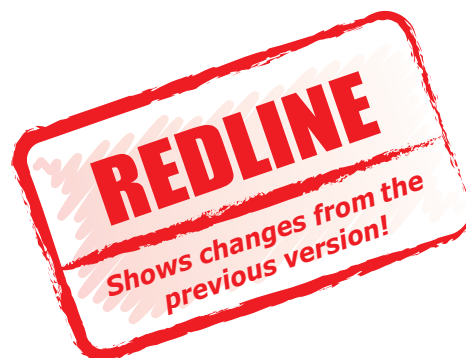


IEEE Standard for Standard SystemC[®] Language Reference Manual

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee



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

9 January 2012

IEEE Std 1666[™]-2011
(Revision of
IEEE Std 1666-2005)

This is a Redline Document produced by Techstreet, a business of Thomson Reuters.

This document is intended to provide users with an indication of changes from one edition to the next. It includes a full-text version of the new document, plus an indication of changes from the previous version.

Redlines are designed to save time and improve efficiencies by using the latest software technology to find and highlight document changes. More professionals are using valuable new technologies like redlines, to help improve outcomes in a fastpaced global business world.

Because it may not be technically possible to capture all changes accurately, it is recommended that users consult previous editions as appropriate. In all cases, only the current base version of this publication is to be considered the official document.

Redline Processing Notes:

1. ~~Red Text~~ - Red strikethrough text denotes deletions.
2. Blue Text - Blue underlined text denotes modifications and additions.

IEEE Standard for Standard SystemC® Language Reference Manual

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 10 September 2011

IEEE-SA Standards Board

Grateful acknowledgment is made to the Open SystemC Initiative (OSCI) for the permission to use the following source material:

— OSCI TLM Language Reference Manual Version 2.0.1

Note that a merger of OSCI and Accellera, announced on 5 December 2011, created a new organization, Accellera Systems Initiative.

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 the users of the class library.

Keywords: C++, computer languages, digital systems, discrete event simulation, electronic design automation, electronic system level, electronic systems, embedded software, fixed-point, hardware description language, hardware design, hardware verification, IEEE 1666, 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 © 2012 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 9 January 2012. 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 in the U.S. Patent & Trademark Office, owned by the Accellera Systems Initiative.

Print: ISBN 978-0-7381-6801-2 STD97162
PDF: ISBN 978-0-7381-6802-9 STDPD97162

IEEE prohibits discrimination, harassment, and bullying. For more information, visit <http://www.ieee.org/web/aboutus/whatis/policies/p9-26.html>.

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 08854-4141
USA

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 1666-2011, IEEE Standard for 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 Accellera Systems Initiative. 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 user's guide or to provide an introduction to SystemC. Readers requiring a SystemC tutorial or information on the intended use of SystemC should consult the Accellera Systems Initiative Web site (www.accellera.org) to locate the many books and training classes available.

Note that the Open Systems Initiative (OSCI) announced in December 2011 its merger with Accellera to form the "Accellera Systems Initiative." The name of the new organization has been used throughout this document instead of the name "Open SystemC Initiative" or the acronym "OSCI." The exceptions are in statements that refer to previous OSCI actions, and standards such as "OSCI TLM-2.0.1," whose names have not been changed.

Notice to users

Laws and regulations

Users of these documents should consult all applicable laws and regulations. Compliance with the provisions of this standard does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Copyrights

This document is copyrighted by the IEEE. It is made available for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making this document available for use and adoption by public authorities and private users, the IEEE does not waive any rights in copyright to this document.

Updating of IEEE documents

Users of IEEE standards should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect. In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE Standards Association website at <http://ieeexplore.ieee.org/xpl/standards.jsp>, or contact the IEEE at the address listed previously.

For more information about the IEEE Standards Association or the IEEE standards development process, visit the IEEE-SA website at <http://standards.ieee.org>.

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: <http://standards.ieee.org/findstds/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/findstds/interps/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:

Stan Krolikoski, *Chair*
Jerome Cornet, *Vice Chair*
John Aynsley, *Technical Lead and Author*
David Long, *Co-Author (SystemC Data Types)*
Dennis Brophy, *Secretary*
Sofie Vandeputte, *Typographical Editor*

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

Accellera Organization	JEITA	STARC
Cadence Design Systems	Mentor Graphics	STMicroelectronics
Freescale Semiconductors	NXP Semiconductors	Synopsys
Intel Corporation	OSCI	Texas Instruments

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

Accellera Organization	Mentor Graphics	STARC
Cadence Design Systems	NXP Semiconductors	STMicroelectronics
Intel Corporation	OSCI	Synopsys
JEITA	Qualcomm Incorporated	Texas Instruments
	SGCC	

