



IEEE Standard SystemC[®] Language Reference Manual

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

1666TM

IEEE
3 Park Avenue
New York, NY 10016-5997, USA
31 March 2006

IEEE Std 1666TM-2005

*Recognized as an
American National Standard (ANSI)*

IEEE Std 1666™-2005

IEEE Standard SystemC® Language Reference Manual

Sponsor

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

Approved 28 March 2006

American National Standards Institute

Approved 6 December 2005

IEEE-SA Standards Board

Grateful acknowledgment is made to Open SystemC Initiative for the permission to use the following source material:
SystemC® Language Reference Manual Version 2.1

Abstract: SystemC® is defined in this standard. SystemC is an ANSI standard C++ class library for system and hardware design for use by designers and architects who need to address complex systems that are a hybrid between hardware and software. This standard provides a precise and complete definition of the SystemC class library so that a SystemC implementation can be developed with reference to this standard alone. The primary audiences for this standard are the implementors of the SystemC class library, the implementors of tools supporting the class library, and users of the class library.

Keywords: C++, computer languages, digital systems, discrete event simulation, electronic design automation, electronic systems, electronic system level, embedded software, fixed-point, hardware description language, hardware design, hardware verification, SystemC, system modeling, system-on-chip, transaction level

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Copyright © 2006 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 31 March 2006. Printed in the United States of America.

IEEE is a registered trademark in the U.S. Patent & Trademark Office, owned by the Institute of Electrical and Electronics Engineers, Incorporated.

SystemC® is a registered trademark of Open SystemC Initiative.

Print: ISBN 0-7381-4871-7 SH95505
PDF: ISBN 0-7381-4870-9 SS95505

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an IEEE Standard is wholly voluntary. The IEEE disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other IEEE Standard document.

The IEEE does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEEE Standards documents are supplied “**AS IS.**”

The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

In publishing and making this document available, the IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position, explanation, or interpretation of the IEEE.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
Piscataway, NJ 08854
USA

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

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Introduction

This introduction is not part of IEEE Std 1666-2005, IEEE Standard SystemC[®] Language Reference Manual.

This document defines SystemC, which is a C++ class library.

As the electronics industry builds more complex systems involving large numbers of components including software, there is an increasing need for a modeling language that can manage the complexity and size of these systems. SystemC provides a mechanism for managing this complexity with its facility for modeling hardware and software together at multiple levels of abstraction. This capability is not available in traditional hardware description languages.

Stakeholders in SystemC include Electronic Design Automation (EDA) companies who implement SystemC class libraries and tools, Integrated Circuit (IC) suppliers who extend those class libraries and use SystemC to model their intellectual property, and end users who use SystemC to model their systems.

Before the publication of this standard, SystemC was defined by an open source proof-of-concept C++ library, also known as *the reference simulator*, available from the Open SystemC Initiative (OSCI). In the event of discrepancies between the behavior of the reference simulator and statements made in this standard, this standard shall be taken to be definitive.

This standard is not intended to serve as a users' guide or to provide an introduction to SystemC. Readers requiring a SystemC tutorial or information on the intended use of SystemC should consult the OSCI Web site (www.systemc.org) to locate the many books and training classes available.

Notice to users

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: <http://standards.ieee.org/reading/ieee/updates/errata/index.html>. Users are encouraged to check this URL for errata periodically.

Interpretations

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

Patents

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

Participants

This entity-based standard was created under the leadership of the following individuals:

Victor Berman, *Chair*
Stuart Swan, *Technical Working Group Chair*
Oz Levia, *Vice-Chair*
John Aynsley, *Co-Author (Core)*
David Long, *Co-Author (Data Types)*
Sofie Vandeputte, *Typographical Editor*
Dennis Brophy, *Secretary*

At the time this entity-based standard was completed, the LRM Working Group had the following membership:

Cadence Design Systems	CoWare	Mentor Graphics
Calypto	Jeda Technologies	Synopsys
	JEITA	

The following members of the entity-based balloting committee voted on this standard. Balloting entities may have voted for approval, disapproval, or abstention.

Cadence Design Systems	Jeda Technologies	Sony
Calypto	JEITA	Sunburst Design
CoWare	Mentor Graphics	Synopsys

The working group gratefully acknowledges the contributions of the following participants.

El Mustapha Aboulhamid	Skip Hovsmith	César Quiroz
Mike Baird	Hiroshi Imai	Adam Rose
Bishnupriya Bhattacharya	Masaharu Imai	Ray Ryan
David C Black	Martin Janssen	Kurt Schwartz
Dundar Dumlogal	Masaru Kakimoto	Yasuyuki Shimizu
Abhijit Ghosh	Masamichi Kawarabayashi	Minoru Shoji
Mark Glasser	Kevin Kranen	Bob Shur
Andy Goodrich	Evan Lavelle	Kazuyoshi Takemura
Serge Goossens	Mike Meredith	Yasutaka Tsukamoto
Kazunori Goto	Kazuya Morii	Vincent Viteau
Robert Graulich	Wolfgang Mueller	Masashi Watanabe
Thorsten Groetker	Kazuhide Nakamura	Akihisa Yamada
Takashi Hasegawa	Junji Nakano	Eugene Zhang
	Takashi Ohsaka	

When the IEEE-SA Standards Board approved this standard on 6 December 2005, it had the following membership:

Steve M. Mills, *Chair*
Richard H. Hulett, *Vice Chair*
Don Wright, *Past Chair*
Judith Gorman, *Secretary*

Mark D. Bowman
Dennis B. Brophy
Joseph Bruder
Richard Cox
Bob Davis
Julian Forster*
Joanna N. Guenin
Mark S. Halpin
Raymond Hapeman

William B. Hopf
Lowell G. Johnson
Herman Koch
Joseph L. Koepfinger*
David J. Law
Daleep C. Mohla
Paul Nikolic

T. W. Olsen
Glenn Parsons
Ronald C. Petersen
Gary S. Robinson
Frank Stone
Malcolm V. Thaden
Richard L. Townsend
Joe D. Watson
Howard L. Wolfman

*Member Emeritus

Also included are the following nonvoting IEEE-SA Standards Board liaisons:

Satish K. Aggarwal, *NRC Representative*
Richard DeBlasio, *DOE Representative*
Alan H. Cookson, *NIST Representative*

Jennie Steinhagen
IEEE Standards Project Editor

CONTENTS

1.	Overview.....	1
1.1	Scope.....	1
1.2	Purpose.....	1
1.3	Subsets	1
1.4	Relationship with C++	1
1.5	Guidance for readers	2
2.	References.....	3
3.	Terminology and conventions used in this standard.....	4
3.1	Terminology.....	4
3.1.1	Shall, should, may, can.....	4
3.1.2	Implementation, application.....	4
3.1.3	Call, called from, derived from	4
3.1.4	Specific technical terms	4
3.2	Syntactical conventions	6
3.2.1	Implementation-defined.....	6
3.2.2	Disabled	6
3.2.3	Ellipsis (...).....	6
3.2.4	Class names.....	6
3.2.5	Embolded text	7
3.3	Semantic conventions	7
3.3.1	Class definitions and the inheritance hierarchy	7
3.3.2	Function definitions and side-effects	7
3.3.3	Functions whose return type is a reference or a pointer.....	7
3.3.4	Namespaces and internal naming.....	9
3.3.5	Non-compliant applications and errors	9
3.4	Notes and examples	10
4.	Elaboration and simulation semantics	11
4.1	Elaboration.....	11
4.1.1	Instantiation	11
4.1.2	Process macros.....	13
4.1.3	Port binding and export binding	13
4.1.4	Setting the time resolution	14
4.2	Simulation.....	14
4.2.1	The scheduling algorithm	15
4.2.2	Cycles in the scheduling algorithm.....	17
4.3	Running elaboration and simulation	18
4.3.1	Function declarations.....	18
4.3.2	Function <code>sc_elab_and_sim</code>	18
4.3.3	Functions <code>sc_argc</code> and <code>sc_argv</code>	19
4.3.4	Running under application control using functions <code>sc_main</code> and <code>sc_start</code>	19
4.3.5	Running under control of the kernel	20
4.4	Elaboration and simulation callbacks	20
4.4.1	<code>before_end_of_elaboration</code>	21
4.4.2	<code>end_of_elaboration</code>	22
4.4.3	<code>start_of_simulation</code>	23

4.4.4	end_of_simulation	23
4.5	Other functions related to the scheduler	24
4.5.1	Function declarations	24
4.5.2	Function sc_stop, sc_set_stop_mode, and sc_get_stop_mode	24
4.5.3	Function sc_time_stamp	25
4.5.4	Function sc_delta_count	26
4.5.5	Function sc_is_running	26
5.	Core language class definitions	27
5.1	Class header files	27
5.1.1	#include "systemc"	27
5.1.2	#include "systemc.h"	27
5.2	sc_module	29
5.2.1	Description	29
5.2.2	Class definition	29
5.2.3	Constraints on usage	31
5.2.4	kind	31
5.2.5	SC_MODULE	31
5.2.6	Constructors	32
5.2.7	SC_CTOR	32
5.2.8	SC_HAS_PROCESS	33
5.2.9	SC_METHOD, SC_THREAD, SC_CTHREAD	33
5.2.10	Method process	34
5.2.11	Thread and clocked thread processes	35
5.2.12	Clocked thread processes and reset_signal_is	36
5.2.13	sensitive	37
5.2.14	dont_initialize	38
5.2.15	set_stack_size	39
5.2.16	next_trigger	39
5.2.17	wait	41
5.2.18	Positional port binding	43
5.2.19	before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation	44
5.2.20	get_child_objects	45
5.2.21	sc_gen_unique_name	45
5.2.22	sc_behavior and sc_channel	46
5.3	sc_module_name	47
5.3.1	Description	47
5.3.2	Class definition	47
5.3.3	Constraints on usage	47
5.3.4	Module hierarchy	48
5.3.5	Member functions	48
5.4	sc_sensitive	50
5.4.1	Description	50
5.4.2	Class definition	50
5.4.3	Constraints on usage	50
5.4.4	operator<<	50
5.5	sc_spawn_options and sc_spawn	52
5.5.1	Description	52
5.5.2	Class definition	52
5.5.3	Constraints on usage	53
5.5.4	Constructors	53
5.5.5	Member functions	53

