IEEE SA
STANDARDS ASSOCIATION

STANDARDS

IEEE Standard for Standard SystemC® Language Reference Manual

IEEE Computer Society

Developed by the
Design Automation Standards Committee

IEEE Std 1666™-2023
(Revision of IEEE Std 1666-2011)

IEEE

Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

IEEE Std 1666™-2023
(Revision of IEEE Std 1666-2011)

IEEE Standard for Standard SystemC® Language Reference Manual

Developed by

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 5 June 2023

IEEE SA Standards Board

Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Abstract: SystemC® is defined in this standard. SystemC is an ISO 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 © 2023 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 8 September 2023. 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 978-1-5044-9867-8 STD26278
PDF: ISBN 978-1-5044-9868-5 STDPD26278

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.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Important Notices and Disclaimers Concerning IEEE Standards Documents

IEEE Standards documents are made available for use subject to important notices and legal disclaimers. These notices and disclaimers, or a reference to this page (https://standards.ieee.org/ipr/disclaimers.html), appear in all standards and may be found under the heading "Important Notices and Disclaimers Concerning IEEE Standards Documents."

Notice and Disclaimer of Liability Concerning the Use of IEEE Standards Documents

IEEE Standards documents are developed within IEEE Societies and subcommittees of IEEE Standards Association (IEEE SA) Board of Governors. IEEE develops its standards through an accredited consensus development process, which brings together volunteers representing varied viewpoints and interests to achieve the final product. IEEE Standards are documents developed by volunteers with scientific, academic, and industry-based expertise in technical working groups. Volunteers are not necessarily members of IEEE or IEEE SA and participate without compensation from IEEE. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

IEEE makes no warranties or representations concerning its standards, and expressly disclaims all warranties, express or implied, concerning this standard, including but not limited to the warranties of merchantability, fitness for a particular purpose and non-infringement. In addition, IEEE does not warrant or represent that the use of the material contained in its standards is free from patent infringement. IEEE Standards documents are supplied "AS IS" and "WITH ALL FAULTS."

Use of an IEEE standard is wholly voluntary. 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.

In publishing and making its standards available, IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity, nor is IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing any IEEE Standards document, should rely upon his or her own independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

IN NO EVENT SHALL IEEE BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO: THE NEED TO PROCURE SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE PUBLICATION, USE OF, OR RELIANCE UPON ANY STANDARD, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE AND REGARDLESS OF WHETHER SUCH DAMAGE WAS FORESEEABLE.

Translations

The IEEE consensus development process involves the review of documents in English only. In the event that an IEEE standard is translated, only the English version published by IEEE is the approved IEEE standard.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Official statements

A statement, written or oral, that is not processed in accordance with the IEEE SA Standards Board Operations Manual shall not be considered or inferred to be the official position of IEEE or any of its committees and shall not be considered to be, nor be relied upon as, a formal position of IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that the presenter's views should be considered the personal views of that individual rather than the formal position of IEEE, IEEE SA, the Standards Committee, or the Working Group.

Comments on standards

Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE or IEEE SA. However, **IEEE does not provide interpretations, consulting information, or advice pertaining to IEEE Standards documents**.

Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important that any responses to comments and questions also receive 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 comments, or questions except in those cases where the matter has previously been addressed. For the same reason, IEEE does not respond to interpretation requests. Any person who would like to participate in evaluating comments or in revisions to an IEEE standard is welcome to join the relevant IEEE working group. You can indicate interest in a working group using the Interests tab in the Manage Profile & Interests area of the IEEE SA myProject system. An IEEE Account is needed to access the application.

Comments on standards should be submitted using the Contact Us form.

Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document does not constitute 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.

Data privacy

Users of IEEE Standards documents should evaluate the standards for considerations of data privacy and data ownership in the context of assessing and using the standards in compliance with applicable laws and regulations.

Copyrights

IEEE draft and approved standards are copyrighted by IEEE under U.S. and international copyright laws. They are made available by IEEE and are adopted 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 these documents available for use and adoption by public authorities and private users, IEEE does not waive any rights in copyright to the documents.

Photocopies

Subject to payment of the appropriate licensing fees, IEEE will grant users a limited, non-exclusive license to photocopy portions of any individual standard for company or organizational internal use or individual, non-commercial use only. To arrange for payment of licensing fees, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400; https://www.copyright.com/. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Updating of IEEE Standards documents

Users of IEEE Standards documents 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.

Every IEEE standard is subjected to review at least every 10 years. When a document is more than 10 years old and has not undergone a revision process, 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 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 IEEE Xplore or contact IEEE. For more information about the IEEE SA or IEEE's standards development process, visit the IEEE SA Website.

Errata

Errata, if any, for all IEEE standards can be accessed on the IEEE SA Website. Search for standard number and year of approval to access the web page of the published standard. Errata links are located under the Additional Resources Details section. Errata are also available in IEEE Xplore. Users are encouraged to periodically check for errata.

Patents

IEEE Standards are developed in compliance with the IEEE SA Patent Policy.

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 by the IEEE with respect to the existence or validity of any patent rights in connection therewith. If a patent holder or patent applicant has filed a statement of assurance via an Accepted Letter of Assurance, then the statement is listed on the IEEE SA Website at https://standards.ieee.org/about/sasb/patcom/patents.html. Letters of Assurance may indicate whether the Submitter is willing or unwilling to grant licenses under patent rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses.

Essential Patent Claims may exist for which a Letter of Assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

IMPORTANT NOTICE

IEEE Standards do not guarantee or ensure safety, security, health, or environmental protection, or ensure against interference with or from other devices or networks. IEEE Standards development activities consider research and information presented to the standards development group in developing any safety recommendations. Other information about safety practices, changes in technology or technology implementation, or impact by peripheral systems also may be pertinent to safety considerations during implementation of the standard. Implementers and users of IEEE Standards documents are responsible for determining and complying with all appropriate safety, security, environmental, health, and interference protection practices and all applicable laws and regulations.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Introduction

This introduction is not part of IEEE Std 1666-2023, 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.

Acknowledgments

Contributed updates to IEEE Std 1666-2011 from the Accellera Systems Initiative SystemC Language Working Group are acknowledged.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Participants

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

Jérôme Cornet, *Chair*
Praveen Wadikar, *Vice Chair*
Mike Meredith, *Secretary*
Diane Lacey, *Technical Editor*

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

Organization Represented	*Name of Representative*
Accellera Systems Initiative, Inc..	Andy Goodrich
Analog Devices	Isaac Molina
ARM, Ltd.	Rohit Kurup
Cadence Design Systems, Inc.	Mike Meredith
Intel Corporation	Eric Roesler
NVIDIA Corporation	Praveen Wadikar
NXP Semiconductors	Martin Barnasconi
Siemens Corporation	Stuart Swan
STMicroelectronic	Jérôme Cornet
Synopsys, Inc.	Bart Vanthournout

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

Martin Barnasconi
David Black
Mark Burton
Jérôme Cornet
Karsten Einwich
Mark Glasser
Andy Goodrich
Philipp A. Hartmann
Peter de Jager
Laurent Maillet-Contoz
Mike Meredith
Roman Popov
Eric Roesler
Stuart Swan
Bart Vanthournout
Praveen Wadikar

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

Accellera Systems Initiative, Inc.
Cadence Design Systems, Inc.
Far East Smarter Energy Co., Ltd.
IBM
Institute of Biomedical Engineering
Intel Corporation
NVIDIA Corporation
NXP Semiconductors
Siemens Corporation
STMicroelectronics
Synopsys, Inc.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

When the IEEE SA Standards Board approved this standard on 5 June 2023, it had the following membership:

David J. Law, *Chair*
Ted Burse, *Vice Chair*
Gary Hoffman, *Past Chair*
Konstantinos Karachalios, *Secretary*

Sara R. Biyabani	Joseph S. Levy	Paul Nikolich
Doug Edwards	Howard Li	Annette D. Reilly
Ramy Ahmed Fathy	Johnny Daozhuang Lin	Robby Robson
Guido R. Hiertz	Gui Lin	Lei Wang
Yousef Kimiagar	Xiaohui Liu	F. Keith Waters
Joseph L. Koepfinger*	Kevin W. Lu	Karl Weber
Thomas Koshy	Daleep C. Mohla	Philip B. Winston
John D. Kulick	Andrew Myles	Don Wright

*Member Emeritus

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Contents

1. Overview 23

1.1 Scope 23
1.2 Purpose 23
1.3 Word usage 23
1.4 Subsets 24
1.5 Relationship with C++ standard 24
1.6 Guidance for readers 24

2. Normative references 26

3. Terminology and conventions used in this standard 27

3.1 Terminology 27
3.1.1 Shall, should, may, can 27
3.1.2 Implementation, application 27
3.1.3 Call, called from, derived from 27
3.1.4 Specific technical terms 27
3.2 Syntactical conventions 29
3.2.1 Implementation-defined 29
3.2.2 Disabled 29
3.2.3 Ellipsis (...) 29
3.2.4 Class names 29
3.2.5 Embolded text 30
3.3 Semantic conventions 30
3.3.1 Class definitions and the inheritance hierarchy 30
3.3.2 Function definitions and side-effects 30
3.3.3 Functions that use const char* as parameter 30
3.3.4 Functions whose return type is a reference or a pointer 31
3.3.5 Namespaces and internal naming 33
3.3.6 Non-compliant applications and errors 33
3.4 Notes and examples 33

4. Elaboration and simulation semantics 34

4.1 Overview 34
4.2 Elaboration 34
4.2.1 Overview 34
4.2.2 Instantiation 35
4.2.3 Process macros 36
4.2.4 Port binding and export binding 36
4.2.5 Setting the time resolution 37
4.3 Simulation 38
4.3.1 Overview 38
4.3.2 The scheduling algorithm 38
4.3.3 Initialization, cycles, pauses,and suspension in the scheduling algorithm 41
4.4 Running elaboration and simulation 42
4.4.1 Overview 42
4.4.2 Function declarations 42
4.4.3 sc_elab_and_sim 43
4.4.4 sc_argc and sc_argv 43

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

4.4.5 Running under application control using functions sc_main and sc_start ... 44
4.4.6 Running under control of the kernel ... 46
4.5 Elaboration and simulation callbacks ... 46
4.5.1 Overview ... 46
4.5.2 before_end_of_elaboration ... 47
4.5.3 end_of_elaboration ... 48
4.5.4 start_of_simulation ... 49
4.5.5 end_of_simulation ... 50
4.6 Other functions related to the scheduler ... 50
4.6.1 Function declarations ... 50
4.6.2 sc_pause ... 52
4.6.3 sc_suspend_all, sc_unsuspend_all, sc_unsuspendable, and sc_suspendable ... 53
4.6.4 sc_stop, sc_set_stop_mode, and sc_get_stop_mode ... 53
4.6.5 sc_time_stamp ... 54
4.6.6 sc_delta_count and sc_delta_count_at_current_time ... 55
4.6.7 sc_is_running ... 55
4.6.8 Functions to detect pending activity ... 56
4.6.9 sc_get_status ... 57
4.6.10 sc_stage_callback_if ... 58
4.6.11 sc_register_stage_callback ... 60
4.6.12 sc_unregister_stage_callback ... 60
4.6.13 Callbacks execution ... 60

5. Core language class definitions ... 61

5.1 Class header files ... 61
5.1.1 Overview ... 61
5.1.2 #include "systemc" ... 61
5.1.3 #include "systemc.h" ... 61
5.2 sc_module ... 62
5.2.1 Description ... 62
5.2.2 Class definition ... 62
5.2.3 Constraints on usage ... 65
5.2.4 kind ... 65
5.2.5 SC_MODULE ... 65
5.2.6 Constructors ... 66
5.2.7 SC_CTOR ... 66
5.2.8 SC_METHOD, SC_THREAD, SC_CTHREAD ... 67
5.2.9 SC_NAMED ... 68
5.2.10 Method process ... 68
5.2.11 Thread and clocked thread processes ... 69
5.2.12 Clocked thread processes ... 70
5.2.13 reset_signal_is and async_reset_signal_is ... 72
5.2.14 sensitive ... 73
5.2.15 dont_initialize ... 74
5.2.16 set_stack_size ... 75
5.2.17 next_trigger ... 75
5.2.18 wait ... 77
5.2.19 Positional port binding ... 78
5.2.20 before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation ... 80
5.2.21 get_child_objects and get_child_events ... 80
5.2.22 sc_gen_unique_name ... 81
5.2.23 sc_behavior and sc_channel ... 81

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

5.3 sc_module_name ... 82
5.3.1 Description ... 82
5.3.2 Class definition ... 82
5.3.3 Constraints on usage ... 83
5.3.4 Module hierarchy ... 83
5.3.5 Member functions ... 84
5.4 *sc_sensitive†* ... *85*
5.4.1 Description ... 85
5.4.2 Class definition ... 85
5.4.3 Constraints on usage ... 85
5.4.4 operator<< ... 85
5.5 sc_spawn_options and sc_spawn ... 86
5.5.1 Description ... 86
5.5.2 Class definition ... 86
5.5.3 Constraints on usage ... 88
5.5.4 Constructors ... 88
5.5.5 Member functions ... 88
5.5.6 sc_spawn ... 89
5.5.7 SC_FORK and SC_JOIN ... 92
5.6 sc_process_handle ... 93
5.6.1 Description ... 93
5.6.2 Class definition ... 93
5.6.3 Constraints on usage ... 95
5.6.4 Constructors ... 95
5.6.5 Member functions ... 95
5.6.6 Member functions for process control ... 98
5.6.7 sc_get_current_process_handle ... 114
5.6.8 sc_is_unwinding ... 115
5.7 sc_event_finder and sc_event_finder_t ... 116
5.7.1 Description ... 116
5.7.2 Class definition ... 116
5.7.3 Constraints on usage ... 116
5.8 sc_event_and_list and sc_event_or_list ... 118
5.8.1 Description ... 118
5.8.2 Class definition ... 118
5.8.3 Constraints and usage ... 119
5.8.4 Constructors, destructor, assignment ... 119
5.8.5 Member functions and operators ... 119
5.9 *sc_event_and_expr†* and *sc_event_or_expr†* ... 121
5.9.1 Description ... 121
5.9.2 Class definition ... 121
5.9.3 Constraints on usage ... 122
5.9.4 Operators ... 122
5.10 sc_event ... 123
5.10.1 Description ... 123
5.10.2 Class definition ... 123
5.10.3 Constraints on usage ... 124
5.10.4 Constructors, destructor, and event naming ... 124
5.10.5 Functions for naming and hierarchy traversal ... 125
5.10.6 notify and cancel ... 126
5.10.7 Event lists ... 127
5.10.8 None event ... 127
5.10.9 Multiple event notifications ... 127
5.11 sc_time ... 128

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

5.11.1 Description ... 128
5.11.2 Class definition ... 128
5.11.3 Constructors ... 129
5.11.4 Functions and operators ... 130
5.11.5 Time resolution ... 130
5.11.6 sc_max_time ... 131
5.11.7 SC_ZERO_TIME ... 131
5.12 sc_port ... 131
5.12.1 Description ... 131
5.12.2 Class definition ... 131
5.12.3 Template parameters ... 133
5.12.4 Constraints on usage ... 134
5.12.5 Constructors ... 135
5.12.6 kind ... 135
5.12.7 Named port binding ... 135
5.12.8 Member functions for bound ports and port-to-port binding ... 136
5.12.9 before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation ... 140
5.13 sc_export ... 140
5.13.1 Description ... 140
5.13.2 Class definition ... 140
5.13.3 Template parameters ... 141
5.13.4 Constraints on usage ... 141
5.13.5 Constructors ... 142
5.13.6 kind ... 142
5.13.7 Export binding ... 142
5.13.8 Member functions for bound exports and export-to-export binding ... 143
5.13.9 before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation ... 144
5.14 sc_interface ... 144
5.14.1 Description ... 144
5.14.2 Class definition ... 145
5.14.3 Constraints on usage ... 145
5.14.4 register_port ... 145
5.14.5 default_event ... 146
5.15 sc_prim_channel ... 147
5.15.1 Description ... 147
5.15.2 Class definition ... 147
5.15.3 Constraints on usage ... 148
5.15.4 Constructors, destructor, and hierarchical names ... 148
5.15.5 kind ... 149
5.15.6 request_update and update ... 149
5.15.7 async_attach_suspending and async_detach_suspending ... 150
5.15.8 next_trigger and wait ... 150
5.15.9 before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation ... 151
5.16 sc_object ... 152
5.16.1 Description ... 152
5.16.2 Class definition ... 152
5.16.3 Constraints on usage ... 153
5.16.4 Constructors and destructor ... 153
5.16.5 name, basename, and kind ... 154
5.16.6 print and dump ... 155
5.16.7 Functions for object hierarchy traversal ... 155

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

5.16.8 Member functions for attributes 157
5.16.9 get_hierarchy_scope 158
5.17 Hierarchical naming of objects and events 158
5.17.1 Overview 158
5.17.2 sc_hierarchical_name_exists 159
5.17.3 sc_register_hierarchical_name, sc_unregister_hierarchical_name 160
5.18 sc_attr_base 160
5.18.1 Description 160
5.18.2 Class definition 160
5.18.3 Member functions 161
5.19 sc_attribute 161
5.19.1 Description 161
5.19.2 Class definition 161
5.19.3 Template parameters 161
5.19.4 Member functions and data members 161
5.20 sc_attr_cltn 162
5.20.1 Description 162
5.20.2 Class definition 162
5.20.3 Constraints on usage 162
5.20.4 Iterators 162
5.21 sc_hierarchy_scope 163
5.21.1 Description 163
5.21.2 Class definition 163
5.21.3 Constraints on usage 163
5.21.4 Constructor 164
5.21.5 get_root 164

6. Predefined channel class definitions 165

6.1 sc_signal_in_if 165
6.1.1 Description 165
6.1.2 Class definition 165
6.1.3 Member functions 165
6.2 sc_signal_in_if<bool> and sc_signal_in_if<sc_dt::sc_logic> 166
6.2.1 Description 166
6.2.2 Class definition 166
6.2.3 Member functions 167
6.3 sc_signal_inout_if 167
6.3.1 Description 167
6.3.2 Class definition 167
6.3.3 Member functions 168
6.4 sc_signal 168
6.4.1 Description 168
6.4.2 Class definition 169
6.4.3 Template parameter T 169
6.4.4 Reading and writing signals 170
6.4.5 Constructors 171
6.4.6 register_port 171
6.4.7 Member functions for reading 172
6.4.8 Member functions for writing 172
6.4.9 Member functions for events 172
6.4.10 Diagnostic member functions 173
6.4.11 operator<< 173
6.5 sc_signal<bool,WRITER_POLICY> and sc_signal<sc_dt::sc_logic,WRITER_POLICY> .. 174

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.5.1 Description ... 174
6.5.2 Class definition ... 174
6.5.3 Member functions ... 176
6.6 sc_buffer ... 177
6.6.1 Description ... 177
6.6.2 Class definition ... 177
6.6.3 Constructors ... 177
6.6.4 Member functions ... 178
6.7 sc_clock ... 179
6.7.1 Description ... 179
6.7.2 Class definition ... 179
6.7.3 Characteristic properties ... 180
6.7.4 Constructors ... 180
6.7.5 write ... 180
6.7.6 Diagnostic member functions ... 181
6.7.7 before_end_of_elaboration ... 181
6.7.8 sc_in_clk ... 181
6.8 sc_in ... 181
6.8.1 Description ... 181
6.8.2 Class definition ... 181
6.8.3 Member functions ... 182
6.8.4 sc_trace ... 183
6.8.5 end_of_elaboration ... 183
6.9 sc_in<bool> and sc_in<sc_dt::sc_logic> ... 183
6.9.1 Description ... 183
6.9.2 Class definition ... 183
6.9.3 Member functions ... 185
6.10 sc_inout ... 186
6.10.1 Description ... 186
6.10.2 Class definition ... 186
6.10.3 Member functions ... 187
6.10.4 initialize ... 187
6.10.5 sc_trace ... 187
6.10.6 end_of_elaboration ... 188
6.10.7 Binding ... 188
6.11 sc_inout<bool> and sc_inout<sc_dt::sc_logic> ... 188
6.11.1 Description ... 188
6.11.2 Class definition ... 188
6.11.3 Member functions ... 190
6.12 sc_out ... 190
6.12.1 Description ... 190
6.12.2 Class definition ... 190
6.12.3 Member functions ... 191
6.13 sc_signal_resolved ... 191
6.13.1 Description ... 191
6.13.2 Class definition ... 191
6.13.3 Constructors ... 192
6.13.4 Resolution semantics ... 192
6.13.5 Member functions ... 193
6.14 sc_in_resolved ... 194
6.14.1 Description ... 194
6.14.2 Class definition ... 194
6.14.3 Member functions ... 195
6.15 sc_inout_resolved ... 195

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.15.1 Description ... 195
6.15.2 Class definition ... 195
6.15.3 Member functions ... 196
6.16 sc_out_resolved ... 196
6.16.1 Description ... 196
6.16.2 Class definition ... 196
6.16.3 Member functions ... 197
6.17 sc_signal_rv ... 197
6.17.1 Description ... 197
6.17.2 Class definition ... 197
6.17.3 Semantics and member functions ... 198
6.18 sc_in_rv ... 198
6.18.1 Description ... 198
6.18.2 Class definition ... 198
6.18.3 Member functions ... 199
6.19 sc_inout_rv ... 199
6.19.1 Description ... 199
6.19.2 Class definition ... 199
6.19.3 Member functions ... 200
6.20 sc_out_rv ... 200
6.20.1 Description ... 200
6.20.2 Class definition ... 200
6.20.3 Member functions ... 201
6.21 sc_fifo_in_if ... 201
6.21.1 Description ... 201
6.21.2 Class definition ... 201
6.21.3 Member functions ... 202
6.22 sc_fifo_out_if ... 202
6.22.1 Description ... 202
6.22.2 Class definition ... 202
6.22.3 Member functions ... 203
6.23 sc_fifo ... 204
6.23.1 Description ... 204
6.23.2 Class definition ... 204
6.23.3 Template parameter T ... 205
6.23.4 Constructors ... 205
6.23.5 register_port ... 205
6.23.6 Member functions for reading ... 206
6.23.7 Member functions for writing ... 206
6.23.8 The update phase ... 207
6.23.9 Member functions for events ... 207
6.23.10 Member functions for available values and free slots ... 207
6.23.11 Diagnostic member functions ... 207
6.23.12 operator<< ... 208
6.24 sc_fifo_in ... 208
6.24.1 Description ... 208
6.24.2 Class definition ... 209
6.24.3 Member functions ... 209
6.25 sc_fifo_out ... 209
6.25.1 Description ... 209
6.25.2 Class definition ... 210
6.25.3 Member functions ... 210
6.26 sc_mutex_if ... 212
6.26.1 Description ... 212

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.26.2 Class definition ... 212
6.26.3 Member functions ... 212
6.27 sc_mutex ... 212
6.27.1 Description ... 212
6.27.2 Class definition ... 213
6.27.3 Constructors ... 213
6.27.4 Member functions ... 213
6.28 sc_semaphore_if ... 214
6.28.1 Description ... 214
6.28.2 Class definition ... 214
6.28.3 Member functions ... 215
6.29 sc_semaphore ... 215
6.29.1 Description ... 215
6.29.2 Class definition ... 215
6.29.3 Constructors ... 215
6.29.4 Member functions ... 216
6.30 sc_event_queue ... 216
6.30.1 Description ... 216
6.30.2 Class definition ... 217
6.30.3 Constraints on usage ... 217
6.30.4 Constructors ... 217
6.30.5 kind ... 217
6.30.6 Member functions ... 218
6.31 sc_stub, sc_unbound, sc_tie ... 219

7. SystemC data types ... 220

7.1 Introduction ... 220
7.2 Common characteristics ... 222
7.2.1 Overview ... 222
7.2.2 Initialization and assignment operators ... 223
7.2.3 Precision of arithmetic expressions ... 223
7.2.4 Base class default word length ... 224
7.2.5 Word length ... 225
7.2.6 Bit-select ... 225
7.2.7 Part-select ... 226
7.2.8 Concatenation ... 227
7.2.9 Reduction operators ... 228
7.2.10 Integer conversion ... 228
7.2.11 String input and output ... 229
7.2.12 Conversion of application-defined types in integer expressions ... 229
7.3 String literals ... 230
7.4 *sc_value_base†* ... *231*
7.4.1 Description ... 231
7.4.2 Class definition ... 232
7.4.3 Constraints on usage ... 232
7.4.4 Member functions ... 232
7.5 Limited-precision integer types ... 233
7.5.1 Type definitions ... 233
7.5.2 sc_int_base ... 233
7.5.3 sc_uint_base ... 238
7.5.4 sc_int ... 242
7.5.5 sc_uint ... 245
7.5.6 Bit-selects ... 247

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.5.7 Part-selects ... 251
7.6 Finite-precision integer types ... 256
7.6.1 Type definitions ... 256
7.6.2 Constraints on usage ... 256
7.6.3 sc_signed ... 257
7.6.4 sc_unsigned ... 263
7.6.5 sc_bigint ... 269
7.6.6 sc_biguint ... 271
7.6.7 Bit-selects ... 273
7.6.8 Part-selects ... 276
7.7 Integer concatenations ... 281
7.7.1 Description ... 281
7.7.2 Class definition ... 281
7.7.3 Constraints on usage ... 283
7.7.4 Assignment operators ... 283
7.7.5 Implicit type conversion ... 283
7.7.6 Explicit type conversion ... 284
7.7.7 Other member functions ... 284
7.8 Generic base proxy class ... 284
7.8.1 Description ... 284
7.8.2 Class definition ... 284
7.8.3 Constraints on usage ... 284
7.9 Logic and vector types ... 285
7.9.1 Type definitions ... 285
7.9.2 sc_logic ... 285
7.9.3 sc_bv_base ... 289
7.9.4 sc_lv_base ... 295
7.9.5 sc_bv ... 300
7.9.6 sc_lv ... 302
7.9.7 Bit-selects ... 304
7.9.8 Part-selects ... 308
7.9.9 Concatenations ... 313
7.10 Fixed-point types ... 320
7.10.1 Overview ... 320
7.10.2 Fixed-point representation ... 320
7.10.3 Fixed-point type conversion ... 322
7.10.4 Fixed-point data types ... 322
7.10.5 Fixed-point expressions and operations ... 323
7.10.6 Bit and part selection ... 327
7.10.7 Variable-precision fixed-point value limits ... 327
7.10.8 Fixed-point word length and mode ... 327
7.10.9 Conversions to character string ... 330
7.10.10 Finite word-length effects ... 331
7.10.11 sc_fxnum ... 354
7.10.12 sc_fxnum_fast ... 358
7.10.13 sc_fxval ... 363
7.10.14 sc_fxval_fast ... 368
7.10.15 sc_fix ... 372
7.10.16 sc_ufix ... 375
7.10.17 sc_fix_fast ... 378
7.10.18 sc_ufix_fast ... 381
7.10.19 sc_fixed ... 384
7.10.20 sc_ufixed ... 386
7.10.21 sc_fixed_fast ... 388

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.22sc_ufixed_fast 390
7.10.23Bit-selects 393
7.10.24Part-selects 396
7.11 Contexts 402
7.11.1 Overview 402
7.11.2 sc_length_param 402
7.11.3 sc_length_context 403
7.11.4 sc_fxtype_params 404
7.11.5 sc_fxtype_context 407
7.11.6 sc_fxcast_switch 408
7.11.7 sc_fxcast_context 409
7.12 Control of string representation 410
7.12.1 Description 410
7.12.2 Class definition 410
7.12.3 Functions 410

8. SystemC utilities 411

8.1 Trace files 411
8.1.1 Overview 411
8.1.2 Class definition and function declarations 411
8.1.3 sc_trace_file 411
8.1.4 sc_create_vcd_trace_file 412
8.1.5 sc_close_vcd_trace_file 412
8.1.6 sc_write_comment 412
8.1.7 sc_trace 412
8.2 sc_report 414
8.2.1 Description 414
8.2.2 Class definition 414
8.2.3 Constraints on usage 415
8.2.4 sc_verbosity 415
8.2.5 sc_severity 416
8.2.6 Copy constructor and assignment 416
8.2.7 Member functions 416
8.3 sc_report_handler 417
8.3.1 Description 417
8.3.2 Class definition 417
8.3.3 Constraints on usage 419
8.3.4 sc_actions 419
8.3.5 report 420
8.3.6 set_actions 420
8.3.7 stop_after 421
8.3.8 get_count 422
8.3.9 Verbosity level 422
8.3.10 suppress and force 423
8.3.11 set_handler 423
8.3.12 get_new_action_id 424
8.3.13 sc_interrupt_here and sc_stop_here 424
8.3.14 get_cached_report and clear_cached_report 424
8.3.15 set_log_file_name and get_log_file_name 425
8.4 sc_exception 425
8.4.1 Description 425
8.4.2 Class definition 425
8.5 sc_vector 426

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

8.5.1 Description ... 426
8.5.2 Class definition ... 426
8.5.3 Constraints on usage ... 429
8.5.4 Constructors and destructors ... 429
8.5.5 init and create_element ... 430
8.5.6 Incremental additions to sc_vector during elaboration phase ... 432
8.5.7 kind, size, get_elements ... 432
8.5.8 operator[] and at ... 432
8.5.9 Iterators ... 433
8.5.10 bind ... 433
8.5.11 sc_assemble_vector ... 435
8.6 Utility functions ... 437
8.6.1 Function declarations ... 437
8.6.2 sc_abs ... 437
8.6.3 sc_max ... 438
8.6.4 sc_min ... 438
8.6.5 Version and copyright ... 438

9. Overview of TLM-2.0 and compliance with TLM-2.0 standard ... 440

9.1 Overview ... 440
9.2 Compliance with the TLM-2.0 standard ... 441

10. Introduction to TLM-2.0 ... 442

10.1 Background ... 442
10.2 Transaction-level modeling, use cases, and abstraction ... 442
10.3 Coding styles ... 443
10.3.1 Overview ... 443
10.3.2 Untimed coding style ... 443
10.3.3 Loosely-timed coding style and temporal decoupling ... 444
10.3.4 Synchronization in loosely-timed models ... 445
10.3.5 Approximately-timed coding style ... 445
10.3.6 Characterization of loosely-timed and approximately-timed coding styles ... 446
10.3.7 Switching between loosely-timed and approximately-timed modeling ... 446
10.3.8 Cycle-accurate modeling ... 446
10.3.9 Blocking versus non-blocking transport interfaces ... 446
10.3.10 Use cases and coding styles ... 447
10.4 Initiators, targets, sockets, and transaction bridges ... 447
10.5 DMI and debug transport interfaces ... 449
10.6 Combined interfaces and sockets ... 449
10.7 Namespaces ... 450
10.8 Header files and version numbers ... 450
10.8.1 Overview ... 450
10.8.2 Software version information ... 450
10.8.3 Definitions ... 450
10.8.4 Rules ... 451

11. TLM-2.0 core interfaces ... 452

11.1 Overview ... 452
11.2 Transport interfaces ... 452
11.2.1 Overview ... 452
11.2.2 Blocking transport interface ... 452

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

11.2.3 Non-blocking transport interface ... 457
11.2.4 Timing annotation with the transport interfaces ... 465
11.2.5 Migration path from TLM-1 ... 468
11.3 Direct memory interface ... 468
11.3.1 Introduction ... 468
11.3.2 Class definition ... 469
11.3.3 get_direct_mem_ptr ... 470
11.3.4 template argument and tlm_generic_payload class ... 471
11.3.5 tlm_dmi class ... 472
11.3.6 invalidate_direct_mem_ptr ... 475
11.3.7 DMI versus transport ... 475
11.3.8 DMI and temporal decoupling ... 476
11.3.9 Optimization using a DMI hint ... 476
11.4 Debug transport interface ... 476
11.4.1 Introduction ... 476
11.4.2 Class definition ... 477
11.4.3 TRANS template argument and tlm_generic_payload class ... 477
11.4.4 Rules ... 477

12. TLM-2.0 global quantum ... 480

12.1 Introduction ... 480
12.2 Header file ... 480
12.3 Class definition ... 480
12.4 tlm_global_quantum ... 481

13. Combined TLM-2.0 interfaces and sockets ... 482

13.1 Combined interfaces ... 482
13.1.1 Introduction ... 482
13.1.2 Class definition ... 482
13.2 Initiator and target sockets ... 483
13.2.1 Introduction ... 483
13.2.2 Class definition ... 483
13.2.3 tlm_base_socket_if ... 487
13.2.4 tlm_base_initiator_socket_b and tlm_base_target_socket_b ... 487
13.2.5 tlm_base_initiator_socket and tlm_base_target_socket ... 488
13.2.6 tlm_initiator_socket and tlm_target_socket ... 490

14. TLM-2.0 generic payload ... 493

14.1 Introduction ... 493
14.2 Extensions and interoperability ... 493
14.2.1 Overview ... 493
14.2.2 Use the generic payload directly, with ignorable extensions ... 494
14.2.3 Define a new protocol traits class containing a typedef for tlm_generic_payload ... 495
14.2.4 Define a new protocol traits class and a new transaction type ... 495
14.3 Generic payload attributes and member functions ... 496
14.4 Class definition ... 496
14.5 Generic payload memory management ... 498
14.6 Constructors, assignment, and destructor ... 502
14.7 Default values and modifiability of attributes ... 502
14.8 Option attribute ... 504
14.9 Command attribute ... 506

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

14.10 Address attribute 506
14.11 Data pointer attribute 507
14.12 Data length attribute 508
14.13 Byte enable pointer attribute 508
14.14 Byte enable length attribute 509
14.15 Streaming width attribute 510
14.16 DMI allowed attribute 510
14.17 Response status attribute 511
14.17.1 Overview 511
14.17.2 The standard error response 512
14.18 Endianness 515
14.18.1 Introduction 515
14.18.2 Rules 516
14.19 Helper functions to determine host endianness 518
14.19.1 Introduction 518
14.19.2 Definition 518
14.19.3 Rules 519
14.20 Helper functions for endianness conversion 519
14.20.1 Introduction 519
14.20.2 Definition 520
14.20.3 Rules 520
14.21 Generic payload extensions 522
14.21.1 Introduction 522
14.21.2 Rationale 522
14.21.3 Extension pointers, objects and transaction bridges 523
14.21.4 Rules 523

15. TLM-2.0 base protocol and phases 529

15.1 Phases 529
15.1.1 Introduction 529
15.1.2 Class definition 529
15.1.3 Rules 530
15.2 Base protocol531
15.2.1 Introduction 531
15.2.2 Class definition 532
15.2.3 Base protocol phase sequences 532
15.2.4 Permitted phase transitions 534
15.2.5 Ignorable phases 537
15.2.6 Base protocol timing parameters and flow control 539
15.2.7 Base protocol rules concerning timing annotation 543
15.2.8 Base protocol rules concerning b_transport 544
15.2.9 Base protocol rules concerning request and response ordering 544
15.2.10 Base protocol rules for switching between b_transport and nb_transport 545
15.2.11 Other base protocol rules 546
15.2.12 Summary of base protocol transaction ordering rules 546
15.2.13 Guidelines for creating base-protocol-compliant components 547

16. TLM-2.0 utilities 550

16.1 Overview 550
16.2 Convenience sockets 550
16.2.1 Introduction 550
16.2.2 Simple sockets 551

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

16.2.3 Tagged simple sockets ... 557
16.2.4 Multi-sockets ... 560
16.3 Quantum keeper ... 566
16.3.1 Introduction ... 566
16.3.2 Header file ... 567
16.3.3 Class definition ... 567
16.3.4 General guidelines for processes using temporal decoupling ... 567
16.3.5 tlm_quantumkeeper ... 568
16.4 Payload event queue ... 570
16.4.1 Introduction ... 570
16.4.2 Header file ... 571
16.4.3 Class definition ... 571
16.4.4 Rules ... 571
16.5 Instance-specific extensions ... 572
16.5.1 Introduction ... 572
16.5.2 Header file ... 573
16.5.3 Class definition ... 573

17. TLM-1 message passing interface and analysis ports ... 575

17.1 Overview ... 575
17.2 Put, get, peek, and transport interfaces ... 575
17.2.1 Description ... 575
17.2.2 Class definition ... 575
17.2.3 Blocking versus non-blocking interfaces ... 577
17.2.4 Blocking interface methods ... 578
17.2.5 Non-blocking interface methods ... 578
17.2.6 Argument passing and transaction lifetime ... 579
17.2.7 Constraints on the transaction data type ... 580
17.3 TLM-1 fifo interfaces ... 580
17.3.1 Description ... 580
17.3.2 Class definition ... 580
17.3.3 Member functions ... 581
17.4 tlm_fifo ... 582
17.4.1 Description ... 582
17.4.2 Class definition ... 582
17.4.3 Template parameter T ... 583
17.4.4 Constructors and destructor ... 583
17.4.5 Member functions ... 584
17.4.6 Delta cycle semantics ... 585
17.5 Analysis interface and analysis ports ... 587
17.5.1 Overview ... 587
17.5.2 Class definition ... 587
17.5.3 Rules ... 588

Annex A (informative) Glossary ... 590

Annex B (informative) Introduction to SystemC ... 607

Annex C (informative) Deprecated features ... 611

Annex D (informative) Changes between IEEE Std 1666-2011 and IEEE Std 1666-2023 ... 613

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

IEEE Standard for Standard SystemC® Language Reference Manual

1. Overview

1.1 Scope

This standard defines SystemC® with Transaction Level Modeling (TLM) as an ISO standard C++ class library for system and hardware design.[1]

1.2 Purpose

The general purpose of 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.

