statement
    endfunction

type ::=
    integer
|
    real

function_item_declaration ::=
    input_declaration
|
    block_item_declaration

block_item_declaration ::=
    parameter_declaration
|
    integer_declaration
|
    real_declaration

```

D.5 Module instantiation

```

module_instantiation ::=
    module_identifier [ parameter_value_assignment ] instance_list

instance_list ::=
    module_instance { , module_instance } ;

module_instance ::=
    name_of_instance ( [ list_of_module_connections ] )

name_of_instance ::=
    module_instance_identifier [ range ]

list_of_module_connections ::=
    ordered_port_connection { , ordered_port_connection }
|
    named_port_connection { , named_port_connection }

ordered_port_connection ::=
    [ node_expression | ground ]

named_port_connection ::=
    .port_identifier ( [ node_expression | ground ] )

parameter_value_assignment ::=
    # ( ordered_param_override_list )
|
    # ( named_param_override_list )

ordered_param_override_list ::=
    constant_or_constant_array_expression { , constant_or_constant_array_expression }

```

```

named_param_override_list ::=
    named_param_override { , named_param_override }

named_param_override ::=
    . parameter_identifier ( constant_or_constant_array_expression )

constant_or_constant_array_expression ::=
    constant_expression
    | constant_array_expression

node_expression ::=
    node_identifier
    | node_identifier [ expression ]
    | node_identifier [ msb_constant_expression : lsb_constant_expression ]
    | node_concatenation

node_concatenation ::=
    { node_expression_list }

node_expression_list ::=
    node_expression { , node_expression }

```

D.6 Connect statements

D.7 Behavioral statements

```

analog_block ::=
    analog analog_statement

analog_statement ::=
    analog_block_statement
    | analog_branch_contribution
    | analog_indirect_branch_assignment
    | analog_procedural_assignment
    | analog_conditional_statement
    | analog_for_statement
    | analog_case_statement
    | analog_event_controlled_statement
    | system_task_enable
    | statement

statement ::=
    block_statement
    | procedural_assignment
    | conditional_statement
    | loop_statement
    | case_statement

analog_block_statement ::=
    begin [ : block_identifier { block_item_declaration } ]

```

```

        { analog_statement_or_null }
    end

analog_statement_or_null ::=
    analog_statement ! :

analog_branch_contribution ::=
    bvalue <+ analog_expression ;

analog_indirect_branch_assignment ::=
    bvalue : nexpr == analog_expression ;

nexpr ::=
    bvalue
    | ddt ( bvalue )
    | idt ( bvalue )

analog_procedural_assignment ::=
    lexpr = analog_expression ;

lexpr ::=
    integer_identifier
    | real_identifier
    | array_element

array_element ::=
    integer_identifier [ expression ]
    | real_identifier [ expression ]

analog_conditional_statement ::=
    if ( genvar_expression ) analog_statement_or_null
    [ else analog_statement_or_null ]

analog_case_statement ::=
    case ( analog_expression )
    analog_case_item { analog_case_item }
    endcase

analog_case_item ::=
    analog_expression { , analog_expression } : analog_statement_or_null
    | default [ : ] analog_statement_or_null

analog_for_statement ::=
    for ( genvar_assignment ; genvar_expression ;
    genvar_assignment ) analog_statement

event_controlled_statement ::=
    @ ( event_expression ) statement_or_null

event_expression ::=
    simple_event [ or event_expression ]

simple_event ::=
    global_event
    | event_function
    | identifier
    | digital_event_identifier

```

```

        | posedge digital_expression
        | negedge digital_expression

digital_event_expression ::=
        digital_expression
        | simple_event
        | digital_event_expression or digital_event_expression

global_event ::=
        initial_step [ ( analysis_list ) ]
        | final_step [ ( analysis_list ) ]

analysis_list ::=
        analysis_name { , analysis_name }

analysis_name ::=
        " analysis_identifier "

event_function ::=
        cross_function
        | timer_function

cross_function ::=
        cross ( arg_list )

timer_function ::=
        timer ( arg_list )

statement_or_null ::=
        statement ! :

system_task_enable ::=
        system_task_name [ ( expression { , expression } ) ] ;

system_task_name ::=
        $identifier

Note: The $ may not be followed by a space.

block_statement ::=
        begin [ : block_identifier { block_item_declaration } ]
        { statement }
        end

procedural_assignment ::=
        lexpr = expression ;

conditional_statement ::=
        if ( expression ) statement_or_null
        [ else statement_or_null ]

acase_statement ::=
        case ( expression )
        case_item { case_item }
        endcase

```

```

case_item ::=
    expression { , expression } : statement_or_null
    | default [ : ] statement_or_null

loop_statement ::=
    | repeat ( expression ) statement
    | while ( expression ) statement
    | for ( procedural_assignment ; expression ;
    |   procedural_assignment ) statement

```

D.8 Analog Expressions

```

analog_expression ::=
    expression
    | analog_operator ( arg_list )

analog_operator ::=
    ddt | idt | idtm | delay | transition | slew | bound_step
    | laplace_zd | laplace_zp | laplace_np | laplace_nd | discontinuity
    | zi_zp | zi_zd | zi_np | zi_nd | last_crossing | ac_stim | limexp
    | white_noise | flicker_noise | noise_table

genvar_expression ::=
    genvar_primary
    | unary_operator genvar_primary
    | genvar_expression binary_operator genvar_primary
    | genvar_expression ? genvar_expression : genvar_expression
    | string

genvar_primary ::=
    constant_primary
    | genvar_identifier
    | genvar_identifier [ genvar_expression ]
    | analysis ( arg_list )

genvar_assignment ::=
    genvar_identifier = genvar_expression

```

D.9 Expressions

```

range ::=
    [ constant_expression : constant_expression ]

constant_expression ::=
    constant_primary
    | string
    | unary_operator constant_primary
    | constant_expression binary_operator constant_expression
    | constant_expression ? constant_expression : constant_expression
    | constant_array_expression

```

```

        | attribute_reference
        | built_in_function(const_arg_list)

const_arg_list ::=
    constant_expression { , constant_expression }

attribute_reference ::=
    node_identifier . pot_or_flow . attribute_identifier

constant_primary ::=
    number
    | parameter_identifier
    | constant_concatenation

constant_array_expression ::=
    { constant_arrayinit_element { , constant_arrayinit_element } }

constant_arrayinit_element ::=
    | constant_expression
    | constant_expression { constant_expression }

expression ::=
    primary
    | unary_operator primary
    | expression binary_operator expression
    | expression ? expression : expression
    | function_call
    | access_function_reference
    | built_in_function ( arg_list )
    | system_function ( arg_list )

function_call ::=
    function_identifier ( arg_list )

arg_list ::=
    argument { , argument }

argument ::=
    expression
    | constant_array_expression

constant_array_expression ::=
    { constant_array_init_elemnet_list }

constant_array_init_elemnet_list ::=
    NULL
    | constant_array_init_element { , constant_array_init_element }

constant_array_init_element ::=
    constant_expression
    | constant_expression { , constant_expression }

```

```

access_function_reference ::=
    bvalue
    |
    pvalue
bvalue ::=
    access_identifier ( analog_signal_list )
analog_signal_list ::=
    branch_identifier
    |
    array_branch_identifier [ genvar_expression ]
    |
    node_or_port_scalar_expression
    |
    node_or_port_scalar_identifier , node_or_port_scalar_identifier
node_or_port_scalar_expression ::=
    node_or_port_identifier
    |
    array_node_or_port_identifier [ genvar_expression ]
    |
    buss_node_or_port_identifier [ genvar_expression ]
pvalue ::=
    flow_access_identifier ( < port_scalar_expression > )
port_scalar_expression ::=
    port_identifier
    |
    array_port_identifier [ genvar_expression ]
    |
    buss_port_identifier [ genvar_expression ]

unary_operator ::=
    + | - | ! | ~

binary_operator ::=
    + | - | * | / | % | == | != | && | ||
    |
    < | <= | > | >= | & | | | ^ | ^~ | ~^ | >> | <<

digital_primary ::=
    primary

primary ::=
    number
    |
    identifier
    |
    identifier [ expression ]
    |
    identifier [ digital_msb_constant_expression : digital_lsb_constant_expression ]
    |
    digital_concatenation
    |
    digital_multiple_concatenation
    |
    digital_function_call
    |
    ( digital_mintypmax_expression )
    |
    string
    |
    nexpr
    |
    ( expression )

number ::=
    decimal_number
    |
    real_number

decimal_number ::=
    [ sign ] unsigned_num

real_number ::=
    [ sign ] unsigned_num . unsigned_num
    |
    [ sign ] unsigned_num [ . unsigned_num ] e [ sign ] unsigned_num

```

```

        | [ sign ] unsigned_num [ . unsigned_num ] E [ sign ] unsigned_num
        | [ sign ] unsigned_num [ . unsigned_num ] scale_factor

concatenation ::=
    { expression { , expression } }

sign ::=
    +
    | -

unsigned_num ::=
    decimal_digit { _ | decimal_digit }

decimal_digit ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

scale_factor ::=
    T | G | M | K | k | m | u | n | p | f | a

built_in_function ::=
    ln | log | exp | sqrt | min | max | abs | pow | ceil | floor
    | sin | cos | tan | asin | acos | atan | atan2
    | sinh | cosh | tanh | asinh | acosh | atanh | hypot

driver_access_function ::=
    driver_count | driver_active | driver_local | driver_state |
    driver_strength | driver_delay | driver_next_state |
    driver_next_strength

system_function ::=
    $limexp | $realtime | $temperature | $vt

```

D.10 General

```

comment ::=
    short_comment
    | long_comment

short_comment ::=
    // comment_text \n

long_comment ::=
    /* comment_text */

comment_text ::=
    { Any_ASCII_character }

string ::=
    " { Any_ASCII_character_except_newline } "

identifier ::=
    IDENTIFIER [ { . IDENTIFIER } ]

```

NOTE: The period in identifier may not be preceded or followed by a space.