5.5.6	sc_spawn	54
5.5.7	SC_FORK and SC_JOIN	56
5.6	sc_process_handle	58
5.6.1	Description	58
5.6.2	Class definition.....	58
5.6.3	Constraints on usage	59
5.6.4	Constructors	59
5.6.5	Member functions	59
5.6.6	sc_get_current_process_handle.....	61
5.7	sc_event_finder and sc_event_finder_t	62
5.7.1	Description	62
5.7.2	Class definition.....	62
5.7.3	Constraints on usage.....	62
5.8	sc_event_and_list [†] and sc_event_or_list [†]	65
5.8.1	Description	65
5.8.2	Class definition.....	65
5.8.3	Constraints on usage.....	65
5.8.4	Event lists	65
5.9	sc_event	66
5.9.1	Description	66
5.9.2	Class definition.....	66
5.9.3	Constraints on usage.....	66
5.9.4	notify and cancel	66
5.9.5	Event lists	67
5.9.6	Multiple event notifications	67
5.10	sc_time.....	68
5.10.1	Description	68
5.10.2	Class definition.....	68
5.10.3	Time resolution	69
5.10.4	Functions and operators	69
5.10.5	SC_ZERO_TIME.....	69
5.11	sc_port.....	71
5.11.1	Description	71
5.11.2	Class definition	71
5.11.3	Template parameters	72
5.11.4	Constraints on usage	73
5.11.5	Constructors	74
5.11.6	kind	74
5.11.7	Named port binding	74
5.11.8	Member functions for bound ports and port-to-port binding.....	75
5.11.9	before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation.....	79
5.12	sc_export.....	80
5.12.1	Description	80
5.12.2	Class definition.....	80
5.12.3	Template parameters	81
5.12.4	Constraints on usage.....	81
5.12.5	Constructors	81
5.12.6	kind.....	81
5.12.7	Export binding.....	82
5.12.8	Member functions for bound exports and export-to-export binding.....	83
5.12.9	before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation.....	84

5.13	sc_interface	85
5.13.1	Description	85
5.13.2	Class definition.....	85
5.13.3	Constraints on usage.....	85
5.13.4	register_port	86
5.13.5	default_event	86
5.14	sc_prim_channel	88
5.14.1	Description	88
5.14.2	Class definition.....	88
5.14.3	Constraints on usage	89
5.14.4	Constructors	89
5.14.5	kind.....	89
5.14.6	request_update and update	89
5.14.7	next_trigger and wait.....	90
5.14.8	before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation.....	90
5.15	sc_object	92
5.15.1	Description	92
5.15.2	Class definition.....	92
5.15.3	Constraints on usage.....	93
5.15.4	Constructors and hierarchical names.....	93
5.15.5	name, basename, and kind.....	94
5.15.6	print and dump	95
5.15.7	Functions for object hierarchy traversal.....	95
5.15.8	Member functions for attributes.....	97
5.16	sc_attr_base.....	99
5.16.1	Description	99
5.16.2	Class definition	99
5.16.3	Member functions	99
5.17	sc_attribute.....	100
5.17.1	Description	100
5.17.2	Class definition	100
5.17.3	Template parameters.....	100
5.17.4	Member functions and data members.....	100
5.18	sc_attr_cltn.....	101
5.18.1	Description	101
5.18.2	Class definition.....	101
5.18.3	Constraints on usage.....	101
5.18.4	Iterators.....	101
6.	Predefined channel class definitions.....	103
6.1	sc_signal_in_if.....	103
6.1.1	Description	103
6.1.2	Class definition.....	103
6.1.3	Member functions	103
6.2	sc_signal_in_if<bool> and sc_signal_in_if<sc_dt::sc_logic>.....	104
6.2.1	Description	104
6.2.2	Class definition.....	104
6.2.3	Member functions	105
6.3	sc_signal_inout_if.....	106
6.3.1	Description	106
6.3.2	Class definition.....	106
6.3.3	write.....	106

6.4	sc_signal	107
6.4.1	Description	107
6.4.2	Class definition	107
6.4.3	Template parameter T	108
6.4.4	Reading and writing signals	108
6.4.5	Constructors	109
6.4.6	register_port	109
6.4.7	Member functions for reading	109
6.4.8	Member functions for writing	110
6.4.9	Member functions for events	110
6.4.10	Diagnostic member functions	110
6.4.11	operator<<	111
6.5	sc_signal<bool> and sc_signal<sc_dt::sc_logic>	113
6.5.1	Description	113
6.5.2	Class definition	113
6.5.3	Member functions	114
6.6	sc_buffer	116
6.6.1	Description	116
6.6.2	Class definition	116
6.6.3	Constructors	116
6.6.4	Member functions	117
6.7	sc_clock	119
6.7.1	Description	119
6.7.2	Class definition	119
6.7.3	Characteristic properties	120
6.7.4	Constructors	120
6.7.5	write	120
6.7.6	Diagnostic member functions	120
6.7.7	before_end_of_elaboration	121
6.7.8	sc_in_clk	121
6.8	sc_in	122
6.8.1	Description	122
6.8.2	Class definition	122
6.8.3	Member functions	123
6.8.4	Function sc_trace	123
6.8.5	end_of_elaboration	123
6.9	sc_in<bool> and sc_in<sc_dt::sc_logic>	124
6.9.1	Description	124
6.9.2	Class definition	124
6.9.3	Member functions	126
6.10	sc_inout	127
6.10.1	Description	127
6.10.2	Class definition	127
6.10.3	Member functions	128
6.10.4	initialize	128
6.10.5	Function sc_trace	128
6.10.6	end_of_elaboration	129
6.10.7	Binding	129
6.11	sc_inout<bool> and sc_inout<sc_dt::sc_logic>	130
6.11.1	Description	130
6.11.2	Class definition	130
6.11.3	Member functions	132
6.12	sc_out	133
6.12.1	Description	133

6.12.2	Class definition.....	133
6.12.3	Member functions	133
6.13	sc_signal_resolved	134
6.13.1	Description	134
6.13.2	Class definition	134
6.13.3	Constructors	134
6.13.4	Resolution semantics	135
6.13.5	Member functions	136
6.14	sc_in_resolved	137
6.14.1	Description	137
6.14.2	Class definition	137
6.14.3	Member functions	137
6.15	sc_inout_resolved	138
6.15.1	Description	138
6.15.2	Class definition.....	138
6.15.3	Member functions	138
6.16	sc_out_resolved	139
6.16.1	Description	139
6.16.2	Class definition.....	139
6.16.3	Member functions.....	139
6.17	sc_signal_rv	140
6.17.1	Description	140
6.17.2	Class definition	140
6.17.3	Semantics and member functions	140
6.18	sc_in_rv.....	142
6.18.1	Description	142
6.18.2	Class definition	142
6.18.3	Member functions	142
6.19	sc_inout_rv.....	143
6.19.1	Description	143
6.19.2	Class definition.....	143
6.19.3	Member functions	143
6.20	sc_out_rv.....	145
6.20.1	Description	145
6.20.2	Class definition.....	145
6.20.3	Member functions	145
6.21	sc_fifo_in_if.....	146
6.21.1	Description	146
6.21.2	Class definition.....	146
6.21.3	Member functions	146
6.22	sc_fifo_out_if.....	148
6.22.1	Description	148
6.22.2	Class definition	148
6.22.3	Member functions	148
6.23	sc_fifo	150
6.23.1	Description	150
6.23.2	Class definition.....	150
6.23.3	Template parameter T	151
6.23.4	Constructors	151
6.23.5	register_port	152
6.23.6	Member functions for reading.....	152
6.23.7	Member functions for writing	152
6.23.8	The update phase.....	153
6.23.9	Member functions for events.....	153

6.23.10	Member functions for available values and free slots	153
6.23.11	Diagnostic member functions.....	154
6.23.12	operator<<.....	154
6.24	sc_fifo_in	156
6.24.1	Description	156
6.24.2	Class definition.....	156
6.24.3	Member functions	156
6.25	sc_fifo_out	157
6.25.1	Description	157
6.25.2	Class definition.....	157
6.25.3	Member functions	157
6.26	sc_mutex_if.....	160
6.26.1	Description.....	160
6.26.2	Class definition	160
6.26.3	Member functions.....	160
6.27	sc_mutex	161
6.27.1	Description	161
6.27.2	Class definition	161
6.27.3	Constructors	161
6.27.4	Member functions	162
6.28	sc_semaphore_if	163
6.28.1	Description	163
6.28.2	Class definition	163
6.28.3	Member functions	163
6.29	sc_semaphore.....	164
6.29.1	Description	164
6.29.2	Class definition	164
6.29.3	Constructors	164
6.29.4	Member functions	165
6.30	sc_event_queue	166
6.30.1	Description	166
6.30.2	Class definition.....	166
6.30.3	Constraints on usage.....	166
6.30.4	Constructors	166
6.30.5	kind.....	167
6.30.6	Member functions	167
7.	Data types	169
7.1	Introduction.....	169
7.2	Common characteristics.....	172
7.2.1	Initialization and assignment operators.....	172
7.2.2	Precision of arithmetic expressions.....	173
7.2.3	Base class default word length	174
7.2.4	Word length.....	175
7.2.5	Bit-select.....	175
7.2.6	Part-select	175
7.2.7	Concatenation.....	176
7.2.8	Reduction operators.....	178
7.2.9	Integer conversion	178
7.2.10	String input and output.....	178
7.2.11	Conversion of application-defined types in integer expressions.....	179
7.3	String literals.....	180
7.4	sc_value_base†	182

7.4.1	Description	182
7.5	Limited-precision integer types	184
7.5.1	Type definitions	184
7.5.2	sc_int_base	185
7.5.3	sc_uint_base	190
7.5.4	sc_int	194
7.5.5	sc_uint	197
7.5.6	Bit-selects	199
7.5.7	Part-selects	203
7.6	Finite-precision integer types	209
7.6.1	Type definitions	209
7.6.2	Constraints on usage	209
7.6.3	sc_signed	209
7.6.4	sc_unsigned	216
7.6.5	sc_bigint	223
7.6.6	sc_biguint	225
7.6.7	Bit-selects	227
7.6.8	Part-selects	231
7.7	Integer concatenations	236
7.7.1	Description	236
7.7.2	Class definition	236
7.7.3	Constraints on usage	237
7.7.4	Assignment operators	238
7.7.5	Implicit type conversion	238
7.7.6	Explicit type conversion	238
7.7.7	Other member functions	238
7.8	Generic base proxy class	239
7.8.1	Description	239
7.8.2	Class definition	239
7.8.3	Constraints on usage	239
7.9	Logic and vector types	240
7.9.1	Type definitions	240
7.9.2	sc_logic	240
7.9.3	sc_bv_base	244
7.9.4	sc_lv_base	251
7.9.5	sc_bv	257
7.9.6	sc_lv	259
7.9.7	Bit-selects	261
7.9.8	Part-selects	264
7.9.9	Concatenations	270
7.10	Fixed-point types	278
7.10.1	Fixed-point representation	278
7.10.2	Fixed-point type conversion	279
7.10.3	Fixed-point data types	279
7.10.4	Fixed-point expressions and operations	281
7.10.5	Bit and part selection	284
7.10.6	Variable-precision fixed-point value limits	284
7.10.7	Fixed-point word length and mode	284
7.10.8	Conversions to character string	287
7.10.9	Finite word-length effects	289
7.10.10	sc_fxnum	313
7.10.11	sc_fxnum_fast	317
7.10.12	sc_fxval	322
7.10.13	sc_fxval_fast	326