1.3 Word usage

The word *shall* indicates mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted (*shall* equals *is required to*).[2, 3]

The word *should* indicates that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required (*should* equals *is recommended that*).

The word *may* is used to indicate a course of action permissible within the limits of the standard (*may* equals *is permitted to*).

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

[2] The use of the word *must* is deprecated and cannot be used when stating mandatory requirements; *must* is used only to describe unavoidable situations.

[3] The use of *will* is deprecated and cannot be used when stating mandatory requirements; *will* is only used in statements of fact.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The word *can* is used for statements of possibility and capability, whether material, physical, or causal (*can* equals is *able to*).

See also Clause 3 for information about other terminology, syntactical conventions, semantic conventions, and notes and examples used in this standard.

1.4 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 9.2 for a description of TLM-2.0-compliance.

1.5 Relationship with C++ standard

This standard is closely related to the C++ programming language and adheres to the terminology used in ISO/IEC 14882:2017 (i.e., C++17).[4] As a result, C++17 is the baseline for SystemC. Implementations and tools may support other C++ versions (i.e., greater than 2017); however, they shall support at least C++17 to claim compliance with IEEE Std 1666-2023. This standard does not seek to restrict the usage of C++17; a SystemC application may use any of the facilities provided by C++17, 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.[5]

1.6 Guidance for readers

Readers who are not entirely familiar with SystemC should start with the introduction to SystemC in Annex B, which provides a brief informal summary of the subject intended to aid in the understanding of the normative definitions given throughout this standard. Such readers may also find it helpful to scan the examples embedded in the normative definitions and to review the glossary in Annex A.

In addition to the overview and normative references in Clause 1 and Clause 2, respectively, all readers should pay close attention to Clause 3, where certain terminology and conventions used in this standard are explained. An understanding of the terminology defined in Clause 3 is necessary for a precise interpretation of this standard.

Clause 4 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 on the foundations laid in Clause 4.

[4] Information on references can be found in Clause 2.

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

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Clause 5 through Clause 8 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.

Clause 9 through Clause 16 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 a glossary giving informal descriptions of the terms used in this standard.

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

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 IEEE Std 1666-2011 and IEEE Std 1666-2023.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

2. Normative references

The following referenced documents are indispensable for the application of this document (i.e., they must be understood and used; therefore, 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 publication:

ISO/IEC 14882:2017, Programming Languages — C++.[6]

[6] ISO/IEC publications are available from the International Organization for Standardization (https://www.iso.org), the International Electrotechnical Commission (https://www.iec.ch), and the American National Standards Institute (https://www.ansi.org).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

3. Terminology and conventions used in this standard

3.1 Terminology

3.1.1 Shall, should, may, can

See 1.3 for IEEE's usage conventions regarding the words *shall, should*, *may*, and *can*.

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**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

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

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

a particular callback cannot be inferred from the terms *during elaboration* and *during simulation* alone but is defined in detail in 4.5. 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.

The term *after elaboration* indicates the moment where all **end_of_elaboration** callbacks have been executed.

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 as follows:

sc_bind_proxy†	*sc_fxnum_fast_bitref*†	*sc_int_subref_r*†	*sc_uint_bitref_r*†
sc_bitref†	*sc_fxnum_fast_subref*†	*sc_sensitive*†	*sc_uint_subref*†
sc_bitref_r†	*sc_fxnum_subref*†	*sc_signed_bitref*†	*sc_uint_subref_r*†

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

sc_concatref†	*sc_fxnum_bitref_r*	*sc_signed_bitref_r*†	*sc_unsigned_bitref*†
sc_concref†	*sc_fxnum_fast_bitref_r*	*sc_signed_subref*†	*sc_unsigned_bitref_r*†
sc_concref_r†	*sc_fxnum_fast_subref_r*	*sc_signed_subref_r*†	*sc_unsigned_subref*†
sc_context_begin†	*sc_fxnum_subref_r*	*sc_subref*†	*sc_unsigned_subref_r*†
sc_event_and_expr†	*sc_int_bitref*†	*sc_subref_r*†	*sc_value_base*†
sc_event_or_expr†	*sc_int_bitref_r*†	*sc_switch*†	*sc_vector_iter*†
sc_fxnum_bitref†	*sc_int_subref*†	*sc_uint_bitref*†	

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 uppercase 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 that use const char* as parameter

An application shall not pass a null pointer as argument to functions that use const char* as parameter. Instead, an application shall pass a null-terminated character string, which may be an empty string. An implementation shall throw an error if a null pointer is passed as argument to a function that use const char* as parameter.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

3.3.4 Functions whose return type is a reference or a pointer

3.3.4.1 Overview

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. Subclause 3.3.4 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.

3.3.4.2 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.

3.3.4.3 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.

3.3.4.4 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 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.

3.3.4.5 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_event *
c) sc_attr_base *
d) std::string& // Property of an attribute object

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The functions concerned are as follows:

```
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:

— std::vector< sc_object * > &
— std::vector< sc_event * > &
— sc_attr_cltn *

The functions concerned are as follows:

```
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.

3.3.4.6 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.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

3.3.5 Namespaces and internal naming

An implementation shall place every declaration specified by this standard, with the one exception of **sc_main**, within one of the five namespaces **sc_core**, **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.

3.3.6 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 uppercase 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.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

4. Elaboration and simulation semantics

4.1 Overview

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

4.2 Elaboration

4.2.1 Overview

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 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.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

4.2.2 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 4.4.5.2 and 4.4.6).

Instances of class **sc_module** and class **sc_prim_channel** may only be created within a module or from function **sc_main** (or a function called from **sc_main**), or in the absence of an **sc_main** (see 4.4.6), 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 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 s**c_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, 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 simulation, as may objects of user-defined classes.

Example:

```
#include <systemc>

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

struct S {
    Mod m;                          // Not recommended coding style - module instance within struct
    S(char* name_) : m(name_) {}
};

struct Top : sc_core::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"
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        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");        // Not recommended coding style
    }                                   // m3->name() returns "top.m3"
};

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

4.2.3 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 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.

4.2.4 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.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

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 needs to be bound is dependent on 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 5.12.3). 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 5.14).

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 the 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—The member function **register_port** is defined in the class **sc_interface** from which every channel is derived.

4.2.5 Setting the time resolution

The simulation time resolution can be set during elaboration and only during elaboration. The time resolution is set by calling the function **sc_set_time_resolution** (see 5.11.5).

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

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

4.3 Simulation

4.3.1 Overview

Subclause 4.3 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 5.10).

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 5.11.1).

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

- In response to the process instance having been made runnable during the initialization phase (see 4.3.2.2)
- 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
- In response to a call to a process control member function of the class **sc_process_handle**

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.

4.3.2 The scheduling algorithm

4.3.2.1 Overview

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.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

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** or **async_request_update** of class **sc_prim_channel** (see 5.15.6).

An *immediate notification* results from, and only from, a call to member function **notify** of class **sc_event** with no arguments (see 5.10.6).

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 5.2.17 and 5.2.18).

The scheduler starts by executing the initialization phase.

4.3.2.2 Initialization phase

Perform the following three steps in the order given:

a) Run the update phase as defined in 4.3.2.4 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 4.3.2.5. At the end of the delta notification phase, go to the evaluation phase.

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

4.3.2.3 Evaluation phase

From the set of runnable processes, select a process instance, remove it from the set, and only then trigger or resume its execution. Run the process instance immediately and without interruption up to the point where it either returns or, in the case of a thread or clocked thread process, calls the function **wait** or calls the member function **suspend** of a process handle associated with the process instance itself.

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 preempt 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.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

A process may execute an immediate notification, in which case all process instances that are currently sensitive to the notified event shall be added to the set of runnable processes. Such process instances shall be executed in the current evaluation phase. The currently executing process instance shall not be added to the set of runnable processes as the result of an immediate notification executed by the process instance itself.

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** or **async_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.

A process may call a process control member function of the class **sc_process_handle**, which may cause a process instance to become runnable in the current evaluation phase or may remove a process instance from the set of runnable processes.

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

The scheduler is not preemptive. 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.

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.

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 to constrain their execution to match the co-routine semantics.

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, unless those processes happen to be statically sensitive to the given event.

4.3.2.4 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 or resulting from calls to function **async_request_update**. Function **update** shall be called no more than once for each primitive channel instance in each update phase.

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

4.3.2.5 Delta notification phase

If at least one process requested suspension and no process has identified itself as unsuspendable, the scheduler ceases execution and the simulation is suspended (see 4.3.3). Otherwise, 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.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

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.

4.3.2.6 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,

— And if at least one primitive channel has attached itself to request suspension using **async_attach_suspending**, the scheduler ceases execution and the simulation is suspended (see 4.3.3).
— Otherwise, the end of simulation has been reached and the scheduler exits.

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

NOTE—On exiting the scheduler, the value of the simulation time will depend on how the scheduler was invoked (see 4.4.5.3).

4.3.3 Initialization, cycles, pauses,and suspension 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.

Update requests may be created during elaboration or before the initialization phase, and they shall be scheduled to execute in the update phase during the initialization phase.

Delta notifications and timed notifications may be created during elaboration or before the initialization phase. Such delta notifications shall be scheduled to occur in the delta notification phase during the initialization phase.

The scheduler repeatedly executes whole delta cycles and may cease execution at the boundary between the delta notification and evaluation phases. The only circumstances in which the scheduler can cease execution other than at this boundary are after a call to function **sc_stop**, when an exception is thrown, or when simulation is stopped or aborted by the report handler (see 8.3).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

When **sc_pause** is called, the scheduler shall cease execution at the end of a delta notification phase, and if it is to resume execution, the scheduler shall resume at the start of the next evaluation phase.

An application may create update requests and event notifications while the scheduler is paused, and these shall be treated by the scheduler as if they had been created in the evaluation phase immediately following resumption, in the following sense: update requests shall be queued to be executed in the first update phase following resumption, immediate notifications shall make any sensitive processes runnable in the first evaluation phase, and delta notifications shall make any sensitive processes runnable in the second evaluation phase following resumption.

When the simulation is suspended (either following a call to **sc_suspend_all** or because a channel has attached itself for suspension and the simulation has run out of timed notifications or time-outs), the only way to resume execution is through a call to **async_request_update** from an external operating system thread.

Update requests created before the initialization phase or while the scheduler is paused shall not be associated with any process instance with respect to the rules for updating the state of any primitive channel, for example, the writer policy of **sc_signal**.

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 an evaluation phase, followed by an update phase, followed by a delta notification phase, and back to an evaluation phase. This loop advances simulation by one delta cycle.
- The timed notification loop takes the path of an evaluation phase, followed by an update phase, followed by a delta notification phase, followed by a timed notification phase, and back to an 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).

4.4 Running elaboration and simulation

4.4.1 Overview

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 s**c_start**
- Under control of the kernel

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

4.4.2 Function declarations

namespace sc_core {

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

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

    enum sc_starvation_policy {
        SC_RUN_TO_TIME,
        SC_EXIT_ON_STARVATION
    };

    void sc_start();
    void sc_start( const sc_time&, sc_starvation_policy p = SC_RUN_TO_TIME );
    void sc_start( double , sc_time_unit, sc_starvation_policy p = SC_RUN_TO_TIME );
}
```

4.4.3 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.

4.4.4 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();
```

The function **sc_argc** shall return an integer whose value is the number of non-null pointers returned in the array returned by **sc_argv**.

The function **sc_argv** shall return a copy of the arguments passed to function **main** or function **sc_elab_and_sim**, which allows them to be modified. The value returned shall be an array of pointers to null-terminated char strings, with the final pointer being a null pointer.

NOTE—The definition and usage of **sc_argc** and **sc_argv** are intended to be consistent with the use of argc and argv in the C++ main function.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

4.4.5 Running under application control using functions sc_main and sc_start

4.4.5.1 Overview

The application provides a function **sc_main** and calls the function **sc_start**, as defined in 4.4.5.2 and 4.4.5.3.

4.4.5.2 sc_main

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

```
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 4.2, 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.

4.4.5.3 sc_start

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

```
void sc_start();
void sc_start( const sc_time&, sc_starvation_policy p = SC_RUN_TO_TIME );
void sc_start( double , sc_time_unit, sc_starvation_policy p = SC_RUN_TO_TIME );
```

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

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

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, and as determined by the starvation policy argument. The scheduler shall execute the initialization phase before executing the first evaluation phase, as described in 4.3.2.2.

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, and as determined by the starvation policy argument. The scheduler shall execute from the evaluation phase of a new delta cycle, as described in 4.3.3.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

When a time is passed as an argument, the scheduler shall execute up to and including the latest timed notification phase with simulation time less than or equal to the end time (calculated by adding the time given as an argument to the simulation time when function **sc_start** is called). On return from **sc_start**, if the argument of type **sc_starvation_policy** has the value SC_RUN_TO_TIME, the implementation shall set simulation time equal to the end time, regardless of the time of the most recent event notification or time-out. If the argument of type **sc_starvation_policy** has the value SC_EXIT_ON_STARVATION, the implementation shall set simulation time equal to the time of the most recent event notification or time-out, which may be less than the end time.

When function **sc_start** is called without any arguments, the scheduler shall run until there is no remaining activity, unless otherwise interrupted. In other words, except when **sc_stop** or **sc_pause** has been called or an exception has been thrown, control shall be returned from **sc_start** only when the set of runnable processes, the set of update requests, the set of delta notifications and time-outs, and the set of timed notifications and time-outs are all empty, and no suspension has been requested. On return from **sc_start**, the implementation shall set simulation time equal to the time of the most recent event notification or time-out.

When function **sc_start** is called with a zero-valued time argument, the scheduler shall run for one delta cycle, that is, an evaluation phase, an update phase, and a delta notification phase, in that order. The value of the starvation policy argument shall be ignored. Simulation time shall not be advanced. If this is the first call to **sc_start**, the scheduler shall execute the initialization phase before executing the evaluation phase, as described in 4.3.2.2. If, when **sc_start** is called with a zero-valued time argument, the set of runnable processes is empty, the set of update requests is empty, and the set of delta notifications and time-outs is empty; that is, if **sc_pending_activity_at_current_time() == false**, the implementation shall issue a warning and the value returned from **sc_delta_count** shall not be incremented.

Once started, the scheduler shall run until either it reaches the end time, there is no remaining activity, or the application calls the function **sc_pause** or **sc_stop**, an exception occurs, or simulation is stopped or aborted by the report handler (see 8.3). If the function **sc_pause** has been called, function **sc_start** may be called again. Once the function **sc_stop** has been called, function **sc_start** shall not be called again. If neither **sc_pause** nor **sc_stop** have been called, the implementation shall insert an implicit call to **sc_pause** before returning control from function **sc_start**, and function **sc_start** may be called again.

Update requests, timed notifications, and delta notifications may be created before the first call to **sc_start**, but immediate notifications shall not be created before the first call to **sc_start**. Update requests and event notifications, including immediate notifications, may be created between or after calls to **sc_start**.

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

Applications are recommended to call function **sc_stop** before returning control from **sc_main** so that the **end_of_simulation** callbacks are called (see 4.5.5).

Example:

```
int sc_main( int argc, char* argv[] ) {
    using namespace sc_core;

    Top top("top");                                  // Instantiate module hierarchy

    sc_start(100.0, SC_NS); // Run for exactly 100 ns
    sc_start();                                      // Run until no more activity

    if (sc_get_status() == SC_PAUSED) {
        SC_REPORT_INFO("", "sc_stop called to terminate a paused simulation");
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_stop();
    }

    return 0;
}
```

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

It shall be an error to call function **sc_start** with a negative floating-point time argument. In this case, the time shall be set to **SC_ZERO_TIME**.

4.4.6 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 it is not obliged to do so.

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

In this case, an implementation shall honor all obligations set out in this standard with the exception of those in 4.4.5.

4.5 Elaboration and simulation callbacks

4.5.1 Overview

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 subclauses that follow.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

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.

4.5.2 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 4.4 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 the global properties of the module hierarchy and that 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**:

- The instantiation of objects of class **sc_module**, **sc_port**, **sc_export**, **sc_prim_channel**.
- The instantiation of objects of other classes derived from class **sc_object**.
- Port binding.
- Export binding.
- The macros SC_METHOD, SC_THREAD, and SC_CTHREAD.
- The member **sensitive** and member functions **dont_initialize**, **set_stack_size**, **reset_signal_is**, and **async_reset_signal_is**, of the class **sc_module**.
- Calls to event finder functions.
- Calls to function **sc_spawn** to create static, spawned processes.
- Calls to member function **set_sensitivity** of class **sc_spawn_options** to set the sensitivity of a spawned process using an event, an interface, or an event finder function. A port cannot be used because port binding may have been deferred.
- Calls to member functions **reset_signal_is** and **async_reset_signal_is** of the class **sc_spawn_options**.
- Calls to member functions **request_update** or **async_request_update** of class **sc_prim_channel** to create update requests (for example, by calling member function **initialize** of class **sc_inout**).
- Calls to member functions **notify(const sc_time&)** and **notify(double,sc_time_unit)** of class **sc_event** to create delta and timed notifications.
- Calls to the process control member functions **suspend**, **resume**, **disable**, **enable**, **sync_reset_on**, and **sync_reset_off** of class **sc_process_handle**.

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

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

— Calls to member function **notify** of class **sc_event** with an empty argument list to create immediate notifications.
— Calls to the process control member functions **kill**, **reset**, or **throw_it** of class **sc_process_handle**.

The following constructs cannot 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**:

— The macro SC_CTOR

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.

4.5.3 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**:

— The instantiation of objects of classes derived from class **sc_object** but excluding classes **sc_module**, **sc_port**, **sc_export**, and **sc_prim_channel**.
— The macros SC_METHOD and SC_THREAD.
— The member **sensitive** and member functions **dont_initialize** and **set_stack_size** of the class **sc_module**.
— Calls to function **sc_spawn** to create dynamic spawned processes.
— Calls to member functions **request_update** or **async_request_update** of class **sc_prim_channel** to create update requests (for example, by calling member function **write** of class **sc_inout**).
— Calls to member functions **notify(const sc_time&)** and **notify(double,sc_time_unit)** of class **sc_event** to create delta and timed notifications.
— Calls to the process control member functions **suspend**, **resume**, **disable**, **enable**, **sync_reset_on**, and **sync_reset_off** of class **sc_process_handle**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

— 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**:

— The instantiation of objects of class **sc_module**, **sc_port**, **sc_export**, **sc_prim_channel**.
— Port binding.
— Export binding.
— The macros SC_CTOR and SC_CTHREAD.
— The member functions **reset_signal_is** and **async_reset_signal_is** of the class **sc_module**.
— Calls to event finder functions.
— Calls to member function **notify** of class **sc_event** with an empty argument list to create immediate notifications.
— Calls to the process control member functions **kill**, **reset**, or **throw_it** of class **sc_process_handle**.

4.5.4 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**:

— The instantiation of objects of classes derived from class **sc_object** but excluding classes **sc_module**, **sc_port**, **sc_export**, and **sc_prim_channel**.
— Calls to function **sc_spawn** to create dynamic spawned processes.
— Calls to member functions **request_update** or **async_request_update** of class **sc_prim_channel** to create update requests (for example by calling member function **write** of class **sc_inout**).
— Calls to member functions **notify(const sc_time&)** and **notify(double,sc_time_unit)** of class **sc_event** to create delta and timed notifications.
— Calls to the process control member functions **suspend**, **resume**, **disable**, **enable**, **sync_reset_on**, and **sync_reset_off** of class **sc_process_handle**.
— 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**:

— The instantiation of objects of class **sc_module**, **sc_port**, **sc_export**, **sc_prim_channel**.
— Port binding.
— Export binding.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

— The macros SC_CTOR, SC_METHOD, SC_THREAD, and SC_CTHREAD.
— The member **sensitive** and member functions **dont_initialize**, **set_stack_size**, **reset_signal_is**, and **async_reset_signal_is** of the class **sc_module**.
— Calls to event finder functions.
— Calls to member function **notify** of class **sc_event** with an empty argument list to create immediate notifications.
— Calls to the process control member functions **kill**, **reset**, or **throw_it** of class **sc_process_handle**.

4.5.5 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 4.6.4) 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:

— The instantiation of objects of classes derived from class **sc_object**.
— Calls to function **sc_spawn** to create dynamic spawned processes.
— Calls to member functions **request_update** or **async_request_update** of class **sc_prim_channel** to create update requests (for example by calling member function **write** of class **sc_inout**).
— Calls to member function **notify** of the class **sc_event**.

Whether any of these actions causes an error is implementation-defined.

4.6 Other functions related to the scheduler

4.6.1 Function declarations

```
namespace sc_core {

  void sc_pause();

  enum sc_stop_mode
  {
      SC_STOP_FINISH_DELTA,
      SC_STOP_IMMEDIATE
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    };

    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();
    sc_dt::uint64 sc_delta_count();
    sc_dt::uint64 sc_delta_count_at_current_time();
    bool sc_is_running();
    bool sc_pending_activity_at_current_time();
    bool sc_pending_activity_at_future_time();
    bool sc_pending_activity();
    sc_time sc_time_to_pending_activity();

    enum sc_status
    {
        SC_ELABORATION,
        SC_BEFORE_END_OF_ELABORATION,
        SC_END_OF_ELABORATION,
        SC_START_OF_SIMULATION,
        SC_RUNNING,
        SC_PAUSED,
        SC_SUSPENDED,
        SC_STOPPED,
        SC_END_OF_SIMULATION
    };

    sc_status sc_get_status();

    enum sc_stage
    {
        SC_POST_BEFORE_END_OF_ELABORATION,
        SC_POST_END_OF_ELABORATION,
        SC_POST_START_OF_SIMULATION,
        SC_POST_UPDATE,
        SC_PRE_TIMESTEP,
        SC_PRE_PAUSE,
        SC_PRE_SUSPEND,
        SC_POST_SUSPEND,
        SC_PRE_STOP,
        SC_POST_END_OF_SIMULATION
    };

    void sc_register_stage_callback( const sc_stage_callback_if&, int );
    void sc_unregister_stage_callback( const sc_stage_callback_if&, int );

}
```

The actual integer values for **sc_status** are implementation-defined and shall be chosen to make possible bitwise combinations of the various values.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

4.6.2 sc_pause

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

```
void sc_pause();
```

Function **sc_pause** shall cause the scheduler to cease execution at the end of the current delta cycle such that the scheduler can be resumed again later. If **sc_start** was called, **sc_pause** shall cause the scheduler to return control from **sc_start** at the end of the current delta cycle such that the scheduler can be resumed by a subsequent call to **sc_start**.

Function **sc_pause** shall be non-blocking; that is, the calling function shall continue to execute until it yields or returns.

If the function **sc_pause** is called during the evaluation phase or the update phase, the scheduler shall complete the current delta cycle before ceasing execution; that is, the scheduler shall complete the evaluation phase, the update phase, and the delta notification phase.

Function **sc_pause** may be called during elaboration, from function **sc_main**, or from one of the callbacks **before_end_of_elaboration**, **end_of_elaboration**, **start_of_simulation**, or **end_of_simulation**, in which case it shall have no effect and shall be ignored by the implementation.

If **sc_stop** has already been called, a call to **sc_pause** shall have no effect.

The following operations are permitted while simulation is paused:

- The instantiation of objects of a type derived from **sc_object** provided that instantiation of those objects is permitted during simulation.
- Calls to function **sc_spawn** to create dynamic processes.
- Calls to member functions **request_update** and **async_request_update** of class **sc_prim_channel** to create update requests.
- Calls to member function **notify** of class **sc_event** to create immediate, delta, or timed notifications.
- Calls to the process control member functions **suspend**, **resume**, **disable**, **enable**, **sync_reset_on**, and **sync_reset_off** of class **sc_process_handle**.
- Interface method calls using **operator->** and **operator[]** of class **sc_port**, provided those functions do not themselves perform any operations that are forbidden while simulation is paused.
- Calls to function **sc_stop**.

It shall be an error to perform any of the following operations while simulation is paused:

- The instantiation of objects of class **sc_module**, **sc_port**, **sc_export**, or **sc_prim_channel**.
- Port binding.
- Export binding.
- Invocation of the macros SC_METHOD, SC_THREAD, or SC_CTHREAD.
- Use of the member **sensitive** of class **sc_module** to create static sensitivity.
- Calls to the member functions **dont_initialize**, **set_stack_size**, **reset_signal_is**, or **async_reset_signal_is** of the class **sc_module**.
- Calls to event finder functions.
- Calls to the process control member functions **kill**, **reset**, or **throw_it** of class **sc_process_handle**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

— Calls to the member functions **wait** or **next_trigger** of classes **sc_module** and **sc_prim_channel** or to the non-member functions **wait** or **next_trigger**.

4.6.3 sc_suspend_all, sc_unsuspend_all, sc_unsuspendable, and sc_suspendable

```
void sc_suspend_all();
void sc_unsuspend_all();
void sc_unsuspendable();
void sc_suspendable();
```

Function **sc_suspend_all** requests the kernel to suspend all processes (using the same semantics as the thread **suspend** call). This is atomic in that the kernel will only suspend all the processes together. Once all processes are suspended, the simulator itself is said to be suspended.

A process may cancel its request for suspension by calling **sc_unsuspend_all** before the next delta notification phase. Suspension will occur only if at least one process requested suspension.

Processes can opt out of **sc_suspend_all** by calling **sc_unsuspendable** and cancel opting out by calling **sc_suspendable** afterwards. **sc_suspend_all** will have no effect if at least one process opted out of **sc_suspend_all**.

NOTE 1—Multiple consecutive calls to **sc_suspend_all** without any call to **sc_unsuspend_all** in between will be ignored. Similarly, multiple consecutive calls to **sc_unsuspendable** without any call to **sc_suspendable** in between will also be ignored.

NOTE 2—One possible use case for this feature is to suspend the simulation to synchronize its simulation time with another external simulation(s). The kernel will resume execution upon external call to **async_request_update**. Other processes may have to opt out of such suspension if they themselves are reacting to stimuli coming from external simulation(s) and they cannot be suspended until the end of the reaction.

4.6.4 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. Function **sc_set_stop_mode** may be called during elaboration or from one of the callbacks **before_end_of_elaboration**, **end_of_elaboration**, or **start_of_simulation**. If **sc_set_stop_mode** is called more than once, the most recent call shall take precedence. It shall be an error for the application to call function **sc_set_stop_mode** from the initialization phase onward.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

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 4.5.5.

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).

The default stop mode shall be SC_STOP_FINISH_DELTA.

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**, avoiding an infinite regression.

A call to **sc_stop** shall take precedence over a call to **sc_pause**.

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.

4.6.5 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.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

4.6.6 sc_delta_count and sc_delta_count_at_current_time

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 in which at least one process is triggered or resumed and, thus, returns a count of the absolute number of delta cycles that have occurred during simulation, starting from zero, ignoring any delta cycles in which there were no runnable processes. The delta count shall be 0 during elaboration, during the initialization phase, and during the first evaluation phase. The delta count shall first be incremented after the evaluation phase of the first whole delta cycle. The delta count shall be incremented between the evaluation phase and the update phase of each delta cycle in which there were runnable processes.

A delta cycle in which there are no runnable processes can occur when function **sc_start** is called with a zero-valued time argument or when the scheduler is resumed following a call to **sc_pause**.

When the scheduler is resumed following a call to function **sc_pause**, the delta count shall continue to be incremented as if **sc_pause** had not been called.

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();
```

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

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

The function **sc_delta_count_at_current_time** is similar to **sc_delta_count**; however, it shall return the number of delta cycles for the current time, i.e., it resets at each new time advance.

4.6.7 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, paused, or suspended, including the initialization phase, and shall return the value **false** during elaboration, during the callbacks **start_of_simulation** and **end_of_simulation**, and on return from **sc_start** after **sc_stop** has been called.

The following relation shall hold:

```
sc_assert( sc_is_running() == (sc_get_status() & (SC_RUNNING | SC_PAUSED | SC_SUSPENDED)) );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

4.6.8 Functions to detect pending activity

The implementation shall provide four functions to detect pending activity with the following declarations:

```
bool sc_pending_activity_at_current_time();
bool sc_pending_activity_at_future_time();
bool sc_pending_activity();
sc_time sc_time_to_pending_activity();
```

The function **sc_pending_activity_at_current_time** shall return the value **true** if and only if the set of runnable processes is not empty, the set of update requests is not empty, or the set of delta notifications and time-outs is not empty. By implication, if **sc_pending_activity_at_current_time** were to return the value **false**, the next action of the scheduler would be to execute the timed notification phase.

The function **sc_pending_activity_at_future_time** shall return the value **true** if and only if the set of timed notifications and time-outs is not empty.

The function **sc_pending_activity** shall return the value **true** if and only if **sc_pending_activity_at_current_time** or **sc_pending_activity_at_future_time** would return **true**.

The function **sc_time_to_pending_activity** shall return the time until the earliest pending activity, calculated as follows:

— If **sc_pending_activity_at_current_time() == true**, the value returned is SC_ZERO_TIME.
— Otherwise, if **sc_pending_activity_at_future_time() == true**, the value returned is **T - sc_time_stamp()**, where T is the time of the earliest timed notification or time-out in the set of timed notifications and time-outs.
— Otherwise, the value returned is **sc_max_time() - sc_time_stamp()**.

These four functions may be called at any time during or following elaboration or simulation.

The function **sc_pending_activity** may return **false** at the end of elaboration. Therefore, care should be taken when calling **sc_start** in a loop conditional on any of the functions that detect pending activity.