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

John Aynsley	Andy Goodrich	Michael McNamara
Bishnupriya Bhattacharya	Philipp A. Hartmann	Mike Meredith
David C. Black	Hiroshi Imai	Eric E. Roesler
Jerome Cornet	Martin Janssen	Stuart Swan
Alan Fitch	Tor Jeremiassen	Bart Vanthournout
Mark Glasser	David Long	Yossi Veller
Puneet Goel		Kaz Yoshinaga

When the IEEE-SA Standards Board approved this standard on 10 September 2011, it had the following membership:

Richard H. Hulett, *Chair*
John Kulick, *Vice Chair*
Robert Grow, *Past Chair*
Judith Gorman, *Secretary*

Masayuki Ariyoshi
William Bartley
Ted Burse
Clint Chaplin
Wael Diab
Jean-Philippe Faure
Alex Gelman
Paul Houzé

Jim Hughes
Joseph L. Koepfinger*
David Law
Thomas Lee
Hung Ling
Oleg Logvinov
Ted Olsen

Gary Robinson
Jon Rosdahl
Sam Sciacca
Mike Seavey
Curtis Siller
Phil Winston
Howard Wolfman
Don Wright

*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*

Michelle D. Turner
IEEE Standards Program Manager, Document Development

Joan Woolery
IEEE Standards Program Manager, Technical Program Development

Contents

1.	Overview.....	1
1.1	Scope.....	1
1.2	Purpose.....	1
1.3	Subsets	2
1.4	Relationship with C++	2
1.5	Guidance for readers	2
2.	Normative references	4
3.	Terminology and conventions used in this standard.....	5
3.1	Terminology.....	5
3.1.1	Shall, should, may, can	5
3.1.2	Implementation, application	5
3.1.3	Call, called from, derived from.....	5
3.1.4	Specific technical terms	5
3.2	Syntactical conventions	7
3.2.1	Implementation-defined.....	7
3.2.2	Disabled	7
3.2.3	Ellipsis (...).	7
3.2.4	Class names.....	7
3.2.5	Embolded text	8
3.3	Semantic conventions	8
3.3.1	Class definitions and the inheritance hierarchy	8
3.3.2	Function definitions and side-effects	8
3.3.3	Functions whose return type is a reference or a pointer	8
3.3.4	Namespaces and internal naming	10
3.3.5	Non-compliant applications and errors.....	11
3.4	Notes and examples	11
4.	Elaboration and simulation semantics	12
4.1	Elaboration.....	12
4.1.1	Instantiation	12
4.1.2	Process macros.....	14
4.1.3	Port binding and export binding	14
4.1.4	Setting the time resolution	15
4.2	Simulation.....	15
4.2.1	The scheduling algorithm	16
4.2.2	Initialization, cycles, and pauses in the scheduling algorithm.....	19
4.3	Running elaboration and simulation	20
4.3.1	Function declarations	20
4.3.2	Function <code>sc_elab_and_sim</code>	20
4.3.3	Functions <code>sc_argc</code> and <code>sc_argv</code>	21
4.3.4	Running under application control using functions <code>sc_main</code> and <code>sc_start</code>	21
4.3.5	Running under control of the kernel	23
4.4	Elaboration and simulation callbacks	23
4.4.1	<code>before_end_of_elaboration</code>	24
4.4.2	<code>end_of_elaboration</code>	25
4.4.3	<code>start_of_simulation</code>	26

4.4.4	end_of_simulation	27
4.5	Other functions related to the scheduler	27
4.5.1	Function declarations	27
4.5.2	Function sc_pause	28
4.5.3	Function sc_stop, sc_set_stop_mode, and sc_get_stop_mode	29
4.5.4	Function sc_time_stamp	30
4.5.5	Function sc_delta_count	31
4.5.6	Function sc_is_running	31
4.5.7	Functions to detect pending activity	31
4.5.8	Function sc_get_status	32
5.	Core language class definitions	35
5.1	Class header files	35
5.1.1	#include "systemc"	35
5.1.2	#include "systemc.h"	35
5.2	sc_module	37
5.2.1	Description	37
5.2.2	Class definition	37
5.2.3	Constraints on usage	39
5.2.4	kind	39
5.2.5	SC_MODULE	40
5.2.6	Constructors	40
5.2.7	SC_CTOR	40
5.2.8	SC_HAS_PROCESS	41
5.2.9	SC_METHOD, SC_THREAD, SC_CTHREAD	42
5.2.10	Method process	43
5.2.11	Thread and clocked thread processes	43
5.2.12	Clocked thread processes	44
5.2.13	reset_signal_is and async_reset_signal_is	46
5.2.14	sensitive	48
5.2.15	dont_initialize	48
5.2.16	set_stack_size	49
5.2.17	next_trigger	50
5.2.18	wait	52
5.2.19	Positional port binding	53
5.2.20	before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation	55
5.2.21	get_child_objects and get_child_events	55
5.2.22	sc_gen_unique_name	56
5.2.23	sc_behavior and sc_channel	56
5.3	sc_module_name	57
5.3.1	Description	57
5.3.2	Class definition	57
5.3.3	Constraints on usage	58
5.3.4	Module hierarchy	58
5.3.5	Member functions	58
5.4	sc_sensitive	60
5.4.1	Description	60
5.4.2	Class definition	60
5.4.3	Constraints on usage	60
5.4.4	operator<<	60
5.5	sc_spawn_options and sc_spawn	61
5.5.1	Description	61