7.10.14	sc_fix	331
7.10.15	sc_ufix	334
7.10.16	sc_fix_fast.....	337
7.10.17	sc_ufix_fast.....	340
7.10.18	sc_fixed.....	343
7.10.19	sc_ufixed.....	345
7.10.20	sc_fixed_fast.....	347
7.10.21	sc_ufixed_fast.....	350
7.10.22	Bit-selects	352
7.10.23	Part-selects.....	354
7.11	Contexts	361
7.11.1	sc_length_param.....	361
7.11.2	sc_length_context.....	362
7.11.3	sc_fxtype_params.....	363
7.11.4	sc_fxtype_context.....	366
7.11.5	sc_fxcast_switch.....	367
7.11.6	sc_fxcast_context	368
7.12	Control of string representation	370
7.12.1	Description	370
7.12.2	Class definition	370
7.12.3	Functions.....	370
8.	Utility class definitions	371
8.1	Trace files	371
8.1.1	Class definition and function declarations	371
8.1.2	sc_trace_file	371
8.1.3	sc_create_vcd_trace_file	372
8.1.4	sc_close_vcd_trace_file	372
8.1.5	sc_write_comment	372
8.1.6	sc_trace.....	372
8.2	sc_report.....	375
8.2.1	Description	375
8.2.2	Class definition	375
8.2.3	Constraints on usage	375
8.2.4	sc_severity.....	376
8.2.5	Copy constructor and assignment	376
8.2.6	Member functions	376
8.3	sc_report_handler.....	378
8.3.1	Description	378
8.3.2	Class definition.....	378
8.3.3	Constraints on usage.....	380
8.3.4	sc_actions	380
8.3.5	report	380
8.3.6	set_actions	381
8.3.7	stop_after.....	381
8.3.8	get_count	382
8.3.9	suppress and force	383
8.3.10	set_handler	383
8.3.11	get_new_action_id	384
8.3.12	sc_interrupt_here and sc_stop_here	384
8.3.13	get_cached_report and clear_cached_report	384
8.3.14	set_log_file_name and get_log_file_name.....	385
8.4	sc_exception.....	386

8.4.1	Description	386
8.4.2	Definition	386
8.5	Utility functions	387
8.5.1	Function declarations	387
8.5.2	sc_abs	387
8.5.3	sc_max	387
8.5.4	sc_min	387
8.5.5	sc_copyright	387
8.5.6	sc_version	388
8.5.7	sc_release	388
Annex A (informative) Introduction to SystemC		389
Annex B (informative) Glossary		393
Annex C (informative) Deprecated features		403
Annex D (informative) Changes between the different SystemC versions		405
Index		407

IEEE Standard SystemC[®] Language Reference Manual

1. Overview

1.1 Scope

This standard defines SystemC[®]¹ as an ANSI standard C++ class library for system and hardware design.

1.2 Purpose

The general purpose of SystemC is to provide a C++-based standard for designers and architects who need to address complex systems that are a hybrid between hardware and software.

The specific purpose of this standard is to provide a precise and complete definition of the SystemC class library so that a SystemC implementation can be developed with reference to this standard alone. This standard is not intended to serve as a users' guide or to provide an introduction to SystemC, but does contain useful information for end users.

1.3 Subsets

It is anticipated that tool vendors will create implementations that support only a subset of this standard or that impose further constraints on the use of this standard. Such implementations are not fully compliant with this standard but may nevertheless claim partial compliance with this standard and may use the name SystemC.

1.4 Relationship with C++

This standard is closely related to the C++ programming language and adheres to the terminology used in ISO/IEC 14882:2003. This standard does not seek to restrict the usage of the C++ programming language; a SystemC application may use any of the facilities provided by C++, which in turn may use any of the facilities provided by C. However, where the facilities provided by this standard are used, they shall be used in accordance with the rules and constraints set out in this standard.

This standard defines the public interface to the SystemC class library and the constraints on how those classes may be used. The SystemC class library may be implemented in any manner whatsoever, provided only that the obligations imposed by this standard are honored.

A C++ class library may be extended using the mechanisms provided by the C++ language. Implementors and users are free to extend SystemC in this way, provided that they do not violate this standard.

¹SystemC[®] is a registered trademark of Open SystemC Initiative.

NOTE—It is possible to create a well-formed C++ program that is legal according to the C++ programming language standard but that violates this standard. An implementation is not obliged to detect every violation of this standard.²

1.5 Guidance for readers

Readers who are not entirely familiar with SystemC should start with Annex A, “Introduction to SystemC,” which provides a brief informal summary of the subject intended to aid in the understanding of the normative definitions. Such readers may also find it helpful to scan the examples embedded in the normative definitions and to see Annex B, “Glossary.”

Readers should pay close attention to Clause 3, “Terminology and conventions used in this standard.” An understanding of the terminology defined in Clause 3 is necessary for a precise interpretation of this standard.

Clause 4, “Elaboration and simulation semantics,” defines the behavior of the SystemC kernel and is central to an understanding of SystemC. The semantic definitions given in the subsequent clauses detailing the individual classes are built upon the foundations laid in Clause 4.

The clauses from Clause 5 onward define the public interface to the SystemC class library. The following information is listed for each class:

- a) A C++ source code listing of the class definition
- b) A statement of any constraints on the use of the class and its members
- c) A statement of the semantics of the class and its members
- d) For certain classes, a description of functions, typedefs, and macros associated with the class.
- e) Informative examples illustrating both typical and atypical uses of the class

Readers should bear in mind that the primary obligation of a tool vendor is to implement the abstract semantics defined in Clause 4, using the framework and constraints provided by the class definitions starting in Clause 5.

Annex A is intended to aid the reader in the understanding of the structure and intent of the SystemC class library.

Annex B is a glossary giving informal descriptions of the terms used in this standard.

Annex C lists the deprecated features, that is, features that were present in version 2.0.1 of the Open SystemC Initiative (OSCI) open source proof-of-concept SystemC implementation but are not part of this standard.

Annex D lists the changes between SystemC version 2.0.1 and version 2.1 Beta Oct 12 2004, and the changes between SystemC 2.1 Beta Oct 12 2004 and this standard.

²Notes in text, tables, and figures are given for information only, and do not contain requirements needed to implement the standard.

2. References

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

This standard shall be used in conjunction with the following publications:

ISO/IEC 14882:2003, Programming Languages—C++.³

IEEE Std 1364TM-2001, IEEE Standard Verilog® Hardware Description Language.^{4, 5}

³ISO/IEC publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembe, CH-1211, Genève 20, Switzerland/Suisse (<http://www.iso.ch/>). ISO/IEC publications are also available in the United States from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112, USA (<http://global.ihs.com/>). Electronic copies are available in the United States from the American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (<http://www.ansi.org/>).

⁴IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (<http://standards.ieee.org/>).

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

3. Terminology and conventions used in this standard

3.1 Terminology

3.1.1 Shall, should, may, can

The word *shall* is used to indicate a mandatory requirement.

The word *should* is used to recommend a particular course of action, but does not impose any obligation.

The word *may* is used to mean shall be permitted (in the sense of being legally allowed).

The word *can* is used to mean shall be able to (in the sense of being technically possible).

In some cases, word usage is qualified to indicate on whom the obligation falls, such as *an application may* or *an implementation shall*.

3.1.2 Implementation, application

The word *implementation* is used to mean any specific implementation of the full SystemC class library as defined in this standard, only the public interface of which need be exposed to the application.

The word *application* is used to mean a C++ program, written by an end user, that uses the SystemC class library, that is, uses classes, functions, or macros defined in this standard.

3.1.3 Call, called from, derived from

The term *call* is taken to mean call directly or indirectly. Call indirectly means call an intermediate function which in turn calls the function in question, where the chain of function calls may be extended indefinitely.

Similarly, *called from* means called from directly or indirectly.

Except where explicitly qualified, the term *derived from* is taken to mean derived directly or indirectly from. Derived indirectly from means derived from one or more intermediate base classes.

3.1.4 Specific technical terms

The following terms are sometimes used to refer to classes and sometimes used to refer to objects of those classes. When the distinction is important, the usage of the term may be qualified. For example, a *port instance* is an object of a class derived from the class **sc_port**, whereas a *port class* is a class derived from class **sc_port**.

A *module* is a class derived from the class **sc_module**.

A *port* is either a class derived from the class **sc_port** or an object of class **sc_port**.

An *export* is an object of class **sc_export**.

An *interface* is a class derived from the class **sc_interface**.

An *interface proper* is an abstract class derived from the class **sc_interface** but not derived from the class **sc_object**.

A *primitive channel* is a non-abstract class derived from one or more interfaces and also derived from the class **sc_prim_channel**.

A *hierarchical channel* is a non-abstract class derived from one or more interfaces and also derived from the class **sc_module**.

A *channel* is a non-abstract class derived from one or more interfaces. A channel may be a primitive channel or a hierarchical channel. If not, it is strongly recommended that a channel be derived from the class **sc_object**.

An *event* is an object of the class **sc_event**.

A *signal* is an object of the class **sc_signal**.

A *process instance* is an object of an implementation-defined class derived from the class **sc_object** and created by one of the three macros **SC_METHOD**, **SC_THREAD**, or **SC_CTHREAD** or by calling the function **sc_spawn**.

The term *process* refers to either a process instance or to the member function that is associated with a process instance when it is created. The meaning is made clear by the context.

A *static process* is a process created during the construction of the module hierarchy or from the **before_end_of_elaboration** callback.

A *dynamic process* is a process created from the **end_of_elaboration** callback or during simulation.

An *unspawned process* is a process created by invoking one of the three macros **SC_METHOD**, **SC_THREAD**, or **SC_CTHREAD**. An unspawned process is typically a static process, but would be a dynamic process if invoked from the **end_of_elaboration** callback.

A *spawned process* is a process created by calling the function **sc_spawn**. A spawned process is typically a dynamic process, but would be a static process if **sc_spawn** is called before the end of elaboration.

A *process handle* is an object of the class **sc_process_handle**.

The *module hierarchy* is the total set of module instances constructed during elaboration. The term is sometimes used to include all of the objects instantiated within those modules during elaboration. The module hierarchy is a subset of the *object hierarchy*.

The *object hierarchy* is the total set of objects of class **sc_object**. Part of the object hierarchy is constructed during elaboration (the *module hierarchy*) and includes module, port, primitive channel, and static process instances. Part is constructed dynamically and destroyed dynamically during simulation and includes dynamic process instances (see 5.15).

A given instance is *within* module M if the constructor of the instance is called (explicitly or implicitly) from the constructor of module M and if the instance is not within another module instance that is itself within module M.

A given module is said to *contain* a given instance if the instance is within that module.

A *child* of a given module is an instance that is within that module.

A *parent* of a given instance is a module having that instance as a child.

A *top-level module* is a module that is not instantiated within any other module.

The concepts of *elaboration* and *simulation* are defined in Clause 4. The terms *during elaboration* and *during simulation* indicate that an action may happen at that time. The implementation makes a number of callbacks to the application during elaboration and simulation. Whether a particular action is allowed within a particular callback cannot be inferred from the terms *during elaboration* and *during simulation* alone but is defined in detail in 4.4. For example, a number of actions that are permitted *during elaboration* are explicitly forbidden during the **end_of_elaboration** callback.

3.2 Syntactical conventions

3.2.1 Implementation-defined

The italicized term *implementation-defined* is used where part of a C++ definition is omitted from this standard. In such cases, an implementation shall provide an appropriate definition that honors the semantics defined in this standard.

3.2.2 Disabled

The italicized term *disabled* is used within a C++ class definition to indicate a group of member functions that shall be disabled by the implementation so that they cannot be called by an application. The disabled member functions are typically the default constructor, the copy constructor, or the assignment operator.