Example:

```
int sc_main( int argc, char* argv[] ) {
    using namespace sc_core;

    // Instantiate top-level module
    ...

    sc_start( SC_ZERO_TIME );                    // Run the initialization phase to create pending activity

    while( sc_pending_activity() ) {
        sc_start( sc_time_to_pending_activity() );   // Run up to the next activity
    }

    return 0;
}
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

4.6.9 sc_get_status

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

```
sc_status sc_get_status();
```

The function shall return a value of type **sc_status** to indicate the current phase of elaboration or simulation as defined by Table 1.

Table 1—Value returned by sc_get_status

Value	When called
SC_ELABORATION	During the construction of the module hierarchy, or if **sc_start** is being used, before the first call to **sc_start**
SC_BEFORE_END_OF_ELABORATION	From the callback function **before_end_of_elaboration**
SC_END_OF_ELABORATION	From the callback function **end_of_elaboration**
SC_START_OF_SIMULATION	From the callback function **start_of_simulation**
SC_RUNNING	From the initialization, evaluation, or update phase
SC_PAUSED	When the scheduler is not running and **sc_pause** has been called
SC_SUSPENDED	When the scheduler is not running because it has been suspended, due to an explicit request or because it ran out of pending notifications and time-outs, with at least one primitive channel attached for suspension
SC_STOPPED	When the scheduler is not running and **sc_stop** has been called
SC_END_OF_SIMULATION	From the callback function **end_of_simulation**

The function **sc_get_status** shall be thread-safe, i.e., it can be also called reliably from an operating system thread other than those in which the SystemC kernel and any SystemC thread processes are executing.

When called from **before_end_of_elaboration**, the function **sc_get_status** shall return SC_BEFORE_END_OF_ELABORATION even when called from a constructor (of a module or other object) that is called directly or indirectly from **before_end_of_elaboration**. In other words, **sc_get_status** permits the application to distinguish between the construction of the module hierarchy defined in 4.4 and the extended construction of the module hierarchy defined in 4.5.2.

When **sc_start** is being used, **sc_get_status** shall return SC_ELABORATION when called before the first call to **sc_start** and shall return SC_PAUSED, SC_SUSPENDED, or SC_STOPPED when called between or after calls to **sc_start**.

If **sc_pause** and **sc_stop** have both been called, **sc_get_status** shall return SC_STOPPED.

When called from **end_of_simulation**, the function **sc_get_status** shall return SC_END_OF_SIMULATION irrespective of whether **sc_pause** or **sc_stop** have been called.

Example:

```
{
    using namespace sc_core;
    if (sc_get_status() & (SC_RUNNING | SC_PAUSED | SC_STOPPED) )
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    ...
}
```

Figure 1 shows the possible transitions between the values of type **sc_status** as returned from function **sc_get_status** when called during the various phases of simulation.

NOTE—When **sc_start** is not being used, the mechanisms for running elaboration and simulation are implementation-defined, as are the callbacks to **end_of_simulation** in the absence of a call to **sc_stop** (see 4.4 and 4.5.5).

4.6.10 sc_stage_callback_if

4.6.10.1 Description

Class **sc_stage_callback_if** provides the interface to enable user-defined callbacks during elaboration or simulation at simulation stages that are otherwise not available to the application.

4.6.10.2 Class definition

```
namespace sc_core {

    class sc_stage_callback_if
    {
    public:
        virtual ~sc_stage_callback_if() {}

        virtual void stage_callback(const sc_stage & stage) = 0;
    };
}       // namespace sc_core
```

4.6.10.3 stage_callback

Member function **stage_callback** shall be called by the implementation when the simulation reaches the corresponding stage, and the callback has been previously registered using function **sc_register_stage_callback** (see 4.6.11). An implementation shall stop calling the member function **stage_callback** when the callback has been unregistered using function **sc_unregister_stage_callback** (see 4.6.12).

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

- The instantiation of objects of class **sc_module**, **sc_port**, **sc_export**, **sc_prim_channel**.
- Port and export binding.
- Macros **SC_CTOR**, **SC_METHOD**, **SC_THREAD**, and **SC_CTHREAD**.
- Member functions **reset_signal_is** and **async_reset_signal_is** of the class **sc_module**.
- Calls to event finder functions.
- Calls to member function **notify** of class **sc_event** with or without argument to create immediate, delta or timed notifications.
- Calls to function **sc_spawn** to create static, spawned processes.
- Calls to member functions **request_update** or **async_request_update** of class **sc_prim_channel** to create update requests.
- Calls to the process control member functions **kill**, **reset**, **throw_it**, **suspend**, **resume**, **disable**, **enable**, **sync_reset_on**, and **sync_reset_off** of class **sc_process_handle**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

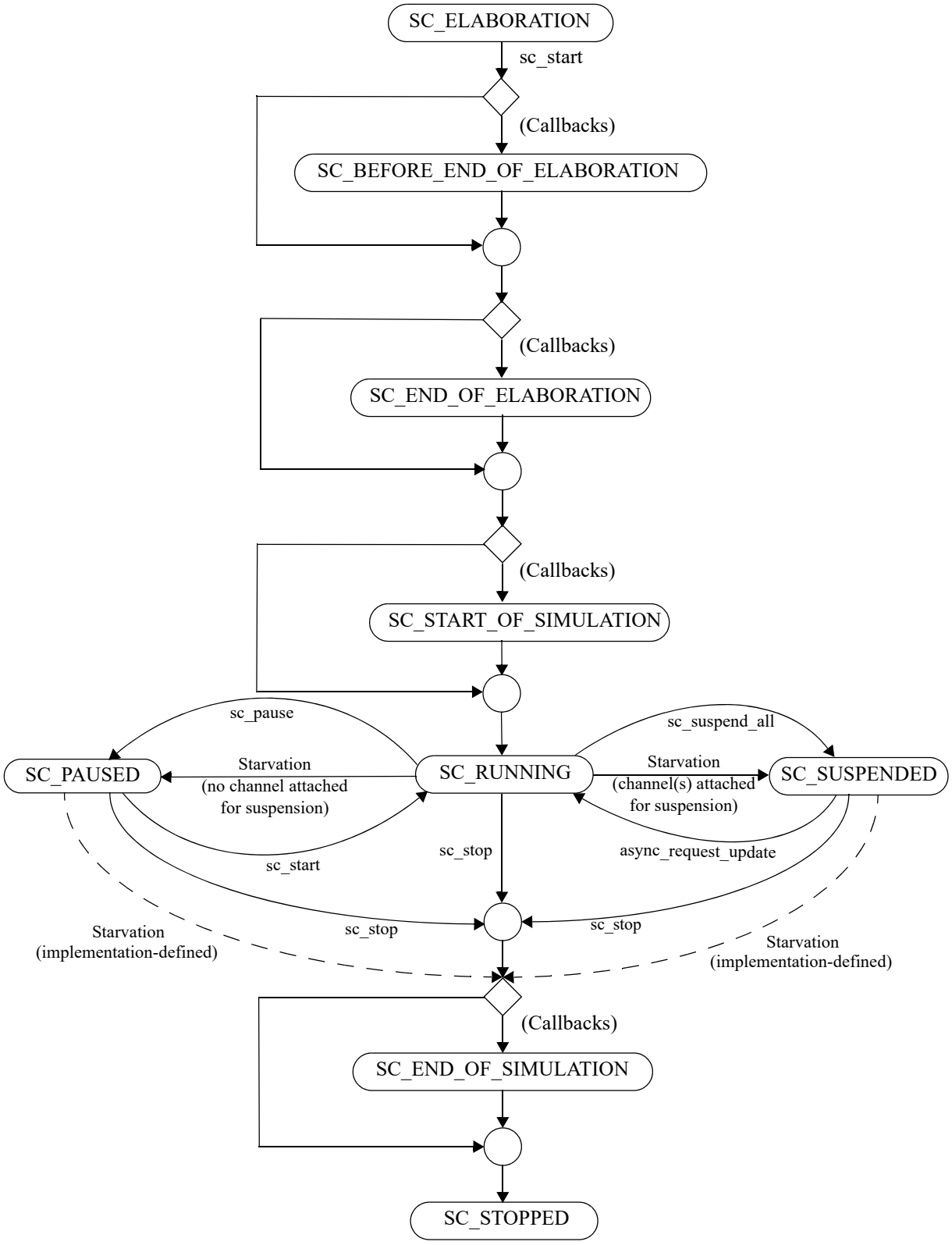

Figure 1—Transitions between values returned by sc_get_status

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

An application may call function **sc_get_status** in this callback, but the return value is implementation-defined.

4.6.11 sc_register_stage_callback

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

void **sc_register_stage_callback**(const sc_stage_callback_if&, int)

Function **sc_register_stage_callback** shall register a callback for the object of type **sc_stage_callback_if** passed as first argument. An implementation shall support bitwise operations of the enumeration type **sc_stage** passed as second argument.

4.6.12 sc_unregister_stage_callback

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

void **sc_unregister_stage_callback**(const sc_stage_callback_if& , int)

Function **sc_unregister_stage_callback** shall unregister a stage callback for the object of type **sc_stage_callback_if** passed as first argument. An implementation shall support bitwise operations of the enumeration type **sc_stage** passed as second argument. An implementation may generate a warning in case a simulation phase callback is unregistered without prior registration but is not obliged to do so.

4.6.13 Callbacks execution

The callbacks registered with **sc_register_stage_callback** shall be called at specific stages of the simulation, based on the actual enumeration constants of **sc_stage**:

- SC_POST_BEFORE_END_OF_ELABORATION: after execution of all **before_end_of_elaboration** callbacks.
- SC_POST_END_OF_ELABORATION: after execution of all **end_of_elaboration** callbacks.
- SC_POST_START_OF_SIMULATION: after execution of all **start_of_simulation** callbacks.
- SC_POST_UPDATE: between the end of the Update phase and the beginning of the delta notification phase.
- SC_PRE_TIMESTEP: during the Timed notification phase, before the simulator advances the simulation time to the time of the earliest pending timed notification or time-out.
- SC_PRE_PAUSE: before the simulator is about to return control from **sc_start**, due to **sc_pause** having been called.
- SC_PRE_SUSPEND: before the simulator is about to enter the suspended state.
- SC_POST_SUSPEND: before the simulator is about to exit the suspended state.
- SC_PRE_STOP: before the simulator is about to return control from **sc_start**, due to **sc_stop** having been called.
- SC_POST_END_OF_SIMULATION: after execution of all **end_of_simulation** callbacks.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

5. Core language class definitions

5.1 Class header files

5.1.1 Overview

To use the SystemC class library features, an application shall include either of the C++ header files specified in 5.1 at appropriate positions in the source code as required by the scope and linkage rules of C++. The TLM-1 and TLM-2.0 classes and TLM-2.0 utilities are in separate header files (see 10.8).

5.1.2 #include "systemc"

The header file named **systemc** shall add the names **sc_core**, **sc_dt**, and **sc_unnamed** to the declarative region in which it is included, and these three 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;
```

5.1.3 #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 the name **sc_unnamed** and 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;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
...

// 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 {
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
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;
        virtual const std::vector<sc_event*>& get_child_events() 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_inout<bool>& , bool );
        void reset_signal_is( const sc_out<bool>& , bool );
        void reset_signal_is( const sc_signal_in_if<bool>& , bool );

        void async_reset_signal_is( const sc_in<bool>& , bool );
        void async_reset_signal_is( const sc_inout<bool>& , bool );
        void async_reset_signal_is( const sc_out<bool>& , bool );
        void async_reset_signal_is( const sc_signal_in_if<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( const sc_event_or_list & );
        void next_trigger( const 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& , const sc_event_or_list &);
        void next_trigger( double , sc_time_unit , const sc_event_or_list & );
        void next_trigger( const sc_time& , const sc_event_and_list & );
        void next_trigger( double , sc_time_unit , const sc_event_and_list & );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

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

private:
        // Disabled
        sc_module( const sc_module& );
        sc_module& operator= ( const sc_module& );
};

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

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

#define SC_MODULE(name)          struct name : sc_module
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
#define SC_CTOR(name, ...)          implementation-defined; name(sc_module_name, implementation-defined)
 #define SC_NAMED(identifier, ...)  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_core::sc_module_name) {}
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    ...
};

class M : public sc_core::sc_module {
    public:
    M(sc_core::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 shall be provided for convenience when declaring or defining a constructor of a module. Macro SC_CTOR shall be used only 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 mandatory argument to the macro. Additional arguments of the macro can be added optionally to declare further constructor parameters.

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

Macro SC_CTOR may also be used to add user-defined arguments to the constructor.

Example:

```
SC_MODULE(M1) {
    SC_CTOR(M1, int a, int b)                   // Additional constructor parameters
    {}
    ...
};
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
SC_MODULE(M2) {
    M2(sc_core::sc_module_name nm, int a, int b)   // Additional constructor parameters
    : sc_core::sc_module(nm) {}
    ...
};
```

The use of macro SC_CTOR is not obligatory, but shall be used at most once per module. If an application needs to pass the **sc_module_name** argument at a position other than the first, such constructors shall be provided explicitly.

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 declare the constructor with a parameter of type **sc_module_name** at the first parameter.

5.2.8 SC_METHOD, SC_THREAD, SC_CTHREAD

The argument passed to the macro SC_METHOD or SC_THREAD or the first argument passed to SC_CTHREAD shall be the name of a member function. The macro shall associate that function with a *method process instance*, a *thread process instance*, or a *clocked thread process instance*, respectively. This shall be the only way in which an unspawned process instance can be created (see 4.2.3).

The second argument passed to the macro SC_CTHREAD shall be an expression of the type **sc_event_finder**.

In each case, the macro invocation shall be terminated with a semicolon.

These three macros shall only be invoked in the body of the constructor, in the **before_end_of_elaboration** or **end_of_elaboration** callbacks of a module, or in a member function called from the constructor or callback. Macro SC_CTHREAD shall not be invoked from the **end_of_elaboration** callback. The first argument shall be the name of a member function of that same module.

A member function associated with an unspawned process instance shall have a return type of **void** and shall have no arguments. (Note that a function associated with a spawned process instance may have a return type and may have arguments.)

A single member function can be associated with multiple process instances within the same module. Each process instance is a distinct object of a class derived from class **sc_object**, and each macro shall use the member function name (in quotation marks) as the string name ultimately passed as an argument to the constructor of the base class sub-object of class **sc_object**. Each process instance can have its own static sensitivity and shall be triggered or resumed independently of other process instances.

Associating a member function with a process instance does not impose any explicit restrictions on how that member function may be used by the application. For example, such a function may be called directly by the application, as well as by the kernel.

A process instance can be suspended, resumed, disabled, enabled, killed, or reset using the member functions of class **sc_process_handle**.

Example:

```
SC_MODULE(M) {
    sc_core::sc_in<bool> clk;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    SC_CTOR(M)
    : clk("clk") {
        SC_METHOD(a_method);
        SC_THREAD(a_thread);
        SC_CTHREAD(a_cthread, clk.pos());
    }

    void a_method();
    void a_thread();
    void a_cthread();
    ...
};
```

5.2.9 SC_NAMED

Macro SC_NAMED shall be provided to declare a variable with a matching name. This macro passes the identifier converted to a string as (first) constructor parameter.

Example:

```
SC_MODULE(M) {
    sc_core::sc_in<bool> port_i;
    sc_core::sc_out<bool> SC_NAMED(port_o);    // in-class member initializer
    sc_core::sc_signal<bool> sig;

    SC_CTOR(M)
    : SC_NAMED(port_i)
    , SC_NAMED(sig, true)                      // supports multiple parameters
    {}
    ...
};
```

5.2.10 Method process

This subclause shall apply to both spawned and unspawned process instances.

A method process is said to be *triggered* when the kernel calls the function associated with the process instance. When a method process is triggered, the associated function executes from beginning to end, then returns control to the kernel. A method process can only be *terminated* by calling the **kill** method of a process handle associated with that process.

A method process instance may have static sensitivity. A method process, and only a method process, may call the function **next_trigger** to create dynamic sensitivity. Function **next_trigger** is a member function of class **sc_module**, a member function of class **sc_prim_channel**, and a non-member function.

A method process instance cannot be made runnable as a result of an immediate notification executed by the process itself, regardless of the static sensitivity or dynamic sensitivity of the method process instance (see 4.3.2.3).

An implementation is not obliged to run a method process in a separate software thread. A method process may run in the same execution context as the simulation kernel.

Member function **kind** of the implementation-defined class associated with a method process instance shall return the string **"sc_method_process"**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

NOTE 1—Any local variables declared within the process will be destroyed on return from the process. Data members of the module should be used to store persistent state associated with the method process.

NOTE 2—Function **next_trigger** can be called from a member function of the module itself, from a member function of a channel, or from any function subject only to the rules of C++, provided that the function is ultimately called from a method process.

5.2.11 Thread and clocked thread processes

This subclause shall apply to both spawned and unspawned process instances.

A function associated with a thread or clocked thread process instance is called once and only once by the kernel, except when a process is reset; in which case, the associated function may be called again (see 5.2.12).

A thread or clocked thread process, and only such a process, may call the function **wait**. Such a call causes the calling process to suspend execution. Function **wait** is a member function of class **sc_module**, a member function of class **sc_prim_channel**, and a non-member function.

A thread or clocked thread process instance is said to be *resumed* when the kernel causes the process to continue execution, starting with the statement immediately following the most recent call to function **wait**. When a thread or clocked thread process is resumed, the process executes until it reaches the next call to function **wait**. Then, the process is suspended once again.

A thread process instance may have static sensitivity. A thread process instance may call function **wait** to create dynamic sensitivity. A clocked thread process instance is statically sensitive only to a single clock.

A thread or clocked thread process instance cannot be made runnable as a result of an immediate notification executed by the process itself, regardless of the static sensitivity or dynamic sensitivity of the thread process instance (see 4.3.2.3).

Each thread or clocked thread process requires its own execution stack. As a result, context switching between thread processes may impose a simulation overhead when compared with method processes.

If the thread or clocked thread process executes the entire function body or executes a return statement and thus returns control to the kernel, the associated function shall not be called again for that process instance. The process instance is then said to be *terminated*.

Member function **kind** of the implementation-defined class associated with a thread process instance shall return the string **"sc_thread_process"**.

Member function **kind** of the implementation-defined class associated with a clocked thread process instance shall return the string **"sc_cthread_process"**.

NOTE 1—It is a common coding idiom to include an infinite loop containing a call to function **wait** within a thread or clocked thread process in order to avoid the process from terminating prematurely.

NOTE 2—When a process instance is resumed, any local variables defined within the process will retain the values they had when the process was suspended.

NOTE 3—If a thread or clocked thread process executes an infinite loop that does not call function **wait**, the process will never suspend. Since the scheduler is not preemptive, no other process will be able to execute.

NOTE 4—Function **wait** can be called from a member function of the module itself, from a member function of a channel, or from any function subject only to the rules of C++, provided that the function is ultimately called from a thread or clocked thread process.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Example:

```
SC_MODULE(synchronous_module) {
    sc_in<bool> SC_NAMED(clock);

    SC_CTOR(synchronous_module) {
        SC_THREAD(thread);
        sensitive << clock.pos();
    }

    void thread() {                    // Member function called once only
        for (;;) {
            wait();                    // Resume on positive edge of clock
            ...
        }
    }
    ...
};
```

5.2.12 Clocked thread processes

A clocked thread process shall be a static process; clocked threads cannot be spawned processes.

A clocked thread process shall be statically sensitive to a single clock, as determined by the event finder passed as the second argument to macro SC_CTHREAD. The clocked thread process shall be statically sensitive to the event returned from the given event finder.

A clocked thread process may call either of the following functions:

void **wait**();
void **wait**(int);

It shall be an error for a clocked thread process to call any other overloaded form of the function **wait**.

A clocked thread process may have any number of synchronous and asynchronous reset signals specified using **reset_signal_is** and **async_reset_signal_is**, respectively.

Any of the process control member functions of class **sc_process_handle** may be called for a clocked thread process, although certain caveats apply to the use of member functions **suspend**, **resume**, **reset**, and **throw_it** due to the fact that they can be called asynchronously with respect to the clock. These caveats are described below.

It is recommended to call **disable** and **enable** rather than **suspend** and **resume** for clocked thread processes. If **resume** is called for a clocked thread process, the call shall be made during the evaluation phase that immediately follows the delta notification or timed notification phase in which the clock event notification occurs. If a clocked thread is resumed at any other time, that is, if the call to **resume** is not synchronized with the clock, the behavior shall be implementation-defined.

The process control member functions **kill**, **reset**, and **throw_it** may be called asynchronously with respect to the clock and their effect is immediate, even in the case of a clocked thread process. Similarly, the effect of an asynchronous reset attaining its active value is immediate for a clocked thread process.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

In summary, a clocked thread process shall have exactly one clock signal and may have any number of synchronous and asynchronous reset signals as well as being reset by calling the process control member function of class **sc_process_handle**. As a consequence, the body of the associated function should normally consist of reset behavior followed by a loop statement that contains a call to **wait** followed by the behavior to be executed in each clock cycle (see the example below).

The first time the clock event is notified, the function associated with a clocked thread process shall be called whether or not the reset signal is active. If a clocked thread process instance has been terminated, the clock event shall be ignored for that process instance. A terminated process cannot be reset.

Example:

```
SC_MODULE(M) {
    sc_core::sc_in<bool> SC_NAMED(clock);
    sc_core::sc_in<bool> SC_NAMED(reset);
    sc_core::sc_in<bool> SC_NAMED(async_reset);
    ...

    SC_CTOR(M) {
        SC_CTHREAD(CT1, clock.pos());
            reset_signal_is(reset, true);

        SC_CTHREAD(CT2, clock.pos());
            async_reset_signal_is(async_reset, true);
    }

    void CT1() {
        if (reset.read()) {
        ...                             // Reset actions
        }
        while(true) {
            wait(true);                 // Wait for 1 clock cycle
            ...                         // Clocked actions
        }
    }

    void CT2() {
        ...                             // Reset actions
        while(true) {
            try {
                while (true) {
                    wait();             // Wait for 1 clock cycle
                    ...                 // Regular behavior
                }
            } catch (...) {             // Some exception has been thrown
                ...                     // Handle exception and go back to waiting for clock
            }
        }
    }
    ...
};
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

5.2.13 reset_signal_is and async_reset_signal_is

void **reset_signal_is**(const sc_in<bool>& , bool);
void **reset_signal_is**(const sc_inout<bool>& , bool);
void **reset_signal_is**(const sc_out<bool>& , bool);
void **reset_signal_is**(const sc_signal_in_if<bool>& , bool);

void **async_reset_signal_is**(const sc_in<bool>& , bool);
void **async_reset_signal_is**(const sc_inout<bool>& , bool);
void **async_reset_signal_is**(const sc_out<bool>& , bool);
void **async_reset_signal_is**(const sc_signal_in_if<bool>& , bool);

Member functions **reset_signal_is** and **async_reset_signal_is** of class **sc_module** shall determine the synchronous and asynchronous reset signals, respectively, of a thread, clocked thread, or method process. These two member functions shall only be called in the body of the constructor, in the **before_end_of_elaboration** callback of a module, or in a member function called from the constructor or callback, and only after having created a process instance within that same constructor or callback.

The order of execution of the statements within the body of the constructor or the **before_end_of_elaboration** callback shall be used to associate the call to **reset_signal_is** or **async_reset_signal_is** with a particular process instance; the call is associated with the most recently created process instance. If a module is instantiated within the constructor or callback between the process being created and function **reset_signal_is** or **async_reset_signal_is** being called, the effect of calling **reset_signal_is** or **async_reset_signal_is** shall be undefined.

The first argument passed to function **reset_signal_is** or **async_reset_signal_is** shall identify the signal instance to be used as the reset. The signal may be identified indirectly by passing a port instance that is bound to the reset signal. The second argument shall be the active level of the reset signal, meaning that the process is to be reset only when the value of the reset signal is equal to the value of this second argument.

When the reset signal (as specified by the first argument) attains its active value (as specified by the second argument), the process instance shall enter the *synchronous reset state*. While in the synchronous reset state, a process instance shall be reset each time it is resumed, whether due to an event notification or to a time-out. Being reset in this sense shall have the same effect as would a call to the member function **reset** of a process handle associated with the given process instance. The synchronous reset state is described more fully in 5.6.6.4.

If a process instance enters the synchronous reset state while it is suspended (meaning that **suspend** has been called), the behavior shall be implementation-defined (see 5.6.6.12).

In the case of **async_reset_signal_is** only, the process shall also be reset whenever the reset signal attains its active value. Being reset in this sense shall have the same effect as would a call to the member function **reset** of a process handle associated with the given process instance. Since the reset signal is itself a primitive channel that can only change value in the update phase, the process instance shall be reset by effectively calling the member function **reset** of an associated process handle at some time during the evaluation phase immediately following the reset signal attaining its active value and in an order determined by the kernel. An asynchronous reset does not take priority over other processes that are due to run in the same evaluation phase.

When an asynchronous reset signal attains its active value, the consequent reset shall have the same priority as a call to **reset**. When a process instance is reset while in the synchronous reset state, the consequent reset shall have the same priority as **sync_reset_on**, regardless of whether the reset signal is synchronous or asynchronous (see 5.6.6.12).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

A given process instance may have any number of synchronous and asynchronous reset signals. When all such reset signals attain the negation of their active value, the process instance shall leave the synchronous reset state unless a call to **sync_reset_on** is in force for the given process instance (see 5.6.6.4).

reset_signal_is and **async_reset_signal_is** are permitted for method processes. The effect of resetting a method process shall be to cancel any dynamic sensitivity, to restore the static sensitivity, and to call the function associated with that process instance.

Example:

```
SC_MODULE(M) {
    sc_core::sc_in<bool> SC_NAMED(clock);
    sc_core::sc_in<bool> SC_NAMED(reset);
    ...

    SC_CTOR(M) {
        SC_METHOD(rtl_proc);
        sensitive << clock.pos();
        async_reset_signal_is(reset, true);
        ...
    }

    void rtl_proc() {
        if (reset.read()) {
        ...                          // Asynchronous reset behavior, executed whenever the reset is active
        }
        else {
        ...                          // Synchronous behavior, only executed on a positive clock edge
        }
    }
    ...
};
```

5.2.14 sensitive

sc_sensitive† sensitive;

This subclause describes the static sensitivity of an unspawned process. Static sensitivity for a spawned process is created using member function **set_sensitivity** of class **sc_spawn_options** (see 5.5).

Data member **sensitive** of class **sc_module** can be used to create the static sensitivity of an unspawned process instance using **operator<<** of class *sc_sensitive*† (see 5.4). This shall be the only way to create static sensitivity for an unspawned process instance. However, static sensitivity may be enabled or disabled by calling function **next_trigger** (see 5.2.17) or function **wait** (see 5.2.18).

Static sensitivity shall only be created in the body of the constructor, in the **before_end_of_elaboration** or **end_of_elaboration** callbacks of a module, or in a member function called from the constructor or callback, and only after having created an unspawned process instance within that same constructor or callback. It shall be an error to modify the static sensitivity of an unspawned process during simulation.

The order of execution of the statements within the body of the constructor or the **before_end_of_elaboration** or **end_of_elaboration** callbacks is used to associate static sensitivity with a particular unspawned process instance; sensitivity is associated with the process instance most recently created within the body of the current constructor or callback.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

A clocked thread process cannot have static sensitivity other than to the clock itself. Using data member **sensitive** to create static sensitivity for a clocked thread process shall have no effect.

NOTE 1—Unrelated statements may be executed between creating an unspawned process instance and creating the static sensitivity for that same process instance. Static sensitivity may be created in a different function body from the one in which the process instance was created.

NOTE 2—Data member **sensitive** can be used more than once to add to the static sensitivity of any particular unspawned process instance; each call to **operator<<** adds further events to the static sensitivity of the most recently created process instance.

5.2.15 dont_initialize

void **dont_initialize**();

This subclause describes member function **dont_initialize** of class **sc_module**, which determines the behavior of an unspawned process instance during initialization. The initialization behavior of a spawned process is determined by the member function **dont_initialize** of class **sc_spawn_options** (see 5.5).

Member function **dont_initialize** of class **sc_module** shall have the effect that a particular unspawned process instance will not be made runnable during the initialization phase of the scheduler. In other words, the member function associated with the given process instance shall not be called by the scheduler until the process instance is triggered or resumed because of the occurrence of an event.

dont_initialize shall only be called in the body of the constructor, in the **before_end_of_elaboration** or **end_of_elaboration** callbacks of a module, or in a member function called from the constructor or callback, and only after having created an unspawned process instance within that same constructor or callback.

The order of execution of the statements within the body of the constructor or the **before_end_of_elaboration** or **end_of_elaboration** callbacks is used to associate the call to **dont_initialize** with a particular unspawned process instance; it is associated with the most recently created process instance. If a module is instantiated within the constructor or callback between the process being created and function **dont_initialize** being called, the effect of calling **dont_initialize** shall be undefined.

dont_initialize shall have no effect if called for a clocked thread process, which is not made runnable during the initialization phase in any case. An implementation may generate a warning but is not obliged to do so.

Example:

```
SC_MODULE(Mod) {
    sc_core::sc_signal<bool> SC_NAMED(A), SC_NAMED(B), SC_NAMED(C),
                             SC_NAMED(D), SC_NAMED(E);
    SC_CTOR(Mod) {
    sensitive << A;                 // Has no effect. Not recommended coding style.

    SC_THREAD(T);
    sensitive << B << C;            // Thread process T is made sensitive to B and C.

    SC_METHOD(M);
    f();                            // Method process M is made sensitive to D.
    sensitive << E;                 // Method process M is made sensitive to E as well as D.
    }

    void f() { sensitive << D; }    // Not recommended coding style.
    void T();
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    void M();
    ...
};
```

5.2.16 set_stack_size

void **set_stack_size**(size_t);

This subclause describes member function **set_stack_size** of class **sc_module**, which sets the stack size of an unspawned process instance during initialization. The stack size of a spawned process is set by the member function **set_stack_size** of class **sc_spawn_options** (see 5.5).

An application may call member function **set_stack_size** to request a change to the size of the execution stack for the thread or clocked thread process instance for which the function is called. The effect of this function is implementation-defined.

set_stack_size shall only be called in the body of the constructor, in the **before_end_of_elaboration** or **end_of_elaboration** callbacks of a module, or in a member function called from the constructor or callback, and only after having created an unspawned process instance within that same constructor or callback. It shall be an error to call **set_stack_size** at other times or to call **set_stack_size** for a method process instance.

The order of execution of the statements within the body of the constructor or the **before_end_of_elaboration** or **end_of_elaboration** callbacks is used to associate the call to **set_stack_size** with a particular unspawned process instance; it is associated with the most recently created unspawned process instance.

5.2.17 next_trigger

This subclause shall apply to both spawned and unspawned process instances.

This subclause shall apply to member function **next_trigger** of class **sc_module**, member function **next_trigger** of class **sc_prim_channel**, and non-member function **next_trigger**.

When called with one or more arguments, the function **next_trigger** shall set the dynamic sensitivity of the method process instance from which it is called for the very next occasion on which that process instance is triggered, and for that occasion only. The dynamic sensitivity is determined by the arguments passed to function **next_trigger**.

If function **next_trigger** is called more than once during a single execution of a particular method process instance, the last call to be executed shall prevail. The effects of earlier calls to function **next_trigger** for that particular process instance shall be cancelled.

If function **next_trigger** is not called during a particular execution of a method process instance, the method process instance shall next be triggered according to its static sensitivity.

A call to the function **next_trigger** with one or more arguments shall override the static sensitivity of the process instance.

It shall be an error to call function **next_trigger** from a thread or clocked thread process.

NOTE—The function **next_trigger** does not suspend the method process instance; a method process cannot be suspended but always executes to completion before returning control to the kernel.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

void **next_trigger**();

The process shall be triggered on the static sensitivity. In the absence of static sensitivity for this particular process instance, the process shall not be triggered again during the current simulation. Calling **next_trigger** with an empty argument list shall cancel the effect of any earlier calls to **next_trigger** during the current execution of the method process instance and, thus, in effect shall be equivalent to not calling **next_trigger** at all during a given process execution.

void **next_trigger**(const sc_event&);

The process shall be triggered when the event passed as an argument is notified.

void **next_trigger**(const sc_event_or_list &);

The argument shall take the form of a list of events separated by the **operator|** of class **sc_event** or **sc_event_or_list**. The process shall be triggered when any one of the given events is notified. The occurrence or non-occurrence of the other events in the list shall have no effect on that particular triggering of the process. If a particular event appears more than once in the list, the behavior shall be the same as if that event had appeared only once. It shall be an error to pass an empty event list object as an argument to **next_trigger** (see 5.8 and 5.9).

void **next_trigger**(const sc_event_and_list &);

The argument shall take the form of a list of events separated by the **operator&** of class **sc_event** or **sc_event_and_list**. In order for the process to be triggered, every single one of the given events shall be notified, with no explicit constraints on the time or order of those notifications. The process is triggered when the last such event is notified, last in the sense of being at the latest point in simulation time, not last in the list. An event in the list may be notified more than once before the last event is notified. If a particular event appears more than once in the list, the behavior shall be the same as if that event had appeared only once. It shall be an error to pass an empty event list object as an argument to **next_trigger** (see 5.8 and 5.9).

void **next_trigger**(const sc_time&);

The process shall be triggered after the time given as an argument has elapsed. The time shall be taken to be relative to the time at which function **next_trigger** is called. When a process is triggered in this way, a *time-out* is said to have occurred. In this method call and in those that follow below, the argument that specifies the time-out shall not be negative.

void **next_trigger**(double v , sc_time_unit tu);

is equivalent to the following:

void next_trigger(sc_time(v , tu));

void **next_trigger**(const sc_time& , const sc_event&);

The process shall be triggered after the given time or when the given event is notified, whichever occurs first.

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

Each of these compound forms combines a time with an event or event list. The semantics of these compound forms shall be deduced from the rules given for the simple forms. In each case, the process shall be triggered after the given time-out or in response to the given event or event list, whichever is satisfied first.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Example:

```
SC_MODULE(M) {
    SC_CTOR(M) {
        SC_METHOD(entry);
        sensitive << sig;
    }

    void entry() {                                          // Run first at initialization.
        if (sig == 0) next_trigger(e1 | e2);                // Trigger on event e1 or event e2 next time
        else if (sig == 1) next_trigger(1, sc_core::SC_NS); // Time-out after 1 nanosecond.
        else next_trigger();                                // Trigger on signal sig next time.
    }

    sc_core::sc_signal<int> SC_NAMED(sig);
    sc_core::sc_event SC_NAMED(e1), SC_NAMED(e2);
    ...
};
```

5.2.18 wait

This subclause shall apply to both spawned and unspawned process instances.

In addition to causing the process instance to suspend, the function **wait** may set the dynamic sensitivity of the thread or clocked thread process instance from which it is called for the very next occasion on which that process instance is resumed, and for that occasion only. The dynamic sensitivity is determined by the arguments passed to function **wait**.

A call to the function **wait** with an empty argument list or with a single integer argument shall use the static sensitivity of the process instance. This is the only form of **wait** permitted within a clocked thread process.

A call to the function **wait** with one or more non-integer arguments shall override the static sensitivity of the process instance.

When calling function **wait** with a passed-by-reference parameter, the application shall be obliged to ensure that the lifetimes of any actual arguments passed by reference extend from the time the function is called to the time the function call reaches completion, and moreover in the case of a parameter of type **sc_time**, the application shall not modify the value of the actual argument during that period.

It shall be an error to call function **wait** from a method process.

void **wait**();

> The process shall be resumed on the static sensitivity. In the absence of static sensitivity for this particular process, the process shall not be resumed again during the current simulation.

void **wait**(int);

> A call to this function shall be equivalent to calling the function **wait** with an empty argument list for a number of times in immediate succession, the number of times being passed as the value of the argument. It shall be an error to pass an argument value less than or equal to zero. The implementation is expected to optimize the execution speed of this function for clocked thread processes.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

void **wait**(const sc_event&);

The process shall be resumed when the event passed as an argument is notified.

void **wait**(const sc_event_or_list &);

The argument shall take the form of a list of events separated by the **operator|** of classes **sc_event** and **sc_event_or_list**. The process shall be resumed when any one of the given events is notified. The occurrence or non-occurrence of the other events in the list shall have no effect on the resumption of that particular process. If a particular event appears more than once in the list, the behavior shall be the same as if it appeared only once. It shall be an error to pass an empty event list object as an argument to **wait** (see 5.8 and 5.9).

void **wait**(const sc_event_and_list &);

The argument shall take the form of a list of events separated by the **operator&** of classes **sc_event** and **sc_event_and_list**. In order for the process to be resumed, every single one of the given events shall be notified, with no explicit constraints on the time or order of those notifications. The process is resumed when the last such event is notified, last in the sense of being at the latest point in simulation time, not last in the list. An event in the list may be notified more than once before the last event is notified. If a particular event appears more than once in the list, the behavior shall be the same as if it appeared only once. It shall be an error to pass an empty event list object as an argument to **wait** (see 5.8 and 5.9).

void **wait**(const sc_time&);

The process shall be resumed after the time given as an argument has elapsed. The time shall be taken to be relative to the time at which function **wait** is called. When a process is resumed in this way, a *time-out* is said to have occurred. In this method call and in those that follow below, the argument that specifies the time-out shall not be negative.

void **wait**(double v , sc_time_unit tu);

is equivalent to the following:

void wait(sc_time(v, tu));

void **wait**(const sc_time& , const sc_event&);

The process shall be resumed after the given time or when the given event is notified, whichever occurs first.

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

Each of these compound forms combines a time with an event or event list. The semantics of these compound forms shall be deduced from the rules given for the simple forms. In each case, the process shall be resumed after the given time-out or in response to the given event or event list, whichever is satisfied first.

5.2.19 Positional port binding

Ports can be bound using either positional binding or named binding. Positional binding is performed using the **operator()** defined in the current subclause. Named binding is performed using the **operator()** or the function **bind** of the class **sc_port** (see 5.12).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

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

This operator shall bind the port instances within the module instance for which the operator is called to the channel instances and port instances passed as actual arguments to the operator, the port order being determined by the order in which the ports were constructed. The first port to be constructed shall be bound to the first argument, the second port to the second argument, and so forth. It shall be an error if the number of actual arguments is greater than the number of ports to be bound.

A multiport instance (see 5.12.3) shall be treated as a single port instance when positional binding is used and may only be bound once, to a single channel instance or port instance. However, if a multiport instance P is bound by position to another multiport instance Q, the child multiport P may be bound indirectly to more than one channel through the parent multiport Q. A given multiport shall not be bound both by position and by name.

This operator shall only bind ports, not exports. Any export instances contained within the module instance shall be ignored by this operator.

An implementation may permit more than 64 ports to be bound in a single call to **operator()** by allowing more than 64 arguments but is not obliged to do so. **operator()** shall not be called more than once for a given module instance.

The following objects, and these alone, can be used as actual arguments to **operator()**:

a) A channel, which is an object of a class derived from class **sc_interface**
b) A port, which is an object of a class derived from class **sc_port**

The *type of a port* is the name of the interface passed as a template argument to class **sc_port** when the port is instantiated. The interface implemented by the channel in case a) or the type of the port in case b) shall be the same as or derived from the type of the port being bound.

An implementation may defer the completion of port binding until a later time during elaboration because the port to which a port is bound may not yet itself have been bound. Such deferred port binding shall be completed by the implementation before the callbacks to function **end_of_elaboration**.

NOTE 1—To bind more than 64 ports of a single module instance, named binding should be used.

NOTE 2—Class *sc_bind_proxy*†, the parameter type of **operator()**, may provide user-defined conversions in the form of two constructors, one having a parameter type of **sc_interface**, and the other a parameter type of **sc_port_base**.

NOTE 3—The actual argument cannot be an export, because this would require the C++ compiler to perform two implicit conversions. However, it is possible to pass an export as an actual argument by explicitly calling the user-defined conversion **sc_export::operator IF&**. It is also possible to bind a port to an export using named port binding.

Example:

```
SC_MODULE(M1) {
    sc_core::sc_inout<int> SC_NAMED(P), SC_NAMED(Q), SC_NAMED(R); // Ports
    SC_CTOR(M1) { ... }
    ...
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
};

SC_MODULE(Top1) {
    sc_core::sc_inout <int> SC_NAMED(A), SC_NAMED(B);
    sc_core::sc_signal<int> SC_NAMED(C);

    M1 m1;                          // Module instance

    SC_CTOR(Top1)
    : m1("m1") {
        m1(A, B, C);                // Binds P-to-A, Q-to-B, R-to-C
    }
    ...
};

SC_MODULE(M2) {
    sc_core::sc_inout<int> SC_NAMED(S);
    sc_core::sc_inout<int> *T;      // Pointer-to-port - not recommended coding style
    sc_core::sc_inout<int> SC_NAMED(U);

    SC_CTOR(M2) { T = new sc_inout<int>("T"); }
    ...
};

SC_MODULE(Top2) {
    sc_core::sc_inout <int> SC_NAMED(D), SC_NAMED(E);
    sc_core::sc_signal<int> SC_NAMED(F);
    M2 m2;                          // Module instance

    SC_CTOR(Top2)
    : m2("m2") {
        m2(D, E, F);                // Binds S-to-D, U-to-E, (*T)-to-F
                                    // Note that binding order depends on the order of port construction
    }
    ...
};
```

5.2.20 before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation

See 4.5.

5.2.21 get_child_objects and get_child_events

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

Member function **get_child_objects** shall return a **std::vector** containing a pointer to every instance of class **sc_object** that lies within the module in the object hierarchy. This shall include pointers to all module, port, primitive channel, unspawned process, and spawned process instances within the module and any other application-defined objects derived from class **sc_object** within the module.

virtual const std::vector<sc_event*>& **get_child_events**() const;

Member function **get_child_events** shall return a **std::vector** containing a pointer to every object of type **sc_event** that is a hierarchically named event and whose parent is the current module.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

NOTE 1—The phrase *within a module* does not include instances nested within modules instances but only includes the immediate children of the given module.

NOTE 2—An application can identify the instances by calling the member functions **name** and **kind** of class **sc_object** or can determine their types using a dynamic cast.

NOTE 3—An event may have a hierarchical name and may have a parent in the object hierarchy, but in any case events are not themselves part of the object hierarchy.

Example:

```
int sc_main (int argc, char* argv[]) {
    Top_level_module top("top");
    std::vector<sc_core::sc_object*> children = top.get_child_objects();

    // Print out names and kinds of top-level objects
    for (unsigned i = 0; i < children.size(); i++)
        std::cout << children[i]->name() << " " << children[i]->kind() << std::endl;

    sc_core::sc_start();
    return 0;
}
```

5.2.22 sc_gen_unique_name

const char* **sc_gen_unique_name**(const char* seed);

The function **sc_gen_unique_name** shall return a unique character string that depends on the context from which the function is called. For this purpose, each module instance shall have a separate space of unique string names, and there shall be a single global space of unique string names for calls to **sc_gen_unique_name** not made from within any module. These spaces of unique string names shall be maintained by function **sc_gen_unique_name** and are only visible outside this function in so far as they affect the value of the strings returned from this function. Function **sc_gen_unique_name** shall only guarantee the uniqueness of strings within each space of unique string names. There shall be no guarantee that the generated name does not clash with a string that was not generated by function **sc_gen_unique_name**.

The unique string shall be constructed by appending a string of two or more characters as a suffix to the character string passed as argument **seed**, subject to the rules given in the remainder of this subclause. The appended suffix shall take the form of a single underscore character, followed by a series of one of more decimal digits from the character set 0-9. The number and choice of digits shall be implementation-defined.

There shall be no restrictions on the character set of the **seed** argument to function **sc_gen_unique_name**. The **seed** argument may be the empty string.

String names are case-sensitive, and every character in a string name is significant. For example, "a", "A", “a_", and "A_" are each unique string names with respect to one another.

NOTE—The intended use of **sc_gen_unique_name** is to generate unique string names for objects of class **sc_object**. Class **sc_object** does impose restrictions on the character set of string names passed as constructor arguments. The value returned from function **sc_gen_unique_name** may be used for other unrelated purposes.

5.2.23 sc_behavior and sc_channel

typedef sc_module **sc_behavior**;

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

typedef sc_module **sc_channel**;

The typedefs **sc_behavior** and **sc_channel** are provided for users to express their intent.

NOTE—There is no distinction between a *behavior* and a hierarchical channel other than a difference of intent. Either may include both ports and public member functions.

Example:

```
class bus_interface : virtual public sc_core::sc_interface {
 public:
    virtual void write(int addr, int data) = 0;
    virtual void read (int addr, int& data) = 0;
};

class bus_adapter : public bus_interface, public sc_core::sc_channel {
 public:
    virtual void write(int addr, int data);              // Interface methods implemented in channel
    virtual void read (int addr, int& data);

    // Ports
    sc_core::sc_in<bool> SC_NAMED(clock);
    sc_core::sc_out<bool> SC_NAMED(wr), SC_NAMED(rd);
    sc_core::sc_out<int> SC_NAMED(addr_bus);
    sc_core::sc_out<int> SC_NAMED(data_out);
    sc_core::sc_in<int> SC_NAMED(data_in);

    SC_CTOR(bus_adapter) {
    ...                                                  // Module constructor
    }

 private:
    ...
};
```

5.3 sc_module_name

5.3.1 Description

Class **sc_module_name** acts as a container for the string name of a module and provides the mechanism for building the hierarchical names of instances in the module hierarchy during elaboration.

When an application creates an object of a class derived directly or indirectly from class **sc_module**, the application typically passes an argument of type **char*** to the module constructor, which itself has a single parameter of class **sc_module_name** and thus the constructor **sc_module_name(const char*)** is called as an implicit conversion. On the other hand, when an application derives a new class directly or indirectly from class **sc_module**, the derived class constructor calls the base class constructor with an argument of class **sc_module_name** and thus the copy constructor **sc_module_name(const sc_module_name&)** is called.

5.3.2 Class definition

```
namespace sc_core {
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
class sc_module_name
{
    public:
        sc_module_name( const char* );
        sc_module_name( const sc_module_name& );

        ~sc_module_name();

        operator const char*() const;

    private:
        // Disabled
        sc_module_name();
        sc_module_name& operator= ( const sc_module_name& );
};

}       // namespace sc_core
```

5.3.3 Constraints on usage

Class **sc_module_name** shall only be used as the type of a parameter of a constructor of a class derived from class **sc_module**. Moreover, every such constructor shall have exactly one parameter of type **sc_module_name**, which need not be the first parameter of the constructor.

In the case that the constructor of a class C derived directly or indirectly from class **sc_module** is called from the constructor of a class D derived directly from class C, the parameter of type **sc_module_name** of the constructor of class D shall be passed directly through as an argument to the constructor of class C. In other words, the derived class constructor shall pass the **sc_module_name** through to the base class constructor as a constructor argument.

NOTE 1—The macro SC_CTOR defines such a constructor.

NOTE 2—In the case of a class C derived directly from class **sc_module**, the constructor for class C is not obliged to pass the **sc_module_name** through to the constructor for class **sc_module**. The default constructor for class **sc_module** may be called explicitly or implicitly from the constructor for class C.

5.3.4 Module hierarchy

To keep track of the module hierarchy during elaboration, the implementation may maintain an internal stack of pointers to objects of class **sc_module_name**, referred to below as *the stack*. For the purpose of building hierarchical names, when objects of class **sc_module**, **sc_port**, **sc_export**, **sc_prim_channel**, or **sc_event** are constructed or when spawned or unspawned processes instances are created, they are assumed to exist within the module identified by the **sc_module_name** object on the top of the stack. In other words, each instance in the module hierarchy is named as if it were a child of the module identified by the item on the top of the stack at the point when the instance is created.

The implementation is not obliged to use these particular mechanisms (a stack of pointers), but if not, the implementation shall substitute an alternative mechanism that is semantically equivalent.

NOTE 1—The *hierarchical name* of an instance in the object hierarchy is returned from member function **name** of class **sc_object**, which is the base class of all such instances.

NOTE 2—An object of type **sc_event** may have a hierarchical name and may have a parent in the object hierarchy, but in any case **sc_event** is not derived from **sc_object** and events are not themselves part of the object hierarchy.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

5.3.5 Member functions

sc_module_name(const char*);

This constructor shall push a pointer to the object being constructed onto the top of the stack. The constructor argument shall be used as the string name of the module being instantiated within the module hierarchy by ultimately being passed as an argument to the constructor of class **sc_object**.

sc_module_name(const sc_module_name&);

This constructor shall copy the constructor argument but shall not modify the stack.

~sc_module_name();

If and only if the object being destroyed was constructed by **sc_module_name**(const char*), the destructor shall remove the **sc_module_name** pointer from the top of the stack.

operator const char*() const;

This conversion function shall return the string name (not the hierarchical name) associated with the **sc_module_name**.

NOTE 1—When a *complete object* of a class derived from **sc_module** is constructed, the constructor for that derived class is passed an argument of type **char***. The first constructor above will be called to perform an implicit conversion from type **char*** to type **sc_module_name**, thus pushing the newly created module name onto the stack and signifying the entry into a new level in the module hierarchy. On return from the constructor for the class of the complete object, the destructor for class **sc_module_name** will be called and will remove the module name from the stack.

NOTE 2—When an **sc_module_name** is passed as an argument to the constructor of a base class, the above copy constructor is called. The **sc_module_name** parameter of the base class may be unused. The reason for mandating that every such constructor have a parameter of class **sc_module_name** (even if the parameter is unused) is to ensure that every such derived class can be instantiated as a module in its own right.

Example:

```
struct A : sc_core::sc_module {
    A(sc_core::sc_module_name) {} // Calls sc_core::sc_module()
};

struct B : sc_core::sc_module {
    B(sc_core::sc_module_name n)
    : sc_core::sc_module(n) {}        // Calls sc_core::sc_module(sc_core::sc_module_name&)
};

struct C: B {                         // One module derived from another
    C(sc_core::sc_module_name n)
    : B(n) {}                         // Calls sc_core::sc_module_name(sc_core::sc_module_name&)
                                      // then B(sc_core::sc_module_name)
};

struct Top : sc_core::sc_module {
    A a;
    C c;

    Top(sc_core::sc_module_name n)
    : sc_core::sc_module(n),          // Calls sc_core::sc_module(sc_core::sc_module_name&)
    a("a"),                           // Calls sc_core::sc_module_name(char*)
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
                                    // then calls A(sc_core::sc_module_name)
    c("c") {}                       // Calls sc_core::sc_module_name(char*)
                                    // then calls C(sc_core::sc_module_name)
};
```

5.4 *sc_sensitive†*

5.4.1 Description

Class *sc_sensitive†* provides the operators used to build the static sensitivity of an unspawned process instance. To create static sensitivity for a spawned process, use the member function **set_sensitivity** of the class **sc_spawn_options** (see 5.5).

5.4.2 Class definition

```
namespace sc_core {

class sc_sensitive†
{
    public:
        sc_sensitive†& operator<< ( const sc_event& );
        sc_sensitive†& operator<< ( const sc_interface& );
        sc_sensitive†& operator<< ( const sc_port_base& );
        sc_sensitive†& operator<< ( sc_event_finder& );

        template <typename Collection>
        sc_sensitive†& operator<< ( const Collection& );

        // Other members
        implementation-defined
};

}        // namespace sc_core
```

5.4.3 Constraints on usage

An application shall not explicitly create an object of class *sc_sensitive†*.

Class **sc_module** shall have a data member named **sensitive** of type *sc_sensitive†*. The use of **sensitive** to create static sensitivity is described in 5.2.14.

5.4.4 operator<<

sc_sensitive†& **operator<<** (const sc_event&);

The event passed as an argument shall be added to the static sensitivity of the process instance.

sc_sensitive†& **operator<<** (const sc_interface&);

The event returned by member function **default_event** of the channel instance passed as an argument to **operator<<** shall be added to the static sensitivity of the process instance.

NOTE 1—If the channel passed as an argument does not override function **default_event**, the member function **default_event** of class **sc_interface** is called through inheritance.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

NOTE 2—An export can be passed as an actual argument to this operator because of the existence of the user-defined conversion **sc_export<IF>::operator**.

sc_sensitive† & **operator<<** (const sc_port_base&);

The event returned by member function **default_event** of the channel instance to which the port instance passed as an argument to **operator<<** is bound shall be added to the static sensitivity of the process instance. In other words, the process is made sensitive to the given port, calling function **default_event** to determine to which particular event it should be made sensitive. If the port instance is a multiport (see 5.12.3), the events returned by calling member function **default_event** for each and every channel instance to which the multiport is bound shall be added to the static sensitivity of the process instance.

sc_sensitive† & **operator<<** (sc_event_finder&);

The event found by the event finder passed as an argument to **operator<<** shall be added to the static sensitivity of the process instance (see 5.7).

NOTE 3—An event finder is necessary to create static sensitivity when the application needs to select between multiple events defined in the channel. In a such a case, the **default_event** mechanism is inadequate.

template <typename Collection>
sc_sensitive& **operator<<** (const Collection&);

All elements of the collection passed as an argument to **operator<<** shall be added to the static sensitivity of the process instance. This operator shall participate in the overload resolution if and only if the following conditions are met:

— For an instance c of Collection, std::begin(c) and std::end(c) are defined and return appropriate iterators.
— For an instance s of **sc_sensitive** and an element e of c, the expression s << e is valid.

NOTE 4—Typical collections are **sc_vector**s (see 8.5) or C arrays of ports, exports, channels, or events.

5.5 sc_spawn_options and sc_spawn

5.5.1 Description

Function **sc_spawn** is used to create a static or dynamic spawned process instance.

Class **sc_spawn_options** is used to create an object that is passed as an argument to function **sc_spawn** when creating a spawned process instance. The spawn options determine certain properties of the spawned process instance when used in this way. Calling the member functions of an **sc_spawn_options** object shall have no effect on any process instance unless the object is passed as an argument to **sc_spawn**.

5.5.2 Class definition

```
namespace sc_core {

class sc_spawn_options
{
    public:
        sc_spawn_options();

        void spawn_method();
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        void dont_initialize();
        void set_stack_size( int );

        void set_sensitivity( const sc_event* );
        void set_sensitivity( sc_port_base* );
        void set_sensitivity( sc_export_base* );
        void set_sensitivity( sc_interface* );
        void set_sensitivity( sc_event_finder* );

        void reset_signal_is( const sc_in<bool>& , bool );
        void reset_signal_is( const sc_inout<bool>& , bool );
        void reset_signal_is( const sc_out<bool>& , bool );
        void reset_signal_is( const sc_signal_in_if<bool>& , bool );

        void async_reset_signal_is( const sc_in<bool>& , bool );
        void async_reset_signal_is( const sc_inout<bool>& , bool );
        void async_reset_signal_is( const sc_out<bool>& , bool );
        void async_reset_signal_is( const sc_signal_in_if<bool>& , bool );

    private:
        // Disabled
        sc_spawn_options( const sc_spawn_options& );
        sc_spawn_options& operator= ( const sc_spawn_options& );
};

template <typename T>
sc_process_handle sc_spawn(
    T object ,
    const char* name_p = 0 ,
    const sc_spawn_options* opt_p = 0 );

template <typename T>
sc_process_handle sc_spawn(
    typename T::result_type* r_p ,
    T object ,
    const char* name_p = 0 ,
    const sc_spawn_options* opt_p = 0 );

#define sc_bind    boost::bind
#define sc_ref(r)  boost::ref(r)
#define sc_cref(r) boost::cref(r)

#define SC_FORK implementation-defined
#define SC_JOIN implementation-defined

}          // namespace sc_core

namespace sc_unnamed {

implementation-defined _1;
implementation-defined _2;
implementation-defined _3;
implementation-defined _4;
implementation-defined _5;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

implementation-defined _6;
implementation-defined _7;
implementation-defined _8;
implementation-defined _9;

} // namespace sc_unnamed

5.5.3 Constraints on usage

Function **sc_spawn** may be called during elaboration or from a static, dynamic, spawned, or unspawned process during simulation. Similarly, objects of class **sc_spawn_options** may be created or modified during elaboration or simulation.

5.5.4 Constructors

sc_spawn_options ();

The default constructor shall create an object having the default values for the properties set by the functions **spawn_method**, **dont_initialize**, **set_stack_size**, and **set_sensitivity**.

5.5.5 Member functions

void **spawn_method**();

Member function **spawn_method** shall set a property of the spawn options to indicate that the spawned process shall be a method process. The default is a thread process.

void **dont_initialize**();

Member function **dont_initialize** shall set a property of the spawn options to indicate that the spawned process instance shall not be made runnable during the initialization phase or when it is created. By default, this property is not set, and thus by default the spawned process instance shall be made runnable during the initialization phase of the scheduler if spawned during elaboration, or it shall be made runnable in the current or next evaluation phase if spawned during simulation irrespective of the static sensitivity of the spawned process instance. If the process is spawned during elaboration, member function **dont_initialize** of class **sc_spawn_options** shall provide the same behavior for spawned processes as the member function **dont_initialize** of class **sc_module** provides for unspawned processes.

void **set_stack_size**(int);

Member function **set_stack_size** shall set a property of the spawn options to set the stack size of the spawned process. This member function shall provide the same behavior for spawned processes as the member function **set_stack_size** of class **sc_module** provides for unspawned processes. The effect of calling this function is implementation-defined.

It shall be an error to call **set_stack_size** for a method process.

void **set_sensitivity**(const sc_event*);
void **set_sensitivity**(sc_port_base*);
void **set_sensitivity**(sc_export_base*);
void **set_sensitivity**(sc_interface*);
void **set_sensitivity**(sc_event_finder*);

Member function **set_sensitivity** shall set a property of the spawn options to add the object passed as an argument to **set_sensitivity** to the static sensitivity of the spawned process, as described for

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

operator<< in 5.4.4, or if the argument is the address of an export, the process is made sensitive to the channel instance to which that export is bound. If the argument is the address of a multiport, the process shall be made sensitive to the events returned by calling member function **default_event** for each and every channel instance to which the multiport is bound. By default, the static sensitivity is empty. Calls to **set_sensitivity** are cumulative: each call to **set_sensitivity** extends the static sensitivity as set in the spawn options. Calls to the four different overloaded member functions can be mixed.

```
void reset_signal_is( const sc_in<bool>& , bool );
void reset_signal_is( const sc_inout<bool>& , bool );
void reset_signal_is( const sc_out<bool>& , bool );
void reset_signal_is( const sc_signal_in_if<bool>& , bool );

void async_reset_signal_is( const sc_in<bool>& , bool );
void async_reset_signal_is( const sc_inout<bool>& , bool );
void async_reset_signal_is( const sc_out<bool>& , bool );
void async_reset_signal_is( const sc_signal_in_if<bool>& , bool );
```

Member functions **reset_signal_is** and **async_reset_signal_is** shall set a property of the spawn options to add the object passed as an argument as a synchronous or an asynchronous reset signal of the spawned process instance, respectively, as described in 5.2.13. Each call to either of these member functions shall add a reset signal to the given spawn options object. The signal may be identified indirectly by passing a port instance that is bound to the reset signal. A spawned process instance may have any number of synchronous and asynchronous reset signals.

NOTE 1—There are no member functions to set the spawn options to spawn a thread process or to make a process runnable during initialization. This functionality is reliant on the default values of the **sc_spawn_options** object.

NOTE 2—It is not possible to spawn a dynamic clocked thread process.

5.5.6 sc_spawn

```
template <typename T>
sc_process_handle sc_spawn(
    T object ,
    const char* name_p = 0 ,
    const sc_spawn_options* opt_p = 0 );

template <typename T>
sc_process_handle sc_spawn(
    typename T::result_type* r_p ,
    T object ,
    const char* name_p = 0 ,
    const sc_spawn_options* opt_p = 0 );

#define sc_bind    boost::bind
#define sc_ref(r)  boost::ref(r)
#define sc_cref(r) boost::cref(r)
```

Function **sc_spawn** shall create a static or dynamic spawned process instance.

Function **sc_spawn** may be called during elaboration, in which case the spawned process is a *child* of the module instance within which function **sc_spawn** is called or is a top-level object if function **sc_spawn** is called from function **sc_main**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Function **sc_spawn** may be called during simulation, in which case the spawned process is a child of the process that called function **sc_spawn**. Function **sc_spawn** may be called from a method process, a thread process, or a clocked thread process.

The process or module from which **sc_spawn** is called is the *parent* of the spawned process. Thus a set of dynamic process instances may have a hierarchical relationship, similar to the module hierarchy, which will be reflected in the hierarchical names of the process instances.

If function **sc_spawn** is called during the evaluation phase, the spawned process shall be made runnable in the current evaluation phase (unless **dont_initialize** has been called for this process instance). If function **sc_spawn** is called during the update phase, the spawned process shall be made runnable in the very next evaluation phase (unless **dont_initialize** has been called for this process instance).

The argument of type **T** shall be either a function pointer or a function object, that is, an object of a class that overloads **operator()** as a member function and shall specify the function associated with the spawned process instance, that is, the function to be spawned. This shall be the only mandatory argument to function **sc_spawn**.

If present, the argument of type **T::result_type*** shall pass a pointer to a memory location that shall receive the value returned from the function associated with the process instance. In this case, the argument of type **T** shall be a function object of a class that exposes a nested type named **result_type**. Furthermore, **operator()** of the function object shall have the return type **result_type**. It is the responsibility of the application to ensure that the memory location is still valid when the spawned function returns. For example, the memory location could be a data member of an enclosing **sc_module** but should not be a stack variable that would have been deallocated by the time the spawned function returns. See the example below.

The macros **sc_bind**, **sc_ref**, and **sc_cref** are provided for convenience when using the free Boost C++ libraries to bind arguments to spawned functions. Passing arguments to spawned processes is a powerful mechanism that allows processes to be parameterized when they are spawned and permits processes to update variables over time through reference arguments. **boost::bind** provides a convenient way to pass value arguments, reference arguments, and const reference arguments to spawned functions, but its use is not mandatory. See the examples below and the Boost documentation available on the Internet.

The only purpose of namespace **sc_unnamed** is to allow the implementation to provide a set of argument placeholders for use with **sc_bind**. **_1**, **_2**, **_3** ... shall be provided by the implementation to give access to the placeholders of the same names from the Boost libraries. These placeholders can be passed as arguments to **sc_bind** in order to defer the binding of function arguments until the call to the function object returned from **sc_bind**. Again, see the Boost documentation for details.

The argument of type **const char*** shall give the string name of the spawned process instance and shall be passed by the implementation to the constructor for the **sc_object** that forms the base class sub-object of the spawned process instance. If no such argument is given or if the argument is an empty string, the implementation shall create a string name for the process instance by calling function **sc_gen_unique_name** with the seed string **"thread_p"** in the case of a thread process or **"method_p"** in the case of a method process.

The argument of type **sc_spawn_options*** shall set the spawn options for the spawned process instance. If no such argument is provided, the spawned process instance shall take the default values as defined for the member functions of class **sc_spawn_options**. The application is not obliged to keep the **sc_spawn_options** object valid after the return from function **sc_spawn**.

Function **sc_spawn** shall return a valid process handle to the spawned process instance. A process instance can be suspended, resumed, disabled, enabled, killed, or reset using the member functions of class **sc_process_handle**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

If a spawn options argument is given, a process string name argument shall also be given, although that string name argument may be an empty string.

NOTE—Function **sc_spawn** provides a superset of the functionality of the macros SC_THREAD and SC_METHOD. In addition to the functionality provided by these macros, function **sc_spawn** provides the passing of arguments and return values to and from processes spawned during elaboration or simulation. The macros are retained for compatibility with earlier versions of SystemC.

Example:

```
int f();

struct Functor {
    typedef int result_type;
    result_type operator() ();
};

Functor::result_type Functor::operator() () { return f(); }

int h(int a, int& b, const int& c);

struct MyMod: sc_core::sc_module {
    sc_core::sc_signal<int> SC_NAMED(sig);
    void g();

    SC_CTOR(MyMod) {
        SC_THREAD(T);
    }

    int ret;

    void T() {
        using namespace sc_core;

        sc_spawn(&f);                   // Spawn a function without arguments and discard any return value.
                                        // Spawn a similar process and create a process handle.
        sc_process_handle handle = sc_spawn(&f);

        Functor fr;
        sc_spawn(&ret, fr);             // Spawn a function object and catch the return value.

        sc_spawn_options opt;
            opt.spawn_method();
            opt.set_sensitivity(&sig);
            opt.dont_initialize();
        sc_spawn(f, "f1", &opt);        // Spawn a method process named "f1", sensitive to sig, not initialized.
                                        // Spawn a similar process named "f2" and catch the return value.
        sc_spawn(&ret, fr, "f2", &opt);
                                        // Spawn a member function using Boost bind.
        sc_spawn(sc_bind(&MyMod::g, this));

        int A = 0, B, C;
                                        // Spawn a function using Boost bind, pass arguments
                                        // and catch the return value.
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    sc_spawn(&ret, sc_bind(&h, A, sc_ref(B), sc_cref(C)));
    }
};
```

5.5.7 SC_FORK and SC_JOIN

#define SC_FORK *implementation-defined*
#define SC_JOIN *implementation-defined*

The macros SC_FORK and SC_JOIN can only be used as a pair to bracket a set of calls to function **sc_spawn** from within a thread or clocked thread process. It is an error to use the fork-join construct in a method process. Each call to **sc_spawn** (enclosed between SC_FORK and SC_JOIN) shall result in a separate process instance being spawned when control enters the fork-join construct. The child processes shall be spawned without delay and may potentially all become runnable in the current evaluation phase (depending on their spawn options). The spawned process instances shall be thread processes. It is an error to spawn a method process within a fork-join construct. Control leaves the fork-join construct when all the spawned process instances have terminated.

The text between SC_FORK and SC_JOIN shall consist of a series of one or more calls to function **sc_spawn** separated by commas. The value returned from the function call may be discarded or the function call may be the only expression on the right-hand-side of an assignment to a variable, where the variable will be set to the process handle returned from **sc_spawn**.

The comma after the final call to **sc_spawn** and immediately before SC_JOIN shall be optional. There shall be no other characters other than white space separating SC_FORK, the function calls, or variable assignments, the commas, and SC_JOIN. If an application violates these rules, the effect shall be undefined.

Example 1:

```
{
    using namespace sc_core;
    SC_FORK
        sc_spawn( arguments ) ,
        sc_spawn( arguments ) ,
        sc_spawn( arguments )
    SC_JOIN
}
```

Example 2:

```
{
    using namespace sc_core;
    sc_process_handle h1, h2, h3;

    SC_FORK
        h1 = sc_spawn( arguments ) ,
        h2 = sc_spawn( arguments ) ,
        h3 = sc_spawn( arguments )
    SC_JOIN
}
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

5.6 sc_process_handle

5.6.1 Description

Class **sc_process_handle** provides a process handle to an underlying spawned or unspawned process instance. A process handle can be in one of two states: *valid* or *invalid*. A *valid* process handle shall be associated with a single underlying process instance, which may or may not be in the terminated state. An *invalid* process handle may be associated with a single underlying process instance provided that process instance has terminated. An *empty* process handle is an invalid handle that is not associated with any underlying process instance. In other words, an invalid process handle is either an empty handle or is associated with a terminated process instance. A process instance may be associated with zero, one or many process handles, and the number and identity of such process handles may change over time.

Since dynamic process instances can be created and destroyed dynamically during simulation, it is in general unsafe to manipulate a process instance through a raw pointer to the process instance (or to the base class sub-object of class **sc_object**). The purpose of class **sc_process_handle** is to provide a safe and uniform mechanism for manipulating both spawned and unspawned process instances without reliance on raw pointers. If control returns from the function associated with a thread process instance (that is, the process terminates), the underlying process instance may be deleted, but the process handle will continue to exist.

The provision of **operator<,** which supplies a strict weak ordering relation on the underlying process instances, allows process handles to be stored in a standard C++ container such as **std::map**.

5.6.2 Class definition

```
namespace sc_core {

enum sc_curr_proc_kind
{
     SC_NO_PROC_ ,
     SC_METHOD_PROC_ ,
     SC_THREAD_PROC_ ,
     SC_CTHREAD_PROC_
};


enum sc_descendant_inclusion_info
{
    SC_NO_DESCENDANTS,
    SC_INCLUDE_DESCENDANTS
};

class sc_unwind_exception: public std::exception
{
public:
    virtual const char* what() const throw();
    virtual bool is_reset() const;

protected:
    sc_unwind_exception();
    sc_unwind_exception( const sc_unwind_exception& );
    virtual ~sc_unwind_exception() throw();
};
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
class sc_process_handle
{
    public:
        sc_process_handle();
        sc_process_handle( const sc_process_handle& );
        explicit sc_process_handle( sc_object* );
        ~sc_process_handle();

        bool valid() const;

        sc_process_handle& operator= ( const sc_process_handle& );
        bool operator== ( const sc_process_handle& ) const;
        bool operator!= ( const sc_process_handle& ) const;
        bool operator< ( const sc_process_handle& ) const;
        void swap( sc_process_handle& );

        const char* name() const;
        const char* basename() const;
        sc_curr_proc_kind proc_kind() const;
        const std::vector<sc_object*>& get_child_objects() const;
        const std::vector<sc_event*>& get_child_events() const;
        sc_object* get_parent_object() const;
        sc_object* get_process_object() const;
        bool dynamic() const;
        bool terminated() const;
        const sc_event& terminated_event() const;

        void suspend ( sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS );
        void resume  ( sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS );
        void disable ( sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS );
        void enable  ( sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS );
        void kill    ( sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS );
        void reset   ( sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS );
        bool is_unwinding() const;
        const sc_event& reset_event() const;

        void sync_reset_on
                    ( sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS );
        void sync_reset_off
                    ( sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS );

        template <typename T>
        void throw_it( const T& user_defined_exception,
                     sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS );
};

sc_process_handle sc_get_current_process_handle();
bool sc_is_unwinding();

}       // namespace sc_core
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

5.6.3 Constraints on usage

None. A process handle may be created, copied, or deleted at any time during elaboration or simulation. The handle may be valid or invalid.

5.6.4 Constructors

sc_process_handle();

The default constructor shall create an *empty* process handle. An empty process handle shall be an invalid process handle.

sc_process_handle(const sc_process_handle&);

The copy constructor shall duplicate the process handle passed as an argument. The result will be two handles to the same underlying process instance or two empty handles.

explicit **sc_process_handle**(sc_object*);

If the argument is a pointer to a process instance, this constructor shall create a valid process handle to the given process instance. Otherwise, this constructor shall create an empty process handle.

5.6.5 Member functions

bool **valid**() const;

Member function **valid** shall return **true** if and only if the process handle is valid.

sc_process_handle& **operator=** (const sc_process_handle&);

The assignment operator shall duplicate the process handle passed as an argument. The result will be two handles to the same underlying process instance or two empty handles.

bool **operator==** (const sc_process_handle&) const;

The equality operator shall return **true** if and only if the two process handles are both valid and share the same underlying process instance.

bool **operator!=** (const sc_process_handle&) const;

The inequality operator shall return **false** if and only if the two process handles are both valid and share the same underlying process instance.

bool **operator<** (const sc_process_handle&) const;

The less-than operator shall define a *strict weak ordering* relation on the underlying process instances in the following sense. Given three process handles H1, H2, and H3:

strict means that H1 < H1 shall return **false** (whether H1 is valid, invalid, or empty)

weak means that if H1 and H2 are both handles (valid or invalid) associated with the same underlying process instance or both are empty handles, then H1 < H2 shall return **false** and H2 < H1 shall return **false**. Otherwise, if H1 and H2 are handles associated with different underlying process instances or if only one is an empty handle, then exactly one of H1 < H2 and H2 < H1 shall return **true**.

ordering relation means that if H1 < H2 and H2 < H3, then H1 < H3 (whether H1, H2, or H3 are valid or invalid).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Example:

```
{
    using namespace sc_core;
    sc_process_handle a, b;                 // Two empty handles

    sc_assert( !a.valid() && !b.valid() );  // Both are invalid
    sc_assert( a != b );
    sc_assert( !(a < b) && !(b < a) );

    a = sc_spawn(...);
    b = sc_spawn(...);

    sc_assert( a != b );
    sc_assert( (a < b) || (b < a) );        // Two handles to different processes

    sc_process_handle c = b;

    sc_assert( b == c );
    sc_assert( !(b < c) && !(c < b) );      // Two handles to the same process

    wait( a.terminated_event() & b.terminated_event() );

    sc_assert( (a < b) || (b < a) );        // Same ordering whether handles are valid or not

    if ( b.valid() )                        // Handles may or may not have been invalidated
        sc_assert( b == c );
    else
        sc_assert( b != c );

    sc_assert( b.valid() == c.valid() );    // Invalidation is consistent

    sc_assert( !(b < c) && !(c < b) );  // Two handles to the same process, whether valid or not

    sc_assert( c.terminated() );
}
```

void **swap**(sc_process_handle&);

Member function **swap** shall exchange the process instances underlying the process handles *this and the **sc_process_handle** argument. If H1 < H2 prior to the call H1.swap(H2), then H2 < H1 after the call. Either handle may be invalid.