5.5.2	Class definition	61
5.5.3	Constraints on usage	62
5.5.4	Constructors	62
5.5.5	Member functions	63
5.5.6	sc_spawn	64
5.5.7	SC_FORK and SC_JOIN	66
5.6	sc_process_handle	67
5.6.1	Description	67
5.6.2	Class definition	68
5.6.3	Constraints on usage	69
5.6.4	Constructors	69
5.6.5	Member functions	69
5.6.6	Member functions for process control	73
5.6.7	sc_get_current_process_handle	88
5.6.8	sc_is_unwinding	89
5.7	sc_event_finder and sc_event_finder_t	90
5.7.1	Description	90
5.7.2	Class definition	90
5.7.3	Constraints on usage	90
5.8	sc_event_and_list and sc_event_or_list	92
5.8.1	Description	92
5.8.2	Class definition	92
5.8.3	Constraints and usage	93
5.8.4	Constructors, destructor, assignment	93
5.8.5	Member functions and operators	94
5.9	sc_event_and_expr† and sc_event_or_expr†	95
5.9.1	Description	95
5.9.2	Class definition	95
5.9.3	Constraints on usage	96
5.9.4	Operators	96
5.10	sc_event	97
5.10.1	Description	97
5.10.2	Class definition	97
5.10.3	Constraints on usage	98
5.10.4	Constructors, destructor, and event naming	98
5.10.5	Functions for naming and hierarchy traversal	99
5.10.6	notify and cancel	100
5.10.7	Event lists	101
5.10.8	Multiple event notifications	101
5.11	sc_time	101
5.11.1	Description	101
5.11.2	Class definition	101
5.11.3	Time resolution	102
5.11.4	Function sc_max_time	103
5.11.5	Functions and operators	103
5.11.6	SC_ZERO_TIME	103
5.12	sc_port	104
5.12.1	Description	104
5.12.2	Class definition	104
5.12.3	Template parameters	105
5.12.4	Constraints on usage	106
5.12.5	Constructors	107
5.12.6	kind	107
5.12.7	Named port binding	108

5.12.8	Member functions for bound ports and port-to-port binding	109
5.12.9	before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation	112
5.13	sc_export	113
5.13.1	Description	113
5.13.2	Class definition	113
5.13.3	Template parameters	114
5.13.4	Constraints on usage	114
5.13.5	Constructors	114
5.13.6	kind	114
5.13.7	Export binding	115
5.13.8	Member functions for bound exports and export-to-export binding	116
5.13.9	before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation	117
5.14	sc_interface	117
5.14.1	Description	117
5.14.2	Class definition	117
5.14.3	Constraints on usage	118
5.14.4	register_port	118
5.14.5	default_event	119
5.15	sc_prim_channel	119
5.15.1	Description	119
5.15.2	Class definition	119
5.15.3	Constraints on usage	121
5.15.4	Constructors, destructor, and hierarchical names	121
5.15.5	kind	121
5.15.6	request_update and update	121
5.15.7	next_trigger and wait	123
5.15.8	before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation	123
5.16	sc_object	124
5.16.1	Description	124
5.16.2	Class definition	124
5.16.3	Constraints on usage	125
5.16.4	Constructors and destructor	126
5.16.5	name, basename, and kind	126
5.16.6	print and dump	127
5.16.7	Functions for object hierarchy traversal	127
5.16.8	Member functions for attributes	129
5.17	Hierarchical naming of objects and events	130
5.18	sc_attr_base	131
5.18.1	Description	131
5.18.2	Class definition	131
5.18.3	Member functions	132
5.19	sc_attribute	132
5.19.1	Description	132
5.19.2	Class definition	132
5.19.3	Template parameters	132
5.19.4	Member functions and data members	132
5.20	sc_attr_cltn	133
5.20.1	Description	133
5.20.2	Class definition	133
5.20.3	Constraints on usage	133
5.20.4	Iterators	133