```
IDENTIFIER ::=
    simple_identifier
    | escaped_identifier

simple_identifier ::=
    [a-zA-Z_]{a-zA-Z_$0-9}

escaped_identifier ::=
    \ { Any_ASCII_character_except_white_space } white_space

white_space ::=
    space
    | tab
    | newline
```


Annex E

Keywords

This annex contains the list of all keywords used in Verilog-AMS HDL.

abs	endmodule	macromodule	sqrt
abstol	endfunction	max	strong0
access	endnature	medium	strong1
acos	endprimitive	min	supply0
acosh	endspecify	module	supply1
ac_stim	endtable	nand	table
always	endtask	nature	tan
analog	event	negedge	tanh
analysis	exclude	nmos	task
and	exp	noise_table	temperature
asin	final_step	nor	time
asinh	flicker_noise	not	timer
assign	flow	notif0	to
atan	for	notif1	tran
atan2	force	or	tranif0
atanh	forever	output	tranif1
begin	fork	parameter	transition
bound_step	from	pmos	tri
branch	function	posedge	tri0
buf	generate	potential	tri1
bufif0	genvar	pow	triand
bufif1	ground	primitive	trior
case	highz0	pull0	trireg
casex	highz1	pull1	units
casez	hypot	pullup	vectored
cmos	idt	pulldown	vt
connect	idt_nature	rcmos	wait
cos	if	real	wand
cosh	ifnone	realtime	weak0
cross	inf	reg	weak1
ddt	initial	release	while
ddt_nature	initial_step	repeat	white_noise
deassign	inout	rnmos	wire
default	input	rpmos	with
defparam	integer	rtran	wor
delay	join	rtranif0	xnor
disable	laplace_nd	rtranif1	xor
discipline	laplace_np	scalared	using
discontinuity	laplace_zd	sin	zi_nd
edge	laplace_zp	sinh	zi_np
else	large	slew	zi_zd
end	last_crossing	small	zi_zp
enddiscipline	ln	specify	
endcase	log	specparam	

Annex F

System Tasks and Functions

This annex describes system tasks and functions available in Verilog-AMS HDL.

F.1 Environment parameter functions

These functions return information about the current environment parameters as a real number. \$realtime and \$temperature do not take any input arguments; \$vt can optionally have temperature (in Kelvin units) as an input argument.

Function	Returns
\$realtime	Current simulation time in seconds.
\$temperature	Ambient temperature in kelvin.
\$vt	Thermal voltage (kT/q).
\$vt(temp)	Thermal voltage at given temperature.

F.2 \$random function

Syntax:

```
$random [ ( seed ) ] ;
```

The system function **\$random** provides a mechanism for generating random numbers. The function returns a new 32-bit random number each time it is called. The random number is a signed integer; it can be positive or negative.

The seed parameter controls the numbers that **\$random** returns. The seed parameter must be either a register, an integer, or a time variable. The seed value should be assigned to this variable prior to calling **\$random**.

Examples:

1. Where $b > 0$ the expression (**\$random** % b) gives a number in the following range: $[(-b+1): (b-1)]$. The following code fragment shows an example of random number generation between -59 and 59:

```
integer rand;
rand = $random % 60;
```

2. The following example shows how adding the concatenation operator to the preceding example gives rand a positive value from 0 to 59.

```
integer rand;
rand = {$random} % 60;
```

F.3 \$dist_ functions

Syntax:

```
$dist_uniform (seed, start, end) ;
$dist_normal (seed, mean, standard_deviation) ;
$dist_exponential (seed, mean) ;
$dist_poisson (seed, mean) ;
$dist_chi_square (seed, degree_of_freedom) ;
$dist_t (seed, degree_of_freedom) ;
$dist_erlang (seed, k_stage, mean) ;
```

Figure F-1: Syntax for the probabilistic distribution functions

All parameters to the system functions are real values, except for seed (which is an integer). For the exponential, poisson, chi-square, t, and erlang functions, the parameters mean, degree of freedom, and k_stage must be greater than 0.

Each of these functions returns a pseudo-random number whose characteristics are described by the function name. That is, **\$dist_uniform** returns random numbers uniformly distributed in the interval specified by its parameters.

For each system function, the seed parameter is an in-out parameter; that is, a value is passed to the function and a different value is returned. The system functions will always return the same value given the same seed. This facilitates debugging by making the operation of the system repeatable. The argument for the seed parameter should be an integer variable that is initialized by the user and only updated by the system function. This will ensure that the desired distribution is achieved.

All functions return a real value.

In the **\$dist_uniform** function, the start and end parameters are real inputs which bound the values returned. The start value should be smaller than the end value.

The mean parameter, used by **\$dist_normal**, **\$dist_exponential**, **\$dist_poisson**, and **\$dist_erlang**, is an real input which causes the average value returned by the function to approach the value specified.

The standard deviation parameter used with the **\$dist_normal** function is an real input which helps determine the shape of the density function. Larger numbers for standard

deviation will spread the returned values over a wider range. With a mean of 0 and standard deviation of 1, **\$dist_normal** generates gaussian distribution.

The degree of freedom parameter used with the **\$dist_chi_square** and **\$dist_t** functions is an real input which helps determine the shape of the density function. Larger numbers will spread the returned values over a wider range.

F.4 Simulation control system tasks

There are two simulation control system tasks, **\$finish** and **\$stop**.

F.4.1 \$finish

Syntax:

\$finish [(n)] ;

The **\$finish** system task simply makes the simulator exit and pass control back to the host operating system. If an expression is supplied to this task, then its value determines the diagnostic messages that are printed before the prompt is issued. If no argument is supplied, then a value of 1 is taken as the default.

Parameter Value	Diagnostic Message
0	prints nothing
1	prints simulation time and location
2	prints simulation time, location, and statistics about the memory and CPU time used in simulation

F.4.2 \$stop

Syntax:

\$stop [(n)] ;

The **\$stop** system task causes simulation to be suspended. This task takes an optional expression argument (0, 1, or 2) that determines what type of diagnostic message is printed. The amount of diagnostic messages output increases with the value of the optional argument passed to **\$stop**.

F.5 File operation tasks

F.5.1 \$fopen

Syntax:

```
integer multi_channel_descriptor = $fopen ( " file_name " );
```

The function **\$fopen** opens the file specified as an argument and returns a 32-bit unsigned multichannel descriptor that is uniquely associated with the file. It returns 0 if the file could not be opened for writing.

The multichannel descriptor should be thought of as a set of 32 flags, where each flag represents a single output channel. The least significant bit (bit 0) of a multichannel descriptor always refers to the standard output. The standard output is also called channel 0. The other bits refer to channels that have been opened by the **\$fopen** system function.

The first call to **\$fopen** opens channel 1 and returns a multichannel descriptor value of 2—that is, bit 1 of the descriptor is set. A second call to **\$fopen** opens channel 2 and returns a value of 4—that is, only bit 2 of the descriptor is set. Subsequent calls to **\$fopen** open channels 3, 4, 5, and so on and return values of 8, 16, 32, and so on, up to a maximum of 32 open channels. Thus, a channel number corresponds to an individual bit in a multichannel descriptor.

F.5.2 \$fclose

Syntax:

```
file_close_task ::=  
    $fclose ( multi_channel_descriptor );
```

The **\$fclose** system task closes the channels specified in the multichannel descriptor, and does not allow any further output to the closed channels. The **\$fopen** task will reuse channels that have been closed.

F.6 Displaying results

The system task **\$strobe** provides the ability to display simulation data when the simulator has converged on a solution for all nodes.

The **\$strobe task** displays its arguments in the same order they appear in the argument list. Each argument can be a quoted string, an expression that returns a value, or a null argument.

The contents of string arguments are output literally except when certain escape sequences are inserted to display special characters or specify the display format for a subsequent expression.

Escape sequences are inserted into a string in three ways:

- The special character `\` indicates that the character to follow is a literal or non-printable character (see Table F-1:).
- The special character `%` indicates that the next character should be interpreted as a format specification that establishes the display format for a subsequent expression argument (Table F-2:). For each `%` character that appears in a string, a corresponding expression argument must be supplied after the string.
- The special character string `%%` indicates the display of the percent sign character `%` (see Table F-1:).

Any null argument produces a single space character in the display. (A null argument is characterized by two adjacent commas in the argument list.)

The **\$stroke** task, when invoked without arguments, simply prints a newline character.

F.6.1 Escape sequences for special characters

The following escape sequences, when included in a string argument, cause special characters to be displayed:

Table F-1: : Escape sequences for printing special characters

<code>\n</code>	is the newline character
<code>\t</code>	is the tab character
<code>\\</code>	is the <code>\</code> character
<code>\"</code>	is the <code>"</code> character
<code>\ddd</code>	is a character specified by 1 to 3 octal digits
<code>%%</code>	is the <code>%</code> character

F.6.2 Format specifications

Table F-2: shows the escape sequences used for format specifications. Each escape sequence, when included in a string argument, specifies the display format for a subsequent expression. For each `%` character (except `%m`) that appears in a string, a corresponding expression must follow the string in the argument list. The value of the expression replaces the format specification when the string is displayed.

Table F-2: : Escape sequences for format specifications

<code>%h</code> or <code>%H</code>	display in hexadecimal format
<code>%d</code> or <code>%D</code>	display in decimal format
<code>%o</code> or <code>%O</code>	display in octal format
<code>%b</code> or <code>%B</code>	display in binary format

Table F-2: : Escape sequences for format specifications

%c or %C	display in ASCII character format
%m or %M	display hierarchical name
%s or %S	display as a string

Any expression argument that has no corresponding format specification is displayed using the default decimal format in **\$strobe**.

The format specifications in Table F-3: are used with real numbers and have the full formatting capabilities available in the C language. For example, the format specification %10.3g specifies a minimum field width of 10 with 3 fractional digits.

Table F-3: : Format specifications for real numbers

%e or %E	display 'real' in an exponential format
%f or %F	display 'real' in a decimal format
%g or %G	display 'real' in exponential or decimal format, whichever format results in the shorter printed output

F.6.3 Hierarchical name format

The %m format specifier does not accept an argument. Instead, it causes the display task to print the hierarchical name of the module, task, function, or named block that invokes the system task containing the format specifier. This is useful when there are many instances of the module that calls the system task. One obvious application is timing check messages in a flip-flop or latch module; the %m format specifier will pinpoint the module instance responsible for generating the timing check message.

F.6.4 String format

The %s format specifier is used to print ASCII codes as characters. For each %s specification that appears in a string, a corresponding parameter must follow the string in the argument list. The associated argument is interpreted as a sequence of 8-bit hexadecimal ASCII codes, with each 8 bits representing a single character. If the argument is a variable, its value should be right-justified so that the right-most bit of the value is the least-significant bit of the last character in the string. No termination character or value is required at the end of a string, and leading zeros are never printed.

F.7 Others - from Ian's writeup

Please look at these -- much more is needed.

F.7.1 System tasks and functions

The table below is from Section 14 of the P1364 LRM. Following this is a repeat of the material from 4.2.3, "Environment Parameters".

The following symbols in the first column have been used to flag actions that need to be taken:

- x - task/function is already covered in Annex F
- + - P1364 definition should work in analog context also
- * - see notes later in section
- ! - proposed extension to P1364

F.7.2 Display tasks

- * \$display{,b,h,o}
- \$monitor{,b,h,o}
- \$monitor{on,off}
- x \$strobe{,b,h,o}
- * \$write{,b,h,o}

\$strobe always emits a newline. Propose adding \$write as similar to \$strobe but no newline, and \$display as same as \$strobe. These all read values after convergence and are thus similar in flavor to \$strobe (at end of time slot).

F.7.3 File I/O tasks

- x \$fclose
- * \$fdisplay{,b,h,o}
- \$fmonitor{b,h,o}
- x \$fopen
- * \$fstrobe{,b,h,o}
- * \$fwrite{,b,h,o}
- * \$readmemb
- * \$readmemh
- ! \$readmemr

\$fopen/\$fclose are in Annex F. \$fstrobe, etc, are missing. Propose same approach as for 'display tasks' for \$fstrobe, etc.

Propose augmenting the \$readmem functions with \$readmemr to read real values (including support for scale factors like "1.5u", etc).

F.7.4 Timescale tasks`$printtimescale``$timeformat`**F.7.5 Simulation control tasks**`x $finish``x $stop`**F.7.6 Timing check tasks**`$hold``$period``$setup``$skew``$nochange``$recovery``$setuphold``$width`**F.7.7 PLA modeling tasks**`$async*``$sync*`**F.7.8 Stochastic analysis tasks**`$q_*``x $random`**F.7.9 Simulation time functions**`* $realtime``* $time``* $stime`

There is a problem here. The P1364 versions return time scaled to the timescale unit of the module. The A/MS LRM 'environment parameter' returns "current simulation time

in seconds".

F.7.10 Conversion functions for reals

- * \$bitstotreal
- * \$itor
- * \$realtobits
- * \$rtoi

These should be handled by the mechanism that deals with incompatible discrete disciplines (whatever that is).

F.7.11 Probabilistic distribution functions

- * \$dist_chi_square
- * \$dist_erlang
- * \$dist_exponential
- * \$dist_normal
- * \$dist_poisson
- * \$dist_t
- * \$dist_uniform

There are several problems here. The return type of the P1364 versions is unspecified, but is probably intended to be a 32-bit signed integer, as for \$random. The return type of the A/MS versions has been explicitly defined to be real.

Some of the arguments to these functions need to be integer and some of them need to be real (in P1364 they are all integers):

chi_square: freedom is int
 erlang: mean is real
 exponential: mean is real
 normal: mean, std_dev are real
 poisson: mean is real
 t: freedom is real <g>
 uniform: start, end are integer

F.7.12 Environment Parameters (from 4.2.3 of A/MS LRM)

- * \$realtime
- * \$temperature
- * \$vt
- * \$vt(temp)

As noted above, \$realtime is defined differently (owing to module scaling) between the two contexts.

Is the intention to import \$temperature, etc, into P1364?

Annex G

Compiler Directives

All Verilog-AMS HDL compiler directives are preceded by the (```) character. This character is called accent grave. It is different from the character (`'`), which is the single quote character. The scope of compiler directives extends from the point where it is processed, across all files processed, to the point where another compiler directive supersedes it or the processing completes.

This annex describes the following compiler directives:

```
`default_discipline
`default_transition
`timescale
`define
`else
`endif
`ifdef
`include
`resetall
`undef
```

G.1 ``default_discipline`

The directive ``default_discipline` controls the node type created for implicit node declarations (see section 3.4.4). It can be used only outside of module definitions. It affects all modules that follow the directive, even across source file boundaries. Multiple ``default_discipline` directives are allowed. The latest occurrence of this directive in the source controls the type of nodes that will be implicitly declared. The following is the syntax of the directive:

```
default_discipline_directive ::=
    `default_discipline [discipline_identifier [qualifier] [scope]]

qualifier ::=
    integer | real | reg |
    wire | tri | wand | triand | wor | trior | trireg |
    tri0 | tri1 | supply0 | supply1

scope ::= module_identifier
```

Figure G-1: Syntax for default nodetype compiler directive

G.2 ``timescale`

This directive specifies the time unit and time precision of the modules that follow it. The time unit is the unit of measurement for time values such as the simulation time and delay values.

To use modules with different time units in the same design, the following timescale constructs are useful:

- The ``timescale` compiler directive to specify the unit of measurement for time and precision of time in the modules in the design

The ``timescale` compiler directive specifies the unit of measurement for time and delay values and the degree of accuracy for delays in all modules that follow this directive until another ``timescale` compiler directive is read.

The syntax for the ``timescale` directive is given in Figure G-2:

```
timescale_compiler_directive ::=
    `timescale time_unit / time_precision
```

Figure G-2: Syntax for timescale compiler directive

The `time_unit` argument specifies the unit of measurement for times and delays.

The `time_precision` argument specifies how delay values are rounded before being used in simulation. The values used are accurate to within the unit of time that is specified here. The smallest `time_precision` argument of all the ``timescale` compiler directives in the design determines the time unit of the simulation.

The `time_precision` argument shall be at least as precise as the `time_unit` argument; it cannot specify a longer unit of time than `time_unit`.

The integers in these arguments specify an order of magnitude for the size of the value; the valid integers are 1, 10, and 100. The character strings represent units of measurement; the valid character strings are **s**, **ms**, **us**, **ns**, **ps**, and **fs**.

The units of measurement specified by these character strings are given in Table G-1:

Table G-1: Arguments of `time_precision`

Character string	Unit of measurement
s	seconds
ms	milliseconds
us	microseconds

Table G-1: Arguments of time_precision, continued

Character string	Unit of measurement
ns	nanoseconds
ps	picoseconds
fs	femtoseconds

The following example shows how this directive is used:

```
`timescale 1 ns / 1 ps
```

Here, all time values in the modules that follow the directive are multiples of 1 ns because the time_unit argument is “1 ns”. Delays are rounded to real numbers with three decimal places—or precise to within one thousandth of a nanosecond—because the time_precision argument is “1 ps,” or one thousandth of a nanosecond.

Consider the following example:

```
`timescale 10 us / 100 ns
```

The time values in the modules that follow this directive are multiples of 10 s because the time_unit argument is “10 us”. Delays are rounded to within one tenth of a microsecond because the time_precision argument is “100 ns,” or one tenth of a microsecond.

G.3 **`default_transition**

The transition time directive specifies the default value for rise and fall time for transition filter (section 4.4.8). There are no scope restrictions for this directive. The syntax for this directive is shown below in Figure G-3:

```
default_transition_compiler_directive ::=  
    `default_transition transition_time
```

Figure G-3: Syntax for default transition compiler directive

The transition_time is an integer value. For all transition filters that follow this directive and do not have rise time and fall time arguments specified, transition_time is used as the default rise and fall time value. If another transition time directive is encountered in the subsequent source description, the transition filters following the newly encountered directive derive their default rise and fall time from the transition time value of the newly encountered directive. In other words, the default rise and fall times for a transition filter

are derived from the `transition_time` value of the directive that immediately precedes the transition filter.

If transition time directive is not used in the description, the `transition_time` defaults to the smallest time precision specified by the `timescale` directive.

G.4 ``define` and ``undef`

A text macro substitution facility has been provided so that meaningful names can be used to represent commonly used pieces of text. For example, in the situation where a constant number is repetitively used throughout a description, a text macro would be useful in that only one place in the source description would need to be altered if the value of the constant needed to be changed.

G.4.1 ``define`

The directive ``define` creates a macro for text substitution. This directive can be used both inside and outside module definitions. After a text macro is defined, it can be used in the source description by using the `(`)` character, followed by the macro name. The compiler substitutes the text of the macro for the string ``macro_name`. All compiler directives are considered pre-defined macro names; it is illegal to re-define a compiler directive as a macro name.

A text macro can be defined with arguments. This allows the macro to be customized for each use individually.

The syntax for text macro definitions is as follows:

```
text_macro_definition ::=
    `define text_macro_name macro_text

text_macro_name ::=
    text_macro_identifier [ ( list_of_formal_arguments ) ]

list_of_formal_arguments ::=
    formal_argument_identifier { , formal_argument_identifier }
```

Figure G-4: Syntax for text macro definition

The macro text can be any arbitrary text specified on the same line as the text macro name. If more than one line is necessary to specify the text, the newline must be preceded by a backslash (`\`). The first newline not preceded by a backslash will end the macro text. The newline preceded by a backslash is replaced in the expanded macro with a newline (but without the preceding backslash character).

When formal arguments are used to define a text macro, the scope of the formal arguments extend up to the end of the macro text. A formal argument can be used in the macro text in the same manner as an identifier.

If a one-line comment (that is, a comment specified with the characters `//`) is included in the text, then the comment does not become part of the text substituted. The macro text can be blank, in which case the text macro is defined to be empty and no text is substituted when the macro is used.

The syntax for using a text macro is as follows:

```
text_macro_usage ::=
    `text_macro_identifier [ ( list_of_actual_arguments ) ]

list_of_actual_arguments ::=
    actual_argument { , actual_argument }

actual_argument ::=
    expression
```

Figure G-5: Syntax for text macro usage

For an argument-less macro, the text is substituted “as is” for every occurrence of ``text_macro`. However, a text macro with one or more arguments must be expanded by substituting each formal argument with the expression used as the actual argument in the macro usage.

Once a text macro name has been defined, it can be used anywhere in a source description; that is, there are no scope restrictions. Text macros may be defined and used interactively.

The text specified for macro text can not be split across the following lexical tokens:

- comments
- numbers
- strings
- identifiers
- keywords
- operators

Examples:

```
`define M_PI      3.14159265358979323846
```

```

`define size 8
electrical [1:`size] vout;

//define an adc with variable delay
`define var_adc(dly) adc #(dly)

`var_adc(2) g121 (q21, n10, n11);
`var_adc(5) g122 (q22, n10, n11);

```

The following is illegal syntax because it is split across a string:

```

`define first_half "start of string
$display(`first_half end of string");

```

Note: Text macro names can not be the same as compiler directive keywords.

Note: Text macro names can re-use names being used as ordinary identifiers. For example, `signal_name` and ``signal_name` are different.

Note: Redefinition of text macros is allowed; the latest definition of a particular text macro read by the compiler prevails when the macro name is encountered in the source text.

G.4.2 ``undef`

The directive ``undef` undefines a previously defined text macro. An attempt to undefine a text macro that was not previously defined using a ``define` compiler directive can result in a warning. The syntax for ``undef` compiler directive is as follows:

```

undefine_compiler_directive ::=
    `undef text_macro_name

```

Figure G-6: Syntax for undef compiler directive

An undefined text macro has no value.

G.5 ``ifdef`, ``else`, ``endif`

These conditional compilation compiler directives are used to optionally include lines of a Verilog-AMS HDL source description during compilation. The ``ifdef` compiler directive checks for the definition of a variable name. If the variable name is defined then the lines following the ``ifdef` directive are included. If the variable name is not defined and an ``else` directive exists then this source is compiled.

These directives may appear anywhere in the source description.

Situations where the ``ifdef`, ``else`, and ``endif` compiler directives may be useful include:

- selecting different representations of a module such as behavioral, structural, or mixed level
- choosing different timing or structural information
- selecting different stimulus for a given simulation run

The **`ifdef**, **`else**, and **`endif** compiler directives have the following syntax:

```
conditional_compilation_directive ::=
    `ifdef text_macro_name
        first_group_of_lines
    [ `else
        second_group_of_lines
    `endif ]
```

Figure G-7: Syntax for conditional compilation directives

The text macro name is a Verilog-AMS HDL identifier. The first group of lines and the second group of lines are parts of a Verilog-AMS HDL source description. The **`else** compiler directive and the second group of lines are optional.

The **`ifdef**, **`else**, and **`endif** compiler directives work in the following manner:

- When an **`ifdef** is encountered, the text macro name is tested to see if it is defined as a text macro name using **`define** within the Verilog-AMS HDL source description.
- If the text macro name is defined, the first group_of_lines is compiled as part of the description. If there is an **`else** compiler directive, the second group of lines is ignored.
- If the text macro name has not been defined, the first group of lines is ignored. If there is an **`else** compiler directive the second group of lines is compiled.

Note: Any group of lines that the compiler ignores still must follow the Verilog-AMS HDL lexical conventions for white space, comments, numbers, strings, identifiers, keywords, and operators.

Note: These compiler directives may be nested.

G.6 **`include**

The file inclusion (**`include**) compiler directive is used to insert the entire contents of a source file in another file during compilation. The result is as though the contents of the included source file appear in place of the **`include** compiler directive. The **`include** compiler directive can be used to include global or commonly used definitions and tasks without encapsulating repeated code within module boundaries.

Advantages of using the **`include** compiler directive include the following:

- providing an integral part of configuration management
- improving the organization of Verilog-AMS HDL source descriptions
- facilitating the maintenance of Verilog-AMS HDL source descriptions

The syntax for the **`include** compiler directive is as follows:

```
include_compiler_directive ::=
    `include "filename"
```

Figure G-8: Syntax for include compiler directive

The compiler directive **`include** can be specified anywhere within the Verilog-AMS HDL description. The *filename* is the name of the file to be included in the source file. The *filename* can be a full or relative path name.

Only white space or a comment may appear on the same line as the **`include** compiler directive.

A file included in the source using **`include** compiler directive may contain other **`include** compiler directives. The number of nesting levels for included files are finite.

Examples:

Examples of legal comments for the **`include** compiler directive are as follows:

```
`include "parts/count.v"
`include "fileA"
`include "fileB" // including fileB
```

Note: Implementations may limit the maximum number of levels to which include files can be nested, but the limit shall be at least 15.

G.7 **`resetall**

When **`resetall** compiler directive is encountered during compilation, all compiler directives are set to the default values. This is useful for ensuring that only those directives that are desired in compiling a particular source file are active.

The recommended usage is to place **`resetall** at the beginning of each source text file, followed immediately by the directives desired in the file.

Annex H

Standard Definitions

This annex contains the standard definition package for Verilog-AMS HDL

```

`ifdef DISCIPLINES_H
`else
`define DISCIPLINES_H 1

//
// Natures and Disciplines
//

discipline logic
    domain discrete;
enddiscipline

/*
 * Default absolute tolerances may be overridden by setting the
 * appropriate _ABSTOL prior to including this file
 */

// Electrical

// Current in amperes
nature Current
    units    = "A";
    access   = I;
    idt_nature = Charge;
`ifdef CURRENT_ABSTOL
    abstol   = `CURRENT_ABSTOL;
`else
    abstol   = 1e-12;
`endif
endnature

// Charge in coulombs
nature Charge
    units    = "coul";
    access   = Q;
    ddt_nature = Current;
`ifdef CHARGE_ABSTOL
    abstol   = `CHARGE_ABSTOL;
`else
    abstol   = 1e-14;
`endif
endnature

```

```

// Potential in volts
nature Voltage
    units    = "V";
    access   = V;
    idt_nature = Flux;
`ifdef VOLTAGE_ABSTOL
    abstol   = `VOLTAGE_ABSTOL;
`else
    abstol   = 1e-6;
`endif
endnature

// Flux in Webers
nature Flux
    units    = "Wb";
    access   = Phi;
    ddt_nature = Voltage;
`ifdef FLUX_ABSTOL
    abstol   = `FLUX_ABSTOL;
`else
    abstol   = 1e-9;
`endif
endnature

// Conservative discipline
discipline electrical
    potential Voltage;
    flow      Current;
enddiscipline

// Signal flow disciplines
discipline voltage
    potential Voltage;
enddiscipline

discipline current
    potential Current;
enddiscipline

// Magnetic
// Magnetomotive force in Ampere-Turns.
nature Magneto_Motive_Force
    units    = "A*turn";
    access   = MMF;
`ifdef MAGNETO_MOTIVE_FORCE_ABSTOL
    abstol   = `MAGNETO_MOTIVE_FORCE_ABSTOL;
`else
    abstol   = 1e-12;
`endif
endnature

```

```

// Conservative discipline
discipline magnetic
    potential   Magneto_Motive_Force;
    flow        Flux;
enddiscipline

// Thermal

// Temperature in Kelvin
nature Temperature
    units       = "K";
    access      = Temp;
`ifdef TEMPERATURE_ABSTOL
    abstol      = `TEMPERATURE_ABSTOL;
`else
    abstol      = 1e-4;
`endif
endnature

// Power in Watts
nature Power
    units       = "W";
    access      = Pwr;
`ifdef POWER_ABSTOL
    abstol      = `POWER_ABSTOL;
`else
    abstol      = 1e-9;
`endif
endnature

// Conservative discipline
discipline thermal
    potential   Temperature;
    flow        Power;
enddiscipline

// Kinematic

// Position in meters
nature Position
    units       = "m";
    access      = Pos;
    ddt_nature = Velocity;
`ifdef POSITION_ABSTOL
    abstol      = `POSITION_ABSTOL;
`else
    abstol      = 1e-6;
`endif
endnature

```

```

// Velocity in meters per second
nature Velocity
    units      = "m/s";
    access     = Vel;
    ddt_nature = Acceleration;
    idt_nature = Position;
`ifdef VELOCITY_ABSTOL
    abstol    = `VELOCITY_ABSTOL;
`else
    abstol    = 1e-6;
`endif
endnature

// Acceleration in meters per second squared
nature Acceleration
    units      = "m/s^2";
    access     = Acc;
    ddt_nature = Impulse;
    idt_nature = Velocity;
`ifdef ACCELERATION_ABSTOL
    abstol    = `ACCELERATION_ABSTOL;
`else
    abstol    = 1e-6;
`endif
endnature

// Impulse in meters per second cubed
nature Impulse
    units      = "m/s^3";
    access     = Imp;
    idt_nature = Acceleration;
`ifdef IMPULSE_ABSTOL
    abstol    = `IMPULSE_ABSTOL;
`else
    abstol    = 1e-6;
`endif
endnature

// Force in Newtons
nature Force
    units      = "N";
    access     = F;
`ifdef FORCE_ABSTOL
    abstol    = `FORCE_ABSTOL;
`else
    abstol    = 1e-6;
`endif
endnature

// Conservative disciplines
discipline kinematic
    potential Position;
    flow      Force;
enddiscipline

```