3.2.3 Ellipsis (...)

An ellipsis, which consists of three consecutive dots (...), is used to indicate that irrelevant or repetitive parts of a C++ code listing or example have been omitted for clarity.

3.2.4 Class names

Class names italicized and annotated with a superscript dagger ([†]) should not be used explicitly within an application. Moreover, an application shall not create an object of such a class. It is strongly recommended that the given class name be used. However, an implementation may substitute an alternative class name in place of every occurrence of a particular daggered class name.

Only the class name is considered here. Whether any part of the definition of the class is implementation-defined is a separate issue.

The class names are the following:

<i>sc_bind_proxy</i> [†]	<i>sc_fxnum_bitref</i> [†]	<i>sc_signed_bitref</i> [†]	<i>sc_uint_subref</i> [†]
<i>sc_bitref</i> [†]	<i>sc_fxnum_fast_bitref</i> [†]	<i>sc_signed_bitref_r</i> [†]	<i>sc_uint_subref_r</i> [†]
<i>sc_bitref_r</i> [†]	<i>sc_fxnum_fast_subref</i> [†]	<i>sc_signed_subref</i> [†]	<i>sc_unsigned_bitref</i> [†]
<i>sc_concatref</i> [†]	<i>sc_fxnum_subref</i> [†]	<i>sc_signed_subref_r</i> [†]	<i>sc_unsigned_bitref_r</i> [†]
<i>sc_concref</i> [†]	<i>sc_int_bitref</i> [†]	<i>sc_subref</i> [†]	<i>sc_unsigned_subref</i> [†]
<i>sc_concref_r</i> [†]	<i>sc_int_bitref_r</i> [†]	<i>sc_subref_r</i> [†]	<i>sc_unsigned_subref_r</i> [†]
<i>sc_context_begin</i> [†]	<i>sc_int_subref</i> [†]	<i>sc_switch</i> [†]	<i>sc_value_base</i> [†]
<i>sc_event_and_list</i> [†]	<i>sc_int_subref_r</i> [†]	<i>sc_uint_bitref</i> [†]	
<i>sc_event_or_list</i> [†]	<i>sc_sensitive</i> [†]	<i>sc_uint_bitref_r</i> [†]	

3.2.5 Embolded text

Embolding is used to enhance readability in this standard but has no significance in SystemC itself. Embolding is used for names of types, classes, functions, and operators in running text and in code fragments where these names are defined. Embolding is never used for upper-case names of macros, constants, and enum literals.

3.3 Semantic conventions

3.3.1 Class definitions and the inheritance hierarchy

An implementation may differ from this standard in that an implementation may introduce additional base classes, class members, and friends to the classes defined in this standard. An implementation may modify the inheritance hierarchy by moving class members defined by this standard into base classes not defined by this standard. Such additions and modifications may be made as necessary in order to implement the semantics defined by this standard or in order to introduce additional functionality not defined by this standard.

3.3.2 Function definitions and side-effects

This standard explicitly defines the semantics of the C++ functions in the SystemC class library. Such functions shall not have any side-effects that would contradict the behavior explicitly mandated by this standard. In general, the reader should assume the common-sense rule that if it is explicitly stated that a function shall perform action A, that function shall not perform any action other than A, either directly or by calling another function defined in this standard. However, a function may, and indeed in certain circumstances shall, perform any tasks necessary for resource management, performance optimization, or to support any ancillary features of an implementation. As an example of resource management, it is assumed that a destructor will perform any tasks necessary to release the resources allocated by the corresponding constructor. As an example of an ancillary feature, an implementation could have the constructor for class **sc_module** increment a count of the number of module instances in the module hierarchy.

3.3.3 Functions whose return type is a reference or a pointer

Many functions in this standard return a reference to an object or a pointer to an object, that is, the return type of the function is a reference or a pointer. This subclause gives some general rules defining the lifetime and the validity of such objects.

An object returned from a function by pointer or by reference is said to be valid during any period in which the object is not deleted and the value or behavior of the object remains accessible to the application. If an application refers to the returned object after it ceases to be valid, the behavior of the implementation shall be undefined.

3.3.3.1 Functions that return ***this** or an actual argument

In certain cases, the object returned is either an object (***this**) returned by reference from its own member function (for example, the assignment operators), or is an object that was passed by reference as an actual argument to the function being called (for example, **std::ostream& operator<< (std::ostream&, const T&)**). In either case, the function call itself places no additional obligations on the implementation concerning the lifetime and validity of the object following return from the function call.

3.3.3.2 Functions that return `char*`

Certain functions have the return type `char*`, that is, they return a pointer to a null-terminated character string. Such strings shall remain valid until the end of the program with the exception of member function `sc_process_handle::name` and member functions of class `sc_report`, where the implementation is only required to keep the string valid while the process handle or report object itself is valid.

3.3.3.3 Functions that return a reference or pointer to an object in the module hierarchy

Certain functions return a reference or pointer to an object that forms part of the module hierarchy or a property of such an object. The return types of these functions include the following:

- a) `sc_interface *` // Returns a channel
- b) `sc_event&` // Returns an event
- c) `sc_event_finder&` // Returns an event finder
- d) `sc_time&` // Returns a property of primitive channel `sc_clock`

The implementation is obliged to ensure that the returned object is valid until either the channel, event, or event finder is deleted explicitly by the application or until the destruction of the module hierarchy, whichever is sooner.

3.3.3.4 Functions that return a reference or pointer to a transient object

Certain functions return a reference or pointer to an object that may be deleted by the application or the implementation before the destruction of the module hierarchy. The return types of these functions include the following:

- a) `sc_object *`
- b) `sc_attr_base *`
- c) `std::string&` // Property of an attribute object

The functions concerned are the following:

```
sc_object* sc_process_handle::get_parent_object() const;  
sc_object* sc_process_handle::get_process_object() const;  
sc_object* sc_object::get_parent_object() const;  
sc_object* sc_find_object( const char* );  
sc_attr_base* sc_object::get_attribute( const std::string& );  
const sc_attr_base* sc_object::get_attribute( const std::string& ) const;  
sc_attr_base* sc_object::remove_attribute( const std::string& );  
const std::string& sc_attr_base::name() const;
```

The implementation is only obliged to ensure that the returned reference is valid until the `sc_object`, `sc_attr_base`, or `std::string` object itself is deleted.

Certain functions return a reference to an object that represents a transient collection of other objects, where the application may add or delete objects before the destruction of the module hierarchy such that the contents of the collection would be modified. The return types of these functions include the following:

- a) `std::vector< sc_object * > &`
- b) `sc_attr_cltn *`

The functions concerned are the following:

```
virtual const std::vector<sc_object*>& sc_module::get_child_objects() const;
const std::vector<sc_object*>& sc_process_handle::get_child_objects() const;
virtual const std::vector<sc_object*>& sc_object::get_child_objects() const;
const std::vector<sc_object*>& sc_get_top_level_objects();
sc_attr_cltn& sc_object::attr_cltn();
const sc_attr_cltn& sc_object::attr_cltn() const;
```

The implementation is only obliged to ensure that the returned object (the vector or collection) is itself valid until an **sc_object** or an attribute is added or deleted that would affect the collection returned by the function if it were to be called again.

3.3.3.5 Functions **sc_time_stamp** and **sc_signal::read**

The implementation is obliged to keep the object returned from function **sc_time_stamp** valid until the start of the next timed notification phase.

The implementation is obliged to keep the object returned from function **sc_signal::read** valid until the end of the current evaluation phase.

For both functions, it is strongly recommended that the application be written in such a way that it would have identical behavior, whether these functions return a reference to an object or return the same object by value.

3.3.4 Namespaces and internal naming

An implementation shall place every declaration and every macro definition specified by this standard within one of the two namespaces **sc_core** and **sc_dt**. The core language and predefined channels shall be placed in the namespace **sc_core**. The SystemC data types proper shall be placed in the namespace **sc_dt**. The utilities are divided between the two namespaces.

It is recommended that an implementation use nested namespaces within **sc_core** and **sc_dt** in order to reduce to a minimum the number of implementation-defined names in these two namespaces. The names of any such nested namespaces shall be implementation-defined.

In general, the choice of internal, implementation-specific names within an implementation can cause naming conflicts within an application. It is up to the implementor to choose names that are unlikely to cause naming conflicts within an application.

3.3.5 Non-compliant applications and errors

In the case where an application fails to meet an obligation imposed by this standard, the behavior of the SystemC implementation shall be undefined in general. When this results in the violation of a diagnosable rule of the C++ standard, the C++ implementation will issue a diagnostic message in conformance with the C++ standard.

When this standard explicitly states that the failure of an application to meet a specific obligation is an *error* or a *warning*, the SystemC implementation shall generate a diagnostic message by calling the function **sc_report_handler::report**. In the case of an *error*, the implementation shall call function **report** with a severity of **SC_ERROR**. In the case of a *warning*, the implementation shall call function **report** with a severity of **SC_WARNING**.

An implementation or an application may choose to suppress run-time error checking and diagnostic messages because of considerations of efficiency or practicality. For example, an application may call member function **set_actions** of class **sc_report_handler** to take no action for certain categories of report. An application that fails to meet the obligations imposed by this standard remains in error.

There are cases where this standard states explicitly that a certain behavior or result is *undefined*. This standard places no obligations on the implementation in such a circumstance. In particular, such a circumstance may or may not result in an *error* or a *warning*.

3.4 Notes and examples

Notes appear at the end of certain subclauses, designated by the upper-case word NOTE. Notes often describe consequences of rules defined elsewhere in this standard. Certain subclauses include examples consisting of fragments of C++ source code. Such notes and examples are informative to help the reader but are not an official part of this standard.

4. Elaboration and simulation semantics

An implementation of the SystemC class library includes a public *shell* consisting of those predefined classes, functions, macros, and so forth that can be used directly by an application. Such features are defined in Clause 5, Clause 6, Clause 7, and Clause 8 of this standard. An implementation also includes a private *kernel* that implements the core functionality of the class library. The underlying semantics of the kernel are defined in this clause.

The execution of a SystemC application consists of *elaboration* followed by *simulation*. Elaboration results in the creation of the *module hierarchy*. Elaboration involves the execution of application code, the public shell of the implementation (as mentioned in the preceding paragraph), and the private kernel of the implementation. Simulation involves the execution of the *scheduler*, part of the kernel, which in turn may execute *processes* within the application.

In addition to providing support for elaboration and implementing the scheduler, the kernel may also provide implementation-specific functionality beyond the scope of this standard. As an example of such functionality, the kernel may save the state of the module hierarchy after elaboration and run or restart simulation from that point, or it may support the graphical display of state variables on-the-fly during simulation.

The phases of elaboration and simulation shall run in the following sequence:

- a) Elaboration—Construction of the module hierarchy
- b) Elaboration—Callbacks to function **before_end_of_elaboration**
- c) Elaboration—Callbacks to function **end_of_elaboration**
- d) Simulation—Callbacks to function **start_of_simulation**
- e) Simulation—Initialization phase
- f) Simulation—Evaluation, update, delta notification, and timed notification phases (repeated)
- g) Simulation—Callbacks to function **end_of_simulation**
- h) Simulation—Destruction of the module hierarchy

4.1 Elaboration

The primary purpose of elaboration is to create internal data structures within the kernel as required to support the semantics of simulation. During elaboration, the parts of the module hierarchy (modules, ports, primitive channels, and processes) are created, and ports and exports are bound to channels.