6.	Predefined channel class definitions	135
6.1	sc_signal_in_if	135
6.1.1	Description	135
6.1.2	Class definition	135
6.1.3	Member functions	135
6.2	sc_signal_in_if<bool> and sc_signal_in_if<sc_dt::sc_logic>	136
6.2.1	Description	136
6.2.2	Class definition	136
6.2.3	Member functions	137
6.3	sc_signal_inout_if	137
6.3.1	Description	137
6.3.2	Class definition	137
6.3.3	Member functions	138
6.4	sc_signal	139
6.4.1	Description	139
6.4.2	Class definition	139
6.4.3	Template parameter T	140
6.4.4	Reading and writing signals	140
6.4.5	Constructors	141
6.4.6	register_port	141
6.4.7	Member functions for reading	141
6.4.8	Member functions for writing	142
6.4.9	Member functions for events	142
6.4.10	Diagnostic member functions	143
6.4.11	operator<<	143
6.5	sc_signal<bool,WRITER_POLICY> and sc_signal<sc_dt::sc_logic,WRITER_POLICY> ..	144
6.5.1	Description	144
6.5.2	Class definition	144
6.5.3	Member functions	146
6.6	sc_buffer	147
6.6.1	Description	147
6.6.2	Class definition	147
6.6.3	Constructors	147
6.6.4	Member functions	148
6.7	sc_clock	149
6.7.1	Description	149
6.7.2	Class definition	149
6.7.3	Characteristic properties	150
6.7.4	Constructors	150
6.7.5	write	151
6.7.6	Diagnostic member functions	151
6.7.7	before_end_of_elaboration	151
6.7.8	sc_in_clk	151
6.8	sc_in	152
6.8.1	Description	152
6.8.2	Class definition	152
6.8.3	Member functions	153
6.8.4	Function sc_trace	153
6.8.5	end_of_elaboration	153
6.9	sc_in<bool> and sc_in<sc_dt::sc_logic>	153
6.9.1	Description	153
6.9.2	Class definition	154
6.9.3	Member functions	155

6.10	sc_inout	156
6.10.1	Description	156
6.10.2	Class definition	156
6.10.3	Member functions	157
6.10.4	initialize	157
6.10.5	Function sc_trace	157
6.10.6	end_of_elaboration	158
6.10.7	Binding	158
6.11	sc_inout<bool> and sc_inout<sc_dt::sc_logic>	158
6.11.1	Description	158
6.11.2	Class definition	158
6.11.3	Member functions	160
6.12	sc_out	160
6.12.1	Description	160
6.12.2	Class definition	160
6.12.3	Member functions	161
6.13	sc_signal_resolved	161
6.13.1	Description	161
6.13.2	Class definition	161
6.13.3	Constructors	162
6.13.4	Resolution semantics	162
6.13.5	Member functions	163
6.14	sc_in_resolved	164
6.14.1	Description	164
6.14.2	Class definition	164
6.14.3	Member functions	165
6.15	sc_inout_resolved	165
6.15.1	Description	165
6.15.2	Class definition	165
6.15.3	Member functions	166
6.16	sc_out_resolved	166
6.16.1	Description	166
6.16.2	Class definition	166
6.16.3	Member functions	167
6.17	sc_signal_rv	167
6.17.1	Description	167
6.17.2	Class definition	167
6.17.3	Semantics and member functions	167
6.18	sc_in_rv	168
6.18.1	Description	168
6.18.2	Class definition	168
6.18.3	Member functions	169
6.19	sc_inout_rv	169
6.19.1	Description	169
6.19.2	Class definition	169
6.19.3	Member functions	170
6.20	sc_out_rv	170
6.20.1	Description	170
6.20.2	Class definition	170
6.20.3	Member functions	171
6.21	sc_fifo_in_if	171
6.21.1	Description	171
6.21.2	Class definition	171
6.21.3	Member functions	172

6.22	sc_fifo_out_if.....	172
6.22.1	Description.....	172
6.22.2	Class definition	172
6.22.3	Member functions	173
6.23	sc_fifo	173
6.23.1	Description.....	173
6.23.2	Class definition	173
6.23.3	Template parameter T	174
6.23.4	Constructors	175
6.23.5	register_port	175
6.23.6	Member functions for reading	175
6.23.7	Member functions for writing.....	176
6.23.8	The update phase	176
6.23.9	Member functions for events	177
6.23.10	Member functions for available values and free slots	177
6.23.11	Diagnostic member functions	177
6.23.12	operator<<.....	177
6.24	sc_fifo_in	178
6.24.1	Description.....	178
6.24.2	Class definition	178
6.24.3	Member functions	179
6.25	sc_fifo_out	179
6.25.1	Description.....	179
6.25.2	Class definition	179
6.25.3	Member functions	180
6.26	sc_mutex_if.....	182
6.26.1	Description.....	182
6.26.2	Class definition	182
6.26.3	Member functions	182
6.27	sc_mutex	182
6.27.1	Description.....	182
6.27.2	Class definition	182
6.27.3	Constructors	183
6.27.4	Member functions	183
6.28	sc_semaphore_if	184
6.28.1	Description.....	184
6.28.2	Class definition	184
6.28.3	Member functions	184
6.29	sc_semaphore.....	185
6.29.1	Description.....	185
6.29.2	Class definition	185
6.29.3	Constructors	185
6.29.4	Member functions	185
6.30	sc_event_queue.....	186
6.30.1	Description.....	186
6.30.2	Class definition	186
6.30.3	Constraints on usage	187
6.30.4	Constructors	187
6.30.5	kind	187
6.30.6	Member functions	187
7.	SystemC data types	189
7.1	Introduction.....	189