```

discipline kinematic_v
    potential Velocity;
    flow Force;
enddiscipline

// Rotational

// Angle in radians
nature angle
    units = "rads";
    access = Theta;
    ddt_nature = Angular_Velocity;
`ifdef ANGLE_ABSTOL
    abstol = `ANGLE_ABSTOL;
`else
    abstol = 1e-6;
`endif
endnature

// Angular Velocity in radians per second
nature Angular_Velocity
    units = "rads/s";
    access = Omega;
    ddt_nature = Angular_Acceleration;
    idt_nature = Angular_Velocity;
`ifdef ANGULAR_VELOCITY_ABSTOL
    abstol = `ANGULAR_VELOCITY_ABSTOL;
`else
    abstol = 1e-6;
`endif
endnature

// Angular acceleration in radians per second squared
nature Angular_Acceleration
    units = "rads/s^2";
    access = Alpha;
    ddt_nature = Angular_Velocity;
`ifdef ANGULAR_ACCELERATION_ABSTOL
    abstol = `ANGULAR_ACCELERATION_ABSTOL;
`else
    abstol = 1e-6;
`endif
endnature

// Torque in Newtons
nature Angular_Force
    units = "N*m";
    access = Tau;
`ifdef ANGULAR_FORCE_ABSTOL
    abstol = `ANGULAR_FORCE_ABSTOL;
`else
    abstol = 1e-6;
`endif
endnature

```

```
// Conservative disciplines
discipline rotational
  potential    Angle;
  flow        Angular_Force;
enddiscipline

discipline rotational_omega
  potential    Angular_Velocity;
  flow        Angular_Force;
enddiscipline

`endif
```

```

// Mathematical and physical constants
`ifdef CONSTANTS_H
`else
`define CONSTANTS_H 1

// M_ is a mathematical constant
`define      M_E                2.7182818284590452354
`define      M_LOG2E            1.4426950408889634074
`define      M_LOG10E          0.43429448190325182765
`define      M_LN2             0.69314718055994530942
`define      M_LN10            2.30258509299404568402
`define      M_PI              3.14159265358979323846
`define      M_TWO_PI          6.28318530717958647652
`define      M_PI_2            1.57079632679489661923
`define      M_PI_4            0.78539816339744830962
`define      M_1_PI            0.31830988618379067154
`define      M_2_PI            0.63661977236758134308
`define      M_2_SQRTPI        1.12837916709551257390
`define      M_SQRT2           1.41421356237309504880
`define      M_SQRT1_2         0.70710678118654752440

// P_ is a physical constant

// charge of electron in coulombs
`define      P_Q                1.6021918e-19

// speed of light in vacuum in meters/sec
`define      P_C                2.997924562e8

// Boltzman's constant in joules/kelvin
`define      P_K                1.3806226e-23

// Plank's constant in joules*sec
`define      P_H                6.6260755e-34

// permittivity of vacuum in farads/meter
`define      P_EPS0             8.85418792394420013968e-12

// permeability of vacuum in henrys/meter
`define      P_U0               (4.0e-7 * `M_PI)

// zero celsius in kelvin
`define      P_CELSIUS0        273.15

`endif

```


Annex I

SPICE Compatibility

I.1 Introduction

Analog simulation has long been performed with SPICE and SPICE-like simulators. As such, there is a huge legacy of SPICE netlists. In addition, SPICE provides a rich set of predefined models and it is considered neither practical nor desirable to convert these models in to a Verilog behavioral description. In order for Verilog to be embraced by the analog design community, it is important that Verilog provide an appropriate degree of SPICE compatibility. This annex describes the degree of compatibility that Verilog provides and the approach taken to provide that compatibility.

I.1.1 Scope of Compatibility

SPICE is not a single language, but rather is a family of related languages. The first widely used version of SPICE was SPICE2g6 from the University of California at Berkeley. However, SPICE has been enhanced and distributed by many different companies, each of which has added their own extensions to the language and the models. As a result, there is a great deal of incompatibility even among the SPICE languages themselves.

Verilog makes no judgement as to which one of the various SPICE languages should be supported. Instead, it states that if a simulator that supports Verilog is also able to read SPICE netlists of a particular flavor, then certain objects defined in that flavor of SPICE netlist can be referenced from within a Verilog structural description. In particular, SPICE models and subcircuits can be instantiated within a Verilog module. This would also be true for any SPICE primitives that are built into the simulator.

I.1.2 Degree of Incompatibility

There are four primary areas of incompatibility between versions of SPICE simulators.

First the version of the SPICE language accepted by various simulators is different and to some degree proprietary. This issues is not addressed by Verilog. So whether a particular Verilog-AMS simulator is SPICE compatible, and with which particular variant of SPICE it is compatible with, is solely determined by the authors of the simulator.

The second area of incompatibility is that not all SPICE simulators support the same set of component primitives. Thus, a particular SPICE netlist may reference a primitive that is unsupported. Verilog offers no alternative in this case other than the possibility that if the model equations are known, the primitive may be rewritten as a module.

The third area of incompatibility is that the names of the built-in SPICE primitives, their parameters, or their ports might differ from simulator to simulator. This is particularly true because many primitives, parameters, and ports are unnamed in SPICE. When instantiating SPICE primitives in Verilog, the primitives must, and parameters and ports can, be named. Since there are no established standard names, there is a high likelihood of incompatibility cropping up in these names. To reduce this, a list is given as to what names must be used for the more common components in the next section. However, it is not possible to anticipate all SPICE primitives and parameters that will be supported, and so different implementations may end up using different names. However, this level of incompatibility can be overcome by using wrapper modules to map names.

The final area of incompatibility is in the mathematical description of the built-in primitives. As with the netlist syntax, through the years incompatible enhancements of the models creep in. Again, Verilog offers no solution in this case other than the possibility that if the model equations are known, the primitive may be rewritten as a module.

I.2 Accessing SPICE Objects from Verilog

If an implementation of a Verilog tool supports SPICE compatibility, then it is expected to provide the basic set of SPICE primitives (as listed in the “Preferred Primitive, Parameter, and Port Names” section on page -4). It is also expected that it will be able to read SPICE netlists that contain models and subcircuit statements.

SPICE primitives that are built-in to the simulator are treated in the same manner as Verilog built-in primitives. However, while the Verilog built-in primitives are standardized, the SPICE primitives are not. All aspects of SPICE primitives are implementation dependent.

In addition to SPICE primitives, it is also possible to access subcircuits and models defined within SPICE netlists. The subcircuits and models contained within the SPICE netlist are treated as module definitions.

I.2.1 Case Sensitivity

SPICE netlists are case insensitive where as Verilog descriptions are case sensitive. From within Verilog, a mixed case name will first match the same name with identical case if it were defined in a Verilog description. However, if no exact match is found, then the mixed case name will match the same name defined within SPICE regardless of the case.

I.2.2 Examples

I.2.2.1 Accessing SPICE Models

Consider the following SPICE model file being read by a Verilog-AMS simulator.

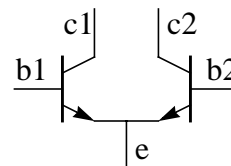
```
.MODEL VERTNPN NPN BF=80 IS=1E-18 RB=100 VAF=50
+ CJE=3PF CJC=2PF CJS=2PF TF=0.3NS TR=6NS
```

This model could be instantiated in a Verilog module as shown below.

```
module diffPair (c1, b1, e, b2, c2);
    electrical c1, b1, e, b2, c2;

    vertNPN Q1 (c1, b1, e, );
    vertNPN Q2 (.c(c2), .b(b2), .e(e));

endmodule
```



Unlike with SPICE, the first letter of the instance name, in this case *Q1* and *Q2*, is not constrained by the primitive type. For example, they can just as easily be *T1* and *T2*.

The ports and params of the BJT are determined by the BJT primitive itself and not by the model statement for the BJT. More on this in the next major section. The BJT has 3 mandatory ports (collector, base, and emitter) and one optional port (the substrate). In the instantiation of *Q1* the ports are passed by order. With *Q2*, the ports are passed by name. In both cases, the optional substrate port is defaulted by simply not giving it.

I.2.2.2 Accessing SPICE Subcircuits

As an example of how a SPICE subcircuit is referenced from Verilog, consider the following SPICE subcircuit definition of an oscillator.

```
.SUBCKT ECPOSC (OUT GND)
    VCC VCC GND 5
    IEE E GND 1MA
    Q1 VCC B1 E VCC VERTNPN
    Q2 OUT B2 E OUT VERTNPN
    L1 VCC OUT 1UH
    C1 VCC OUT 1P IC=1
    C2 OUT B1 272.7PF
    C3 B1 GND 3NF
    R1 B1 GND 10K
    C4 B2 GND 3NF
    R2 B2 GND 10K
.ENDS ECPOSC
```

This oscillator would be referenced from Verilog as follows:

```
module osc (out, gnd);
    electrical out, gnd;

    ecpOsc Osc1 (out, gnd);

endmodule
```

Notice that in Verilog the name of the subcircuit instance is not constrained to start with *X* as it is in SPICE.

I.2.2.3 Accessing SPICE Primitives

To show how various SPICE primitives would be accessed from Verilog, the above subcircuit is translated to native Verilog.