The actions stated in the following subclauses can occur during elaboration and only during elaboration.

NOTE 1—Because these actions can only occur during elaboration, SystemC does not support the dynamic creation or modification of the module hierarchy during simulation, although it does support dynamic processes.

NOTE 2—Other actions besides those listed below may occur during elaboration, provided that they do not contradict any statement made in this standard. For example, objects of class **sc_dt::sc_logic** may be created during elaboration and spawned processes may be created during elaboration, but the function **notify** of class **sc_event** shall not be called during elaboration.

4.1.1 Instantiation

Instances of the following classes (or classes derived from these classes) may be created during elaboration and only during elaboration. Such instances shall not be deleted before the destruction of the module hierarchy at the end of simulation.

sc_module (see 5.2)
 sc_port (see 5.11)
 sc_export (see 5.12)
 sc_prim_channel (see 5.14)

An implementation shall permit an application to have zero or one top-level modules and may permit more than one top-level module (see 4.3.4.1 and 4.3.5).

Instances of class **sc_module** and class **sc_prim_channel** may only be created within a module or within function **sc_main**. Instances of class **sc_port** and class **sc_export** can only be created within a module. It shall be an error to instantiate a module or primitive channel other than within a module or within function **sc_main**, or to instantiate a port or export other than within a module.

The instantiation of a module also implies the construction of objects of class **sc_module_name** and class **sc_sensitive**[†] (see 5.4).

Although these rules allow for considerable flexibility in instantiating the module hierarchy, it is strongly recommended that, wherever possible, module, port, export, and primitive channel instances be data members of a module or their addresses be stored in data members of a module. Moreover, the names of those data members should match the string names of the instances wherever possible.

NOTE 1—The four classes **sc_module**, **sc_port**, **sc_export**, and **sc_prim_channel** are derived from a common base class **sc_object**, and thus have some member functions in common (see 5.15).

NOTE 2—Objects of classes derived from **sc_object** but not derived from one of these four classes may be instantiated during elaboration or during simulation, as may objects of user-defined classes.

Example:

```
#include "systemc.h"

struct Mod: sc_module
{
    SC_CTOR(Mod) {}
};

struct S
{
    Mod m; // Unusual coding style - module instance within struct
    S(char* name_) : m(name_) {}
};

struct Top: sc_module // Five instances of module Mod exist within module Top.
{
    Mod m1; // Recommended coding style
    Mod *m2; // Recommended coding style
    S s1;

    SC_CTOR(Top)
    : m1("m1"), // m1.name() returns "top.m1"
      s1("s1") // s1.m.name() returns "top.s1"
    {
        m2 = new Mod("m2"); // m2->name() returns "top.m2"
        f();
        S *s2 = new S("s2"); // s2->m.name() returns "top.s2"
    }
};
```

```

    }
    void f() {
        Mod *m3 = new Mod("m3"); // Unusual coding style - not recommended
    }                                // m3->name() returns "top.m3"
};

int sc_main(int argc, char* argv[])
{
    Top top("top");
    sc_start();
    return 0;
}

```

4.1.2 Process macros

An *unspawned process instance* is a process created by invoking one of the following three process macros:

```

SC_METHOD
SC_THREAD
SC_CTHREAD

```

The name of a member function belonging to a class derived from class **sc_module** shall be passed as an argument to the macro. This member function shall become the function *associated* with the process instance.

Unspawned processes can be created during elaboration or from the **end_of_elaboration** callback. Spawned processes may be created by calling the function **sc_spawn** during elaboration or during simulation.

The purpose of the process macros is to register the associated function with the kernel such that the scheduler can call back that member function during simulation. It is also possible to use spawned processes for this same purpose. The process macros are provided for backward compatibility with earlier versions of SystemC and to provide clocked threads for hardware synthesis.

4.1.3 Port binding and export binding

Port instances can be *bound* to channel instances, to other port instances, or to export instances. Export instances can be *bound* to channel instances or to other export instances, but not to port instances. Port binding is an asymmetrical relationship, and export binding is an asymmetrical relationship. If a port is *bound* to a channel, it is not true to say that the channel is *bound* to the port. Rather, it is true to say that the channel is the channel to which the port is *bound*.

Ports can be bound by name or by position. Named port binding is performed by a member function of class **sc_port** (see 5.11.7). Positional port binding is performed by a member function of class **sc_module** (see 5.2.18). Exports can only be bound by name. Export binding is performed by a member function of class **sc_export** (see 5.12.7).

A given port instance shall not be bound both by name and by position.

A port should typically be bound within the parent of the module instance containing that port. Hence, when port A is bound to port B, the module containing port A will typically be instantiated within the module containing port B. An export should typically be bound within the module containing the export. A port should typically be bound to a channel or a port that lies within the same module in which the port is bound or to an export within a child module. An export should typically be bound to a channel that lies within the same module in which the export is bound or to an export within a child module.

When port A is bound to port B, and port B is bound to channel C, the effect shall be the same as if port A were bound directly to channel C. Wherever this standard refers to port A being bound to channel C, it shall be assumed this means that port A is bound either directly to channel C or to another port that is itself bound to channel C according to this very same rule. This same rule shall apply when binding exports.

Port and export binding can occur during elaboration and only during elaboration. Whether a port need be bound is dependent upon the port policy argument of the port instance, whereas every export shall be bound exactly once. A module may have zero or more ports and zero or more exports. If a module has no ports, no (positional) port bindings are necessary or permitted for instances of that module. Ports may be bound (by name) in any sequence. The binding of ports belonging to different module instances may be interleaved. Since a port may be bound to another port that has not yet itself been bound, the implementation may defer the completion of port binding until a later time during elaboration, whereas exports shall be bound immediately. Such deferred port binding shall be completed by the implementation before the callbacks to function **end_of_elaboration**.

The channel to which a port is bound shall not be deleted before the destruction of the module hierarchy at the end of simulation.

Where permitted in the definition of the port object, a single port can be bound to multiple channel or port instances. Such ports are known as *multiports* (see 5.11.3). An export can only be bound once. It shall be an error to bind a given port instance to a given channel instance more than once, even if the port is a multiport.

When a port is bound to a channel, the kernel shall call the member function **register_port** of the channel. There is no corresponding function called when an export is bound (see 5.13).

The purpose of port and export binding is to enable a port or export to forward interface method calls made during simulation to the channel instances to which that port was bound during elaboration. This forwarding is performed during simulation by member functions of the class **sc_port** and class **sc_export**, such as **operator->**. A port *requires* the services defined by an interface (that is, the type of the port), whereas an export *provides* the services defined by an interface (that is, the type of the export).

NOTE 1—A phrase such as *bind a channel to a port* is not used in this standard. However, it is recognized that such a phrase may be used informally to mean *bind a port to a channel*.

NOTE 2—A port of a child module instance can be bound to an export of that same child module instance.

NOTE 3—Member function **register_port** is defined in the class **sc_interface** from which every channel is derived.

4.1.4 Setting the time resolution

The simulation time resolution can be set during elaboration and only during elaboration. Time resolution is set by calling the function **sc_set_time_resolution** (see 5.10.3).

NOTE—Time resolution can only be set globally. There is no concept of a local time resolution.

4.2 Simulation

This subclause defines the behavior of the scheduler and the semantics of simulated time and process execution.

The primary purpose of the scheduler is to trigger or resume the execution of the processes that the user supplies as part of the application. The scheduler is event-driven, meaning that processes are executed in response to the occurrence of events. Events occur (are *notified*) at precise points in simulation time. Events are represented by objects of the class **sc_event**, and by this class alone (see 5.9).

Simulation time is an integer quantity. Simulation time is initialized to zero at the start of simulation and increases monotonically during simulation. The physical significance of the integer value representing time within the kernel is determined by the simulation time resolution. Simulation time and time intervals are represented by class **sc_time**. Certain functions allow time to be expressed as a value pair having the signature **double,sc_time_unit** (see 5.10.1).

The scheduler can execute a spawned or unspawned process instance as a consequence of one of the following four causes, and these alone:

- In response to the process instance having been made runnable during the initialization phase (see 4.2.1.1)
- In response to a call to function **sc_spawn** during simulation
- In response to the occurrence of an event to which the process instance is sensitive
- In response to a time-out having occurred

The *sensitivity* of a process instance is the set of events and time-outs that can potentially cause the process to be resumed or triggered. The *static sensitivity* of an unspawned process instance is fixed during elaboration. The *static sensitivity* of a spawned process instance is fixed when the function **sc_spawn** is called. The *dynamic sensitivity* of a process instance may vary over time under the control of the process itself. A process instance is said to be *sensitive* to an event if the event has been added to the static sensitivity or dynamic sensitivity of the process instance. A *time-out* occurs when a given time interval has elapsed.

The scheduler shall also manage event notifications and primitive channel update requests.

4.2.1 The scheduling algorithm

The semantics of the scheduling algorithm are defined in the following subclauses. For the sake of clarity, imperative language is used in this description. The description of the scheduling algorithm uses the following four sets:

- The set of runnable processes
- The set of update requests
- The set of delta notifications and time-outs
- The set of timed notifications and time-outs

An implementation may substitute an alternative scheme, provided the scheduling semantics given here are retained.

A process instance shall not appear more than once in the set of runnable processes. An attempt to add to this set a process instance that is already runnable shall be ignored.

An *update request* results from, and only from, a call to member function **request_update** of class **sc_prim_channel** (see 5.14.6).

An *immediate notification* results from, and only from, a call to member function **notify** of class **sc_event** with no arguments (see 5.9.4).

A *delta notification* results from, and only from, a call to member function **notify** of class **sc_event** with a zero-valued time argument.

A *timed notification* results from, and only from, a call to member function **notify** of class **sc_event** with a non-zero-valued time argument. The time argument determines the time of the notification, relative to the time when function **notify** is called.

A *time-out* results from, and only from, certain calls to functions **wait** or **next_trigger**, which are member functions of class **sc_module**, member functions of class **sc_prim_channel**, and non-member functions. A time-out resulting from a call with a zero-valued time argument is added to the set of delta notifications and time-outs. A time-out resulting from a call with a non-zero-valued time argument is added to the set of timed notifications and time-outs (see 5.2.16 and 5.2.17).

The scheduler starts by executing the initialization phase.

4.2.1.1 Initialization phase

Perform the following three steps in the order given:

- a) Run the update phase as defined in 4.2.1.3 but without continuing to the delta notification phase.
- b) Add every method and thread process instance in the object hierarchy to the set of runnable processes, but exclude those process instances for which the function **dont_initialize** has been called, and exclude clocked thread processes.
- c) Run the delta notification phase, as defined in 4.2.1.4. At the end of the delta notification phase, go to the evaluation phase.

NOTE—The update and delta notification phases are necessary because function **request_update** can be called during elaboration in order to set initial values for primitive channels, for example, from function **initialize** of class **sc_inout**.

4.2.1.2 Evaluation phase

From the set of runnable processes, select a process instance and trigger or resume its execution. Run the process instance immediately and without interruption up to the point where it either returns or calls the function **wait**.