7.2	Common characteristics.....	191
7.2.1	Initialization and assignment operators	192
7.2.2	Precision of arithmetic expressions	193
7.2.3	Base class default word length.....	193
7.2.4	Word length	194
7.2.5	Bit-select	194
7.2.6	Part-select.....	195
7.2.7	Concatenation	196
7.2.8	Reduction operators	197
7.2.9	Integer conversion.....	198
7.2.10	String input and output	198
7.2.11	Conversion of application-defined types in integer expressions	199
7.3	String literals.....	199
7.4	<i>sc_value_base</i>	201
7.4.1	Description.....	201
7.4.2	Class definition	201
7.4.3	Constraints on usage	201
7.4.4	Member functions	202
7.5	Limited-precision integer types	202
7.5.1	Type definitions	202
7.5.2	<i>sc_int_base</i>	203
7.5.3	<i>sc_uint_base</i>	208
7.5.4	<i>sc_int</i>	213
7.5.5	<i>sc_uint</i>	215
7.5.6	Bit-selects.....	217
7.5.7	Part-selects	222
7.6	Finite-precision integer types.....	227
7.6.1	Type definitions	227
7.6.2	Constraints on usage	227
7.6.3	<i>sc_signed</i>	227
7.6.4	<i>sc_unsigned</i>	234
7.6.5	<i>sc_bigint</i>	240
7.6.6	<i>sc_biguint</i>	242
7.6.7	Bit-selects.....	244
7.6.8	Part-selects	248
7.7	Integer concatenations	253
7.7.1	Description.....	253
7.7.2	Class definition	253
7.7.3	Constraints on usage	255
7.7.4	Assignment operators	255
7.7.5	Implicit type conversion	255
7.7.6	Explicit type conversion	255
7.7.7	Other member functions	256
7.8	Generic base proxy class.....	256
7.8.1	Description.....	256
7.8.2	Class definition	256
7.8.3	Constraints on usage	256
7.9	Logic and vector types.....	257
7.9.1	Type definitions	257
7.9.2	<i>sc_logic</i>	257
7.9.3	<i>sc_bv_base</i>	262
7.9.4	<i>sc_lv_base</i>	267
7.9.5	<i>sc_bv</i>	273
7.9.6	<i>sc_lv</i>	275

7.9.7	Bit-selects.....	277
7.9.8	Part-selects.....	280
7.9.9	Concatenations.....	286
7.10	Fixed-point types.....	293
7.10.1	Fixed-point representation.....	293
7.10.2	Fixed-point type conversion.....	294
7.10.3	Fixed-point data types.....	295
7.10.4	Fixed-point expressions and operations.....	296
7.10.5	Bit and part selection.....	299
7.10.6	Variable-precision fixed-point value limits.....	300
7.10.7	Fixed-point word length and mode.....	300
7.10.8	Conversions to character string.....	302
7.10.9	Finite word-length effects.....	304
7.10.10	sc_fxnum.....	327
7.10.11	sc_fxnum_fast.....	332
7.10.12	sc_fxval.....	337
7.10.13	sc_fxval_fast.....	341
7.10.14	sc_fix.....	346
7.10.15	sc_ufix.....	349
7.10.16	sc_fix_fast.....	352
7.10.17	sc_ufix_fast.....	355
7.10.18	sc_fixed.....	358
7.10.19	sc_ufixed.....	360
7.10.20	sc_fixed_fast.....	362
7.10.21	sc_ufixed_fast.....	365
7.10.22	Bit-selects.....	367
7.10.23	Part-selects.....	369
7.11	Contexts.....	375
7.11.1	sc_length_param.....	375
7.11.2	sc_length_context.....	377
7.11.3	sc_fxtype_params.....	378
7.11.4	sc_fxtype_context.....	380
7.11.5	sc_fxcast_switch.....	381
7.11.6	sc_fxcast_context.....	382
7.12	Control of string representation.....	383
7.12.1	Description.....	383
7.12.2	Class definition.....	383
7.12.3	Functions.....	384
8.	SystemC utilities.....	385
8.1	Trace files.....	385
8.1.1	Class definition and function declarations.....	385
8.1.2	sc_trace_file.....	385
8.1.3	sc_create_vcd_trace_file.....	386
8.1.4	sc_close_vcd_trace_file.....	386
8.1.5	sc_write_comment.....	386
8.1.6	sc_trace.....	386
8.2	sc_report.....	388
8.2.1	Description.....	388
8.2.2	Class definition.....	388
8.2.3	Constraints on usage.....	389
8.2.4	sc_verbosity.....	389
8.2.5	sc_severity.....	389