```

module ecpOsc (out, gnd);
    electrical out, gnd;

    vsource #(.dc(5)) Vcc (vcc, gnd);
    isource #(.dc(1m)) Iee (e, gnd);
    vertnpnQ1 (vcc, b1, e, vcc);
    vertnpnQ2 (out, b2, e, out);
    inductor #(.l(1u)) L1 (vcc, out);
    capacitor #(.c(1p), .ic(1)) C1 (vcc, out);
    capacitor #(.c(272.7p)) C2 (out, b1);
    capacitor #(.c(3n)) C3 (b1, gnd);
    resistor #(.r(10k)) R1 (b1, gnd);
    capacitor #(.c(3n)) C4 (b2, gnd);
    resistor #(.r(10k)) R2 (b2, gnd);
endmodule

```

I.3 Preferred Primitive, Parameter, and Port Names

The following table gives required names for primitives, parameters, and ports that are otherwise unnamed in SPICE. For connection by order instead of by name, the ports and

Primitive Name	Port Name	Parameter Name
resistor	p, n	r, tc1, tc2
capacitor	p, n	c, ic
inductor	p, n	l, ic
vsource	p, n	see section I.3.1
isource	sink, src	see section I.3.1
diode*	a, c	area
bjt*	c, b, e, s	area
mosfet*	d, g, s, b	w, l, ad, as, pd, ps, nrd, nrs
jfet*	d, g, s	area
mesfet*	d, g, s	area
vcvs	p, n, ps, ns	gain
vccs	sink, src, ps, ns	gm
tline	t1, b1, t2, b2	z0, td, f, nl

parameters must be given in the order listed. The discipline of the ports for these primitives shall be `electrical` and their descriptions shall be `inout`.

* The names diode, bjt, jfet, and mosfet are never used from within Verilog because these components require model. Thus the model name is used in Verilog, not the primitive name.

I.3.1 Independent Sources

The parameterization of independent source is more complicated than other components. The sources have several different modes, which are selected by the string parameter *type*. This parameter takes the name of a mode as its value. The parameters associated with each mode are given in the following table.

Mode	Name	Description
<i>all modes</i>	type	Waveform type, possible values are “dc”, “pulse”, “pwl”, “sine”, or “exp”.
	mag, phase	Small signal level and phase.
	“dc”	dc
	“pulse”	val0, val1
	delay	Start time of first pulse.
	rise, fall	Pulse rise and fall time.
	width	Pulse width.
	period	Pulse period.
“pwl”	wave	Vector of time/value pairs that define the waveform.
“sine”	dc	DC level of sinusoid.
	ampl	Amplitude of sinusoid.
	freq	Frequency of sinusoid.
	delay	Start time of first pulse.
	damp	Damping factor of sinusoid.
	sinephase	Phase of sinusoid.
	ammodindex	AM index of modulation.
	ammodfreq	AM modulation frequency.
	ammodphase	AM modulation phase.
	fmodindex	FM index of modulation.
	fmodfreq	FM modulation frequency.

Mode	Name	Description
"exp"	val0, val1	Equilibrium levels.
	td0, td1	Start time for transitions to val0, val1.
	tau0, tau1	Time constant for transition to val0, val1.
<i>all modes</i>	mag, phase	Small signal level and phase.

To specify source parameters by order, only the parameter for one waveshape can be given. The first parameter would be the type and the remaining parameters must be given in the order specified in the following table.

Mode	Parameter Order
<i>dc</i>	"dc", dc, mag, phase
<i>pulse</i>	"pulse", val0, val1, delay, rise, fall, width, period
<i>pwl</i>	"pwl", wave
<i>sine</i>	"sine", dc, ampl, freq, delay, damp
<i>exp</i>	"exp", val0, val1, td0, tau0, td1, tau1

I.3.2 Unsupported Components

Verilog does not support the concept of passing an instance name as a parameter. As such, the following components are not supported: ccvs, cccs, and mutual inductors; however, these primitives can be instantiated inside a subcircuit.

I.4 Other Issues

I.4.1 Multiplicity Factor on Subcircuits

Some SPICE simulators support a multiplicity factor or "M" factor parameter on subcircuits without the parameter being explicitly being declared. This factor is typically used to indicate that the subcircuit should be modeled as if there are a specified number of copies in parallel. If supported by the implementation, the automatic "M" factors would be supported for subcircuits defined in SPICE but not for subcircuits defined as a modules in Verilog. Thus in the above examples, if the SPICE subcircuit of the "Accessing Spice Subcircuits" section on page -3 were instantiated a multiplicity factor could be specified (assuming the simulator implementation supports multiplicity factors on SPICE subcircuits. However, one could not specify a multiplicity factor when

instantiating the equivalent Verilog module of the shown in "Accessing Spice Primitives" (section I.2.2.3).

I.4.2 Binning and Libraries

Some SPICE netlists provide mechanisms for mapping an instance to a group of models with the final determination of which model is used based on rules encapsulated in the SPICE netlist. Examples include model binning or corners support. From within an instance statement it appears as if the instance is referencing a simple model, thus supporting these capabilities in Verilog is provided by default.

Annex J

Glossary

Glossary of Terms

B

behavioral description

A mathematical mapping of inputs to outputs for a module, including intermediate variables and control flow.

behavioral model

A version of a module with a unique set of parameters designed to model a specific component.

block

A level within the behavioral description of a module, delimited by *begin* and *end*.

branch

A relationship between two nodes and their attached quantities within the behavioral description of a module. Each branch has two quantities, a value and a flow, with a reference direction for each.

C

component

A fundamental unit within a system that encapsulates behavior and/or structure (also known as an *element*). Modules and models might represent a single component, or a subcircuit with many components.

constitutive relationships

The essential relationships (expressions, statements) between the outputs of a module and its inputs and parameters that define the nature of the module. These relationships constitute a behavioral description.

control flow

The conditional and iterative statements controlling the behavior of a module. These statements evaluate arbitrary variables, (counters, flags, and tokens), to control the operation of different sections of a behavioral description.

child module

A module instantiated inside the behavioral description of another, “parent” module. You must have a complete definition of the child module somewhere. A child module is also known as submodule or instantiated module.

D**declaration**

A definition of the properties of a variable or a node.

dynamic attributes

The characteristics of an expression whose value is derived from the evaluation of a derivative (the *dot* function). Dynamic expressions define time-dependent module behavior. Some functions cannot operate on dynamic expressions.

E**element**

A fundamental unit within the system that encapsulates behavior and/or structure (also known as an *component*).

F**flow**

One of the two fundamental quantities used to simulate the behavior of a system. In electrical systems, flow is current.

G**global declarations**

Declarations of variables and parameters at the beginning of a behavioral description.

I**instance**

Any named occurrence of an element created from a module definition. One module definition can occur in multiple instances.

instantiation

The process of creating an instance from a module definition or simulator primitive, and defining the connectivity and parameters of that instance. (Placing the instance in the circuit or system.)

K**Kirchhoff's Laws**

Physical laws that define the interconnection relationships of nodes, branches, values, and flows. They specify a conservation of flow in and out of a node and a conservation of value around a loop of branches.

L**level**

One block within a behavioral description, delimited by a pair of matching keywords such as begin-end, discipline-enddiscipline.

M**model**

A named instance with a unique group of parameters specifying the behavior of one particular version of a module. You can use models to instantiate elements with parametric specifications different than those in the original module definition.

module

A definition of the interfaces and behavior of a component or a function.

N**NR method**

Newton-Raphson method. A generalized method for solving systems of nonlinear algebraic equations by breaking them into a series of many small linear operations ideally suited for computer processing.

node

A connection point in the system, with access functions for potential and/or flow through underlying discipline.

node declaration

The statement in a module definition, identifying the names of the nodes that are associated with the module ports or are local to the module. A node declaration also identifies the discipline of the node, which in turn identifies the access functions.

P**parameter**

A variable for characterizing the behavior of an instance of a module. Parameters are defined in the first section of a module, the module interface declarations, and can be specified each time a module is called in a netlist instance statement.

parameter declaration

The statement in a module definition, which defines the instance parameters of that module.