Since process instances execute without interruption, only a single process instance can be running at any one time, and no other process instance can execute until the currently executing process instance has yielded control to the kernel. A process shall not pre-empt or interrupt the execution of another process. This is known as *co-routine semantics* or *co-operative multitasking*.

The order in which process instances are selected from the set of runnable processes is implementation-defined. However, if a specific version of a specific implementation runs a specific application using a specific input data set, the order of process execution shall not vary from run to run.

A process may execute an immediate notification, in which case determine which process instances are currently sensitive to the notified event and add all such process instances to the set of runnable processes. Such processes shall be executed subsequently in this very same evaluation phase.

A process may call function **sc_spawn** to create a spawned process instance, in which case the new process instance shall be added to the set of runnable processes (unless function **sc_spawn_options::dont_initialize** is called) and subsequently executed in this very same evaluation phase.

A process may call the member function **request_update** of a primitive channel, which will cause the member function **update** of that same primitive channel to be called back during the very next update phase.

Repeat this step until the set of runnable processes is empty, then go on to the update phase.

NOTE 1—The scheduler is not pre-emptive. An application can assume that a method process will execute in its entirety without interruption, and a thread or clocked thread process will execute the code between two consecutive calls to function **wait** without interruption.

NOTE 2—Because the order in which processes are run within the evaluation phase is not under the control of the application, access to shared storage should be explicitly synchronized to avoid non-deterministic behavior.

NOTE 3—An implementation running on a machine that provides hardware support for concurrent processes may permit two or more processes to run concurrently, provided that the behavior appears identical to the co-routine semantics defined in this subclause. In other words, the implementation would be obliged to analyze any dependencies between processes and constrain their execution to match the co-routine semantics.

NOTE 4—When an immediate notification occurs, only processes that are currently sensitive to the notified event shall be made runnable. This excludes processes that are only made dynamically sensitive to the notified event later in the same evaluation phase.

4.2.1.3 Update phase

Execute any and all pending calls to function **update** resulting from calls to function **request_update** made in the immediately preceding evaluation phase or made during elaboration if the update phase is executed as part of the initialization phase.

If no remaining pending calls to function **update** exist, go on to the delta notification phase (except when executed from the initialization phase).

4.2.1.4 Delta notification phase

If pending delta notifications or time-outs exist (which can only result from calls to function **notify** or function **wait** in the immediately preceding evaluation phase or update phase):

- a) Determine which process instances are sensitive to these events or time-outs.
- b) Add all such process instances to the set of runnable processes.
- c) Remove all such notifications and time-outs from the set of delta notifications and time-outs.

If, at the end of the delta notification phase, the set of runnable processes is non-empty, go back to the evaluation phase.

4.2.1.5 Timed notification phase

If pending timed notifications or time-outs exist:

- a) Advance simulation time to the time of the earliest pending timed notification or time-out.
- b) Determine which process instances are sensitive to the events notified and time-outs lapsing at this precise time.
- c) Add all such process instances to the set of runnable processes.
- d) Remove all such notifications and time-outs from the set of timed notifications and time-outs.

If no pending timed notifications or time-outs exist, the end of simulation has been reached. So, exit the scheduler.

If, at the end of the timed notification phase, the set of runnable processes is non-empty, go back to the evaluation phase.

4.2.2 Cycles in the scheduling algorithm

A *delta cycle* is a sequence of steps in the scheduling algorithm consisting of the following steps in the order given:

- a) An evaluation phase
- b) An update phase
- c) A delta notification phase

The initialization phase does not include a delta cycle.

NOTE 1—The scheduling algorithm implies the existence of three causal loops resulting from immediate notification, delta notification, and timed notification, as follows:

- The immediate notification loop is restricted to a single evaluation phase.
- The delta notification loop takes the path of evaluation phase, followed by update phase, followed by delta notification phase and back to evaluation phase. This loop advances simulation by one delta cycle.
- The timed notification loop takes the path of evaluation phase, followed by update phase, followed by delta notification phase, followed by timed notification phase and back to evaluation phase. This loop advances simulation time.

NOTE 2—The immediate notification loop is non-deterministic in the sense that process execution can be interleaved with immediate notification, and the order in which runnable processes are executed is undefined.

NOTE 3—The delta notification and timed notification loops are deterministic in the sense that process execution alternates with primitive channel updates. If, within a particular application, inter-process communication is confined to using only deterministic primitive channels, the behavior of the application will be independent of the order in which the processes are executed within the evaluation phase (assuming no other explicit dependencies on process order such as external input or output exist).

4.3 Running elaboration and simulation

An implementation shall provide either or both of the following two mechanisms for running elaboration and simulation:

- Under application control using functions **sc_main** and **sc_start**
- Under control of the kernel

Both mechanisms are defined in the following subclauses. An implementation is not obliged to provide both mechanisms.

4.3.1 Function declarations

```
namespace sc_core {  
  
    int sc_elab_and_sim( int argc, char* argv[] );  
    int sc_argc();  
    const char* const* sc_argv();  
    void sc_start();  
    void sc_start( const sc_time& );  
    void sc_start( double, sc_time_unit );  
}
```

4.3.2 Function **sc_elab_and_sim**

The function **main** that is the entry point of the C++ program may be provided by the implementation or by the application. If function **main** is provided by the implementation, function **main** shall initiate the mechanisms for elaboration and simulation as described in this subclause. If function **main** is provided by the application, function **main** shall call the function **sc_elab_and_sim**, which is the entry point into the SystemC implementation.

The implementation shall provide a function **sc_elab_and_sim** with the following declaration:

```
int sc_elab_and_sim( int argc, char* argv[] );
```

Function **sc_elab_and_sim** shall initiate the mechanisms for running elaboration and simulation. The application should pass the values of the parameters from function **main** as arguments to function

sc_elab_and_sim. Whether the application may call function **sc_elab_and_sim** more than once is implementation-defined.

A return value of 0 from function **sc_elab_and_sim** shall indicate successful completion. An implementation may use other return values to indicate other termination conditions.

NOTE—Function **sc_elab_and_sim** was named **sc_main_main** in an earlier version of SystemC.

4.3.3 Functions **sc_argc** and **sc_argv**

The implementation shall provide functions **sc_argc** and **sc_argv** with the following declarations:

```
int sc_argc();
const char* const* sc_argv();
```

These two functions shall return the values of the arguments passed to function **main** or function **sc_elab_and_sim**.

4.3.4 Running under application control using functions **sc_main** and **sc_start**

The application provides a function **sc_main** and calls the function **sc_start**, as defined in 4.3.4.1 and 4.3.4.2.

4.3.4.1 Function **sc_main**

An application shall provide a function **sc_main** with the following declaration. The order and types of the arguments and the return type shall be as shown here:

```
int sc_main( int argc, char* argv[] );
```

This function shall be called once from the kernel and is the only entry point into the application. The arguments **argc** and **argv[]** are command-line arguments. The implementation may pass the values of C++ command-line arguments (as passed to function **main**) through to function **sc_main**. The choice of which C++ command-line arguments to pass is implementation-defined.

Elaboration consists of the execution of the **sc_main** function from the start of **sc_main** to the point immediately before the first call to the function **sc_start**.

A return value of 0 from function **sc_main** shall indicate successful completion. An application may use other return values to indicate other termination conditions.

NOTE 1—As a consequence of the rules defined in 4.1, before calling function **sc_start** for the first time, the function **sc_main** may instantiate modules, instantiate primitive channels, bind the ports and exports of module instances to channels, and set the time resolution. More than one top-level module may exist.

NOTE 2—Throughout this standard the term *call* is taken to mean call directly or indirectly. Hence function **sc_start** may be called indirectly from function **sc_main** by another function or functions.

4.3.4.2 Function **sc_start**

The implementation shall provide a function **sc_start**, overloaded with the following signatures:

```
void sc_start();
void sc_start( const sc_time& );
void sc_start( double, sc_time_unit );
```

The behavior of the latter function shall be equivalent to the following definition:

```
void sc_start( double d, sc_time_unit t ) { sc_start( sc_time(d, t) ); }
```

When called for the first time, function **sc_start** shall start the scheduler, which shall run up to the simulation time passed as an argument (if an argument was passed), unless otherwise interrupted.

When called on the second and subsequent occasions, function **sc_start** shall resume the scheduler from the time it had reached at the end of the previous call to **sc_start**. The scheduler shall run for the time passed as an argument (if an argument was passed), relative to the current simulation time, unless otherwise interrupted.

When a time is passed as an argument, the scheduler shall execute up to and including the timed notification phase that advances simulation time to the end time (calculated by adding the time given as an argument to the simulation time when function **sc_start** is called).

When function **sc_start** is called without any arguments, the scheduler shall run until it reaches completion, unless otherwise interrupted.

When function **sc_start** is called with a zero-valued time argument, the scheduler shall run for one delta cycle.

Once started, the scheduler shall run until either it reaches completion, or the application calls the function **sc_stop**, or an exception occurs, or simulation is stopped or aborted by the report handler (see 8.3). Once the function **sc_stop** has been called, function **sc_start** shall not be called again.

Function **sc_start** may be called from function **sc_main**, and only from function **sc_main**.

On completion, function **sc_start** returns control to the function from which it was called.

NOTE—When the scheduler is paused between successive calls to function **sc_start**, the set of runnable processes need not be empty.

4.3.5 Running under control of the kernel

Elaboration and simulation may be initiated under the direct control of the kernel, in which case the implementation shall not call the function **sc_main**, and the implementation is not obliged to provide a function **sc_start**.

An implementation may permit more than one top-level module but is not obliged to do so.

NOTE 1—In this case, the mechanisms used to initiate elaboration and simulation and to identify top-level modules are implementation-defined.

NOTE 2—In this case, an implementation shall honor all obligations set out in this standard with the exception of those in 4.3.4.

4.4 Elaboration and simulation callbacks

Four callback functions are called by the kernel at various stages during elaboration and simulation. They have the following declarations:

```
virtual void before_end_of_elaboration();  
virtual void end_of_elaboration();  
virtual void start_of_simulation();  
virtual void end_of_simulation();
```

The implementation shall define each of these four callback functions as member functions of the classes **sc_module**, **sc_port**, **sc_export**, and **sc_prim_channel**, and each of these definitions shall have empty function bodies. The implementation also overrides various of these functions as member functions of various predefined channel and port classes and having specific behaviors (see Clause 6). An application may override any of these four functions in any class derived from any of the classes mentioned in this paragraph. If an application overrides any such callback function of a predefined class and the callback has implementation-defined behavior (for example, **sc_in::end_of_elaboration**), the application-defined member function in the derived class may or may not call the implementation-defined function of the base class, and the behavior will differ, depending on whether the member function of the base class is called.

Within each of the four categories of callback functions, the order in which the callbacks are made for objects of class **sc_module**, **sc_port**, **sc_export**, and **sc_prim_channel** shall be implementation-defined.

The implementation shall make callbacks to all such functions for every instance in the module hierarchy, as defined in the following subclauses.

The implementation shall provide the following two functions:

```
namespace sc_core {

    bool sc_start_of_simulation_invoked();
    bool sc_end_of_simulation_invoked();
}
```

Function **sc_start_of_simulation_invoked** shall return **true** after and only after all the callbacks to function **start_of_simulation** have executed to completion. Function **sc_end_of_simulation_invoked** shall return **true** after and only after all the callbacks to function **end_of_simulation** have executed to completion.