Example:

```
{
    using namespace sc_core;
    sc_process_handle a, b = sc_get_current_process_handle();
    sc_assert( b.valid() );

    a.swap( b );
    sc_assert( a == sc_get_current_process_handle() );
    sc_assert( !b.valid() );
}
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

const char* **name**() const;

Member function **name** shall return the hierarchical name of the underlying process instance. If the process handle is invalid, member function **name** shall return an empty string, that is, a pointer to the string "". The implementation is obliged to keep the returned string valid only while the process handle is valid.

const char* **basename**() const;

Member function **basename** shall return the string name of the underlying process instance. If the process handle is invalid, member function **basename** shall return an empty string, that is, a pointer to the string "". The implementation is obliged to keep the returned string valid only while the process handle is valid.

sc_curr_proc_kind **proc_kind**() const;

For a valid process handle, member function **proc_kind** shall return one of the three values SC_METHOD_PROC_, SC_THREAD_PROC_, or SC_CTHREAD_PROC_, depending on the kind of the underlying process instance, that is, method process, thread process, or clocked thread process, respectively. For an invalid process handle, member function **proc_kind** shall return the value SC_NO_PROC_.

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

Member function **get_child_objects** shall return a **std::vector** containing a pointer to every instance of class **sc_object** that is a child of the underlying process instance. This shall include every dynamic process instance that has been spawned during the execution of the underlying process instance and has not yet been deleted, and any other application-defined objects derived from class **sc_object** created during the execution of the underlying process instance that have not yet been deleted. Processes that are spawned from child processes are not included (grandchildren, as it were). If the process handle is invalid, member function **get_child_objects** shall return an empty **std::vector**.

This same function shall be overridden in any implementation-defined classes derived from **sc_object** and associated with spawned and unspawned process instances. Such functions shall have identical behavior provided that the process handle is valid.

const std::vector<sc_event*>& **get_child_events**() const;

Member function **get_child_events** shall return a **std::vector** containing a pointer to every object of type **sc_event** that is a hierarchically named event and whose parent is the current process instance. If the process handle is invalid, member function **get_child_events** shall return an empty **std::vector**.

This same function shall be overridden in any implementation-defined classes derived from **sc_object** and associated with spawned and unspawned process instances. Such functions shall have identical behavior provided that the process handle is valid.

sc_object* **get_parent_object**() const;

Member function **get_parent_object** shall return a pointer to the module instance or process instance from which the underlying process instance was spawned. If the process handle is invalid, member function **get_parent_object** shall return the null pointer. If the parent object is a process instance and that process has terminated, **get_parent_object** shall return a pointer to that process instance. A process instance shall not be deleted (nor any associated process handles invalidated) while the process has surviving children, but it may be deleted once all its child objects have been deleted.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

sc_object* **get_process_object**() const;

If the process handle is valid, member function **get_process_object** shall return a pointer to the process instance associated with the process handle. If the process handle is invalid, member function **get_process_object** shall return the null pointer. An application should test for a null pointer before dereferencing the pointer. Moreover, an application should assume that the pointer remains valid only until the calling process suspends.

bool **dynamic**() const;

Member function **dynamic** shall return **true** if the underlying process instance is a dynamic process and false if the underlying process instance is a static process. If the process handle is invalid, member function **dynamic** shall return the value **false**.

bool **terminated**() const;

Member function **terminated** shall return **true** if and only if the underlying process instance has *terminated*. A thread or clocked thread process is *terminated* after the point when control is returned from the associated function. A process can also be *terminated* by calling the member function **kill** of a process handle associated with that process. This is the only way in which a method process can be terminated. If the process handle is empty, member function **terminated** shall return the value **false**.

When the underlying process instance terminates, if the process instance has no surviving children, an implementation may choose to invalidate any associated process handles, but it is not obliged to do so. An implementation shall not invalidate a process handle while the process instance has child objects. However, if an implementation chooses to invalidate a process handle, it shall invalidate every process handle associated with the underlying process instance at that time. In other words, it shall not be possible to have a valid and an invalid handle to the same underlying process instance in existence at the same time. After the process instance has terminated, function **terminated** will continue to return **true**, even if the process handle becomes invalid. The process instance shall continue to exist as long as the process handle is valid. Once all the handles associated with a given process instance have been invalidated, an implementation is free to delete the process instance, but it is not obliged to do so. With respect to the value of function **terminated** and the ordering relation that defines the behavior of **operator<** and member function **swap**, in effect an invalid process handle remains associated with a process instance even after that process instance has been deleted.

const sc_event& **terminated_event**() const;

Member function **terminated_event** shall return a reference to an event that is notified when the underlying process instance terminates. It shall be an error to call member function **terminated_event** for an invalid process handle.

5.6.6 Member functions for process control

5.6.6.1 Overview

The member functions of class **sc_process_handle** described in 5.6.6 are concerned with process control. Examples of process control include suspending, resuming, killing, or resetting a process instance. Process control involves a *calling process* controlling a *target process*. In each case, the *calling process* calls a member function of a process handle associated with the *target process* (or with its parent object, grandparent object, and so forth).

The calling process and the target process may be distinct process instances or may be the same process instance. In the latter case, certain process control member functions are meaningless, such as having a process instance attempt to resume itself.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Several of the process control member functions are organized as complementary pairs:

- **suspend** and **resume**
- **disable** and **enable**
- **sync_reset_on** and **sync_reset_off**
- **kill**
- **reset**
- **throw_it**

Each of the above member functions may be called where the underlying process instance is a method process, a thread process, or a clocked thread process. There are certain caveats concerning the use of **suspend**, **resume**, **reset**, and **throw_it** with clocked thread processes (see 5.2.12).

The distinction between **suspend/resume** and **disable/enable** lies in the sensitivity of the target process during the period while it is suspended or disabled. With **suspend**, the kernel keeps track of the sensitivity of the target process while it is suspended such that a relevant event notification or time-out while suspended would cause the process to become runnable immediately when **resume** is called. With **disable**, the sensitivity of the target process is nullified while it is suspended such that the process is not made runnable by the call to **enable**, but only on the next relevant event notification or time-out subsequent to the call to **enable**. In other words, with **suspend/resume**, the kernel keeps a record of whether the target process would have awoken while in fact being suspended, whereas with **disable/enable**, the kernel entirely ignores the sensitivity of the target process while disabled. Also see 5.2.12 regarding clocked thread processes.

Example:

```
struct M1 : sc_core::sc_module {
    M1(sc_core::sc_module_name _name)
    {
        SC_THREAD(ticker);
        SC_THREAD(calling);
        SC_THREAD(target);
        t = sc_core::sc_get_current_process_handle();
    }

    sc_core::sc_process_handle t;
    sc_core::sc_event ev;

    void ticker() {
        for (;;) {
            wait(10.0, sc_core::SC_NS);
            ev.notify();
        }
    }

    void calling() {
        using namespace sc_core;

        wait(15.0, SC_NS);
        // Target runs at time 10 NS due to notification

        t.suspend();
        wait(10.0, SC_NS);
        // Target does not run at time 20 NS while suspended
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        t.resume();
        // Target runs at time 25 NS when resume is called

        wait(10.0, SC_NS);
        // Target runs at time 30 NS due to notification

        t.disable();
        wait(10.0, SC_NS);
        // Target does not run at time 40 NS while disabled

        t.enable();
        // Target does not run at time 45 NS when enable is called

        wait(10.0, SC_NS);
        // Target runs at time 50 NS due to notification

        sc_stop();
    }

    void target() {
        for (;;) {
            wait(ev);
            std::cout << "Target awoke at " << sc_core::sc_time_stamp() << std::endl;
        }
    }
};
```

sync_reset_on and **sync_reset_off** provide a procedural way of putting a process instance into and out of the *synchronous reset state* (see 5.6.6.4). This is equivalent in effect to having specified a reset signal using the member function **reset_signal_is** of class **sc_module** or **sc_spawn_options** and that reset signal attaining its active value or the negation of its active value, respectively.

Example:

```
struct M2 : sc_core::sc_module {
    M2(sc_core::sc_module_name _name) {
        SC_THREAD(ticker);
        SC_THREAD(calling);
        SC_THREAD(target);
        t = sc_core::sc_get_current_process_handle();
    }

    sc_core::sc_process_handle t;
    sc_core::sc_event ev;

    void ticker() {
        for (;;) {
            wait(10.0, sc_core::SC_NS);
            ev.notify();
        }
    }

    void calling() {
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        using namespace sc_core;

        wait(15, SC_NS);
        // Target runs at time 10 NS due to notification

        t.sync_reset_on();
        // Target does not run at time 15 NS

        wait(10.0, SC_NS);
        // Target is reset at time 20 NS due to notification

        wait(10.0, SC_NS);
        // Target is reset again at time 30 NS due to notification

        t.sync_reset_off();
        // Target does not run at time 35 NS

        wait(10.0, SC_NS);
        // Target runs at time 40 NS due to notification

        sc_stop();
    }

    void target() {
        std::cout << "Target called/reset at " << sc_core::sc_time_stamp() << std::endl;
        for (;;) {
            wait(ev);
            std::cout << "Target awoke at " << sc_core::sc_time_stamp() << std::endl;
        }
    }
};
```

kill interrupts and irrevocably terminates the target process. **reset** interrupts the target process and, in the case of a thread process, calls the associated function again from the top. **kill** and **reset** both cause an exception to be thrown within the target process, and that exception may be caught and handled by the application. Both have immediate semantics such that the target process is killed or reset before return from the function call.

Example:

```
struct M3 : sc_core::sc_module {
    M3(sc_core::sc_module_name _name) {
        SC_THREAD(ticker);
        SC_THREAD(calling);
        SC_THREAD(target);
        t = sc_core::sc_get_current_process_handle();
    }

    sc_core::sc_process_handle t;
    sc_core::sc_event ev;
    int count;

    void ticker() {
        for (;;) {
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
            wait(10.0, sc_core::SC_NS);
            ev.notify();
        }
    }

    void calling() {
        using namespace sc_core;

        wait(15.0, SC_NS);
        // Target runs at time 10 NS due to notification
        sc_assert( count == 1 );

        wait(10.0, SC_NS);
        // Target runs again at time 20 NS due to notification
        sc_assert( count == 2 );

        t.reset();
        // Target reset immediately at time 25 NS
        sc_assert( count == 0 );

        wait(10.0, SC_NS);
        // Target runs again at time 30 NS due to notification
        sc_assert( count == 1 );

        t.kill();
        // Target killed immediately at time 35 NS
        sc_assert( t.terminated() );

        sc_stop();
    }

    void target() {
        std::cout << "Target called/reset at " << sc_core::sc_time_stamp() << std::endl;
        count = 0;
        for (;;) {
            wait(ev);
            std::cout << "Target awoke at " << sc_core::sc_time_stamp() << std::endl;
            ++count;
        }
    }
};
```

A process may be reset in several ways: by a synchronous or asynchronous reset signal specified using **reset_signal_is** or **async_reset_signal_is**, respectively (see 5.2.13), by calling **sync_reset_on**, or by calling **reset**. Whichever alternative is used, the reset action itself is ultimately equivalent to a call to **reset**.

throw_it permits a user-defined exception to be thrown within the target process.

For each of the nine member functions for process control described in this clause and its subclauses, if the process handle is invalid, the implementation shall generate a warning and the member function shall return immediately without having any other effect.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

In this clause, the phrase *during elaboration or before the process has first executed* shall encompass the following cases:

- At any time during elaboration
- From one of the callbacks **before_end_of_elaboration**, **end_of_elaboration**, or **start_of_simulation**
- During simulation but before the given process instance has executed for the first time

In each case, if the **include_descendants** argument has the value **SC_INCLUDE_DESCENDANTS**, the member function shall be applied to the associated process instance and recursively in bottom-up order to all its children in the object hierarchy that are also process instances, where each process instance shall in turn act as the target process. In other words, the member function shall be applied to the children, grandchildren, great grandchildren, and so forth, of the associated process instance, starting with the deepest descendant, and finishing with the associated process instance itself. If the **include_descendants** argument has the value **SC_NO_DESCENDANTS**, the member function shall only be applied to the associated process instance. There are no restrictions concerning how this argument may be used across pairs of calls to **suspend-resume**, **disable-enable**, or **sync_reset_on-sync_reset_off**. For example, it is permitted to suspend a process and all of its descendants but only resume the process itself.

The member functions for process control shall not be called during the update phase.

Beware that the rules given in the following subclauses are to be understood alongside the rules given in 5.6.6.12 concerning the interaction between the member functions for process control, which shall take precedence where appropriate.

5.6.6.2 suspend and resume

void **suspend**(sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS);

Member function **suspend** shall suspend the target process instance such that it cannot be made runnable again until it is resumed by a call to member function **resume**, at the earliest (but see 5.6.6.12 for the rules concerning **reset**). If the process instance is in the set of runnable processes, it shall be removed from that set immediately such that it does not run in the current evaluation phase. While the process is suspended in this way, if an event notification or time-out occurs to which the process was sensitive at the time the process was suspended, and if **resume** is called subsequently, the process instance shall become runnable in the evaluation phase in which **resume** is called. In other words, the implementation shall in effect add an implicit event to the sensitivity of the process instance using the *event_and_list* **operator&**, and shall create an immediate notification for this event when **resume** is called.

Calling **suspend** on a process instance that is already suspended shall have no effect on that particular process instance, although it may cause descendants to be suspended. Only a single call to **resume** is required to resume a suspended process instance.

If a method process suspends itself, the associated function shall run to the end before returning control to the kernel. If that same method process calls the **resume** method before returning control to the kernel, the effect of the call to **suspend** shall be removed as if **suspend** had not been called.

If a thread process suspends itself, the process instance shall be suspended immediately without control being returned from the member function **suspend** back to the associated function.

If **suspend** is called for a target process instance that is not suspended (meaning that **suspend** has not been called) and is in the synchronous reset state, the behavior shall be implementation-defined (see 5.6.6.12).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Calling **suspend** on a terminated process instance shall have no effect on that particular process instance, although it may cause the suspension of any non-terminated descendant process instances.

Calling **suspend** during elaboration or before the process has first executed is permitted. If **dont_initialize** is in force, the implementation shall in effect add an implicit event to the sensitivity of the process instance using the *event_and_list* **operator&**, and shall create an immediate notification for this event when **resume** is called. Otherwise, the implementation shall make the process instance runnable immediately when **resume** is called, in effect deferring initialization until the process instance is resumed.

void **resume**(sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS);

Member function **resume** shall remove the effect of any previous call to **suspend** such that the process instance shall be made runnable in the current evaluation phase if and only if the sensitivity of the process instance would have caused it to become runnable while it was in fact suspended. If the process instance was in the set of runnable processes at the time is was suspended, member function **resume** shall cause the process instance to be made runnable in the current evaluation phase (but see 5.6.6.12 for the rules concerning the interaction between process control member functions). If the process instance was not in the set of runnable processes at the time it was suspended and if there were no event notifications or time-outs that would have caused the process instance to become runnable while it was in fact suspended, member function **resume** shall not make the target process instance runnable.

Member function **resume** shall restore the sensitivity of the target process as it was when **suspend** was called.

If a **resume** is preceded by a **reset**, any event notifications or time-outs that occurred before the time of the reset shall be ignored when determining the effect of **resume**.

A thread process shall be resumed from the executable statement following the call to **suspend**. A method process shall be resumed by calling the associated function.

Calling **resume** on a process instance that is not suspended shall have no effect on that particular process instance, although it may cause suspended descendants to be resumed. As a consequence, multiple calls to **resume** from the same or from multiple processes within the same evaluation phase may cause the target process to run only once or to run more than once within that evaluation phase, depending on the precise order of process execution.

If multiple target processes become runnable within an evaluation phase as a result of multiple calls to **resume**, they will be run in an implementation-defined order (see 4.3.2.3).

If **resume** is called for a target process instance that is both suspended (meaning that **suspend** has been called) and disabled, the behavior shall be implementation-defined (see 5.6.6.12).

Calling **resume** on a terminated process instance shall have no effect on that particular process instance, although it may cause any non-terminated descendant process instances to be resumed.

Calling **resume** during elaboration or before the process has first executed is permitted, in which case the above rules shall apply.

The function associated with any process instance can call **suspend** or **resume**: there is no obligation that a process instance be suspended and resumed by the same process instance.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

5.6.6.3 disable and enable

void **disable**(sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS);

Member function **disable** shall disable the target process instance such that it cannot be made runnable again until it is enabled by a call to member function **enable**, at the earliest. If the disabled process instance is in the set of runnable processes, it shall not be removed from that set but shall be allowed to run in the current evaluation phase. While a process instance is disabled in this way, any events or time-outs to which it is sensitive shall be ignored for that particular process instance such that it will not be inserted into the set of runnable processes. As a consequence of this rule, a disabled process instance may run once and only once while it remains disabled, and then only if it was already runnable at the time it was disabled.

Calling **disable** on a process instance that is already disabled shall have no effect on that particular process instance, although it may cause descendants to be disabled. Only a single call to **enable** is required to enable a disabled process instance.

If a process disables itself, whether it be a method process or a thread process the associated function shall continue to run to the point where it calls **wait** or to the end before returning control to the kernel. If that same process calls the member function **enable** before returning control to the kernel, the effect of the call to **disable** shall be removed as if **disable** had not been called.

If **disable** is called for a target process instance that is not disabled and is waiting on a time-out (with or without an event or event list), the behavior shall be implementation-defined (see 5.6.6.12).

Calling **disable** on a terminated process instance shall have no effect on that particular process instance, although it may cause any non-terminated descendant process instances to be disabled.

Calling **disable** during elaboration or before the process has first executed is permitted: if the process is already in the set of runnable processes at the time **disable** is called, during initialization, for example, it shall be allowed to run.

void **enable**(sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS);

Member function **enable** shall remove the effect of any previous **disable** such that the process instance can be made runnable by subsequent event notifications or time-outs. Unlike **resume**, an enabled process shall not become runnable due to event notification or time-outs that occurred while it was disabled.

If a time-out occurs while a process instance is disabled and that process instance is sensitive to no other events aside from the time-out itself, the process instance will not run again (unless it is reset) and the implementation may issue a warning.

When the enabled process instance next executes, if it is a waiting thread process it shall execute from the statement following the call to **wait** at which it was disabled, and if a method process it shall execute by calling the associated function.

Calling **enable** on a process instance that is not disabled shall have no effect on that particular process instance, although it may cause disabled descendants to be enabled.

Calling **enable** on a terminated process instance shall have no effect on that particular process instance, although it may cause any non-terminated descendant process instances to be enabled.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Calling **enable** during elaboration or before the process has first executed is permitted, in which case the above rules shall apply.

If **disable** is called during elaboration and **enable** is only called after the initialization phase, the target process instance shall not become runnable during the initialization phase, and shall not become runnable when **enable** is called.

The function associated with any process instance can call **disable** or **enable**: there is no obligation that a process instance be disabled and enabled by the same process instance.

5.6.6.4 Member functions to reset processes

A process instance can be reset in one of three ways:

- By a call to member function **reset** of a process handle associated with that process instance, which shall reset the target process instance immediately.
- By a reset signal specified using **async_reset_signal_is** attaining its reset value, at which time the process instance shall be reset immediately (see 5.2.13).
- By the process instance being resumed while it is in the *synchronous reset state*, defined as follows:

A process instance can be put into the *synchronous reset state* in one of three ways:

- By a call to member function **sync_reset_on** of a process handle associated with that process instance.
- When any reset signal specified using **reset_signal_is** for that process instance attains its active value.
- When any reset signal specified using **async_reset_signal_is** for that process instance attains its active value.

A process instance can only leave the synchronous reset state when all of the following apply:

- Member function **sync_reset_off** is called or member function **sync_reset_on** has not yet been called for a process handle associated with that process instance.
- Every reset signal specified using **reset_signal_is** for that process instance has the negation of its active value.
- Every reset signal specified using **async_reset_signal_is** for that process instance has the negation of its active value.

In each case, any resultant reset shall be equivalent to a call to member function **reset**, and hence, the event returned by member function **reset_event** shall be notified.

Any given process instance can have multiple resets specified using **reset_signal_is** and **async_reset_signal_is** in addition to being the target of calls to member functions **reset** and **sync_reset_on** of an associated process handle.

5.6.6.5 kill

void **kill**(sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS);

Member function **kill** shall kill the target process instance such that an **sc_unwind_exception** shall be thrown (see 5.6.6.7) for the killed process instance and the killed process instance shall not be made runnable again during the current simulation. A killed process shall be terminated.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

In the case of an exception thrown for a kill, member function **is_reset** of class **sc_unwind_exception** shall return the value **false**.

kill shall have immediate effect; that is, the killed process instance shall be removed from the set of runnable processes, the call stack unwound, the process instance put into the terminated state, and the terminated event notified before the return from the **kill** function. Calls to **kill** can have side-effects. If a process kills another process, control shall return to the function that called **kill**. If a process kills itself, the statements following the call to **kill** shall not be executed again during the current simulation, and control shall return to the kernel.

Calling **kill** on a terminated process instance shall have no effect on that particular process instance, although it may cause any non-terminated descendant process instances to be killed.

Calling **kill** before the process has first executed is permitted, in which case the target process instance shall not be run during the current simulation.

It shall be an error to call **kill** during elaboration, before the initialization phase, or while the simulation is paused or stopped.

5.6.6.6 reset

void **reset**(sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS);

Member function **reset** shall reset the target process instance such that the process shall be removed from the set of runnable processes, an **sc_unwind_exception** shall be thrown for the process instance, any dynamic sensitivity associated with the process removed, and the static sensitivity restored. The target process shall not be terminated. The process handle shall remain valid. The associated function shall then be called again and, as a consequence, will execute until it calls **wait**, suspends itself, or returns.

reset shall have immediate effect; that is, the target process instance shall be removed from the set of runnable processes, the call stack unwound, and the reset event notified before the return from the **reset** function. Calls to **reset** can have side-effects. In particular, care should be taken to avoid mutual recursion between processes that reset one another. **sc_unwind_exception** may be caught provided it is immediately re-thrown.

In the case of an **sc_unwind_exception** thrown for a reset, member function **is_reset** of class **sc_unwind_exception** shall return the value **true**.

A method process may be reset, although the associated function will not have a call stack to be unwound except in the case where a method process resets itself. When a method process is reset, any dynamic sensitivity associated with the process shall be removed, the static sensitivity restored, and the associated function called again.

A thread or a method process instance may reset itself (by a call to the **reset** method of an associated process handle), in which case the call stack shall be unwound immediately.

Calling **reset** on a terminated process instance shall have no effect on that particular process instance, although it may cause any non-terminated descendant process instances to be reset.

Calling **reset** before the process has first executed is permitted, but it shall have no effect other than to notify the event returned by **reset_event**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

It shall be an error to call **reset** during elaboration, before the initialization phase, or while the simulation is paused or stopped.

5.6.6.7 sc_unwind_exception

Class **sc_unwind_exception** is the type of the exception thrown by member functions **kill** and **reset**.

In the case of a thread process suspended by a call to **wait**, or a thread or method process that kills or resets itself, the **sc_unwind_exception** shall cause the call stack of the associated function to be unwound and any local objects to have their destructors called. In the case of a method process, the associated function does not have a call stack to be unwound unless the process kills or resets itself, but a call to **kill** shall cause the method process to be terminated nonetheless.

As a process is being killed or reset, the unwinding of the call stack may have side-effects, including calls to destructors for local objects. Such side-effects shall not include calls to **wait** or **next_trigger**, but they may include calls to process control member functions for other process instances, including the calling process itself. In other words, calls to **kill** or **reset** can be nested (as can calls to **throw_it**, as described in 5.6.6.11).

(The default actions of the SystemC report handler following an error include the SC_THROW action, which by default throws an exception. An implementation is obliged to catch the **sc_unwind_exception** before having the **sc_report_handler** throw another exception.)

The target process is permitted to catch the **sc_unwind_exception** within its associated function body, in which case it shall re-throw the exception such that the implementation can properly execute the kill or reset. It shall be an error for an application to catch the **sc_unwind_exception** without re-throwing it.

The catch block in the target process shall execute in the context of the target process, not the calling process. As a consequence, the function **sc_get_current_process_handle** shall return a handle to the target process instance when called from the catch block.

virtual const char* **what**() const;

Member function **what** shall return an implementation-defined string describing the exception.

virtual bool **is_reset**() const;

Member function **is_reset** shall return the value **true** if the exception is thrown by **reset** and **false** if the exception is thrown by **kill**.

sc_unwind_exception();
sc_unwind_exception(const sc_unwind_exception&);
virtual ~**sc_unwind_exception**();

This exception shall only be thrown by the kernel. Hence, the constructors and destructor shall be protected members.

Example:

```
SC_MODULE(m) {
    SC_CTOR(m) {
        SC_THREAD(run);
    }

    void run() {
        try {
        ...
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        } catch (const sc_core::sc_unwind_exception& ex) {
            // Perform clean-up
            if (ex.is_reset())
            ...
            else
            ...
            throw ex;
        }
    }
    ...
};
```

5.6.6.8 is_unwinding

bool **is_unwinding**() const;

Member function **is_unwinding** shall return the value **true** from the point when the kernel throws an **sc_unwind_exception** to the point when the kernel catches that same **sc_unwind_exception**, and otherwise shall return the value **false**. Hence, **is_unwinding** shall return **true** when called during the unwinding of the call stack following a call to **kill** or **reset**, but not following a call to **throw_it**. The intent is that this member function can be called from the destructor of a local object defined within the function associated with a process instance, where it can be used to differentiate between the case where a process is killed or reset and the end of simulation.

If called for an invalid process handle, the implementation shall generate a warning and **is_unwinding** shall return the value **false**.

There also exists a non-member function that calls **is_unwinding** for the currently executing process (see 5.6.8).

5.6.6.9 reset_event

const sc_event& **reset_event**() const;

Member function **reset_event** shall return a reference to an event that is notified whenever the target process instance is reset, whether that be through an explicit call to member function **reset**, through a process instance being resumed while it is in the synchronous reset state, or through a reset signal attaining its active value. The reset event shall be scheduled using an immediate notification in the evaluation phase in which the explicit or implicit call to **reset** occurs. It shall be an error to call member function **reset_event** for an invalid process handle.

5.6.6.10 sync_reset_on and sync_reset_off

void **sync_reset_on**(sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS);
void **sync_reset_off**(sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS);

Member function **sync_reset_on** shall cause the target process instance to enter the *synchronous reset state*. While in the synchronous reset state, a process instance shall be reset each time it is resumed, whether due to an event notification or to a time-out. The call to **sync_reset_on** shall not itself cause the process to be reset immediately; the process shall only be reset each time it is subsequently resumed. Being reset in this sense shall have the same effect as would a call to the member function **reset**, as described above.

Member function **sync_reset_off** shall cause the target process instance to leave the *synchronous reset state* (unless any reset signal specified using **reset_signal_is** or **async_reset_signal_is** for that process instance has its active value), thereby reversing the effect of any previous call to

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

sync_reset_on for that process instance. **sync_reset_off** shall not modify the effect of any reset signal and shall not modify the effect of any explicit call to the member function **reset**.

As a consequence of the above rules, if a call to **sync_reset_on** is followed by a call to **sync_reset_off** before a particular process instance has been resumed, that process instance cannot have been reset due to the **sync_reset_on** call, although it may have been reset due to the effect of a reset signal (specified using **reset_signal_is** or **async_reset_signal_is**) or a call to the member function **reset**.

It is permissible to call **sync_reset_on** and **sync_reset_off** with the **include_descendants** argument having the value **SC_INCLUDE_DESCENDANTS** and where the descendants are a mixture of method and thread processes; the appropriate action shall be taken for each descendant process instance according to whether it is a method process or a thread process.

Calling **sync_reset_on** for a process instance that is already in the synchronous reset state shall have no effect on that particular process instance, although it may cause descendants to enter the synchronous reset state. Only a single call to **sync_reset_off** is required for a particular process instance to leave the synchronous reset state (assuming there is no reset signal active).

Calling **sync_reset_off** for a process instance that is not currently in the synchronous reset state shall have no effect on that particular process instance, although it may cause descendants to leave the synchronous reset state.

A process instance may call **sync_reset_on** or **sync_reset_off** with itself as the target. As a consequence of the above rules, the effect of the call will only be visible the next time the process instance is resumed.

If **sync_reset_on** is called for a target process instance that is not in the synchronous reset state and is suspended (meaning that **suspend** has been called), the behavior shall be implementation-defined (see 5.6.6.12).

Calling **sync_reset_on** or **sync_reset_off** for a terminated process instance shall have no effect on that particular process instance, although it may cause descendant process instances to enter or leave the synchronous reset state.

Calling **sync_reset_on** or **sync_reset_off** during elaboration or before the process has first executed is permitted, in which case the target process instance shall enter or leave the synchronous reset state accordingly. Being in the synchronous reset state shall have no effect on the initialization of a process instance; that is, the process instance shall be resumed during the first evaluation phase and shall run up to the point where it either returns or calls **wait**. If a process instance is still in the synchronous reset state when it is first resumed (from a call to **wait**), then it shall be reset as described above.

The function associated with any process instance can call **sync_reset_on** or **sync_reset_off**: there is no obligation that **sync_reset_on** and **sync_reset_off** should be called by the same process instance.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

5.6.6.11 throw_it

```
template <typename T>
void throw_it( const T& user_defined_exception,
    sc_descendant_inclusion_info include_descendants = SC_NO_DESCENDANTS );
```

Member function **throw_it** shall throw an exception within the target process instance. The exception to be thrown shall be passed as the first argument to function **throw_it**. The exception shall be thrown in the context of the target process, not in the context of the caller. Excepting the special case where a process throws an exception to itself, this shall require two context switches: from the caller to the target process and from the target process back to the caller.

It is recommended that the exception being thrown should be derived from class **std::exception**, but an application may throw an exception not derived from class **std::exception**.

Any dynamic sensitivity associated with the target process instance shall be removed and the static sensitivity restored. The target process instance shall not be terminated by virtue of the fact that **throw_it** was called, although a thread process may immediately terminate if control is returned from the associated function after the exception has been caught.

throw_it shall have immediate effect; that is, control shall be passed from the caller to the target process and the exception thrown and caught before the return from the **throw_it** function. Calls to **throw_it** can have side-effects.

The target process instance shall catch the exception in its associated function. It shall be an error for the target process instance not to catch the exception. After catching the exception, the associated function may return (and thus terminate in the case of a thread process) or may call **wait**. It shall be an error for the target process to throw another exception while handling the first exception.

If a process throws an exception to another process, control shall return to the function that called **throw_it**. If a process throws an exception to itself, function **throw_it** does not return control to the caller because its execution is interrupted by the exception, but control shall pass to a catch block in the function associated with the process.

The act of catching and handling the exception may have side-effects, including calls to destructors for local objects. Such side-effects shall not include calls to **wait** or **next_trigger**, but they may include calls to process control member functions for other process instances, including the calling process itself. In other words, calls to **throw_it** can be nested (as can calls to **kill** or **reset**, as described in 5.6.6.7).

throw_it is only applicable when the target is a non-terminated thread process. Calls to **throw_it** where the target process instance is a method process or a terminated process are permitted and shall have no effect except that an implementation may issue a warning. This shall include the case where a method process throws an exception to itself. In particular, it is permissible to call **throw_it** with the **include_descendants** argument having the value **SC_INCLUDE_DESCENDANTS** and where the descendants are a mixture of method and thread processes, some of which may have terminated; the appropriate action shall be taken for each descendant process instance. If the target process is a non-terminated method process, that process shall not be terminated by the call to **throw_it**.

throw_it is only applicable when the target process instance has been suspended during execution, that is, has called **wait** or has been the target of a call to **suspend** or **disable**. A call to **throw_it** in any other context during simulation shall have no effect except that an implementation may generate a warning. In particular, a call to **throw_it** during simulation before the target process has first executed shall have no effect except to generate a warning.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

It shall be an error to call **throw_it** during elaboration, before the initialization phase, or while the simulation is paused or stopped.

5.6.6.12 Interactions between member functions for process control

Table 2—When process control member functions take effect

Member function	When it takes effect on the target process instance
suspend	The current evaluation phase
resume	The current evaluation phase
disable	The next evaluation phase
enable	The current evaluation phase
kill	Immediate
reset	Immediate
throw_it	Immediate
sync_reset_on	The current evaluation phase
sync_reset_off	The current evaluation phase

In Table 2, the phrase *The current evaluation phase* means that the effect on the target process becomes visible during the evaluation phase in which the process control member function is called. For example, a call to **suspend** could remove the target process from the set of runnable processes, and an immediate notification following a call to **enable** could cause the target process to run in the current evaluation phase. *The next evaluation phase* implies that **disable** would not prevent the target process from running in the current evaluation phase, but it would prevent the target process from running in the subsequent evaluation phase. *Immediate* means that an exception is thrown and any associated actions are completed in the target process before return to the caller.

As described above, member functions **kill** and **reset** each achieve their effect by throwing an object of class **sc_unwind_exception**, while member function **throw_it** throws an exception of a user-defined class. Calls to **kill, reset**, and **throw_it** may be nested such that a process catching an exception may kill or reset another process instance or may throw an exception in another process instance before itself returning control to the kernel. Similarly, the destructor of an object that is going out-of-scope during stack unwinding (as a result of an **sc_unwind_exception** having been thrown) may itself kill or reset another process instance or may throw an exception in another process instance. It is possible that such a chain of calls can get interrupted by a process instance killing or resetting itself, either directly or indirectly. For example, if a given process instance attempts to kill all the descendants of its parent process with the call **handle.kill(SC_INCLUDE_DESCENDANTS)**, the descendant process instances will be killed one-by-one until the calling process instance is reached, at which point the caller itself will be killed and the call stack unwound.

Care should be taken to avoid mutual recursion between processes that kill or reset one another, as such mutual recursion can result in stack overflow.

The relative priorities of the process control member functions are given below, ordered from highest priority to lowest priority:

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

1) **kill**, **reset**, **throw_it**
2) **disable**, **enable**
3) **suspend**, **resume**
4) **sync_reset_on**, **sync_reset_off**

For each process instance, the implementation shall in effect maintain three independent flags to track the state of calls to the member functions **disable**/**enable**, **suspend**/**resume**, and **sync_reset_on**/**sync_reset_off**, and shall then act according to the relative priority of those calls as listed above. Each of these three flags shall be set and cleared by calls to the corresponding pair of member functions irrespective of the values of the remaining two flags and irrespective of the fact that certain interactions between the process control member functions are implementation-defined. For example, given two successive calls to **resume** with no intervening call to **suspend**, the second call to **resume** would have no effect, irrespective of any intervening calls to **disable** or **enable**.

Member functions **kill**, **reset**, and **throw_it** have immediate semantics that are executed immediately even if the target process instance is disabled, suspended, or in the synchronous reset state. In such a case, the target process shall remain disabled, suspended, or in the synchronous reset state after the immediate action has been completed. For example, a call to **throw_it** where the target process was disabled might cause the target process to wake up, catch the exception, call **wait** within its catch block, and finally yield control back to the kernel, all the while remaining in the disabled state.

The behavior of certain interactions between the process control member functions shall be implementation-defined, as follows:

a) If **resume** is called for a target process instance that is currently both suspended and disabled, the behavior shall be implementation-defined (where suspended means **suspend** has been called and disabled means **disable** has been called).
b) If **sync_reset_on** is called or if a synchronous or asynchronous reset signal attains its active value for a target process instance that is currently both suspended and not in the synchronous reset state, the behavior shall be implementation-defined.
c) If **suspend** is called for a target process instance that is currently both in the synchronous reset state (whether by means of a call to **sync_reset_on** or by means of a synchronous or asynchronous reset signal having attained its active value) and not suspended, the behavior shall be implementation-defined.
d) If **disable** is called for a target process instance that is currently both not disabled and waiting on a time-out (whether or not the time-out is accompanied by an event or an event list), the behavior shall be implementation-defined.

If **reset** is called while a target process instance is suspended, **reset** shall remove the dynamic sensitivity of the target process such that any event notification or time-out that occurred before the time of the **reset** will be ignored when determining the behavior of a subsequent call to **resume** for that same target process. In other words, a **reset** shall wipe the slate clean when determining the behavior of **resume**.

It shall be an error to call the three member functions with immediate semantics during elaboration, before the initialization phase, or while the simulation is paused or stopped, but the remaining member functions may be called while in those states.

NOTE—The intent of making certain interactions between the process control member functions implementation-defined is to shield the user from surprising behavior while also allowing for more specific semantics to be defined in the future as and when the use cases for the process control member functions are clarified.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Example:

```
sc_core::sc_process_handle t;
...
t.sync_reset_on();      // Target process put into the synchronous reset state
A();                    // Blocking call

t.suspend();            // Target process suspended
B();                    // Blocking call

t.disable();            // Target process disabled
C();                    // Blocking call

t.reset();              // Target process immediately reset but still disabled
                        // Static sensitivity restored for target process
D();                    // Blocking call

t.enable();             // Target process enabled but still suspended
E();                    // Blocking call

t.disable();            // Target process disabled
F();                    // Blocking call

t.enable();             // Target process enabled but still suspended
G();                    // Blocking call

t.resume();             // Target process resumed but still in the synchronous reset state
                        // Target process is made runnable if sensitive to an event notified by E or G
                        // Target process not made runnable if sensitive to an event notified by A, B, C, D,
                        // or F
```

5.6.7 sc_get_current_process_handle

sc_process_handle **sc_get_current_process_handle**();

The value returned from function **sc_get_current_process_handle** shall depend on the context in which it is called. When called during elaboration from the body of a module constructor or from a function called from the body of a module constructor, **sc_get_current_process_handle** shall return a handle to the spawned or unspawned process instance most recently created within that module, if any. When called from one of the callbacks **before_end_of_elaboration** or **end_of_elaboration**, **sc_get_current_process_handle** shall return a handle to the spawned or unspawned process instance most recently created within that specific callback function, if any. If the most recently created process instance was not within the current module, or in the case of **before_end_of_elaboration** or **end_of_elaboration** was not created within that specific callback, an implementation may return either a handle to the most recently created process instance or an invalid handle. When called from **sc_main** during elaboration or from the callback **start_of_simulation**, **sc_get_current_process_handle** may return either a handle to the most recently created process instance or an invalid handle. When called during simulation, **sc_get_current_process_handle** shall return a handle to the currently executing spawned or unspawned process instance, if any. If there is no such process instance, **sc_get_current_process_handle** shall return an invalid handle. When called from **sc_main** during simulation, **sc_get_current_process_handle** shall return an invalid handle.

Example:

```
SC_MODULE(Mod){
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    ...
    sc_in<bool> in;
    SC_CTOR(Mod) {
        SC_METHOD(run);
        sensitive << in;
            sc_core::sc_process_handle h1 = sc_core::sc_get_current_process_handle();
                                                                    // Returns a handle to process run
    }

    void run() {
        using namespace sc_core;

        sc_process_handle h2 = sc_get_current_process_handle();  // Returns a handle to process run

        if (h2.proc_kind() == SC_METHOD_PROC_)
        ...                                                         // Running a method process
        sc_object *parent = h2.get_parent_object();                 // Returns a pointer to the
                                                                    // module instance
        if (parent) {
            handle = sc_process_handle(parent);                     // Invalid handle - parent is not a process
            if (handle.valid())
            ...                                                 // Executed if parent were a valid process
        }
    }
    ...
};
```

5.6.8 sc_is_unwinding

bool **sc_is_unwinding**();

Function **sc_is_unwinding** shall return the value returned from member function **is_unwinding** of the process handle returned from **sc_get_current_process_handle**. In other words, **sc_is_unwinding** shall return **true** if and only if the caller is currently the target process instance of a call to **kill** or **reset**.

Example:

```
struct wait_on_exit {
    ~wait_on_exit() {
        if( !sc_core::sc_is_unwinding() )               // needed, because the process might get killed
            wait( 10.0, sc_core::SC_NS ) ;              // ... and this wait would be illegal
    }
};

void some_module::some_process() {
    while( true ) {
        try {
            wait_on_exit w;                             // local object, destroyed before catch
            ...
        } catch( const sc_core::sc_unwind_exception& ) {
        ...                                             // some other cleanup
            throw;
        }
    }
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

}

5.7 sc_event_finder and sc_event_finder_t

5.7.1 Description

An *event finder* is a member function of a port class with a return type of **sc_event_finder&**. When a port instance is bound to a channel instance containing multiple events, an event finder permits a specific event from the channel to be retrieved through the port instance and added to the static sensitivity of a process instance. **sc_event_finder_t** is a templated wrapper for class **sc_event_finder**, where the template parameter is the interface type of the port.

An event finder function is called when creating static sensitivity to events through a port. Because port binding may be deferred, it may not be possible for the implementation to retrieve an event to which a process is to be made sensitive at the time the process instance is created. Instead, an application should call an event finder function, in which case the implementation shall defer the adding of events to the static sensitivity of the process until port binding has been completed. These deferred actions shall be completed by the implementation before the callbacks to function **end_of_elaboration**.

If an event finder function is called for a multiport bound to more than one channel instance, the events for all such channel instances shall be added to the static sensitivity of the process.

5.7.2 Class definition

```
namespace sc_core {

class sc_event_finder implementation-defined ;

template <class IF>
class sc_event_finder_t
: public sc_event_finder
{
    public:
        sc_event_finder_t( const sc_port_base& port_, const sc_event& (IF::*event_method_) () const );

        // Other members
        implementation-defined
};

}       // namespace sc_core
```

5.7.3 Constraints on usage

An application shall only use class **sc_event_finder** as the return type (passed by reference) of a member function of a port class, or as the base class for an application-specific event finder class template that may possess additional template parameters and event method parameters.

An application shall only use class **sc_event_finder_t<interface>** in constructing the object returned from an event finder.

An event finder shall have a return type of **sc_event_finder&** and shall return an object of class **sc_event_finder_t<interface>** or an application-specific event finder class template, where

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

a) *interface* shall be the name of an interface to which said port can be bound, and
b) the first argument passed to the constructor for said object shall be the port object itself, and
c) the second argument shall be the address of a member function of said interface. The event *found by* the event finder is the event returned by this function.

An event finder member function may only be called when creating the static sensitivity of a process using **operator<<**, function **set_sensitivity**, or macro SC_CTHREAD. An event finder member function shall only be called during elaboration, either from a constructor or from the **before_end_of_elaboration** callback. An event finder member function shall not be called from the **end_of_elaboration** callback or during simulation. Instead, an application may make a process directly sensitive to an event.

In the case of a multiport, an event finder member function cannot find an event from an individual channel instance to which the multiport is bound using an index number. An application can work around this restriction by getting the events from the individual channel instances in the **end_of_elaboration** callback after port binding is complete (see example below).

Example:

```
#include <systemc>

class if_class : virtual public sc_core::sc_interface {
 public:
    virtual const sc_core::sc_event& ev_func() const = 0;
    ...
};

class chan_class : public if_class, public sc_core::sc_prim_channel {
public:
    virtual const sc_core::sc_event& ev_func() const { return an_event; }
    ...
private:
    sc_core::sc_event an_event;
};

template<int N = 1>
class port_class : public sc_core::sc_port<if_class, N> {
public:
    sc_core::sc_event_finder& event_finder() const {
        return *new sc_core::sc_event_finder_t<if_class>( *this , &if_class::ev_func );
    }
    ...
};

SC_MODULE(mod_class) {
    port_class<1> port_var;
    port_class<0> multiport;

    SC_CTOR(mod_class) {
        SC_METHOD(method);
        sensitive << port_var.event_finder();                    // Sensitive to chan_class::an_event
    }

    void method();
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    ...

    void end_of_elaboration() {
        SC_METHOD(method2);
        for (int i = 0; i < multiport.size(); i++)
            sensitive << multiport[i]->ev_func();          // Sensitive to chan_class::an_event
    }

    void method2();
    ...
};
```

NOTE—For particular examples of event finders, refer to the functions **pos** and **neg** of class **sc_in<bool>** (see 6.9).

5.8 sc_event_and_list and sc_event_or_list

5.8.1 Description

The classes **sc_event_and_list** and **sc_event_or_list** are used to represent explicit event list objects which may be constructed and manipulated prior to being passed as arguments to the functions **next_trigger** (see 5.2.17) and **wait** (see 5.2.18). An event list object shall store pointers or references to zero or more event objects. An event list may contain multiple references to the same event object. The order of the events within the event list shall have no effect on the behavior of the event list when it is used to create dynamic sensitivity.

5.8.2 Class definition

```
namespace sc_core {

class sc_event_and_list
{
public:
    sc_event_and_list();
    sc_event_and_list( const sc_event_and_list& );
    sc_event_and_list( const sc_event& );
    sc_event_and_list& operator= ( const sc_event_and_list& );
    ~sc_event_and_list();

    int size() const;
    void swap( sc_event_and_list& );

    sc_event_and_list& operator&= ( const sc_event& );
    sc_event_and_list& operator&= ( const sc_event_and_list& );

    sc_event_and_expr† operator&  ( const sc_event& ) const;
    sc_event_and_expr† operator&  ( const sc_event_and_list& ) const;
};

class sc_event_or_list
{
public:
    sc_event_or_list();
    sc_event_or_list( const sc_event_or_list& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    sc_event_or_list( const sc_event& );
    sc_event_or_list& operator= ( const sc_event_or_list& );
    ~sc_event_or_list();

    int size() const;
    void swap( sc_event_or_list& );

    sc_event_or_list& operator|= ( const sc_event& );
    sc_event_or_list& operator|= ( const sc_event_or_list& );

    sc_event_or_expr† operator| ( const sc_event& ) const;
    sc_event_or_expr† operator| ( const sc_event_or_list& ) const;
};

} // namespace sc_core
```

5.8.3 Constraints and usage

The intended usage for objects of class **sc_event_and_list** and **sc_event_or_list** is that they are passed as arguments to the functions **wait** and **next_trigger**. Unlike the case of event expression objects, which are deleted automatically by the implementation (see 5.9.3), the application shall be responsible for deleting event list objects when they are no longer required. The application shall be responsible for ensuring that an event list object is still valid (that is, has not been destroyed) when a process instance resumes or is triggered as a direct result of the notification of one or more events in the event list. If the event list object has already been destroyed at this point, the behavior of the implementation shall be undefined.

5.8.4 Constructors, destructor, assignment

sc_event_and_list();
sc_event_or_list();

Each default constructor shall construct an empty event list object. It shall be an error to pass an empty event list object as an argument to the functions **wait** or **next_trigger**.

sc_event_and_list(const sc_event_and_list&);
sc_event_or_list(const sc_event_or_list&);
sc_event_and_list& **operator=** (const sc_event_and_list&);
sc_event_or_list& **operator=** (const sc_event_or_list&);

The copy constructors and assignment operators shall permit event list objects to be initialized and assigned by the application.

sc_event_and_list(const sc_event&);
sc_event_or_list(const sc_event&);

These constructors shall construct event list objects containing the single event passed as an argument to the constructor.

~sc_event_and_list();
~sc_event_or_list();

An event list object may be constructed as a stack or as a heap variable. In the case of the heap, the application is responsible for deleting an event list object when it is no longer required.

5.8.5 Member functions and operators

int **size**() const;

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Member function **size** shall return the number of events in the event list. Any duplicate events in the event list shall not count toward the value of **size**.

Example:

```
{
    using namespace sc_core;
    sc_event ev;
    sc_event_or_list list = ev | ev;
    sc_assert( list.size() == 1 );
}
```

void **swap**(sc_event_and_list&);
void **swap**(sc_event_or_list&);

Member function **swap** shall exchange the current event list object ***this** with the event list passed as an argument.

sc_event_and_list& **operator&=** (const sc_event&);
sc_event_and_list& **operator&=** (const sc_event_and_list&);
sc_event_and_expr† **operator&** (const sc_event&) const;
sc_event_and_expr† **operator&** (const sc_event_and_list&) const;
sc_event_or_list& **operator|=** (const sc_event&);
sc_event_or_list& **operator|=** (const sc_event_or_list&);
sc_event_or_expr† **operator|** (const sc_event&) const;
sc_event_or_expr† **operator|** (const sc_event_or_list&) const;

Each of these operators shall append the event or event list passed as an argument to the current event list object ***this**, and shall return the resultant elongated event list as the value of the operator. In the case of the assignment operators **&=** and **|=**, the operator shall return a reference to the current object, which shall have been modified to contain the elongated event list. In the case of the remaining operators **&** and **|**, the current object ***this** shall not be modified.

Example:

```
struct M : sc_core::sc_module {
    sc_core::sc_port<sc_core::sc_signal_in_if<int>, 0> p;          // Multiport

    M(sc_core::sc_module_name _name) {
        SC_THREAD(T);
    }

    sc_core::sc_event_or_list all_events() const {
        sc_core::sc_event_or_list or_list;
        for (int i = 0; i < p.size(); i++)
            or_list |= p[i]->default_event();
        return or_list;
    }

    sc_core::sc_event event1, event2;
    ...

    void T() {
        using namespace sc_core;
        for (;;) {
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
            wait(all_events());
            ...

            sc_event_and_list list;
            sc_assert(list.size() == 0);

            list = list & event1;
            sc_assert(list.size() == 1);

            list &= event2;
            sc_assert(list.size() == 2);

            wait(list);
            sc_assert(list.size() == 2);
            ...
        }
    }
};
```

5.9 *sc_event_and_expr†* and *sc_event_or_expr†*

5.9.1 Description

The classes *sc_event_and_expr†* and *sc_event_or_expr†* provide the **&** and | operators used to construct the event expressions passed as arguments to the functions **wait** (see 5.2.17) and **next_trigger** (see 5.2.18). An event expression object shall store pointers or references to zero or more event objects. An event expression may contain multiple references to the same event object. The order of the events within the event expression shall have no effect on the behavior of the event expression when it is used to create dynamic sensitivity.

5.9.2 Class definition

```
namespace sc_core {

class sc_event_and_expr†
{
    public:
        operator const sc_event_and_list &() const;

        // Other members
        implementation-defined
};

sc_event_and_expr† operator& ( sc_event_and_expr† , sc_event const&  );
sc_event_and_expr† operator& ( sc_event_and_expr† , sc_event_and_list const& );

class sc_event_or_expr†
{
    public:
        operator const sc_event_or_list &() const;

        // Other members
        implementation-defined
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
};

sc_event_or_expr† operator| ( sc_event_or_expr† , sc_event const&  );
sc_event_or_expr† operator| ( sc_event_or_expr† , sc_event_or_list const& );

}       // namespace sc_core
```

5.9.3 Constraints on usage

An application shall not explicitly create an object of class *sc_event_and_expr*† or *sc_event_or_expr*†. An application wishing to create explicit event list objects should use the classes **sc_event_and_list** and **sc_event_or_list**.

Classes *sc_event_and_expr*† and *sc_event_or_expr*† are the return types of **operator&** and **operator|**, respectively, of classes **sc_event**, **sc_event_and_list**, and **sc_event_or_list**.

The existence of the type conversion operators from type *sc_event_and_expr*† to type **const sc_event_and_list &** and from type *sc_event_or_expr*† to type **const sc_event_or_list &** allow expressions of type *sc_event_and_expr*† and *sc_event_or_expr*† to be passed as arguments to the functions **wait** and **next_trigger**, for example, **wait(ev1 & ev2)**. The objects of type *sc_event_and_expr*† and *sc_event_or_expr*† are temporaries returned by **operator&** or **operator|** in an event expression, and would be destroyed automatically after the call to **wait** or **next_trigger**. The implementation shall delete the event list object when the process instance resumes or is triggered as a direct result of the notification of one or more events in the event expression. In other words, the implementation shall be responsible for the creation, deletion, and memory management of event list objects returned from the above type conversion operators.

5.9.4 Operators

operator const sc_event_and_list &() const;
operator const sc_event_or_list &() const;

Each of these type conversion operators shall return a reference to an event list object equivalent to the event list expression represented by the current object ***this**, equivalent in the sense that the event list object shall contain references to the same set of events as the event list expression.

```
sc_event_and_expr† operator& ( sc_event_and_expr† , sc_event const&  );
sc_event_and_expr† operator& ( sc_event_and_expr† , sc_event_and_list const&  );
sc_event_or_expr† operator| ( sc_event_or_expr† , sc_event const&  );
sc_event_or_expr† operator| ( sc_event_or_expr† , sc_event_or_list const&  );
```

A call to either operator shall append the event or events passed as the second argument to the event expression passed as the first argument, and shall return the resultant elongated event expression as the value of the operator.

Example:

```
sc_core::sc_event event1, event2;

void thread_process() {
    ...
    wait( event1 | event2 );
    // When the thread process resumes following the notification of event1 or event2,
    // the implementation shall delete the object of type sc_event_or_expr returned from operator|
    ...
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

}

5.10 sc_event

5.10.1 Description

An event is an object of class **sc_event** used for process synchronization. A process instance may be triggered or resumed on the *occurrence* of an event, that is, when the event is notified. Any given event may be notified on many separate occasions.

Events can be *hierarchically named* or not. An event without a hierarchical name shall have an implementation-defined name. A hierarchically named event shall either have a parent in the object hierarchy (in which case the event would be returned from member function **get_child_events** of that parent) or be a top-level event (in which case the event would be returned by function **sc_get_top_level_events**). If an event has a parent, that parent shall be either a module instance or a process instance. An event without a hierarchical name shall not have a parent in the object hierarchy.

A *kernel event* is an event that is instantiated by the implementation, such as an event within a predefined primitive channel. A kernel event shall not be hierarchically named but shall have an implementation-defined name, regardless of whether it is instantiated during elaboration or during simulation.

With the exception of kernel events, every event that is instantiated during elaboration or before the initialization phase shall be a hierarchically named event.

NOTE 1—Events without a hierarchical name are provided for backward compatibility with earlier versions of this standard and to avoid the potential performance impact of creating hierachical names during simulation.

NOTE 2—Although an event may have a parent of type **sc_object**, an event object is not itself an **sc_object**, and hence an event cannot be returned by member function **get_child_objects**. Member function **get_child_events** is provided for the purpose of retrieving the events within a given module or process instance.

NOTE 3—In the case of a hierarchically named event, the rules for hierachical naming imply that the name of the event is formed by concatenating the hierarchical name of the parent of the event with the basename of the event, with the two parts separated by a period character.

5.10.2 Class definition

```
namespace sc_core {

class sc_event
{
    public:
        sc_event();
         explicit sc_event( const char* );
        ~sc_event();

        const char* name() const;
        const char* basename() const;
        bool in_hierarchy() const;
        sc_object* get_parent_object() const;
        bool triggered() const;

        void notify();
        void notify( const sc_time& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        void notify( double , sc_time_unit );
        void cancel();

        sc_event_and_expr† operator& ( const sc_event& ) const;
        sc_event_and_expr† operator& ( const sc_event_and_list& ) const;
        sc_event_or_expr† operator| ( const sc_event& ) const;
        sc_event_or_expr† operator| ( const sc_event_or_list& ) const;
        static const sc_event none;

    private:
        // Disabled
        sc_event( const sc_event& );
        sc_event& operator= ( const sc_event& );
};

const std::vector<sc_event*>& sc_get_top_level_events();
sc_event* sc_find_event( const char* );

}       // namespace sc_core
```

5.10.3 Constraints on usage

Objects of class **sc_event** may be constructed during elaboration or simulation. Events may be notified during elaboration or simulation, except that it shall be an error to create an immediate notification during elaboration or from one of the callbacks **before_end_of_elaboration**, **end_of_elaboration**, or **start_of_simulation**.

5.10.4 Constructors, destructor, and event naming

```
sc_event();
explicit sc_event( const char* );
```

Calling the constructor **sc_event(const char*)** with an empty string shall have the same effect as calling the default constructor, regardless of when it is called by the application.

When called by the application during elaboration or before the initialization phase, each of the above two constructors shall create a hierarchically named event.

When the default constructor is called by the application from the initialization phase onwards, whether or not a hierarchically named event is created shall be implementation-defined on a per-instance basis.

When the constructor **sc_event(const char*)** is called by the application from the initialization phase onward with a non-empty string argument, the implementation shall create a hierachically named event.

When a hierarchically named event is constructed, if a non-empty string is passed as a constructor argument, that string shall be used to set the string name of the event. Otherwise, the string name shall be set to **"event"**. The string name shall be used to determine the hierarchical name as described in 5.17.

If a constructor needs to substitute a new string name in place of the original string name as the result of a name clash, the constructor shall generate a single warning.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

An event that is not hierarchically named shall have a non-empty implementation-defined name. That name shall take the form of a hierarchical name as described in 5.17 but may contain one or more characters that are not in the recommended character set for application-defined hierarchical names (such as punctuation characters and mathematical symbols). The intent is that an implementation may use special characters to distinguish implementation-defined names from application-defined names, although it is not obliged to do so.

Kernel events shall obey the same rules as application-defined events that are not hierarchically named.

NOTE—An implementation is not required to guarantee that each implementation-defined event name is unique. The intent is that the implementation will use characters not recommended in application-defined names to reduce the probability of a name clash between a hierarchical name and an implementation-defined name.

~sc_event();

The destructor shall delete the object and shall remove the event from the object hierarchy such that the event is no longer a top-level event or a child event.

5.10.5 Functions for naming and hierarchy traversal

const char* **name**() const;

Member function **name** shall return the hierarchical name of the event instance in the case of a hierarchically named event, or otherwise, it shall return the implementation-defined name of the event instance. The name shall always be non-empty.

const char* **basename**() const;

Member function **basename** shall return the string name of the event instance in the case of a hierarchically named event, or otherwise, it shall return an implementation-defined name. This is the string name created when the event instance was constructed. In the case of an implementation-defined name, the relationship between the strings returned from member functions **name** and **basename** is also implementation-defined.

bool **in_hierarchy**() const;

Member function **in_hierarchy** shall return the value **true** if and only if the event is a hierarchically named event.

bool **triggered**() const;

Member function **triggered** shall return the value **true** if and only if the event has been triggered in the immediately preceding delta notification phase or it has been triggered in the current evaluation phase via an immediate notification.

NOTE—This function may be used to determine whether an event has recently been triggered, e.g., to determine which event(s) in an **sc_event_or_list** triggered a process or whether a time-out has been reached.

The three functions below return information that supports the traversal of events that have a parent in the object hierarchy. An implementation shall allow each of these three functions to be called at any stage during elaboration or simulation. If called before elaboration is complete, they shall return information concerning the partially constructed object hierarchy as it exists at the time the functions are called. In other words, a function shall return pointers to any **sc_event** objects that have been constructed before the time the function is called but will exclude any objects constructed after the function is called.

sc_object* **get_parent_object**() const;

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Member function **get_parent_object** shall return a pointer to the **sc_object** that is the parent of the current **sc_event** object in the case of hierarchically named events, or otherwise, it shall return the null pointer. **get_parent_object** shall also return a null pointer for a top-level event. If the parent object is a process instance and that process has terminated, **get_parent_object** shall return a pointer to that process instance. A process instance shall not be deleted (nor any associated process handles invalidated) while the process has surviving children, but it may be deleted once all its child objects have been deleted.

const std::vector<sc_event*>& **sc_get_top_level_events**();

Function **sc_get_top_level_events** shall return a **std::vector** containing pointers to all of the top-level **sc_event** objects, that is, events that are hierarchically named but have no parent.

sc_event* **sc_find_event**(const char*);

Function **sc_find_event** shall return a pointer to the hierarchically named **sc_event** object that has a name that exactly matches the value of the string argument or shall return the null pointer if there is no hierarchically named **sc_event** with that name. Function **sc_find_event** shall not return a pointer to an event that has an implementation-defined name.

5.10.6 notify and cancel

void **notify**();

A call to member function **notify** with an empty argument list shall create an immediate notification. Any and all process instances sensitive to the event shall be made runnable before control is returned from function **notify**, with the exception of the currently executing process instance, which shall not be made runnable due to an immediate notification regardless of its static or dynamic sensitivity (see 4.3.2.3).

NOTE 1—Process instances sensitive to the event will not be resumed or triggered until the process that called **notify** has suspended or returned.

NOTE 2—All process instances sensitive to the event will be run in the current evaluation phase and in an order that is implementation-defined. The presence of immediate notification can introduce non-deterministic behavior.

NOTE 3—Member function **update** of class **sc_prim_channel** shall not call **notify** to create an immediate notification.

void **notify**(const sc_time&);
void **notify**(double , sc_time_unit);

A call to member function **notify** with an argument that represents a zero time shall create a delta notification.

A call to function **notify** with an argument that represents a non-zero time shall create a timed notification at the given time, expressed relative to the simulation time when function **notify** is called. In other words, the value of the time argument is added to the current simulation time to determine the time at which the event will be notified. The time argument shall not be negative.

NOTE 4—In the case of a delta notification, all processes that are sensitive to the event in the delta notification phase will be made runnable in the subsequent evaluation phase. In the case of a timed notification, all processes sensitive to the event at the time the event occurs will be made runnable at the time, which will be a future simulation time.

void **cancel**();

Member function **cancel** shall delete any pending notification for this event.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

NOTE 5—At most one pending notification can exist for any given event.

NOTE 6—Immediate notification cannot be cancelled.

5.10.7 Event lists

sc_event_and_expr† **operator&** (const sc_event&) const;
sc_event_and_expr† **operator&** (const sc_event_and_list&) const;
sc_event_or_expr† **operator|** (const sc_event&) const;
sc_event_or_expr† **operator|** (const sc_event_or_list&) const;

A call to either operator shall return an event expression formed by appending the current event object to the event or event list passed as an argument.

NOTE—Event lists are used as arguments to functions **wait** (see 5.2.18) and **next_trigger** (see 5.2.17).

5.10.8 None event

The static member *none* is provided for contexts where an (otherwise unused) event reference is required. A none event is guaranteed to never be notified and const.

NOTE—This static member prevents explicit notifications to be mistakenly called from application code.

5.10.9 Multiple event notifications

A given event shall have no more than one pending notification.

If function **notify** is called for an event that already has a notification pending, only the notification scheduled to occur at the earliest time shall survive. The notification scheduled to occur at the later time shall be cancelled (or never be scheduled in the first place). An immediate notification is taken to occur earlier than a delta notification, and a delta notification earlier than a timed notification. This is irrespective of the order in which function **notify** is called.

Example:

```
{
    using namespace sc_core;
    sc_event e;
    e.notify(SC_ZERO_TIME);      // Delta notification
    e.notify(1.0, SC_NS);        // Timed notification ignored due to pending delta notification
    e.notify();                  // Immediate notification cancels pending delta notification
                                 // e is notified
    e.notify(2.0, SC_NS);        // Timed notification
    e.notify(3.0, SC_NS);        // Timed notification ignored due to earlier pending timed notification
    e.notify(1.0, SC_NS);        // Timed notification cancels pending timed notification
    e.notify(SC_ZERO_TIME);      // Delta notification cancels pending timed notification
                                 // e is notified in the next delta cycle
}
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

5.11 sc_time

5.11.1 Description

Class **sc_time** is used to represent simulation time and time intervals, including delays and time-outs. An object of class **sc_time** is constructed from a **double** and an **sc_time_unit**. Time shall be represented internally as an unsigned integer of at least 64 bits declared as **sc_time::value_type**.

The values of enum **sc_time_unit** shall follow the International System of Units, for example, SC_FS represents a femtosecond or the equivalent of 1.0E-15 seconds.[7]

5.11.2 Class definition

```
namespace sc_core {

enum sc_time_unit { SC_SEC, SC_MS, SC_US, SC_NS, SC_PS, SC_FS, SC_AS, SC_ZS, SC_YS };

class sc_time
{
    public:

        typedef implementation-defined value_type;

        sc_time();
        sc_time( double , sc_time_unit );
        sc_time( const sc_time& );
        explicit sc_time( std::string_view );
        static sc_time from_value( value_type );
        static sc_time from_seconds( double );
        static sc_time from_string( std::string_view );

        sc_time& operator= ( const sc_time& );

        value_type value() const;
        double to_double() const;
        double to_seconds() const;


        bool operator== ( const sc_time& ) const;
        bool operator!= ( const sc_time& ) const;
        bool operator<  ( const sc_time& ) const;
        bool operator<= ( const sc_time& ) const;
        bool operator>  ( const sc_time& ) const;
        bool operator>= ( const sc_time& ) const;

        sc_time& operator+= ( const sc_time& );
        sc_time& operator-= ( const sc_time& );
        sc_time& operator*= ( double );
        sc_time& operator/= ( double );
        sc_time& operator%= (const sc_time& );
```

[7] See https://www.nist.gov/publications/international-system-units-si-2019-edition for the International System of Units publication that was current at the time IEEE Std 1666-2023 was published.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        std::string to_string() const;
        void print( std::ostream& = std::cout ) const;
};

sc_time operator+ ( const sc_time&, const sc_time& );
sc_time operator- ( const sc_time&, const sc_time& );

sc_time operator* ( const sc_time&, double );
sc_time operator* ( double, const sc_time& );
sc_time operator/ ( const sc_time&, double );
double operator/ ( const sc_time&, const sc_time& );
sc_time operator% ( const sc_time&, const sc_time& );


std::ostream& operator<< ( std::ostream&, const sc_time& );

const sc_time SC_ZERO_TIME;

void sc_set_time_resolution( double, sc_time_unit );
sc_time sc_get_time_resolution();
const sc_time& sc_max_time();

}       // namespace sc_core
```

5.11.3 Constructors

sc_time();

The default constructor shall create an object having a time value of zero.

sc_time(**double**, **sc_time_unit**);

The constructor shall create an object having a time value according to the following rules:

- The given floating-point value shall be scaled and rounded to the nearest multiple of the time resolution. Whether the value is rounded up or down is implementation-defined. An implementation may generate a warning if rounding is applied, but is not obliged to do so.
- If the floating-point value is negative, the implementation shall issue a warning and use the value **SC_ZERO_TIME** instead.
- If the floating-point value is positive but smaller than time resolution, the implementation shall issue a warning and use the value **SC_ZERO_TIME** instead.
- If the floating-point value is bigger than **sc_max_time**, the implementation shall issue a warning and use the value **sc_max_time** instead (see 5.11.6).

sc_time(const sc_time&);

The copy constructor shall create an object having a time value equal to the time value of the object passed as argument.

sc_time(**std::string_view**);

If the time is specified as a constructor argument of type **std::string_view**, the string shall at least contain a value in decimal or scientific notation. The string may contain a metric prefix and time unit defined by the International System of Units. If the string does not contain a time unit, the value is interpreted as seconds. It shall be an error if the string cannot be translated into a time object.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Example:

```
{
    using namespace sc_core;

    sc_time("0");          // Equivalent to sc_time( SC_ZERO_TIME )
    sc_time("1 ms");       // Equivalent to sc_time( 1.0, SC_MS )
    sc_time("1.0E-3");     // Equivalent to sc_time( 1.0e-3, SC_SEC )
    sc_time("2.0m");       // Equivalent to sc_time( 2.0, SC_MS )
    sc_time("3 US");       // Invalid, metric prefix and time unit not according to
                           // International System of Units. Shall throw an error
    sc_time("3E us");      // Invalid, incomplete exponent notation. Shall throw an error
    sc_time("ps");         // Invalid, value missing. Shall throw an error
    sc_time("1 fs!");      // Invalid character found. Shall throw an error
}
```

5.11.4 Functions and operators

All arithmetic, relational, equality, and assignment operators declared in 5.11.2 shall be taken to have their natural meanings when performing integer arithmetic on the underlying representation of time. The results of integer underflow and divide-by-zero shall be implementation-defined.

static **sc_time from_value**(**value_type**);
static **sc_time from_seconds**(**double**);
static **sc_time from_string**(**std::string_view**);

These member functions shall return a time object for the given argument.

value_type value() const;
double **to_double**() const;
double **to_seconds**() const;

These member functions shall return the underlying representation of the time value. The member functions **to_double** and **to_seconds** shall convert the time object to a **double**. The member function **to_seconds** shall also scale the resultant value to units of 1 second.

std::string **to_string**() const;
void **print**(std::ostream& = std::cout) const;
std::ostream& **operator<<** (std::ostream& , const sc_time&);

These member functions shall return the time value converted to a string or print that string to the given stream. The format of the string is implementation-defined.

5.11.5 Time resolution

Time shall be represented internally as an integer multiple of the time resolution. The default time resolution is 1 picosecond. Every object of class **sc_time** shall share a single common global time resolution.

The time resolution can only be changed by calling the function **sc_set_time_resolution**. This function shall only be called during elaboration, shall not be called more than once, and shall not be called after constructing an object of type **sc_time** with a non-zero time value. The value of the **double** argument shall be positive and shall be a power of 10. It shall be an error for an application to break the rules given in this paragraph.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The function **sc_get_time_resolution** shall return the time resolution.

5.11.6 sc_max_time

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