pin

An external connection point for a module (also known as a *terminal*).

potential

One of the two fundamental quantities used to simulate the behavior of a system.

primitive

A basic component that is defined entirely in terms of behavior, without reference to any other primitives. A primitive is the smallest and simplest possible portion of a simulated circuit or system.

probe

An artificial branch introduced into a circuit (or system) that does not alter its behavior, but lets the simulator to read out the potential or flow at that point.

R**reference direction**

A convention for determining whether the value of a node, the flow through a branch, the value across a branch, or the flow in or out of a terminal, is positive or negative.

reference node

The global node (which equals zero value) against which all node values are measured. The reference node is ground in an electrical system.

run time binding

The conditional introduction and removal of value and flow sources during a simulation. A value source can replace a flow source and vice versa. Binding a source to a specific node or branch prevents it from going into an unknown state.

S**scope**

The current nesting level of a block statement, which includes all lines of code within one set of braces in a module definition.

structural definitions

Instantiating modules inside other modules through the use of module definitions and declarations to create a hierarchical structure in the module's behavioral description.

T**terminal**

An external connection point for a module (also known as a *pin* or an *analog port*).

V**Verilog-A**

A subset of Verilog-AMS detailing in the analog version of Verilog HDL (see Annex C). This is a language for behavioral description of continuous-time systems that uses a syntax similar to Verilog HDL standard IEEE 1364-1995.

Verilog-AMS

Mixed-signal version of Verilog HDL. A language for behavioral description of continuous-time and discrete-time systems that is based on Verilog HDL standard IEEE 1364.

Index

Symbols

- !
 - logical negation operator 4-1, 4-6
- !=
 - logical inequality operator 4-1, 4-6
- \$dist_ functions F-2
- \$dist_chi_square F-2
- \$dist_erlang F-2
- \$dist_exponential F-2
- \$dist_normal F-2
- \$dist_poisson F-2
- \$dist_t F-2
- \$dist_uniform F-2
- \$fclose F-4
- \$finish F-3
- \$fopen F-4
- \$limexp 4-29
- \$random F-1
- \$realtime F-1
- \$stop F-3
- \$strobe
 - escape sequences F-5
 - format specifications F-5
- \$temperature F-1
- \$transition 4-20
- \$vt F-1
- \$vt(temp) F-1
- %
 - in format specifications F-5
 - modulus operator 4-1
- &
 - bit-wise AND operator 4-1
- &&
 - logical AND operator 4-1, 4-6
- *
 - arithmetic multiplication operator 4-1
- ”
 - in null expressions F-5
- /
 - arithmetic division operator 4-1
- <
 - relational less-than operator 4-1, 4-5
- <+
 - branch contribution operator 5-10
- <<
 - left shift operator 4-2, 4-7
- <=
 - relational less-than-or-equal operator 4-1, 4-5
- ==
 - logical equality operator 4-1, 4-6
- >
 - relational greater-than operator 4-1, 4-5
- >=
 - relational greater-than-or-equal operator 4-1, 4-5
- >>
 - right shift operator 4-2, 4-7
- ?:
 - conditional operator 4-2
- @ operator 6-9
- \
 - for escape sequences in strings F-5
- ^
 - bit-wise exclusive OR operator 4-1
- ^~
 - bit-wise equivalence operator 4-2
- `
 - in compiler directives G-1
- `default_nodetype G-1
- `define G-3, G-4
- `else G-6
- `endif G-6
- `ifdef G-6
- `include G-7
- `resetall G-8
- `timescale G-2
- `undef G-6
- |

- bit-wise inclusive OR operator 4-1
- ||
 - logical OR operator 4-1, 4-6
- ~
 - bit-wise negation operator 4-1
- ~^
 - bit-wise equivalence operator 4-2
- A**
 - absolute tolerance 4-14, 4-15, 4-24, A-4
 - abstol 3-8
 - AC Stimulus 4-32
 - Acceleration H-4
 - access 3-8
 - Access Functions 5-1, I-1
 - A-D converter 4-21
 - always procedural block 6-1
 - analog block 5-10
 - analog bus 3-14
 - analog operators 4-12
 - restrictions 4-12
 - analog procedural block 6-1
 - analysis dependent functions 4-30
 - analysis function 4-30
 - angle H-5
 - Angular_Acceleration H-5
 - Angular_Force H-5
 - Angular_Velocity H-5
 - arithmetic operators 4-1, 4-4
 - % 4-4
 - * 4-4
 - + 4-4
 - / 4-4
 - arrays
 - of integers 3-1
 - of time variables 3-1
 - associated reference directions 1-4
- B**
 - begin-end block statement 6-5
 - bidirectional port 8-9
 - binary operators 4-3
 - precedence 4-3
 - bit-wise operators 4-6–4-7
 - AND 4-1

- and 4-7
- equivalence 4-2
- exclusive nor 4-7
- exclusive OR 4-1
- exclusive or 4-7
- inclusive OR 4-1
- inclusive or 4-7
- negation 4-1
- unary negation 4-7
- block statement
 - naming of 6-3
- bound_step function 6-17
- branch contribution operator 5-10
- branch relations 5-11
- Branches 3-20
- branches 1-4
- built-in primitives 1-5
- C**
 - calltf routines 24-49
 - case statement 6-5
 - cbAfterDelay 24-42
 - cbAssign 24-41
 - cbAtStartOfSimTime 24-42
 - cbDeassign 24-41
 - cbDisable 24-41
 - cbEndOfCompile 24-43, 24-47
 - cbEndOfRestart 24-43, 24-47
 - cbEndOfSave 24-43, 24-47
 - cbEndOfSimulation 24-43, 24-47
 - cbEnterInteractive 24-43, 24-47
 - cbError 24-43, 24-47
 - cbExitInteractive 24-43, 24-47
 - cbForce 24-41
 - cbInteractiveScopeChange 24-43, 24-47
 - cbNextSimTime 24-42
 - cbPLIError 24-43, 24-47
 - cbReadOnlySynch 24-42
 - cbReadWriteSynch 24-42
 - cbRelease 24-41
 - cbStartOfRestart 24-43, 24-47
 - cbStartOfSave 24-43, 24-47
 - cbStartOfSimulation 24-43, 24-47
 - cbStmt 24-41, 24-46
 - cbTchkViolation 24-43, 24-47

- cbUnresolvedSystf 24-43, 24-47
- cbValueChange 24-41, 24-46
- Charge H-1
- circular integrator 4-15
- classes of PLI routines
 - calltf 24-49
 - compiletf 24-49
- comments 2-1
- compatibility rules
 - discrete domain rule 3-18
 - domain incompatibility rule 3-18
 - empty discipline rule 3-18
 - flow compatibility rule 3-18
 - nature compatibility rule 3-18
 - nature incompatibility rule 3-18
 - node connection rule 3-18
 - potential compatibility rule 3-17
 - self rule 3-17
 - units value rule 3-18
- Compiler directives 2-8
- compiletf routines 24-49
- concatenation
 - of names 8-15
- conditional compilation G-6
- conditional operator 4-2, 4-8
- conditional operator ?: 4-3
- conditional statement 6-4
- Connecting module ports by name 8-11
- Connecting module ports by ordered list 8-11
- connecting ports
 - by name 8-11
 - rules 8-12
- conservative branch 3-20
- conservative disciplines 3-12
- conservative nodes 3-12
- constant expression 4-1
- constitutive relationships 1-5, A-1
- contribution statements 6-2
- convergence A-3
- Correlated noise 4-33
- cross function 6-13
- Current H-1
- current H-2

D

- ddt operator 4-14
- ddt_nature 3-8
- decimal notation 2-3
- default
 - in case statement 6-6
- Defining a function 4-34
- defparam 3-4, 8-5–8-6
- defparam statement 8-5
- delay operator 4-17
- delays
 - inertial 24-38
 - pure transport 24-38
 - transport 24-38
- diagnostic messages
 - from \$stop and \$finish F-3
- discipline 3-11
- disciplines
 - conservative 3-12
 - empty 3-13
 - signal-flow 3-12
- discontinuity 6-15
- discrete-time finite difference approximation A-2
- Domain Binding 3-12
- driver_active function 7-22
- driver_count function 7-22
- driver_delay function 7-24
- driver_local function 7-23
- driver_next_state function 7-25
- driver_next_strength function 7-25
- driver_state function 7-23
- driver_strength function 7-24

E

- electrical H-2
- else statement 6-4
- embedding modules 8-1, 8-3
- empty disciplines 3-13
- end
 - sequential block 6-2
- endcase 6-6
- enddiscipline 3-11
- endfunction 4-34
- endmodule 8-2
- equality operators

- != 4-6
- == 4-6
- precedence 4-6
- escape sequences F-4, F-5
- escaped identifiers 2-5
- event
 - OR construct 6-10
- event or 4-2
- event or operator 4-8
- events
 - global 6-10
 - monitored 6-10
- exit simulator F-3
- exponentiation 4-10
- expression
 - evaluation order 4-4
- expressions 4-11
 - constant 4-1

F

- file inclusion G-7
- filters 4-12
- final_step 6-11
- finite-difference approximation A-2
- flicker_noise 4-33
- floating-point literals 2-4
- flow 1-6
- flow probe 5-3
- flow source 5-2
- Flux H-2
- for loop 6-8
- Force H-4
- format specifications F-5
 - ASCII character F-6
 - b or B F-5
 - binary F-5
 - c or C F-6
 - d or D F-5
 - decimal F-5
 - h or H F-5
 - hexadecimal F-5
 - hierarchical name F-6
 - m or M F-6
 - o or O F-5
 - octal F-5

- s or S F-6
- string F-6
- fullname 24-26
- function 4-34
- functions
 - call 4-36
 - definition 4-34
 - distribution F-2
 - probability F-2
 - returning a value 4-35

G

- generate statement C-5
- global events 6-10
- ground 1-4, 5-1

H

- handles
 - vpiHandle data type 22-2
- hierarchical path name 8-14
- hierarchy
 - level 8-15
 - name referencing 8-14, F-6
 - scope 8-15
 - scope rules for naming 8-16
 - top level names 8-15
- hyperbolic functions 4-10

I

- ideal opamp 5-13
- identifiers 2-5
 - escaped 2-5
 - keywords 2-6
- idt operator 4-14
- idt_nature 3-8
- idtmod 4-15
- if-else statement 6-4
 - omitting else from nested if 6-5
- implicit declarations G-1
- implicit equations 5-5