4.4.1 before_end_of_elaboration

The implementation shall make callbacks to member function **before_end_of_elaboration** after the construction of the module hierarchy defined in 4.3 is complete. Function **before_end_of_elaboration** may extend the construction of the module hierarchy by instantiating further modules (and other objects) within the module hierarchy.

The purpose of member function **before_end_of_elaboration** is to allow an application to perform actions during elaboration that depend on global properties of the module hierarchy and which also need to modify the module hierarchy. Examples include the instantiation of top-level modules to monitor events buried within the hierarchy.

The following actions may be performed directly or indirectly from the member function **before_end_of_elaboration**.

- a) The instantiation of objects of class **sc_module**, **sc_port**, **sc_export**, **sc_prim_channel**
- b) The instantiation of objects of other classes derived from class **sc_object**
- c) Port binding
- d) Export binding
- e) The macros **SC_METHOD**, **SC_THREAD**, **SC_CTHREAD**, **SC_HAS_PROCESS**
- f) The member **sensitive** and member functions **dont_initialize**, **set_stack_size**, and **reset_signal_is** of the class **sc_module**
- g) Calls to event finder functions
- h) Calls to function **sc_spawn** to create static spawned processes
- i) Calls to member function **request_update** of class **sc_prim_channel** to create update requests (for example by calling member function **initialize** of class **sc_inout**)

The following constructs shall not be used directly within member function **before_end_of_elaboration**, but may be used where permitted within module instances nested within callbacks to **before_end_of_elaboration**:

- a) The macro **SC_CTOR**
- b) Calls to member function **notify** of the class **sc_event**

operator-> and **operator[]** of class **sc_port** should not be called from the function **before_end_of_elaboration**, because the implementation may not have completed port binding at the time of this callback and hence these operators may return null pointers. The member function **size** may return a value less than its final value.

Any **sc_object** instances created from callback **before_end_of_elaboration** shall be placed at a location in the module hierarchy as if those instances had been created from the constructor of the module to which the callback belongs, or to the parent module if the callback belongs to a port, export, or primitive channel. In other words, it shall be as if the instances were created from the constructor of the object whose callback is called.

Objects instantiated from the member function **before_end_of_elaboration** may themselves override any of the four callback functions, including the member function **before_end_of_elaboration** itself. The implementation shall make all such nested callbacks. An application can assume that every such member function will be called back by the implementation, whatever the context in which the object is instantiated.

4.4.2 end_of_elaboration

The implementation shall call member function **end_of_elaboration** at the very end of elaboration after all callbacks to **before_end_of_elaboration** have completed and after the completion of any instantiation or port binding performed by those callbacks and before starting simulation.

The purpose of member function **end_of_elaboration** is to allow an application to perform housekeeping actions at the end of elaboration that do not need to modify the module hierarchy. Examples include design rule checking, actions that depend on the number of times a port is bound, and printing diagnostic messages concerning the module hierarchy.

The following actions may be performed directly or indirectly from the callback **end_of_elaboration**:

- a) The instantiation of objects of classes derived from class **sc_object** but excluding classes **sc_module**, **sc_port**, **sc_export**, and **sc_prim_channel**
- b) The macros **SC_METHOD**, **SC_THREAD**, and **SC_HAS_PROCESS**
- c) The member **sensitive** and member functions **dont_initialize** and **set_stack_size** of the class **sc_module**
- d) Calls to function **sc_spawn** to create dynamic spawned processes
- e) Calls to member function **request_update** of class **sc_prim_channel** to create update requests (for example by calling member function **write** of class **sc_inout**)
- f) Interface method calls using **operator->** and **operator[]** of class **sc_port**, provided that those calls do not attempt to perform actions prohibited outside simulation such as event notification

The following constructs shall not be used directly or indirectly within callback **end_of_elaboration**:

- a) The instantiation of objects of class **sc_module**, **sc_port**, **sc_export**, **sc_prim_channel**
- b) Port binding
- c) Export binding
- d) The macros **SC_CTOR**, **SC_CTHREAD**

- e) The member function **reset_signal_is** of the class **sc_module**
- f) Calls to event finder functions
- g) Calls to member function **notify** of the class **sc_event**

4.4.3 start_of_simulation

The implementation shall call member function **start_of_simulation** immediately when the application calls function **sc_start** for the first time or at the very start of simulation, if simulation is initiated under the direct control of the kernel. If an application makes multiple calls to **sc_start**, the implementation shall only make the callbacks to **start_of_simulation** on the first such call to **sc_start**. The implementation shall call function **start_of_simulation** after the callbacks to **end_of_elaboration** and before invoking the initialization phase of the scheduler.

The purpose of member function **start_of_simulation** is to allow an application to perform housekeeping actions at the start of simulation. Examples include opening stimulus and response files and printing diagnostic messages. The intention is that an implementation that initiates elaboration and simulation under direct control of the kernel (in the absence of functions **sc_main** and **sc_start**) shall make the callbacks to **end_of_elaboration** at the end of elaboration and the callbacks to **start_of_simulation** at the start of simulation.

The following actions may be performed directly or indirectly from the callback **start_of_simulation**:

- a) The instantiation of objects of classes derived from class **sc_object** but excluding classes **sc_module**, **sc_port**, **sc_export**, and **sc_prim_channel**
- b) Calls to function **sc_spawn** to create dynamic spawned processes
- c) Calls to member function **request_update** of class **sc_prim_channel** to create update requests (for example by calling member function **write** of class **sc_inout**)
- d) Interface method calls using **operator->** and **operator[]** of class **sc_port**, provided that those calls do not attempt to perform actions prohibited outside simulation such as event notification

The following constructs shall not be used directly or indirectly within callback **start_of_simulation**:

- a) The instantiation of objects of class **sc_module**, **sc_port**, **sc_export**, **sc_prim_channel**
- b) Port binding
- c) Export binding
- d) The macros **SC_CTOR**, **SC_METHOD**, **SC_THREAD**, **SC_CTHREAD**, **SC_HAS_PROCESS**
- e) The member **sensitive** and member functions **dont_initialize**, **set_stack_size**, and **reset_signal_is** of the class **sc_module**
- f) Calls to event finder functions
- g) Calls to member function **notify** of the class **sc_event**

4.4.4 end_of_simulation

The implementation shall call member function **end_of_simulation** at the point when the scheduler halts because of the function **sc_stop** having been called during simulation (see 4.5.2) or at the very end of simulation if simulation is initiated under the direct control of the kernel. The **end_of_simulation** callbacks shall only be called once even if function **sc_stop** is called multiple times.

The purpose of member function **end_of_simulation** is to allow an application to perform housekeeping actions at the end of simulation. Examples include closing stimulus and response files and printing diagnostic messages. The intention is that an implementation that initiates elaboration and simulation under

direct control of the kernel (in the absence of functions **sc_main** and **sc_start**) shall make the callbacks to **end_of_simulation** at the very end of simulation whether or not function **sc_stop** has been called.

As a consequence of the language mechanisms of C++, the destructors of any objects in the module hierarchy will be called as these objects are deleted at the end of program execution. Any callbacks to function **end_of_simulation** shall be made before the destruction of the module hierarchy. The function **sc_end_of_simulation_invoked** may be called by the application within a destructor to determine whether the callback has been made.

The implementation is not obliged to support any of the following actions when made directly or indirectly from the member function **end_of_simulation** or from the destructors of any objects in the module hierarchy. Whether any of these actions cause an error is implementation-defined.

- a) The instantiation of objects of classes derived from class **sc_object**
- b) Calls to function **sc_spawn** to create dynamic spawned processes
- c) Calls to member function **request_update** of class **sc_prim_channel** to create update requests (for example by calling member function **write** of class **sc_inout**)
- d) Calls to member function **notify** of the class **sc_event**

4.5 Other functions related to the scheduler

4.5.1 Function declarations

```
namespace sc_core {  
  
    enum sc_stop_mode  
    {  
        SC_STOP_FINISH_DELTA ,  
        SC_STOP_IMMEDIATE  
    };  
  
    extern void sc_set_stop_mode( sc_stop_mode mode );  
    extern sc_stop_mode sc_get_stop_mode();  
    void sc_stop();  
  
    const sc_time& sc_time_stamp();  
    const sc_dt::uint64 sc_delta_count();  
    bool sc_is_running();  
}
```

4.5.2 Function **sc_stop**, **sc_set_stop_mode**, and **sc_get_stop_mode**

The implementation shall provide functions **sc_set_stop_mode**, **sc_get_stop_mode**, and **sc_stop** with the following declarations:

```
enum sc_stop_mode  
{  
    SC_STOP_FINISH_DELTA ,  
    SC_STOP_IMMEDIATE  
};  
  
extern void sc_set_stop_mode( sc_stop_mode mode );  
extern sc_stop_mode sc_get_stop_mode();
```

```
void sc_stop();
```

The function **sc_set_stop_mode** shall set the current stop mode to the value passed as an argument. The function **sc_get_stop_mode** shall return the current stop mode.

The function **sc_stop** may be called by the application from an elaboration or simulation callback, from a process, from the member function **update** of class **sc_prim_channel**, or from function **sc_main**. The implementation may call the function **sc_stop** from member function **report** of class **sc_report_handler**.

A call to function **sc_stop** shall cause elaboration or simulation to halt as described below and control to return to function **sc_main** or to the kernel. The implementation shall print out a message from function **sc_stop** to standard output to indicate that simulation has been halted by this means. The implementation shall make the **end_of_simulation** callbacks as described in 4.4.4.

If the function **sc_stop** is called from one of the callbacks **before_end_of_elaboration**, **end_of_elaboration**, **start_of_simulation**, or **end_of_simulation**, elaboration or simulation shall halt after the current callback phase is complete, that is, after all callbacks of the given kind have been made.

If the function **sc_stop** is called during the evaluation phase or the update phase, the scheduler shall halt as determined by the current stop mode but in any case before the delta notification phase of the current delta cycle. If the current stop mode is **SC_STOP_FINISH_DELTA**, the scheduler shall complete both the current evaluation phase and the current update phase before halting simulation. If the current stop mode is **SC_STOP_IMMEDIATE** and function **sc_stop** is called during the evaluation phase, the scheduler shall complete the execution of the current process and shall then halt without executing any further processes and without executing the update phase. If function **sc_stop** is called during the update phase, the scheduler shall complete the update phase before halting. Whatever the stop mode, simulation shall not halt until the currently executing process has yielded control to the scheduler (such as by calling function **wait** or by executing a return statement).

It shall be an error for the application to call function **sc_start** after function **sc_stop** has been called.

If function **sc_stop** is called a second time before or after elaboration or simulation has halted, the implementation shall issue a warning. If function **stop_after** of class **sc_report_handler** has been used to cause **sc_stop** to be called on the occurrence of a warning, the implementation shall override this report-handling mechanism and shall not make further calls to **sc_stop**, preventing an infinite regression.

NOTE 1—A function **sc_stop** shall be provided by the implementation, whether or not the implementors choose to provide a function **sc_start**.

NOTE 2—Throughout this standard, the term *call* is taken to mean call directly or indirectly. Hence function **sc_stop** may be called indirectly, for example, by an interface method call.