8.2.6	Copy constructor and assignment	390
8.2.7	Member functions	390
8.3	sc_report_handler.....	391
8.3.1	Description	391
8.3.2	Class definition	391
8.3.3	Constraints on usage	393
8.3.4	sc_actions.....	393
8.3.5	report.....	393
8.3.6	set_actions.....	394
8.3.7	stop_after	395
8.3.8	get_count.....	396
8.3.9	Verbosity level.....	396
8.3.10	suppress and force.....	396
8.3.11	set_handler	397
8.3.12	get_new_action_id	398
8.3.13	sc_interrupt_here and sc_stop_here.....	398
8.3.14	get_cached_report and clear_cached_report.....	398
8.3.15	set_log_file_name and get_log_file_name	399
8.4	sc_exception.....	399
8.4.1	Description	399
8.4.2	Class definition	399
8.5	sc_vector	400
8.5.1	Description	400
8.5.2	Class definition	400
8.5.3	Constraints on usage	403
8.5.4	Constructors and destructors.....	403
8.5.5	init and create_element	404
8.5.6	kind, size, get_elements	405
8.5.7	operator[] and at.....	406
8.5.8	Iterators	406
8.5.9	bind	406
8.5.10	sc_assemble_vector	408
8.6	Utility functions	410
8.6.1	Function declarations	410
8.6.2	sc_abs.....	411
8.6.3	sc_max	411
8.6.4	sc_min.....	411
8.6.5	Version and copyright.....	411
9.	Overview of TLM-2.0.....	413
9.1	Compliance with the TLM-2.0 standard.....	414
10.	Introduction to TLM-2.0.....	415
10.1	Background.....	415
10.2	Transaction-level modeling, use cases, and abstraction	415
10.3	Coding styles.....	416
10.3.1	Untimed coding style	416
10.3.2	Loosely-timed coding style and temporal decoupling	417
10.3.3	Synchronization in loosely-timed models.....	418
10.3.4	Approximately-timed coding style	418
10.3.5	Characterization of loosely-timed and approximately-timed coding styles	419
10.3.6	Switching between loosely-timed and approximately-timed modeling	419

10.3.7	Cycle-accurate modeling	419
10.3.8	Blocking versus non-blocking transport interfaces	419
10.3.9	Use cases and coding styles	420
10.4	Initiators, targets, sockets, and transaction bridges	420
10.5	DMI and debug transport interfaces	423
10.6	Combined interfaces and sockets	423
10.7	Namespaces	423
10.8	Header files and version numbers	424
10.8.1	Software version information	424
10.8.2	Definitions	424
10.8.3	Rules	425
11.	TLM-2.0 core interfaces	426
11.1	Transport interfaces	426
11.1.1	Blocking transport interface	426
11.1.2	Non-blocking transport interface	430
11.1.3	Timing annotation with the transport interfaces	438
11.1.4	Migration path from TLM-1	442
11.2	Direct memory interface	442
11.2.1	Introduction	442
11.2.2	Class definition	443
11.2.3	get_direct_mem_ptr method	444
11.2.4	template argument and tlm_generic_payload class	445
11.2.5	tlm_dmi class	445
11.2.6	invalidate_direct_mem_ptr method	448
11.2.7	DMI versus transport	449
11.2.8	DMI and temporal decoupling	449
11.2.9	Optimization using a DMI hint	450
11.3	Debug transport interface	450
11.3.1	Introduction	450
11.3.2	Class definition	450
11.3.3	TRANS template argument and tlm_generic_payload class	451
11.3.4	Rules	451
12.	TLM-2.0 global quantum	453
12.1	Introduction	453
12.2	Header file	453
12.3	Class definition	453
12.4	Class tlm_global_quantum	454
13.	Combined TLM-2.0 interfaces and sockets	455
13.1	Combined interfaces	455
13.1.1	Introduction	455
13.1.2	Class definition	455
13.2	Initiator and target sockets	456
13.2.1	Introduction	456
13.2.2	Class definition	456
13.2.3	Classes tlm_base_initiator_socket_b and tlm_base_target_socket_b	460
13.2.4	Classes tlm_base_initiator_socket and tlm_base_target_socket	460
13.2.5	Classes tlm_initiator_socket and tlm_target_socket	461