```
const sc_time& sc_max_time();
```

The function **sc_max_time** shall return the maximum value of type **sc_time**, calculated after taking into account the time resolution. Every call to **sc_max_time** during a given simulation run shall return an object having the same value and representing the maximum simulation time. The actual value is implementation-defined.

Calling **sc_max_time** shall not freeze the time resolution, which can be modified after the call. Calling **value**, **to_double**, **to_seconds**, **to_string**, **print**, or **operator<<** on any **sc_time** object, including objects created by **sc_max_time**, will force the time resolution.

Whether each call to **sc_max_time** returns a reference to the same object or a different object is implementation-defined.

5.11.7 SC_ZERO_TIME

Constant SC_ZERO_TIME represents a time value of zero. It is good practice to use this constant whenever writing a time value of zero, for example, when creating a delta notification or a delta time-out.

Example:

```
sc_core::sc_event e;
e.notify(sc_core::SC_ZERO_TIME);        // Delta notification
wait(sc_core::SC_ZERO_TIME);            // Delta time-out
```

5.12 sc_port

5.12.1 Description

Ports provide the means by which a module can be written such that it is independent of the context in which it is instantiated. A port forwards interface method calls to the channel to which the port is bound. A port defines a set of services (as identified by the type of the port) that are required by the module containing the port.

If a module is to call a member function belonging to a channel that is outside the module itself, that call should be made using an interface method call through a port of the module. To do otherwise is considered bad coding style. However, a call to a member function belonging to a channel instantiated within the current module may be made directly. This is known as *portless* channel access. If a module is to call a member function belonging to a channel instance within a child module, that call should be made through an export of the child module (see 5.13).

5.12.2 Class definition

```
namespace sc_core {

enum sc_port_policy
{
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    SC_ONE_OR_MORE_BOUND ,                  // Default
    SC_ZERO_OR_MORE_BOUND ,
    SC_ALL_BOUND
};

class sc_port_base
: public sc_object
{
    public:
        virtual sc_interface* get_interface() = 0;
        virtual const sc_interface* get_interface() const = 0;
        virtual std::type_index get_interface_type() const = 0;

    protected:
        explicit sc_port_base( int, sc_port_policy );
        sc_port_base( const char*, int, sc_port_policy );
        virtual ~sc_port_base();
};

template <class IF>
class sc_port_b
: public sc_port_base
{
    public:
        void operator() ( IF& );
        void operator() ( sc_port_b<IF>& );

        virtual void bind( IF& );
        virtual void bind( sc_port_b<IF>& );

        int size() const;

        IF* operator-> ();
        const IF* operator-> () const;

        IF* operator[] ( int );
        const IF* operator[] ( int ) const;

        virtual sc_interface* get_interface();
        virtual const sc_interface* get_interface() const;
        virtual std::type_index get_interface_type() const;

    protected:
        virtual void before_end_of_elaboration();
        virtual void end_of_elaboration();
        virtual void start_of_simulation();
        virtual void end_of_simulation();

        explicit sc_port_b( int , sc_port_policy );
        sc_port_b( const char* , int , sc_port_policy );
        virtual ~sc_port_b();

    private:
        // Disabled
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_port_b();
        sc_port_b( const sc_port_b<IF>& );
        sc_port_b<IF>& operator = ( const sc_port_b<IF>& );
};

template <class IF, int N = 1, sc_port_policy P = SC_ONE_OR_MORE_BOUND>
class sc_port
: public sc_port_b<IF>
{
    public:
        sc_port();
        explicit sc_port( const char* );
        virtual ~sc_port();

        virtual const char* kind() const;

    private:
         // Disabled
        sc_port( const sc_port<IF,N,P>& );
        sc_port<IF,N,P>& operator= ( const sc_port<IF,N,P>& );
};

}       // namespace sc_core
```

5.12.3 Template parameters

The first argument to template **sc_port** shall be the name of an *interface proper*. This interface is said to be the *type of the port*. A port can only be bound to a channel derived from the type of the port or to another port or export with a type derived from the type of the port.

The second argument to template **sc_port** is an optional integer value. If present, this argument shall specify the maximum number of channel instances to which any one instance of the port belonging to any specific module instance may be bound. If the value of this argument is zero, the port may be bound to an arbitrary number of channel instances. It shall be an error to bind a port to more channel instances than the number permitted by the second template argument.

The default value of the second argument is 1. If the value of the second argument is not 1, the port is said to be a *multiport*. If a port is bound to another port, the value of this argument may differ between the two ports.

The third argument to template **sc_port** is an optional port policy of type **sc_port_policy**. The port policy argument determines the rules for binding multiports and the rules for unbound ports.

The policy SC_ONE_OR_MORE_BOUND means that the port instance shall be bound to one or more channel instances, the maximum number being determined by the value of the second template argument. It shall be an error for the port instance to remain unbound at the end of elaboration.

The policy SC_ZERO_OR_MORE_BOUND means that the port instance shall be bound to zero or more channel instances, the maximum number being determined by the value of the second template argument. The port instance may remain unbound at the end of elaboration.

The policy SC_ALL_BOUND means that the port instance shall be bound to exactly the number of channel instances given by value of the second template argument, no more and no less, provided that value is greater than zero. If the value of the second template argument is zero, policy SC_ALL_BOUND shall have

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

the same meaning as policy SC_ONE_OR_MORE_BOUND. It shall be an error for the port instance to remain unbound at the end of elaboration, or to be bound to fewer channel instances than the number required by the second template argument.

It shall be an error to bind a given port instance to a given channel instance more than once, whether directly or through another port.

The port policy shall apply independently to each port instance, even when a port is bound to another port. For example, if a port on a child module with a type **sc_port<IF>** is bound to a port on a parent module with a type **sc_port<IF, 2, SC_ALL_BOUND>**, the two port policies are contradictory and one or other will inevitably result in an error at the end of elaboration.

The port policies shall hold when port binding is completed by the implementation just before the callbacks to function **end_of_elaboration** but are not required to hold any earlier. For example, a port of type **sc_port<IF, 2, SC_ALL_BOUND>** could be bound once in a module constructor and once in the callback function **before_end_of_elaboration**.

Example:

```
{
    using namespace sc_core;

    sc_port<IF>                                 p0;  // Bound to exactly 1 channel instance
    sc_port<IF, 0>                              p1;  // Bound to 1 or more channel instances
                                                     // with no upper limit
    sc_port<IF, 3>                              p2;  // Bound to 1, 2, or 3 channel instances
    sc_port<IF, 0, SC_ZERO_OR_MORE_BOUND>       p3;  // Bound to 0 or more channel instances
                                                     // with no upper limit
    sc_port<IF, 1, SC_ZERO_OR_MORE_BOUND>       p4;  // Bound to 0 or 1 channel instances
    sc_port<IF, 3, SC_ZERO_OR_MORE_BOUND>       p5;  // Bound to 0, 1, 2, or 3 channel instances
    sc_port<IF, 3, SC_ALL_BOUND>                p6;  // Bound to exactly 3 channel instances
}
```

NOTE—A port may be bound indirectly to a channel by being bound to another port or export (see 4.2.4).

5.12.4 Constraints on usage

An implementation shall derive class **sc_port_base** from class **sc_object**.

Ports shall only be instantiated during elaboration and only from within a module. It shall be an error to instantiate a port other than within a module. It shall be an error to instantiate a port during simulation.

The member functions **size** and **get_interface** can be called during elaboration or simulation, whereas **operator->** and **operator[]** should only be called from **end_of_elaboration** or during simulation.

It is strongly recommended that a port within a given module be bound at the point where the given module is instantiated, that is, within the constructor from which the module is instantiated. Furthermore, it is strongly recommended that the port be bound to a channel or another port that is itself instantiated within the module containing the instance of the given module or to an export that is instantiated within a child module. This recommendation may be violated on occasion. For example, it is convenient to bind an otherwise unbound port from the **before_end_of_elaboration** callback of the port instance itself.

The constraint that a port be instantiated *within a module* allows for considerable flexibility. However, it is strongly recommended that a port instance be a data member of a module wherever practical; otherwise, the

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

syntax necessary for named port binding becomes somewhat arcane in that it requires more than simple class member access using the dot operator.

Suppose a particular port is instantiated within module C, and module C is itself instantiated within module P. It is permissible for the port to be bound at some point in the code remote from the point at which module C is instantiated, it is permissible for the port to be bound to a channel (or another port) that is itself instantiated in a module other than the module P, and it is permissible for the port to be bound to an export that is instantiated somewhere other than in a child module of module P. However, all such cases would result in a breakdown of the normal discipline of the module hierarchy and are strongly discouraged in typical usage.

5.12.5 Constructors

explicit **sc_port_b**(int , sc_port_policy);
sc_port_b(const char* , int , sc_port_policy);
virtual ~**sc_port_b**();

> The constructor for class **sc_port_b** shall pass the values of its arguments though to the constructor for the base class sub-object. The character string argument shall be the string name of the instance in the module hierarchy, the **int** argument shall be the maximum number of channel instances, and the **sc_port_policy** argument shall be the port policy.

sc_port();
explicit **sc_port**(const char*);

> The constructor for class **sc_port** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sc_port_b** to set the string name of the instance in the module hierarchy, and shall pass the two template parameters N and P as arguments to the constructor for class **sc_port_b**.

The default constructor shall call function **sc_gen_unique_name("port")** to generate a unique string name that it shall then pass through to the constructor for the base class **sc_object**.

NOTE—A port instance need not be given an explicit string name within the application when it is constructed.

5.12.6 kind

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

5.12.7 Named port binding

Ports can be bound either using the functions listed in this subclause for named binding or using the **operator()** from class **sc_module** for positional binding. An implementation may defer the completion of port binding until a later time during elaboration because the port to which a port is bound may not yet itself have been bound. Such deferred port binding shall be completed by the implementation before the callbacks to function **end_of_elaboration**.

void **operator()** (IF&);
virtual void **bind**(IF&);

> Each of these two functions shall bind the port instance for which the function is called to the channel instance passed as an argument to the function. The actual argument can be an export, in which case the C++ compiler will call the implicit conversion **sc_export<IF>::operator IF&**. The implementation of **operator()** shall achieve its effect by calling the virtual member function **bind**.

void **operator()** (sc_port_b<IF>&);

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

virtual void **bind**(sc_port_b<IF>&);

Each of these two functions shall bind the port instance for which the function is called to the port instance passed as an argument to the function. The implementation of **operator()** shall achieve its effect by calling the virtual member function **bind**.

Example:

```
SC_MODULE(M) {
    // Ports
    sc_core::sc_inout<int> SC_NAMED(P), SC_NAMED(Q), SC_NAMED(R), SC_NAMED(S);
    sc_core::sc_inout<int> *T;                  // Pointer-to-port (not a recommended coding style)

    SC_CTOR(M) {
        T = new sc_core::sc_inout<int>("T");
    }
    ...
};

SC_MODULE(Top) {
    sc_core::sc_inout<int> SC_NAMED(A), SC_NAMED(B);
    sc_core::sc_signal<int> SC_NAMED(C), SC_NAMED(D), SC_NAMED(E);

    M m;                                        // Module instance

    SC_CTOR(Top)
    : m("m") {
        m.P(A);                                 // Binds P-to-A
        m.Q.bind(B);                            // Binds Q-to-B
        m.R(C);                                 // Binds R-to-C
        m.S.bind(D);                            // Binds S-to-D
        m.T->bind(E);                           // Binds T-to-E
    }
    ...
};
```

5.12.8 Member functions for bound ports and port-to-port binding

5.12.8.1 Overview

The member functions described in 5.12.8 return information about ports that have been bound during elaboration. These functions return information concerning the ordered set of channel instances to which a particular port instance (which may or may not be a multiport) is bound.

The ordered set S of channel instances to which a given port is bound (for the purpose of defining the semantics of the functions given in 5.12.8) is determined as follows:

a) When the port or export is bound to a channel instance, that channel instance shall be added to the end of the ordered set S.

b) When the port or export is bound to an export, rules a) and b) shall be applied recursively to the export.

c) When the port is bound to another port, rules a), b), and c) shall be applied recursively to the other port.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Because an implementation may defer the completion of port binding until a later time during elaboration, the number and order of the channel instances as returned from the member functions described in 5.12.8 may change during elaboration and the final order is implementation-defined, but it shall not change during the **end_of_elaboration** callback or during simulation.

NOTE—As a consequence of the above rules, a given channel instance may appear to lie at a different position in the ordered set of channel instances when viewed from ports at different positions in the module hierarchy. For example, a given channel instance may be the first channel instance to which a port of a parent module is bound but the third channel instance to which a port of a child module is bound.

5.12.8.2 size

int **size**() const;

Member function **size** shall return the number of channel instances to which the port instance for which it is called has been bound.

If member function **size** is called during elaboration and before the callback **end_of_elaboration**, the value returned is implementation-defined because the time at which port binding is completed is implementation-defined.

NOTE—The value returned by **size** will be 1 for a typical port but may be 0 if the port is unbound or greater than 1 for a multiport.

5.12.8.3 operator->

IF* **operator-> ()**;
const IF* **operator->** () const;

operator-> shall return a pointer to the first channel instance to which the port was bound during elaboration.

It shall be an error to call **operator->** for an unbound port. If **operator->** is called during elaboration and before the callback **end_of_elaboration**, the behavior is implementation-defined because the time at which port binding is completed is implementation-defined.

NOTE—**operator->** is key to the interface method call paradigm in that it permits a process to call a member function, defined in a channel, through a port bound to that channel.

Example:

```
struct iface : virtual sc_core::sc_interface {
    virtual int read() const = 0;
};

struct chan : iface, sc_core::sc_prim_channel {
    virtual int read() const;
};

int chan::read() const {...}

SC_MODULE(modu) {
    sc_port<iface> SC_NAMED(P);

    SC_CTOR(modu) {
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        SC_THREAD(thread);
    }

    void thread() {
        int i = P->read();                  // Interface method call
    }
};

SC_MODULE(top) {
    modu *mo;
    chan *ch;

    SC_CTOR(top) {
        ch = new chan;
        mo = new modu("mo");
        mo->P(*ch);                         // Port P bound to channel *ch
    }
};
```

5.12.8.4 operator[]

IF* **operator[]** (int);
const IF* **operator[]** (int) const;

operator[] shall return a pointer to a channel instance to which a port is bound. The argument identifies which channel instance shall be returned. The instances are numbered starting from zero in the order in which the port binding was completed, the order being implementation-defined.

The value of the argument shall lie in the range 0 to N – 1, where N is the number of instances to which the port is bound. It shall be an error to call **operator[]** with an argument value that lies outside this range. If **operator[]** is called during elaboration and before the callback **end_of_elaboration**, the behavior is implementation-defined because the time at which port binding is completed is implementation-defined.

operator[] may be called for a port that is not a multiport, in which case the value of the argument should be 0.

Example:

```
class bus_interface;

class peripheral_interface : virtual public sc_core::sc_interface {
 public:
    virtual void peripheral_write(int addr, int data) = 0;
    virtual void peripheral_read(int addr, int &data) = 0;
};

class bus_channel : public bus_interface, public sc_core::sc_module {
public:
    ...
    sc_core::sc_port<peripheral_interface, 0> peripheral_port;
                                                    // Multiport for attaching peripherals to bus

    SC_CTOR(bus_channel) {
        SC_THREAD(action);
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    }

private:
    void action() {
        for (int i = 0; i < peripheral_port.size(); i++) // Function size() returns number of peripherals
        peripheral_port[i]->peripheral_write(0, 0);   // Operator[] indexes peripheral port
    }
};

class memory : public peripheral_interface, public sc_core::sc_module {
public:
    virtual void peripheral_write(int addr, int data);
    virtual void peripheral_read(int addr, int &data);
    ...
};

SC_MODULE(top_level) {
    bus_channel bus;
    memory ram0, ram1, ram2, ram3;

    SC_CTOR(top_level)
    : bus("bus"), ram0("ram0"), ram1("ram1"), ram2("ram2"), ram3("ram3") {
        bus.peripheral_port(ram0);
        bus.peripheral_port(ram1);
        bus.peripheral_port(ram2);
        bus.peripheral_port(ram3);                    // One multiport bound to four memory channels
    }
};
```

5.12.8.5 get_interface

virtual sc_interface* **get_interface**();
virtual const sc_interface* **get_interface**() const;

Member function **get_interface** shall return a pointer to the first channel instance to which the port is bound. If the port is unbound, a null pointer shall be returned. This member function may be called during elaboration to test whether a port has yet been bound. Because the time at which deferred port binding is completed is implementation-defined, it is implementation-defined whether **get_interface** returns a pointer to a channel instance or a null pointer when called during construction or from the callback **before_end_of_elaboration**.

get_interface is intended for use in implementing specialized port classes derived from **sc_port**. In general, an application should call **operator->** instead. However, **get_interface** permits an application to call a member function of the class of the channel to which the port is bound, even if such a function is not a member of the interface type of the port.

NOTE—Function **get_interface** cannot return channels beyond the first channel instance to which a multiport is bound; use **operator[]** instead.

Example:

```
SC_MODULE(Top) {
    sc_core::sc_in<bool> clock;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    void before_end_of_elaboration() {
        using namespace sc_core;
        sc_interface* i_f = clock.get_interface();
        sc_clock* clk = dynamic_cast<sc_clock*>(i_f);
        sc_time t = clk->period();                    // Call method of clock object to which port is bound
        ...
    }
    ...
};
```

5.12.8.6 get_interface_type

Member function **get_interface_type** shall return the type of the *interface proper* of the port.

5.12.9 before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation

See 4.5.

5.13 sc_export

5.13.1 Description

Class **sc_export** allows a module to provide an interface to its parent module. An export forwards interface method calls to the channel to which the export is bound. An export defines a set of services (as identified by the type of the export) that are provided by the module containing the export.

Providing an interface through an export is an alternative to a module simply implementing the interface. The use of an explicit export allows a single module instance to provide multiple interfaces in a structured manner.

If a module is to call a member function belonging to a channel instance within a child module, that call should be made through an export of the child module.

5.13.2 Class definition

```
namespace sc_core {

class sc_export_base
: public sc_object {
    public:
        virtual sc_interface* get_interface() = 0;
        virtual const sc_interface* get_interface() const = 0;
        virtual std::type_index get_interface_type() const = 0;

    protected:
        sc_export_base();
        sc_export_base( const char* );
        virtual ~sc_export_base();
};

template<class IF>
class sc_export
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
: public sc_export_base
{
    public:
        sc_export();
        explicit sc_export( const char*  );
        virtual ~sc_export();

        virtual const char* kind() const;

        void operator() ( IF& );
        virtual void bind( IF& );
        operator IF& ();
        operator const IF& () const;

        IF* operator-> ();
        const IF* operator-> () const;

        virtual sc_interface* get_interface();
        virtual const sc_interface* get_interface() const;
        virtual std::type_index get_interface_type() const;

    protected:
        virtual void before_end_of_elaboration();
        virtual void end_of_elaboration();
        virtual void start_of_simulation();
        virtual void end_of_simulation();

    private
        // Disabled
        sc_export( const sc_export<IF>& );
        sc_export<IF>& operator= ( const sc_export<IF>& );
};

}       // namespace sc_core
```

5.13.3 Template parameters

The argument to template **sc_export** shall be the name of an interface proper. This interface is said to be the *type of the export*. An export can only be bound to a channel derived from the type of the export or to another export with a type derived from the type of the export.

NOTE—An export may be bound indirectly to a channel by being bound to another export (see 4.2.4).

5.13.4 Constraints on usage

An implementation shall derive class **sc_export_base** from class **sc_object**.

Exports shall only be instantiated during elaboration and only from within a module. It shall be an error to instantiate an export other than within a module. It shall be an error to instantiate an export during simulation.

Every export of every module instance shall be bound once and once only during elaboration. It shall be an error to have an export remaining unbound at the end of elaboration. It shall be an error to bind an export to more than one channel.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The member function **get_interface** can be called during elaboration or simulation, whereas **operator->** should only be called during simulation.

It is strongly recommended that an export within a given module be bound within that same module. Furthermore, it is strongly recommended that the export be bound to a channel that is itself instantiated within the current module or implemented by the current module or bound to an export that is instantiated within a child module. Any other usage would result in a breakdown of the normal discipline of the module hierarchy and is strongly discouraged (see 5.12.4).

5.13.5 Constructors

sc_export();
explicit **sc_export**(const char*);

The constructor for class **sc_export** shall pass the character string argument (if there is one) through to the constructor belonging to the base class **sc_object** in order to set the string name of the instance in the module hierarchy.

The default constructor shall call function **sc_gen_unique_name("export")** in order to generate a unique string name that it shall then pass through to the constructor for the base class **sc_object**.

NOTE—An export instance need not be given an explicit string name within the application when it is constructed.

5.13.6 kind

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

5.13.7 Export binding

Exports can be bound using either of the two functions defined here. The notion of positional binding is not applicable to exports. Each of these functions shall bind the export immediately, in contrast to ports for which the implementation may need to defer the binding.

void **operator()** (IF&);
virtual void **bind**(IF&);

Each of these two functions shall bind the export instance for which the function is called to the channel instance passed as an argument to the function. The implementation of **operator()** shall achieve its effect by calling the virtual member function **bind**.

NOTE—The actual argument could be an export, in which case **operator IF&** would be called as an implicit conversion.

Example:

```
struct i_f : virtual sc_core::sc_interface {
    virtual void print() = 0;
};

struct Chan : sc_core::sc_channel, i_f {
    SC_CTOR(Chan) {}
    void print() { std::cout << "I'm Chan, name=" << name() << std::endl; }
};

struct Caller : sc_core::sc_module {
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    sc_core::sc_port<i_f> p;
    ...
    SC_CTOR(Caller) { ... }
};

struct Bottom : sc_core::sc_module {
    sc_core::sc_export<i_f> xp;
    Chan ch;
    SC_CTOR(Bottom)
    : xp("xp"), ch("ch") {
        xp.bind(ch);                    // Bind export xp to channel ch
    }
};

struct Middle : sc_core::sc_module {
    sc_core::sc_export<i_f> xp;
    Bottom *b;
    SC_CTOR(Middle)
    : xp("xp") {
        b = new Bottom("b");
        xp.bind(b->xp);                 // Bind export xp to export b->xp
        b->xp->print();                 // Call method of export within child module
    }
};

struct Top : sc_core::sc_module {
    Caller *c;
    Middle *m;
    SC_CTOR(Top) {
        c = new Caller("c");
        m = new Middle("m");
        c->p(m->xp);                    // Bind port c->p to export m->xp
    }
};
```

5.13.8 Member functions for bound exports and export-to-export binding

5.13.8.1 Overview

The member functions described in 5.13.8 return information about exports that have been bound during elaboration, and hence, these member functions should only be called after the export has been bound during elaboration or simulation. These functions return information concerning the channel instance to which a particular export instance has been bound.

It shall be an error to bind an export more than once. It shall be an error for an export to be unbound at the end of elaboration.

The channel instance to which a given export is bound (for the purpose of defining the semantics of the functions given in 5.13.8) is determined as follows:

a) If the export is bound to a channel instance, that is the channel instance in question.
b) If the export is bound to another export, rules a) and b) shall be applied recursively to the other export.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

5.13.8.2 operator-> and operator IF&

IF* **operator->** ();
const IF* **operator->** () const;
operator IF& ();
operator const IF& () const;

operator-> and **operator IF&** shall both return a pointer to the channel instance to which the export was bound during elaboration.

It shall be an error for an application to call this operator if the export is unbound.

NOTE 1—**operator->** is intended for use during simulation when making an interface method call through an export instance from a parent module of the module containing the export.

NOTE 2—**operator IF&** is intended for use during elaboration as an implicit conversion when passing an object of class **sc_export** in a context that requires an **sc_interface**, for example, when binding a port to an export or when adding an export to the static sensitivity of a process.

NOTE 3—There is no **operator[]** for class **sc_export**, and there is no notion of a multi-export. Each export can only be bound to a single channel.

5.13.8.3 get_interface

virtual sc_interface* **get_interface**();
virtual const sc_interface* **get_interface**() const;

Member function **get_interface** shall return a pointer to the channel instance to which the export is bound. If the export is unbound, a null pointer shall be returned. This member function may be called during elaboration to test whether an export has yet been bound.

5.13.8.4 get_interface_type

Member function **get_interface_type** shall return the type of the *interface proper* of the export.

5.13.9 before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation

See 4.4.

5.14 sc_interface

5.14.1 Description

sc_interface is the abstract base class for all interfaces.

An *interface* is a class derived from the class **sc_interface**. An *interface proper* is an abstract class derived from class **sc_interface** but not derived from class **sc_object**. An interface proper contains a set of pure virtual functions that shall be defined in one or more channels derived from that interface proper. Such a channel is said to *implement* the interface.

NOTE 1—The term *interface proper* is used to distinguish an interface proper from a channel. A channel is a class derived indirectly from class **sc_interface**, and in that sense, a channel is an interface. However, a channel is not an interface proper.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

NOTE 2—As a consequence of the rules of C++, an instance of a channel derived from an interface **IF** or a pointer to such an instance can be passed as the argument to a function with a parameter of type **IF&** or **IF***, respectively, or a port of type **IF** can be bound to such a channel.

5.14.2 Class definition

```
namespace sc_core {

class sc_interface
{
    public:
        virtual void register_port( sc_port_base& , const char* );
        virtual const sc_event& default_event() const;
        virtual ~sc_interface();

    protected:
        sc_interface();

    private:
        // Disabled
        sc_interface( const sc_interface& );
        sc_interface& operator= ( const sc_interface& );
};

}       // namespace sc_core
```

5.14.3 Constraints on usage

An application should not use class **sc_interface** as the direct base class for any class other than an interface proper.

An interface proper shall obey the following rules:

- It shall be publicly derived directly or indirectly from class **sc_interface**.
- If directly derived from class **sc_interface**, it shall use the **virtual** specifier.
- It shall not be derived directly or indirectly from class **sc_object**.

An interface proper should typically obey the following rules:

- It should contain one or more pure virtual functions.
- It should not be derived from any other class that is not itself an interface proper.
- It should not contain any function declarations or function definitions apart from the pure virtual functions.
- It should not contain any data members.

NOTE 1—An interface proper may be derived from another interface proper or from two or more other interfaces proper, thus creating a multiple inheritance hierarchy.

NOTE 2—A channel class may be derived from any number of interfaces proper.

5.14.4 register_port

```
virtual void register_port( sc_port_base& , const char* );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The definition of this function in class **sc_interface** does nothing. An application may override this function in a channel.

The purpose of function **register_port** is to enable an application to perform actions that depend on port binding during elaboration, such as checking connectivity errors.

Member function **register_port** of a channel shall be called by the implementation whenever a port is bound to a channel instance. The first argument shall be a reference to the port instance being bound. The second argument shall be the value returned from the expression **typeid(IF).name()**, where **IF** is the interface type of the port.

Member function **register_port** shall not be called when an export is bound to a channel.

If a port P is bound to another port Q, and port Q is in turn bound to a channel instance, the first argument to member function **register_port** shall be the port P. In other words, **register_port** is *not* passed a reference to a port on a parent module if a port on a child module is in turn bound to that port; instead, it is passed as a reference to the port on the child module, and so on recursively down the module hierarchy.

In the case that multiple ports are bound to the same single channel instance or port instance, member function **register_port** shall be called once for each port so bound.

Example:

```
void register_port( sc_core::sc_port_base& port_, const char* if_typename_ ) {
    std::string nm( if_typename_ );
    if( nm == typeid( my_interface ).name() )
    std::cout << " channel " << name() << " bound to port " << port_.name() << std::endl;
}
```

5.14.5 default_event

virtual const sc_event& **default_event**() const;

Member function **default_event** shall be called by the implementation in every case where a port or channel instance is used to define the static sensitivity of a process instance by being passed directly as an argument to **operator<<** of **class** *sc_sensitive*[†]. In such a case, the application shall override this function in the channel in question to return a reference to an event to which the process instance will be made sensitive.

If this function is called by the implementation but not overridden by the application, the implementation may generate a warning.

Example:

```
struct my_if : virtual sc_core::sc_interface {
    virtual int read() = 0;
};

class my_ch : public my_if, public sc_core::sc_module {
public:
    virtual int read() { return m_val; }
    virtual const sc_core::sc_event& default_event() const { return m_ev; }

private:
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    int m_val;
    sc_core::sc_event m_ev;
    ...
};
```

5.15 sc_prim_channel

5.15.1 Description

sc_prim_channel is the base class for all primitive channels and provides such channels with unique access to the update phase of the scheduler. In common with hierarchical channels, a primitive channel may provide public member functions that can be called using the interface method call paradigm.

This standard provides a number of predefined primitive channels to model common communication mechanisms (see Clause 6).

5.15.2 Class definition

```
namespace sc_core {

class sc_prim_channel
: public sc_object
{
    public:
        virtual const char* kind() const;

    protected:
        sc_prim_channel();
        explicit sc_prim_channel( const char* );
        virtual ~sc_prim_channel();

        void request_update();
        void async_request_update();
        void async_attach_suspending();
        void async_detach_suspending();

        virtual void update();

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

        void wait();
        void wait( int );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

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

    private:
        // Disabled
        sc_prim_channel( const sc_prim_channel& );
        sc_prim_channel& operator= ( const sc_prim_channel& );
};

}       // namespace sc_core
```

5.15.3 Constraints on usage

Objects of class **sc_prim_channel** can only be constructed during elaboration. It shall be an error to instantiate a primitive channel during simulation.

A primitive channel should be *publicly* derived from class **sc_prim_channel**.

A primitive channel shall implement one or more interfaces.

Member functions **request_update** and **async_request_update** may be called during elaboration or simulation. If called during elaboration, the update request shall be executed during the initialization phase.

NOTE—Because the constructors are protected, class **sc_prim_channel** cannot be instantiated directly but may be used as a base class for a primitive channel.

5.15.4 Constructors, destructor, and hierarchical names

sc_prim_channel();
explicit **sc_prim_channel**(const char*);

The constructor for class **sc_prim_channel** shall pass the character string argument (if such argument exists) through to the constructor belonging to the base class **sc_object** to set the string name of the instance in the module hierarchy.

NOTE—A class derived from class **sc_prim_channel** is not obliged to have a constructor, in which case the default constructor for class **sc_object** will generate a unique string name. As a consequence, a primitive channel instance need not be given an explicit string name within the application when it is constructed.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

5.15.5 kind

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

5.15.6 request_update and update

Member functions **request_update** and **async_request_update** each cause the scheduler to queue an update request. An application should not call both **request_update** and **async_request_update** for a given primitive channel instance at any time during elaboration or simulation, but it may do so for different primitive channel instances. If both **request_update** and **async_request_update** are called for the same primitive channel instance, the behavior of the implementation is undefined.

void **request_update**();

Member function **request_update** shall cause the scheduler to queue an update request for the current primitive channel (see 4.3.2.4). No more than one update request shall be queued for any given primitive channel instance in any given update phase; that is, multiple calls to **request_update** in the same evaluation phase shall result in a single update request.

void **async_request_update**();

Member function **async_request_update** shall cause the scheduler to queue an update request for the current primitive channel in a thread-safe manner with respect to the host operating system. The intent of **async_request_update** is that it may be called reliably from an operating system thread other than those in which the SystemC kernel and any SystemC thread processes are executing.

An application may call **async_request_update** at any time during elaboration or simulation, and these calls may be asynchronous with respect to the SystemC kernel. The implementation shall guarantee that each call to **async_request_update** behaves as if it were executed in an evaluation phase. No more than one update request shall be queued for any given primitive channel instance in any given update phase; that is, multiple calls to **async_request_update** received by the kernel between two consecutive update phases shall result in a single update request. The precise phase in which any given call to **async_request_update** is received and processed by the kernel is undefined. As a consequence, two calls to **async_request_update** for the same primitive channel and occurring within a narrow time window may result in a single update request, two update requests in consecutive delta cycles, or two update requests in non-consecutive delta cycles.

The application is responsible for synchronizing access to any shared memory across calls to **async_request_update** and **update**. The software thread that calls **async_request_update** would typically need to write a value to some variable that is read by function **update**. The implementation can give no guarantee with regard to the integrity of any such shared memory.

It is not recommended to call member function **async_request_update** from functions executed in the context of the SystemC kernel, such as SystemC thread or method processes, but only by operating system threads separate from the kernel. Calling **async_request_update** may cause a performance degradation compared to **request_update**.

virtual void **update**();

Member function **update** shall be called back by the scheduler during the update phase in response to a call to **request_update** or **async_request_update**. An application may override this member function in a primitive channel. The definition of this function in class **sc_prim_channel** itself does nothing.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

When overridden in a derived class, member function **update** shall not perform any of the following actions:

a) Call any member function of class **sc_prim_channel** with the exception of member function **update** itself if overridden within a base class of the current object.
b) Call member function **notify()** of class **sc_event** with no arguments to create an immediate notification.
c) Call any of the member functions of class **sc_process_handle** for process control (**suspend** or **kill**, for example).

If the application violates the three rules just given, the behavior of the implementation shall be undefined.

Member function **update** should not change the state of any storage except for data members of the current object. Doing so may result in non-deterministic behavior.

Member function **update** should not read the state of any primitive channel instance other than the current object. Doing so may result in non-deterministic behavior.

Member function **update** should not call interface methods of other channel instances. In particular, member function **update** should not write to any signals.

Member function **update** may call function **sc_spawn** to create a dynamic process instance. Such a process shall not become runnable until the next evaluation phase.

NOTE 1—The purpose of the member functions **request_update** and **update** is to permit simultaneous requests to a channel made during the evaluation phase to be resolved or arbitrated during the update phase. The nature of the arbitration is the responsibility of the application; for example, the behavior of member function **update** may be deterministic or random.

NOTE 2—**update** will typically only read and modify data members of the current object and create delta notifications.

5.15.7 async_attach_suspending and async_detach_suspending

void **async_attach_suspending**();

Member function **async_attach_suspending** shall cause the primitive channel to attach itself to request suspension. When at least one primitive channel has attached itself to request suspension, instead of ending the simulation when running out of internal events, the simulation is suspended until any external call to member function **async_request_update**.

void **async_detach_suspending**();

Member function **async_detach_suspending** shall cause the primitive channel to detach itself from previously requested suspension. If there is no more channel attached to request suspension, the simulation will end when running out of internal events.

NOTE—One possible use case for this feature is for a primitive channel waiting from stimuli external to the simulator to prevent exit of the simulation when running out of internal events.

5.15.8 next_trigger and wait

The behavior of the member functions **wait** and **next_trigger** of class **sc_prim_channel** shall be identical to that of the member functions of class **sc_module** with the same function names and signatures. Aside from the fact that they are members of different classes and so have different scopes, the restrictions concerning

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

the context in which the member functions may be called is also identical. For example, the member function **next_trigger** shall only be called from a method process.