- implicit nodes 3-15
- Impulse H-4
- indirect branch assignement 5-13
- inertial delays 24-38
- initial procedural block 6-1

- initial_step 6-11
- inout port 8-9
- input port 8-9
- instantiation
 - of modules 8-1
- instantiation of modules 8-2
- integer 3-1
- integers
 - division 4-4
- interconnection relationships 1-5

J

- junction diode 5-6

K

- keywords 2-6
- kinematic H-4
- kinematic_v H-5
- Kirchhoff's Flow Law 1-5, A-1, A-4
- Kirchhoff's laws 1-5, A-1
- Kirchhoff's Value Law 1-5

L

- Laplace transform filters 4-23
- laplace_nd 4-25
- laplace_np 4-25
- laplace_zd 4-24
- laplace_zp 4-24
- last_crossing function 4-23
- left shift operator 4-2, 4-7
- lexical token
 - comment 2-1
 - definition of 2-1
 - number 2-2
 - operator 2-2
 - types 2-1
 - white space 2-1
- limited exponential 4-29
- logical operators 4-6
 - ! 4-6
 - && 4-6
 - || 4-6
 - AND 4-1
 - equality 4-1
 - inequality 4-1

- negation 4-1
- OR 4-1
- precedence 4-6
- looping statement
 - for loop 6-8
 - repeat loop 6-7
 - while loop 6-7

M

- M_1_PI H-7
- M_2_PI H-7
- M_2_SQRTPI H-7
- M_E H-7
- M_LN10 H-7
- M_LN2 H-7
- M_LOG10E H-7
- M_LOG2E H-7
- M_PI H-7
- M_PI_2 H-7
- M_PI_4 H-7
- M_SQRT1_2 H-7
- M_SQRT2 H-7
- M_TWO_PI H-7
- magnetic H-3
- Magneto_Motive_Force H-2
- mathematical function 4-9
- mathematical functions 4-9
- minus sign(-)
 - arithmetic subtraction operator 4-1
- module 8-1
 - definition 8-1
 - instance parameter value assignment 8-6
 - instantiation 8-2
 - overriding parameter values 8-5–8-8
 - parameter dependencies 8-8
 - port 8-3
 - terminal 8-4
 - top-level 8-2
- module parameter
 - dependencies 8-8
 - overriding values 8-5–8-8
- modulus operator 4-1
 - definition 4-4
- mtm_flag 24-12, 24-35
- multi-channel descriptor F-4

- multi-way decisions
 - case statement 6-5
 - if-else-if statement 6-5

N

- name 24-26
- named blocks
 - and scope 8-16
 - purpose 6-3
- names
 - of hierarchical paths 8-14
- new line character F-5
- Newton-Raphson method A-3
- nodal analysis A-1
- node 3-6
 - in hierarchical name tree 8-15
- nodes 1-6, 3-15
- noise 4-32
- noise_table 4-33
- null
 - expression F-5
- numbers 2-2

O

- operator
 - circular integrator 4-15
 - idtmod 4-15
- operators 4-1–4-8
 - 4-1
 - ! 4-1, 4-6
 - != 4-1, 4-6
 - % 4-1
 - & 4-1
 - && 4-1, 4-6
 - * 4-1
 - + 4-1
 - / 4-1
 - < 4-1, 4-5
 - << 4-2, 4-7
 - <= 4-1, 4-5
 - == 4-1, 4-6
 - > 4-1, 4-5
 - >= 4-1, 4-5
 - >> 4-2, 4-7
 - ?: 4-2

- ^ 4-1
- ^~ 4-2
- {{}} 4-1
- { } 4-1
- | 4-1
- || 4-1, 4-6
- ~ 4-1
- ~^ 4-2
- analog 4-12
- arithmetic 4-1, 4-4
- binary 2-2, 4-3
- bit-wise 4-6–4-7
- bit-wise AND 4-1
- bit-wise equivalence 4-2
- bit-wise exclusive OR 4-1
- bit-wise inclusive OR 4-1
- bit-wise negation 4-1
- concatenation 4-1
- conditional 2-2, 4-2, 4-8
- definition 2-2
- event or 4-2
- left shift 4-2
- left shift operator 4-7
- logical 4-6
- logical AND 4-1
- logical equality 4-1
- logical inequality 4-1
- logical negation 4-1
- logical OR 4-1
- modulus 4-1
- power 4-10
- relational 4-1, 4-5
- replication 4-1
- right shift 4-2
- right shift operator 4-7
- shift 4-7
- time derivative 4-14
- time integral 4-14
- unary 2-2
- output port 8-9
- overriding module parameter values 8-5–8-8
 - by name 8-7
 - defparam 8-5

P

- P_C H-7
- P_CELSIUS0 H-7
- P_EPS0 H-7
- P_H H-7
- P_K H-7
- P_Q H-7
- P_U0 H-7
- parameter
 - module type 3-2
- parameter assignment by name 8-5
- parameter assignment by order 8-5
- parentheses
 - and changing operator precedence 4-4
- plus sign(+)
 - arithmetic addition operator 4-1
- port 8-8–8-14
 - connecting by name 8-11
 - declaration 8-9
 - definition 8-8
 - module 8-3
 - rules for connecting 8-12
- port access function 4-11
- Port Branches 5-6
- Position H-3
- potential probe 5-3
- potential source 5-2
- pow operator 4-10
- precedence
 - binary operators 4-3
 - equality operators 4-6
 - logical operators 4-6
 - relational operators 4-5
- primitives J-4
- probabilistic distribution functions F-3
 - \$dist_chi_square F-2
 - \$dist_erlang F-2
 - \$dist_exponential F-2
 - \$dist_normal F-2
 - \$dist_poisson F-2
 - \$dist_t F-2
 - \$dist_uniform F-2
 - gaussian distribution F-3
- probe 5-3

- Probes 5-3, I-2
- pulse control 24-12, 24-35
- pulsere_flag 24-12, 24-35
- pure transport delays 24-38

Q

- QAM modulator 4-21
- quantities A-4

R

- real numbers 3-1–3-2
 - format specifications used with F-6
 - operators with real number operands 4-2
- reference direction 1-4
- reference node 1-4, 5-1
- relational operators 4-1, 4-5
 - < 4-5
 - <= 4-5
 - > 4-5
 - >= 4-5
 - precedence 4-5
- relative tolerance A-4
- repeat loop 6-7
- right shift operator 4-2, 4-7
- rotational H-6
- rotational_omega H-6

S

- s
 - in string display format F-6
- s_cb_data structure 24-8, 24-40, 24-45
- s_vpi_delay structure 24-11
- s_vpi_error_info structure 24-2
- s_vpi_strengthval structure 24-18
- s_vpi_systf_data structure 24-15, 24-48
- s_vpi_time structure 24-11, 24-16, 24-39
- s_vpi_value structure 24-18, 24-39
- s_vpi_vecval structure 24-18
- s_vpi_vlog_info structure 24-22
- scalar node 3-14
- scientific notation 2-3
- scope
 - and hierarchical names 8-15
 - rules 8-16
- seed F-2

- shift operators 4-7
 - << 4-7
 - >> 4-7
- signal access functions 4-11
- signal transitions 4-18
- signal-flow branch 3-20
- signal-flow disciplines 3-12
- signal-flow nodes 3-12
- sinusoidal voltage source 6-17
- slew filter 4-22
- slope 4-22
- source branch 5-2
- Sources 5-2
- standard mathematical functions 4-9
- standard output F-4
- stochastic analysis F-3
 - probabilistic distribution functions F-3
- stop F-3
- strings
 - display format F-6
- switch branch 5-2
- system tasks
 - for interrupting the simulator F-3
- System tasks and functions 2-7

T

- Temperature H-3
- terminals 1-4
- text macro substitutions G-4–G-6
 - and `define G-4
 - definition G-4
 - redefinition G-6
 - with arguments G-4
- text output
 - vpi_mcd_close() 24-29
 - vpi_mcd_name() 24-30
 - vpi_mcd_open() 24-31
 - vpi_mcd_printf() 24-32
 - vpi_printf() 24-33
- thermal H-3
- time derivative operator 4-14, A-2
- time integral operator 4-14
- time precision G-2
- time unit G-2
- timer function 6-15

- Tolerances 4-13
- top-level module 8-2
- transient analysis A-2
- transition 4-18
- transition filter 4-18
- transition function 4-20
- transport delays 24-38
- tree structure
 - of hierarchical names 8-14
- trigonometric functions 4-10
- type specification
 - parameter 3-4

U

- unary operators
 - ! 4-6
 - << 4-7
 - >> 4-7
- underscore character 2-3
- units 3-8
- User Defined Attributes 3-10
- User defined functions 4-34

V

- value 1-4
- value range specification
 - parameter 3-5
- vector branch 3-20
- vector node 3-14
- Velocity H-4
- vlog_startup_routines array 24-49
- Voltage H-2
- VPI object diagrams
 - assignments 22-32
 - case statement 22-34
 - continuous assignments 22-27
 - delay controls 22-32
 - event controls 22-32
 - expressions 22-28, 22-29, 22-30
 - for loops 22-33
 - forever loops 22-33
 - function calls 22-26
 - functions 22-14
 - if statement 22-34
 - inter-module paths 22-25

- IO declarations 22-14
- memories 22-21
- module paths 22-25
- modules 22-11, 22-12
- named events 22-20, 22-31
- nets 22-18
- parameters 22-22
- ports 22-15, 22-16, 22-17
- primitives 22-23
- procedural assign statement 22-35
- procedural blocks 22-31
- procedural deassign statement 22-35
- procedural disable statement 22-35
- procedural force statement 22-35
- procedural release statement 22-35
- processes 22-31
- regs 22-19
- repeat controls 22-32
- repeat loops 22-33
- scopes 22-14
- specparams 22-22
- statements 22-31
- task calls 22-26
- tasks 22-14
- timing checks 22-25
- UDPs 22-24
- variables 22-20
- wait control 22-33
- while loops 22-33
- VPI routines
 - callback overview 22-1
 - error handling 22-2
 - key to object diagrams 22-7
 - object access overview 22-2
 - object classifications 22-2
- vpi_chk_error() 24-2
- vpi_compare_objects() 24-3
- vpi_free_object() 24-5
- vpi_get() 24-7
- vpi_get_cb_info() 24-8
- vpi_get_str() 24-14
- vpi_get_systf_info() 24-15
- vpi_get_time() 24-16
- vpi_get_value() 24-17
- vpi_get_vlog_info() 24-22
- vpi_handle() 24-24
- vpi_handle_by_index() 24-25
- vpi_handle_by_name() 24-26
- vpi_handle_multi() 24-27
- vpi_iterate() 24-28
- vpi_mcd_close() 24-29
- vpi_mcd_name() 24-30
- vpi_mcd_open() 24-31
- vpi_mcd_printf() 24-32
- vpi_printf() 24-33
- vpi_put_delays() 24-34
- vpi_put_value() 24-38
- vpi_register_cb() 24-40, 24-45
- vpi_register_systf() 24-45
- vpi_remove_cb() 24-51
- vpi_scan() 24-52
- vpiCancelEvent 24-38
- vpiForceFlag 24-38
- vpiHandle 22-2
- vpiInertialDelay 24-38
- vpiInterModPath 24-27
- vpiIntFunc 24-49
- vpiIterator 24-28
- vpiNoDelay 24-38
- vpiPureTransportDelay 24-38
- vpiRealFunc 24-49
- vpiReleaseFlag 24-38
- vpiReturnEvent 24-38
- vpiScaledRealTime 24-39
- vpiSchedEvent 24-38
- vpiScheduled 24-38
- vpiSizedFunc 24-49
- vpiSysFunction 24-48, 24-49
- vpiSysTask 24-48
- vpiTimeFunc 24-49
- vpiTimeUnit 24-7
- vpiTransportDelay 24-38

W

- Watts H-3
- while loop 6-7
- white space 2-1
- white_noise 4-32

Z

zi_nd 4-28

zi_np 4-28

zi_zd 4-27

zi_zp 4-27

Z-transform filters 4-26