14.	TLM-2.0 generic payload	465
14.1	Introduction.....	465
14.2	Extensions and interoperability	465
14.2.1	Use the generic payload directly, with ignorable extensions.....	466
14.2.2	Define a new protocol traits class containing a typedef for tlm_generic_payload.....	467
14.2.3	Define a new protocol traits class and a new transaction type	467
14.3	Generic payload attributes and methods	467
14.4	Class definition	468
14.5	Generic payload memory management	470
14.6	Constructors, assignment, and destructor	474
14.7	Default values and modifiability of attributes	474
14.8	Option attribute	476
14.9	Command attribute	477
14.10	Address attribute	478
14.11	Data pointer attribute	479
14.12	Data length attribute.....	480
14.13	Byte enable pointer attribute.....	480
14.14	Byte enable length attribute	481
14.15	Streaming width attribute.....	482
14.16	DMI allowed attribute.....	482
14.17	Response status attribute.....	483
14.17.1	The standard error response	484
14.18	Endianness	487
14.18.1	Introduction.....	487
14.18.2	Rules	488
14.19	Helper functions to determine host endianness	490
14.19.1	Introduction.....	490
14.19.2	Definition	490
14.19.3	Rules	491
14.20	Helper functions for endianness conversion	491
14.20.1	Introduction.....	491
14.20.2	Definition	492
14.20.3	Rules	492
14.21	Generic payload extensions	494
14.21.1	Introduction.....	494
14.21.2	Rationale	494
14.21.3	Extension pointers, objects and transaction bridges	495
14.21.4	Rules	495
15.	TLM-2.0 base protocol and phases	500
15.1	Phases.....	500
15.1.1	Introduction.....	500
15.1.2	Class definition	500
15.1.3	Rules	501
15.2	Base protocol	502
15.2.1	Introduction.....	502
15.2.2	Class definition	503
15.2.3	Base protocol phase sequences	503
15.2.4	Permitted phase transitions	505
15.2.5	Ignorable phases	508
15.2.6	Base protocol timing parameters and flow control.....	510
15.2.7	Base protocol rules concerning timing annotation	514

15.2.8	Base protocol rules concerning b_transport.....	514
15.2.9	Base protocol rules concerning request and response ordering.....	515
15.2.10	Base protocol rules for switching between b_transport and nb_transport.....	516
15.2.11	Other base protocol rules	517
15.2.12	Summary of base protocol transaction ordering rules	517
15.2.13	Guidelines for creating base-protocol-compliant components	517
16.	TLM-2.0 utilities.....	521
16.1	Convenience sockets.....	521
16.1.1	Introduction.....	521
16.1.2	Simple sockets	523
16.1.3	Tagged simple sockets	529
16.1.4	Multi-sockets	532
16.2	Quantum keeper.....	537
16.2.1	Introduction.....	537
16.2.2	Header file.....	537
16.2.3	Class definition	537
16.2.4	General guidelines for processes using temporal decoupling.....	538
16.2.5	Class tlm_quantumkeeper.....	539
16.3	Payload event queue	541
16.3.1	Introduction.....	541
16.3.2	Header file.....	542
16.3.3	Class definition	542
16.3.4	Rules	543
16.4	Instance-specific extensions	544
16.4.1	Introduction.....	544
16.4.2	Header file.....	544
16.4.3	Class definition	544
17.	TLM-1 Message passing interface and analysis ports	546
17.1	Put, get, peek, and transport interfaces	546
17.1.1	Description.....	546
17.1.2	Class Definition	546
17.1.3	Blocking versus non-blocking interfaces.....	548
17.1.4	Blocking put, get, peek, and transport	549
17.1.5	Non-blocking interface methods.....	549
17.1.6	Argument passing and transaction lifetime	550
17.1.7	Constraints on the transaction data type	551
17.2	TLM-1 fifo interfaces	551
17.2.1	Description.....	551
17.2.2	Class Definition	551
17.2.3	Member functions	552
17.3	tlm_fifo	552
17.3.1	Description.....	552
17.3.2	Class Definition	553
17.3.3	Template parameter T.....	554
17.3.4	Constructors and destructor	554
17.3.5	Member functions	554
17.3.6	Delta cycle semantics.....	556
17.4	Analysis interface and analysis ports.....	558
17.4.1	Class definition	558
17.4.2	Rules	559

Annex A (informative) Introduction to SystemC	562
Annex B (informative) Glossary.....	566
Annex C (informative) Deprecated features	583
Annex D (informative) Changes between IEEE Std 1666-2005 and IEEE Std 1666-2011	585
Index	589

IEEE Standard for Standard SystemC[®] Language

Reference Manual

IMPORTANT NOTICE: This standard is not intended to ensure safety, security, health, or environmental protection. Implementers of the standard are responsible for determining appropriate safety, security, environmental, and health practices or regulatory requirements.