4.5.3 Function **sc_time_stamp**

The implementation shall provide a function **sc_time_stamp** with the following declaration:

```
const sc_time& sc_time_stamp();
```

The function **sc_time_stamp** shall return the current simulation time. During elaboration and initialization the function shall return a value of zero.

NOTE—The simulation time can only be modified by the scheduler.

4.5.4 Function **sc_delta_count**

The implementation shall provide a function **sc_delta_count** with the following declaration:

```
const sc_dt::uint64 sc_delta_count();
```

The function **sc_delta_count** shall return an integer value that is incremented exactly once in each delta cycle, and thus returns a count of the absolute number of delta cycles that have occurred during simulation, starting from zero. When the delta count reaches the maximum value of type **sc_dt::uint64**, the count shall start again from zero. Hence the delta count in successive delta cycles might be maxvalue-1, maxvalue, 0, 1, 2, and so on.

NOTE—This function is intended for use in primitive channels to detect whether an event has occurred by comparing the delta count with the delta count stored in a variable from an earlier delta cycle. The following code fragment will test whether a process has been executed in two consecutive delta cycles:

```
if (sc_delta_count() == stored_delta_count + 1) { /* consecutive */ }  
stored_delta_count = sc_delta_count();
```

4.5.5 Function **sc_is_running**

The implementation shall provide a function **sc_is_running** with the following declaration:

```
bool sc_is_running();
```

The function **sc_is_running** shall return the value **true** while the scheduler is running, including the initialization phase, and shall return the value **false** during elaboration, during the callbacks **start_of_simulation** and **end_of_simulation** and when called from the destructor of any object in the module hierarchy.

5. Core language class definitions

5.1 Class header files

To use the SystemC class library features, an application shall include either of the C++ header files specified in this subclause at appropriate positions in the source code as required by the scope and linkage rules of C++.

5.1.1 #include "systemc"

The header file named **systemc** shall add the names **sc_core** and **sc_dt** to the declarative region in which it is included, and these two names only. The header file **systemc** shall not introduce into the declarative region in which it is included any other names from this standard or any names from the standard C or C++ libraries.

It is recommended that applications include the header file **systemc** rather than the header file **systemc.h**.

Example:

```
#include "systemc"
using sc_core::sc_module;
using sc_core::sc_signal;
using sc_core::SC_NS;
using sc_core::sc_start;
using sc_dt::sc_logic;
```

```
#include <iostream>
using std::ofstream;
using std::cout;
using std::endl;
```

5.1.2 #include "systemc.h"

The header file named **systemc.h** shall add all of the names from the namespaces **sc_core** and **sc_dt** to the declarative region in which it is included, together with selected names from the standard C or C++ libraries as defined in this subclause. It is recommended that an implementation keep to a minimum the number of additional implementation-specific names introduced by this header file.

The header file **systemc.h** is provided for backward compatibility with earlier versions of SystemC and may be deprecated in future versions of this standard.

The header file **systemc.h** shall include at least the following:

```
#include "systemc"

// Using declarations for all the names in the sc_core namespace specified in this standard
using sc_core::sc_module;
...

// Using declarations for all the names in the sc_dt namespace specified in this standard
using sc_dt::sc_int;
...
```

```
// Using declarations for selected names in the standard libraries
using std::ios;
using std::streambuf;
using std::streampos;
using std::streamsize;
using std::iostream;
using std::istream;
using std::ostream;
using std::cin;
using std::cout;
using std::cerr;
using std::endl;
using std::flush;
using std::dec;
using std::hex;
using std::oct;
using std::fstream;
using std::ifstream;
using std::ofstream;
using std::size_t;
using std::memchr;
using std::memcmp;
using std::memcpy;
using std::memmove;
using std::memset;
using std::strcat;
using std::strncat;
using std::strchr;
using std::strrchr;
using std::strcmp;
using std::strncmp;
using std::strcpy;
using std::strncpy;
using std::strcspn;
using std::strspn;
using std::strlen;
using std::strpbrk;
using std::strstr;
using std::strtok;
```

5.2 **sc_module**

5.2.1 Description

Class **sc_module** is the base class for modules. Modules are the principle structural building blocks of SystemC.

5.2.2 Class definition

```
namespace sc_core {

class sc_bind_proxy† { implementation-defined };
const sc_bind_proxy† SC_BIND_PROXY_NIL;

class sc_module
: public sc_object
{
public:
    virtual ~sc_module();

    virtual const char* kind() const;

    void operator() ( const sc_bind_proxy†& p001,
                     const sc_bind_proxy†& p002 = SC_BIND_PROXY_NIL,
                     const sc_bind_proxy†& p003 = SC_BIND_PROXY_NIL,
                     ...
                     const sc_bind_proxy†& p063 = SC_BIND_PROXY_NIL,
                     const sc_bind_proxy†& p064 = SC_BIND_PROXY_NIL );

    virtual const std::vector<sc_object*>& get_child_objects() const;

protected:
    sc_module( const sc_module_name& );
    sc_module();

    void reset_signal_is( const sc_in<bool>& , bool );
    void reset_signal_is( const sc_signal<bool>& , bool );

    sc_sensitive† sensitive;

    void dont_initialize();
    void set_stack_size( size_t );

    void next_trigger();
    void next_trigger( const sc_event& );
    void next_trigger( sc_event_or_list†& );
    void next_trigger( sc_event_and_list†& );
    void next_trigger( const sc_time& );
    void next_trigger( double , sc_time_unit );
    void next_trigger( const sc_time& , const sc_event& );
    void next_trigger( double , sc_time_unit , const sc_event& );
    void next_trigger( const sc_time& , sc_event_or_list†& );
    void next_trigger( double , sc_time_unit , sc_event_or_list†& );
};
```

```

void next_trigger( const sc_time& , const sc_event_and_list†& );
void next_trigger( double , sc_time_unit , sc_event_and_list†& );

void wait();
void wait( int );
void wait( const sc_event& );
void wait( sc_event_or_list†& );
void wait( sc_event_and_list†& );
void wait( const sc_time& );
void wait( double , sc_time_unit );
void wait( const sc_time& , const sc_event& );
void wait( double , sc_time_unit , const sc_event& );
void wait( const sc_time& , sc_event_or_list†& );
void wait( double , sc_time_unit , sc_event_or_list†& );
void wait( const sc_time& , sc_event_and_list†& );
void wait( double , sc_time_unit , sc_event_and_list†& );

virtual void before_end_of_elaboration();
virtual void end_of_elaboration();
virtual void start_of_simulation();
virtual void end_of_simulation();
};

void next_trigger();
void next_trigger( const sc_event& );
void next_trigger( sc_event_or_list†& );
void next_trigger( sc_event_and_list†& );
void next_trigger( const sc_time& );
void next_trigger( double , sc_time_unit );
void next_trigger( const sc_time& , const sc_event& );
void next_trigger( double , sc_time_unit , const sc_event& );
void next_trigger( const sc_time& , sc_event_or_list†& );
void next_trigger( double , sc_time_unit , sc_event_or_list†& );
void next_trigger( const sc_time& , const sc_event_and_list†& );
void next_trigger( double , sc_time_unit , sc_event_and_list†& );

void wait();
void wait( int );
void wait( const sc_event& );
void wait( sc_event_or_list†& );
void wait( sc_event_and_list†& );
void wait( const sc_time& );
void wait( double , sc_time_unit );
void wait( const sc_time& , const sc_event& );
void wait( double , sc_time_unit , const sc_event& );
void wait( const sc_time& , sc_event_or_list†& );
void wait( double , sc_time_unit , sc_event_or_list†& );
void wait( const sc_time& , sc_event_and_list†& );
void wait( double , sc_time_unit , sc_event_and_list†& );

#define SC_MODULE(name)      struct name : sc_module
#define SC_CTOR(name)        implementation-defined; name(sc_module_name)
#define SC_HAS_PROCESS(name) implementation-defined
#define SC_METHOD(name)      implementation-defined

```

```

#define SC_THREAD(name)      implementation-defined
#define SC_CTHREAD(name,clk) implementation-defined

const char* sc_gen_unique_name( const char* );

typedef sc_module sc_behavior;
typedef sc_module sc_channel;

}      // namespace sc_core

```

5.2.3 Constraints on usage

Objects of class **sc_module** can only be constructed during elaboration. It shall be an error to instantiate a module during simulation.

Every class derived (directly or indirectly) from class **sc_module** shall have at least one constructor. Every such constructor shall have one and only one parameter of class **sc_module_name** but may have further parameters of classes other than **sc_module_name**. That parameter is not required to be the first parameter of the constructor.

A string-valued argument shall be passed to the constructor of every module instance. It is good practice to make this string name the same as the C++ variable name through which the module is referenced, if such a variable exists.

Inter-module communication should typically be accomplished using interface method calls, that is, a module should communicate with its environment through its ports. Other communication mechanisms are permissible, for example, for debugging or diagnostic purposes.

NOTE 1—Because the constructors are protected, class **sc_module** cannot be instantiated directly but may be used as a base class.

NOTE 2—A module should be *publicly* derived from class **sc_module**.

NOTE 3—It is permissible to use class **sc_module** as an indirect base class. In other words, a module can be derived from another module. This can be a useful coding idiom.

5.2.4 kind

Member function **kind** shall return the string "**sc_module**".

5.2.5 SC_MODULE

The macro **SC_MODULE** may be used to prefix the definition of a module, but the use of this macro is not obligatory.

Example:

// The following two class definitions are equally acceptable.

```

SC_MODULE(M)
{
    M(sc_module_name) {}
    ...
};

```

```
class M: public sc_module
{
    public:
        M(sc_module_name) {}
    ...
};
```

5.2.6 Constructors

```
sc_module( const sc_module_name& );
sc_module();
```

Module names are managed by class **sc_module_name**, not by class **sc_module**. The string name of the module instance is initialized using the value of the string name passed as an argument to the constructor of the class **sc_module_name** (see 5.3).

5.2.7 SC_CTOR

This macro is provided for convenience when declaring or defining a constructor of a module. Macro SC_CTOR shall only be used at a place where the rules of C++ permit a constructor to be declared and can be used as the declarator of a constructor declaration or a constructor definition. The name of the module class being constructed shall be passed as the argument to the macro.

Example:

```
SC_MODULE(M1)
{
    SC_CTOR(M1) // Constructor definition
    : i(0)
    {}
    int i;
    ...
};

SC_MODULE(M2)
{
    SC_CTOR(M2); // Constructor declaration
    int i;
    ...
};

M2::M2(sc_module_name) : i(0) {}
```

The use of macro SC_CTOR is not obligatory. Using SC_CTOR, it is not possible to add user-defined arguments to the constructor. If an application needs to pass additional arguments, the constructor shall be provided explicitly. This is a useful coding idiom.

NOTE 1—The macros SC_CTOR and SC_MODULE may be used in conjunction or may be used separately.

NOTE 2—Since macro SC_CTOR is equivalent to declaring a constructor for a module, an implementation shall ensure that the constructor so declared has a parameter of type **sc_module_name**.

NOTE 3—If process macros are invoked but macro SC_CTOR is not used, macro SC_HAS_PROCESS shall be used instead (see 5.2.8).