This IEEE document is made available for use subject to important notices and legal disclaimers. These notices and disclaimers appear in all publications containing this document and may be found under the heading "Important Notice" or "Important Notices and Disclaimers Concerning IEEE Documents." They can also be obtained on request from IEEE or viewed at <http://standards.ieee.org/IPR/disclaimers.html>.

1. Overview

1.1 Scope

This standard defines SystemC^{®1} with Transaction Level Modeling (TLM) as an ANSI standard C++ class library for system and hardware design.

1.2 Purpose

The general purpose of ~~SystemC~~ [this standard](#) 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 [including a TLM library](#) so that a SystemC implementation can be developed with reference to this standard alone. This standard is not intended to serve as a user's guide or to provide an introduction to SystemC, but it does contain useful information for end users.

¹SystemC[®] is a registered trademark of the Accellera Systems Initiative.

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. [See also 9.1 for a description of TLM-2.0-compliance.](#)

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.

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.³

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 ~~up~~ 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

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

³Notes in text, tables, and figures are given for information only and do not contain requirements needed to implement the standard.

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.

The clauses from Clause 9 onward define the public interface to the TLM-2.0 class library, including the classes of the interoperability layer and the utilities.

Clause 17 defines the TLM-1 Message passing interface, including **tlm fifo** and analysis ports.

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~~ [IEEE Std 1666-2005](#) and ~~version 2.1 Beta Oct 12 2004~~, and the changes between ~~SystemC 2.1 Beta Oct 12 2004~~ and this standard [IEEE 1666-2011](#).

2. Normative references

The following referenced documents are indispensable for the application of this document (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced 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}

⁴ISO/IEC publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, 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 it 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, [TLM-1](#), and [TLM-2.0](#) class libraries 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, [TLM-1](#), and [TLM-2.0](#) class libraries, 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 that 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 the class **sc_port**. An *export* is an object of the 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 it 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 it 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 the 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 5.16). Events do not belong to the object hierarchy, although like objects of the class **sc_object**, events may have a hierarchical name.

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.

The term *during elaboration* includes the construction of the module hierarchy and the before end of elaboration callbacks.

The term *during elaboration or simulation* includes every phase from the construction of the module hierarchy up to and including the final delta cycle, but neither includes nor excludes activity subsequent to the final delta cycle. In other words, use of this term infers nothing about whether specific actions are or are not permitted after the final delta cycle.

The term *during simulation* includes the initialization, evaluation, and update phases and any period when simulation is paused.

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_hist[†]exp_{r[†]}</i>	<i>sc_int_subref_r[†]</i>	<i>sc_uint_bitref[†]</i>	<u>sc_vector_iter[†]</u>
<i>sc_event_or_expr[†]</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 `it` 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 `const char*`

Certain functions have the return type `const 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

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

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:

The implementation is obliged to ensure that the returned object is valid either until 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

- a) sc_object *
- b) sc_event *
- c) [sc_attr_base](#) *
- d) std::string& // Property of an attribute 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:

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\_event::get\_parent\_object\(\) const;
```

```

sc_object* sc_find_object( const char* );
sc\_event\* sc\_find\_event\( const char\* \);
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_event](#), **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) [std::vector<sc_event*> &c](#)) `sc_attr_cltn *`

The functions concerned are the following:

```

virtual const std::vector<sc_object*>& sc_module::get_child_objects() const; virtual const
std::vector<sc\_event\*>& sc\_module::get\_child\_events\(\) const;
const std::vector<sc_object*>& sc_process_handle::get_child_objects() const;
const std::vector<sc\_event\*>& sc\_process\_handle::get\_child\_events\(\) const;
virtual const std::vector<sc_object*>& sc_object::get_child_objects() const;
virtual const std::vector<sc\_event\*>& sc\_object::get\_child\_events\(\) const;
const std::vector<sc_object*>& sc_get_top_level_objects();
const std::vector<sc\_event\*>& sc\_get\_top\_level\_events\(\);
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**, [an sc_event](#), 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, [with the one exception of sc_main](#), within one of the ~~two~~[five](#) namespaces ~~sc_core and~~, [sc_dt](#), [sc_unnamed](#), [tlm](#), and [tlm_utils](#). 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 [SystemC](#) utilities are divided between the two namespaces [sc_core](#) and [sc_dt](#).

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 the 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, the 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** cannot 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 5.12)

`sc_export` (see 5.13)

`sc_prim_channel` (see 5.15)

An implementation shall permit an application to have zero or one top-level module 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~~ from function **sc_main** (or a function called from **sc_main**), or in the absence of an **sc_main** (see 4.3.5), in the form of one or more top-level modules. 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**~~ as described above, 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*^t (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, [they](#) have some member functions in common (see 5.16).

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
    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 5.12.7). Positional port binding is performed by a member function of class **sc_module** (see 5.2.19). Exports can only be bound by name. Export binding is performed by a member function of class **sc_export** (see 5.13.7).

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.