5.15.9 before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation

See 4.5.

Example:

```
struct my_if : virtual sc_core::sc_interface {          // An interface proper
    virtual int read() = 0;
    virtual void write(int) = 0;
};

struct my_prim : sc_core::sc_prim_channel, my_if {          // A primitive channel
    my_prim() // Default constructor
    : sc_core::sc_prim_channel(sc_core::sc_gen_unique_name("my_prim")),
        m_req(false),
        m_written(false),
        m_cur_val(0) {}

    virtual void write(int val) {
        if (!m_req) {                                   // Only keeps the 1st value written in any one delta
            m_new_val = val;
            request_update();                           // Schedules an update request
            m_req = true;
        }
    }

    virtual void update() {                             // Called back by the scheduler in the update phase
        m_cur_val = m_new_val;
        m_req = false;
        m_written = true;
        m_write_event.notify(SC_ZERO_TIME);   // A delta notification
    }

    virtual int read() {
        if (!m_written)
            wait(m_write_event);                        // Blocked until update() is called
        m_written = false;
        return m_cur_val;
    }

    bool m_req, m_written;
    sc_core::sc_event m_write_event;
    int m_new_val, m_cur_val;
};
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

5.16 sc_object

5.16.1 Description

Class **sc_object** is the common base class for classes **sc_module**, **sc_port**, **sc_export**, and **sc_prim_channel**, and for the implementation-defined classes associated with spawned and unspawned process instances. The set of **sc_object**s shall be organized into an *object hierarchy*, where each **sc_object** has no more than one parent but may have multiple siblings and multiple children. Only module objects and process objects can have children.

An **sc_object** is a *child* of a module instance if and only if that object lies *within* the module instance, as defined in 3.1.4. An **sc_object** is a *child* of a process instance if and only if that object was created during the execution of the function associated with that process instance. Object P is a *parent* of object C if and only if C is a child of P.

An **sc_object** that has no parent object is said to be a *top-level* object. Module instances, spawned process instances, and objects of an application-defined class derived from class **sc_object** may be top-level objects.

Each call to function **sc_spawn** shall create a spawned process instance that is either a child of the caller or a top-level object. The parent of the spawned process instance so created may be another spawned process instance, an unspawned process instance, or a module instance. Alternatively, the spawned process instance may be a top-level object.

Each **sc_object** shall have a unique hierarchical name reflecting its position in the object hierarchy.

Except where explicitly forbidden (as is the case for the classes **sc_module**, **sc_port**, **sc_export**, and **sc_prim_channel**), objects of a class derived from **sc_object** may be deleted at any time. When such an **sc_object** is deleted, it ceases to be a child. The object associated with a process instance shall not be deleted while the process has surviving children, but it may be deleted by the implementation once all its child objects have been deleted.

Attributes may be added to each **sc_object**.

NOTE—An implementation may permit multiple top-level **sc_object**s (see 4.4).

5.16.2 Class definition

```
namespace sc_core {

class sc_object
{
    public:
        virtual ~sc_object();

        const char* name() const;
        const char* basename() const;
        virtual const char* kind() const;

        virtual void print( std::ostream& = std::cout ) const;
        virtual void dump( std::ostream& = std::cout ) const;

        virtual const std::vector<sc_object*>& get_child_objects() const;
        virtual const std::vector<sc_event*>& get_child_events() const;
        sc_object* get_parent_object() const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        bool add_attribute( sc_attr_base& );
        sc_attr_base* get_attribute( const std::string& );
        const sc_attr_base* get_attribute( const std::string& ) const;
        sc_attr_base* remove_attribute( const std::string& );
        void remove_all_attributes();
        int num_attributes() const;
        sc_attr_cltn& attr_cltn();
        const sc_attr_cltn& attr_cltn() const;

    protected:
        sc_object();
        sc_object(const char*);
        sc_object( const sc_object& );
        sc_object& operator= ( const sc_object& );

        virtual sc_hierarchy_scope get_hierarchy_scope();
};

sc_object* get_current_sc_object();

const std::vector<sc_object*>& sc_get_top_level_objects();
sc_object* sc_find_object( const char* );

}       // namespace sc_core
```

5.16.3 Constraints on usage

An application may use class **sc_object** as a base class for other classes besides modules, ports, exports, primitive channels, and processes. An application may access the hierarchical name of such an object or may add attributes to such an object.

An application shall not define a class that has two or more base class sub-objects of class **sc_object**.

Objects of class **sc_object** may be instantiated during elaboration or may be instantiated during simulation. However, modules, ports, exports, and primitive channels can only be instantiated during elaboration. It is permitted to create a channel that is neither a hierarchical channel nor a primitive channel but is nonetheless derived from class **sc_object**, and to instantiate such a channel either during elaboration or simulation. Portless channel access is permitted for any channel, but a port or export cannot be bound to a channel that is instantiated during simulation.

NOTE 1—Because the constructors are protected, class **sc_object** cannot be instantiated directly.

NOTE 2—Since the classes having **sc_object** as a direct base class (that is, **sc_module**, **sc_port**, **sc_export**, and **sc_prim_channel**) have class **sc_object** as a non-virtual base class, any class derived from these classes can have at most one direct base class derived from class **sc_object**. In other words, multiple inheritance from the classes derived from class **sc_object** is not permitted.

5.16.4 Constructors and destructor

sc_object();
sc_object(const char*);

Both constructors shall register the **sc_object** as part of the object hierarchy and shall construct a hierarchical name for the object using the string name passed as an argument. Calling the constructor **sc_object(const char*)** with an empty string shall have the same behavior as the default constructor; that is, the string name shall be set to **"object"**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The rules for composing a hierarchical name are given in 5.17.

If the constructor needs to substitute a new string name in place of the original string name as the result of a name clash, the constructor shall generate a single warning.

sc_object(const sc_object& arg);

The copy constructor shall create a new **sc_object** as if it had been created by the call **sc_object(arg.basename())**. In other words, the new object shall be a child of a module instance if created from the constructor of that module or a child of a process instance if created from the function associated with that process. The string name of the existing object shall be used as the seed for the string name of the new object. Attributes or children of the existing object shall not be copied to the new object.

sc_object& **operator=** (const sc_object&);

The assignment operator shall not modify the hierarchical name or the parent of the destination object in the object hierarchy. In other words, the destination object shall retain its current position in the object hierarchy. Attributes and children of the destination object shall not be modified. **operator=** shall return a reference to ***this**.

virtual **~sc_object**();

The destructor shall delete the object, shall delete any attribute collection attached to the object, and shall remove the object from the object hierarchy such that it is no longer a child.

NOTE—If an implementation were to create internal objects of class **sc_object**, the implementation would be obliged by the rules of this subclause to exclude those objects from the object hierarchy and from the namespace of hierarchical names. This would necessitate an extension to the semantics of class **sc_object**, and the implementation would be obliged to make such an extension transparent to the application.

5.16.5 name, basename, and kind

const char* **name**() const;

Member function **name** shall return the *hierarchical name* of the **sc_object** instance in the object hierarchy.

const char* **basename**() const;

Member function **basename** shall return the string name of the **sc_object** instance. This is the string name created when the **sc_object** instance was constructed.

virtual const char* **kind**() const;

Member function **kind** returns a character string identifying the *kind* of the **sc_object**. Member function **kind** of class **sc_object** shall return the string **"sc_object"**. Every class that is part of the implementation and that is derived from class **sc_object** shall override member function **kind** to return an appropriate string.

Example:

```
#include <systemc>

SC_MODULE(Mod) {
    sc_core::sc_port<sc_core::sc_signal_in_if<int> > p;

    SC_CTOR(Mod)                    // p.name() returns "top.mod.p"
    : p("p") {}                     // p.basename() returns "p"
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
// p.kind() returns "sc_port"
};

SC_MODULE(Top) {
    Mod *mod;                           // mod->name() returns "top.mod"
    sc_core::sc_signal<int> sig;        // sig.name() returns "top.sig"
    SC_CTOR(Top)
    : sig("sig") {
        mod = new Mod("mod");
        mod->p(sig);
    }
};

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

5.16.6 print and dump

virtual void **print**(std::ostream& = std::cout) const;

Member function **print** shall print the character string returned by member function **name** to the stream passed as an argument. No additional characters shall be printed.

virtual void **dump**(std::ostream& = std::cout) const;

Member function **dump** shall print at least the name and the kind of the **sc_object** to the stream passed as an argument. The formatting shall be implementation-dependent. The purpose of **dump** is to allow an implementation to print diagnostic information to help the user debug an application.

5.16.7 Functions for object hierarchy traversal

The following functions in this subclause return information that supports the traversal of the object hierarchy. An implementation shall allow each of these functions to be called at any stage during elaboration or simulation. If called before elaboration is complete, they shall return information concerning the partially constructed object hierarchy as it exists at the time the functions are called. In other words, a function shall return pointers to any objects that have been constructed before the time the function is called but will exclude any objects constructed after the function is called.

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

Member function **get_child_objects** shall return a **std::vector** containing a pointer to every instance of class **sc_object** that is a child of the current **sc_object** in the object hierarchy. The virtual function **sc_object::get_child_objects** shall return an empty vector but shall be overridden by the implementation in those classes derived from class **sc_object** that do have children, that is, class **sc_module** and the implementation-defined classes associated with spawned and unspawned process instances.

virtual const std::vector<sc_event*>& **get_child_events**() const;

Member function **get_child_events** shall return a **std::vector** containing a pointer to every object of type **sc_event** that is a hierarchically named event and whose parent is the current object. The virtual function **sc_object::get_child_events** shall return an empty vector but shall be overridden by the implementation in those classes derived from class **sc_object** that do have children, that is, class

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

sc_module and the implementation-defined classes associated with spawned and unspawned process instances.

sc_object* **get_parent_object()** const;

Member function **get_parent_object** shall return a pointer to the **sc_object** that is the parent of the current object in the object hierarchy. If the current object is a top-level object, member function **get_parent_object** shall return the null pointer. If the parent object is a process instance and that process has terminated, **get_parent_object** shall return a pointer to that process instance. A process instance shall not be deleted (nor any associated process handles invalidated) while the process has surviving children, but it may be deleted once all its child objects have been deleted.

sc_object* **get_current_sc_object**();

Member function **get_current_sc_object** may be used when the current **sc_object** hierarchy context needs to be known. It shall return the current parent module (during elaboration), or the currently active process (during simulation), or null pointer (when outside the simulator context).

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

Function **sc_get_top_level_objects** shall return a **std::vector** containing pointers to all of the top-level **sc_object**s.

sc_object* **sc_find_object**(const char*);

Function **sc_find_object** shall return a pointer to the **sc_object** that has a hierarchical name that exactly matches the value of the string argument or shall return the null pointer if there is no **sc_object** having a matching name.

Examples:

```
void scan_hierarchy(sc_core::sc_object *obj) {              // Traverse the entire object subhierarchy
                                                            // below a given object
    std::vector<sc_core::sc_object *> children = obj->get_child_objects();
    for (unsigned i = 0; i < children.size(); i++)
        if (children[i])
            scan_hierarchy(children[i]);
}

std::vector<sc_core::sc_object *> tops = sc_core::sc_get_top_level_objects();
for (unsigned i = 0; i < tops.size(); i++)
    if (tops[i])
        scan_hierarchy(tops[i]);                            // Traverse the object hierarchy below
// each top-level object

sc_core::sc_object *obj = sc_core::sc_find_object("foo.foobar");
                                                            // Find an object given its hierarchical name

sc_core::sc_module *m;
if ( (m = dynamic_cast<sc_core::sc_module *>(obj)) )        // Test whether the given object is a module
    ...                                                     // The given object is a module

sc_core::sc_object *parent = obj->get_parent_object();      // Get the parent of the given object
if (parent)                                                 // parent is a null pointer for a top-level object
    std::cout << parent->name() << " " << parent->kind();   // Print the name and kind
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

5.16.8 Member functions for attributes

bool **add_attribute**(sc_attr_base&);

Member function **add_attribute** shall attempt to attach to the object of class **sc_object** the attribute passed as an argument. If an attribute having the same name as the new attribute is already attached to this object, member function **add_attribute** shall not attach the new attribute and shall return the value **false**. Otherwise, member function **add_attribute** shall attach the new attribute and shall return the value **true**. The argument should be an object of class **sc_attribute**, not **sc_attr_base**.

The lifetime of an attribute shall extend until the attribute has been completely removed from all objects. If an application deletes an attribute that is still attached to an object, the behavior of the implementation shall be undefined.

sc_attr_base* **get_attribute**(const std::string&);
const sc_attr_base* **get_attribute**(const std::string&) const;

Member function **get_attribute** shall attempt to retrieve from the object of class **sc_object** an attribute having the name passed as an argument. If an attribute with the given name is attached to this object, member function **get_attribute** shall return a pointer to that attribute. Otherwise, member function **get_attribute** shall return the null pointer.

sc_attr_base* **remove_attribute**(const std::string&);

Member function **remove_attribute** shall attempt to remove from the object of class **sc_object** an attribute having the name passed as an argument. If an attribute with the given name is attached to this object, member function **remove_attribute** shall return a pointer to that attribute and remove the attribute from this object. Otherwise, member function **remove_attribute** shall return the null pointer.

void **remove_all_attributes**();

Member function **remove_all_attributes** shall remove all attributes from the object of class **sc_object**.

int **num_attributes**() const;

Member function **num_attributes** shall return the number of attributes attached to the object of class **sc_object**.

sc_attr_cltn& **attr_cltn**();
const sc_attr_cltn& **attr_cltn**() const;

Member function **attr_cltn** shall return the collection of attributes attached to the object of class **sc_object** (see 5.20).

NOTE—A pointer returned from function **get_attribute** needs to be cast to type **sc_attribute<T>*** in order to access data member **value** of class **sc_attribute**.

Example:

```
{
    using namespace sc_core;
    sc_signal<int> sig;
    ...

    // Add an attribute to an sc_object
    sc_attribute<int> a("number", 1);
    sig.add_attribute(a);
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    // Retrieve the attribute by name and modify the value
    sc_attribute<int>* ap;
    ap = (sc_attribute<int>*)sig.get_attribute("number");
    ++ ap->value;
}
```

5.16.9 get_hierarchy_scope

Member function **get_hierarchy_scope** shall return an object of type **sc_hierarchy_scope** that shall contain the object's hierarchical scope (see 5.21).

Example:

```
SC_MODULE(SignalStub) {
    SC_CTOR(SignalStub) {}

    template<typename T>
    void stub( sc_core::sc_in<T>& port ) {
        // force current SystemC hierarchy to point to current module
        sc_core::sc_hierarchy_scope scope = get_hierarchy_scope();
        sc_core::sc_signal<T>* sig = new sc_core::sc_signal<T>( sc_core::sc_gen_unique_name(
port.basename() ) );
        port.bind( *sig );
    }
};

SC_MODULE(SomeModule) {
    sc_core::sc_in<int>& in;
    SC_CTOR(SomeModule) { ... }
    ...
};

int sc_main(int, char*[]) {
    SignalStub SC_NAMED(sigStub);
    SomeModule SC_NAMED(top);
    sigStub.stub( top.in );
    ...
}
```

5.17 Hierarchical naming of objects and events

5.17.1 Overview

A hierarchical name shall be composed of a set of string names separated by the period character '.', starting with the string name of a top-level **sc_object** instance and including the string name of each module instance or process instance descending down through the object hierarchy until the current **sc_object** or **sc_event** is reached. The hierarchical name shall end with the string name of the **sc_object** or **sc_event** itself.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Hierarchical names are case-sensitive.

It shall be an error if a string name includes the period character (.) or any white-space characters. It is strongly recommended that an application limit the character set of a string name to the following:

a) Lowercase letters a–z
b) Uppercase letters A–Z
c) Decimal digits 0–9
d) Underscore character _

An implementation may generate a warning if a string name contains characters outside this set but is not obliged to do so.

There shall be a single global namespace for hierarchical names. Each **sc_object** and each hierarchically named **sc_event** shall have a unique non-empty hierarchical name. An implementation shall not add any names to this namespace other than the hierarchical names of **sc_object**s explicitly constructed by an application and the hierarchical names of hierarchically named events.

The constructor shall build a hierarchical name from the string name (either passed in as an argument or the default name **"object"** for an **sc_object** or **"event"** for an **sc_event**) and test whether that hierarchical name is unique. If it is unique, that hierarchical name shall become the hierarchical name of the object. If not, the constructor shall call function **sc_gen_unique_name**, passing the string name as a seed. It shall use the value returned as a replacement for the string name and shall repeat this process until a unique hierarchical name is generated.

If function **sc_gen_unique_name** is called more than once in the course of constructing any given object, the choice of seed passed to **sc_gen_unique_name** on the second and subsequent calls shall be implementation-defined but shall in any case be either the string name passed as the seed on the first such call or shall be one of the string names returned from **sc_gen_unique_name** in the course of constructing the given object. In other words, the final string name shall have the original string name as a prefix.

5.17.2 sc_hierarchical_name_exists

This clause describes the rules that determine the hierarchical naming of objects of type **sc_object** and of hierarchically named events. The naming of events that are not hierarchically named is described in 5.10.4.

```
namespace sc_core {

    bool sc_hierarchical_name_exists( const char* );

} // namespace sc_core
```

Function **sc_hierarchical_name_exists** shall return the value **true** if and only if the value of the string argument exactly matches the hierarchical name of an **sc_object** or of a hierarchically named event. If the value of the string argument exactly matches an implementation-defined event name, function **sc_hierarchical_name_exists** shall return the value **false** unless the string argument also matches a hierarchical name.

5.17.3 sc_register_hierarchical_name, sc_unregister_hierarchical_name

```
namespace sc_core {

    bool sc_register_hierarchical_name(std::string_view name);
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    bool sc_register_hierarchical_name(const sc_object* parent, std::string_view name);
    bool sc_unregister_hierarchical_name(std::string_view name);
    bool sc_unregister_hierarchical_name(const sc_object* parent, std::string_view name);

}
```

The function **sc_register_hierarchical_name** shall reserve a hierarchical name to avoid its use when creating new objects or events. In other words, the name registered is taken into account by **sc_gen_unique_name** and the constructor of **sc_object**. These functions are intended to avoid clashes with names that are outside of the SystemC object and event hierarchy.

The function **sc_unregister_hierarchical_name** shall cancel the effect of **sc_register_hierarchical_name**, i.e., freeing that name for other uses. The function shall only have an effect if the name was previously reserved using **sc_register_hierarchical_name**.

The variant with two parameters allows specifying a parent and an instance name (which is equivalent to the hierarchical name consisting of the parent's hierarchical name followed by the second parameter).

5.18 sc_attr_base

5.18.1 Description

Class **sc_attr_base** is the base class for attributes, storing only the name of the attribute. The name is used as a key when retrieving an attribute from an object. Every attribute attached to a specific object shall have a unique name but two or more attributes with identical names may be attached to distinct objects.

5.18.2 Class definition

```
namespace sc_core {

class sc_attr_base
{
    public:
        sc_attr_base( const std::string& );
        sc_attr_base( const sc_attr_base& );
        virtual ~sc_attr_base();

        const std::string& name() const;

    private:
        // Disabled
        sc_attr_base();
        sc_attr_base& operator= ( const sc_attr_base& );
};

}       // namespace sc_core
```

5.18.3 Member functions

The constructors for class **sc_attr_base** shall set the name of the attribute to the string passed as an argument to the constructor.

Member function **name** shall return the name of the attribute.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

5.19 sc_attribute

5.19.1 Description

Class **sc_attribute** stores the value of an attribute. It is derived from class **sc_attr_base**, which stores the name of the attribute. An attribute can be attached to an object of class **sc_object**.

5.19.2 Class definition

```
namespace sc_core {

template <class T>
class sc_attribute
: public sc_attr_base
{
    public:
        sc_attribute( const std::string& );
        sc_attribute( const std::string&, const T& );
        sc_attribute( const sc_attribute<T>& );
        virtual ~sc_attribute();
        T value;

    private:
        // Disabled
        sc_attribute();
        sc_attribute<T>& operator= ( const sc_attribute<T>& );
};

}       // namespace sc_core
```

5.19.3 Template parameters

The argument passed to template **sc_attribute** shall be of a *copy-constructible* type.

5.19.4 Member functions and data members

The constructors shall set the name and value of the attribute using the name (of type **std::string**) and value (of type **T**) passed as arguments to the constructor. If no value is passed to the constructor, the default constructor (of type **T**) shall be called to construct the value.

Data member **value** is the value of the attribute. An application may read or assign this public data member.

5.20 sc_attr_cltn

5.20.1 Description

Class **sc_attr_cltn** is a container class for attributes, as used in the implementation of class **sc_object**. It provides iterators for traversing all of the attributes in an attribute collection.

5.20.2 Class definition

```
namespace sc_core {
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
class sc_attr_cltn
{
    public:
        typedef sc_attr_base* elem_type;
        typedef elem_type* iterator;
        typedef const elem_type* const_iterator;

        iterator begin();
        const_iterator begin() const;
        iterator end();
        const_iterator end() const;

        // Other members
        Implementation-defined

    private:
        // Disabled
        sc_attr_cltn( const sc_attr_cltn& );
        sc_attr_cltn& operator= ( const sc_attr_cltn& );
};

}       // namespace sc_core
```

5.20.3 Constraints on usage

An application shall not explicitly create an object of class **sc_attr_cltn**. An application may use the iterators to traverse the attribute collection returned by member function **attr_cltn** of class **sc_object**.

An implementation is only obliged to keep an attribute collection valid until a new attribute is attached to the **sc_object** or an existing attribute is removed from the **sc_object** in question. Hence, an application should traverse the attribute collection immediately on return from member function **attr_cltn**.

5.20.4 Iterators

```
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
```

Member function **begin** shall return a pointer to the first element of the collection. Each element of the collection is itself a pointer to an attribute.

Member function **end** shall return a pointer to the element following the last element of the collection.

Example:

```
{
    using namespace sc_core;
    sc_signal<int> sig;
    ...

    // Iterate through all the attributes of an sc_object
    sc_attr_cltn& c = sig.attr_cltn();
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    for (sc_attr_cltn::iterator i = c.begin(); i < c.end(); i++) {
        sc_attribute<int>* ap = dynamic_cast<sc_attribute<int>*>(*i);
        if (ap) std::cout << ap->name() << "=" << ap->value << std::endl;
    }
}
```

5.21 sc_hierarchy_scope

5.21.1 Description

Class **sc_hierarchy_scope** provides an interface to allow an application to place objects of type **sc_object** in an object hierarchy outside the current hierarchical scope.

5.21.2 Class definition

```
namespace sc_core {

class sc_hierarchy_scope
{
public:
    sc_hierarchy_scope( sc_hierarchy_scope&& );
    ~sc_hierarchy_scope() noexcept(false);

    static sc_hierarchy_scope get_root();

private:
    // Disabled
    sc_hierarchy_scope ( const sc_hierarchy_scope& ) = delete;
    sc_hierarchy_scope& operator=( const sc_hierarchy_scope& ) = delete;
    void* operator new( std::size_t) = delete;
    void* operator new[]( std::size_ t) = delete;
};

} // namespace sc_core
```

5.21.3 Constraints on usage

The copy constructor, assignment operator, and new operators shall be deleted so that they cannot be used by an application.

5.21.4 Constructor

sc_hierarchy_scope(sc_hierarchy_scope&&);

The constructor **sc_hierarchy_scope** shall set the hierarchical scope which is given as argument.

5.21.5 get_root

static sc_hierarchy_scope **get_root**();

Static member function **get_root** shall return the root of the object hierarchy.

NOTE—An application may call **sc_core::sc_hierarchy_scope(sc_core::sc_hierarchy_scope::get_root**()) to create objects at the top level.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Example:

```
SC_MODULE(TopModule) { SC_CTOR(TopModule){} };

SC_MODULE(SomeModule)
{
    sc_in<bool> SC_NAMED(in);

    SC_CTOR(SomeModule)
    {
        {
           sc_core::sc_hierarchy_scope scope( sc_core::sc_hierarchy_scope::get_root() );
           TopModule * mod = new TopModule("TopModule"); // will be top-level object
        }
    }
};
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6. Predefined channel class definitions

6.1 sc_signal_in_if

6.1.1 Description

Class **sc_signal_in_if** is an interface proper used by predefined channels, including **sc_signal**. Interface **sc_signal_in_if** gives read access to the value of a signal.

6.1.2 Class definition

```
namespace sc_core {

template <class T>
class sc_signal_in_if
: virtual public sc_interface
{
    public:
        virtual const T& read() const = 0;
        virtual const sc_event& value_changed_event() const = 0;
        virtual bool event() const = 0;

    protected:
        sc_signal_in_if();

    private:
        // Disabled
        sc_signal_in_if( const sc_signal_in_if<T>& );
        sc_signal_in_if<T>& operator= ( const sc_signal_in_if<T>& );
};

}       // namespace sc_core
```

6.1.3 Member functions

The following member functions are all pure virtual functions. The descriptions refer to the expected definitions of the functions when overridden in a channel that implements this interface. The precise semantics will be channel-specific.

Member function **read** shall return a reference to the current value of the channel.

Member function **value_changed_event** shall return a reference to an event that is notified whenever the value of the channel is written or modified.

Member function **event** shall return the value **true** if and only if the value of the channel was written or modified in the immediately preceding delta cycle and at the current simulation time.

NOTE—The value of the channel may have been modified in the evaluation phase or in the update phase of the immediately preceding delta cycle, depending on whether it is a hierarchical channel or a primitive channel (for example, **sc_signal**).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.2 sc_signal_in_if<bool> and sc_signal_in_if<sc_dt::sc_logic>

6.2.1 Description

Classes **sc_signal_in_if<bool>** and **sc_signal_in_if<sc_dt::sc_logic>** are interfaces proper that provide additional member functions appropriate for two-valued signals.

6.2.2 Class definition

```
namespace sc_core {

template <>
class sc_signal_in_if<bool>
: virtual public sc_interface
{
    public:
        virtual const T& read() const = 0;

        virtual const sc_event& value_changed_event() const = 0;
        virtual const sc_event& posedge_event() const = 0;
        virtual const sc_event& negedge_event() const = 0;

        virtual bool event() const = 0;
        virtual bool posedge() const = 0;
        virtual bool negedge() const = 0;

    protected:
        sc_signal_in_if();

    private:
        // Disabled
        sc_signal_in_if( const sc_signal_in_if<bool>& );
        sc_signal_in_if<bool>& operator= ( const sc_signal_in_if<bool>& );
};


template <>
class sc_signal_in_if<sc_dt::sc_logic>
: virtual public sc_interface
{
    public:
        virtual const T& read() const = 0;

        virtual const sc_event& value_changed_event() const = 0;
        virtual const sc_event& posedge_event() const = 0;
        virtual const sc_event& negedge_event() const = 0;

        virtual bool event() const = 0;
        virtual bool posedge() const = 0;
        virtual bool negedge() const = 0;

    protected:
        sc_signal_in_if();
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    private:
        // Disabled
        sc_signal_in_if( const sc_signal_in_if<sc_dt::sc_logic>& );
        sc_signal_in_if<sc_dt::sc_logic>& operator= ( const sc_signal_in_if<sc_dt::sc_logic>& );
};

}         // namespace sc_core
```

6.2.3 Member functions

The following list is incomplete. For the remaining member functions, refer to the definitions of the member functions for class **sc_signal_in_if** (see 6.1.3).

Member function **posedge_event** shall return a reference to an event that is notified whenever the value of the channel (as returned by member function **read**) changes and the new value of the channel is **true** or **'1'**.

Member function **negedge_event** shall return a reference to an event that is notified whenever the value of the channel (as returned by member function **read**) changes and the new value of the channel is **false** or **'0'**.

Member function **posedge** shall return the value **true** if and only if the value of the channel changed in the update phase of the immediately preceding delta cycle and at the current simulation time, and the new value of the channel is **true** or **'1'**.

Member function **negedge** shall return the value **true** if and only if the value of the channel changed in the update phase of the immediately preceding delta cycle and at the current simulation time, and the new value of the channel is **false** or **'0'**.

6.3 sc_signal_inout_if

6.3.1 Description

Class **sc_signal_inout_if** is an interface proper that is used by predefined channels, including **sc_signal**. Interface **sc_signal_inout_if** gives both read and write access to the value of a signal. The class shall be derived from both interface proper **sc_signal_write_if** and interface proper **sc_signal_in_if**.

6.3.2 Class definition

```
namespace sc_core {

#define SC_DEFAULT_WRITER_POLICY implementation-defined

enum sc_writer_policy
{
    SC_ONE_WRITER,
    SC_MANY_WRITERS
};

template <class T>
class sc_signal_write_if
: virtual public sc_interface
{
    public:
        virtual sc_writer_policy get_writer_policy() const { return SC_DEFAULT_WRITER_POLICY; }
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        virtual void write( const T& ) = 0;

    protected:
        sc_signal_write_if();

    private:
        // Disabled
        sc_signal_write_if( const sc_signal_write_if<T>& );
        sc_signal_write_if<T>& operator= ( const sc_signal_write_if<T>& );
};

template <class T>
class sc_signal_inout_if
: public sc_signal_in_if<T> , public sc_signal_write_if<T>
{
    protected:
        sc_signal_inout_if();

    private:
        // Disabled
        sc_signal_inout_if( const sc_signal_inout_if<T>& );
        sc_signal_inout_if<T>& operator= ( const sc_signal_inout_if<T>& );
};

}        // namespace sc_core
```

6.3.3 Member functions

The following text describes the expected definitions of the member functions when overridden in a channel that implements this interface. The precise semantics will be channel-specific.

virtual sc_writer_policy **get_writer_policy**() const { return SC_DEFAULT_WRITER_POLICY; }

Member function **get_writer_policy** shall return the value of the writer policy for the channel instance. Unless overridden in a channel, this function shall return the value SC_DEFAULT_WRITER_POLICY. By default, the value SC_DEFAULT_WRITER_POLICY shall be equal to SC_ONE_WRITER for backward compatibility with previous versions of this standard. An implementation may define another value of SC_DEFAULT_WRITER_POLICY.

virtual void **write**(const T&) = 0;

Member function **write** shall modify the value of the channel such that the channel appears to have the new value (as returned by member function **read**) in the next delta cycle but not before then. The new value is passed as an argument to the function.

Member function **write** shall honor the value of the writer policy set for the channel instance; that is, it shall permit either one writer or many writers.

6.4 sc_signal

6.4.1 Description

Class **sc_signal** is a predefined primitive channel intended to model the behavior of a single piece of wire carrying a digital electronic signal.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.4.2 Class definition

```
namespace sc_core {

template <class T, sc_writer_policy WRITER_POLICY = SC_DEFAULT_WRITER_POLICY>
class sc_signal
: public sc_signal_inout_if<T>, public sc_prim_channel
{
    public:
        sc_signal();
        explicit sc_signal( const char* );
        sc_signal( const char* name_, const T& initial_value_ );
        sc_signal( const char* name_, const sc_dt::sc_logic initial_value_ );
        virtual ~sc_signal();

        virtual void register_port( sc_port_base&, const char* );

        virtual const T& read() const;
        operator const T& () const;

        virtual sc_writer_policy get_writer_policy() const;
        virtual void write( const T& );
        sc_signal<T,WRITER_POLICY>& operator= ( const T& );
        sc_signal<T,WRITER_POLICY>& operator= ( const sc_signal<T,WRITER_POLICY>& );

        virtual const sc_event& default_event() const;
        virtual const sc_event& value_changed_event() const;
        virtual bool event() const;

        virtual void print( std::ostream& = std::cout ) const;
        virtual void dump( std::ostream& = std::cout ) const;
        virtual const char* kind() const;

    protected:
        virtual void update();

    private:
        // Disabled
        sc_signal( const sc_signal<T,WRITER_POLICY>& );
};

template <class T, sc_writer_policy WRITER_POLICY>
inline std::ostream& operator<< ( std::ostream&, const sc_signal<T,WRITER_POLICY>& );

}       // namespace sc_core
```

6.4.3 Template parameter T

The argument passed to template **sc_signal** shall be either a C++ type for which the predefined semantics for assignment and equality are adequate (for example, a fundamental type or a pointer), or a type **T** that obeys each of the following rules:

a) The following equality operator shall be defined for the type **T** and should return the value **true** if and only if the two values being compared are to be regarded as indistinguishable for the purposes of

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

signal propagation (that is, an event occurs only if the values are different). The implementation shall use this operator within the implementation of the signal to determine whether an event has occurred.

bool T::**operator==** (const T&);

b) The following stream operator shall be defined and should copy the state of the object given as the second argument to the stream given as the first argument. The way in which the state information is formatted is undefined by this standard. The implementation shall use this operator in implementing the behavior of the member functions **print** and **dump**.

std::ostream& **operator<<** (std::ostream&, const T&);

c) If the default assignment semantics are inadequate (in the sense given in this subclause), the following assignment operator should be defined for the type **T**. In either case (default assignment or explicit operator), the semantics of assignment should be sufficient to assign the state of an object of type **T** such that the value of the left operand is indistinguishable from the value of the right operand using the equality operator mentioned in this subclause. The implementation shall use this assignment operator within the implementation of the signal when assigning or copying values of type **T**.

const T& **operator=** (const T&);

d) If any constructor for type **T** exists, a default constructor for type **T** shall be defined.

e) If the class template is used to define a signal to which a port of type **sc_in**, **sc_inout**, or **sc_out** is bound, the following function shall be defined:

void **sc_trace**(sc_trace_file*, const T&, const std::string&);

NOTE 1—The equality and assignment operators are not obliged to compare and assign the complete state of the object, although they should typically do so. For example, diagnostic information may be associated with an object that is not to be propagated through the signal.

NOTE 2—The SystemC data types proper (**sc_dt::sc_int**, **sc_dt::sc_logic**, and so forth) all conform to the rule set just given.

NOTE 3—It is illegal to pass class **sc_module** (for example) as a template argument to class **sc_signal**, because **sc_module::operator==** does not exist. It is legal to pass type **sc_module*** through a signal, although this would be regarded as an abuse of the module hierarchy and thus bad practice.

6.4.4 Reading and writing signals

A signal is *read* by calling member function **read** or **operator const T& ()**.

A signal is *written* by calling member function **write** or **operator=** of the given signal object. If the template argument WRITER_POLICY has the value SC_ONE_WRITER, it shall be an error to write to a given signal instance from more than one process instance at any time during simulation. If the template argument WRITER_POLICY has the value SC_MANY_WRITERS, it shall be an error to write to a given signal instance from more than one process instance during any given evaluation phase, but different process instances may write to a given signal instance during different delta cycles. A signal may be written during elaboration (including the elaboration and simulation callbacks as described in 4.5) to initialize the value of

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

the signal. A signal may be written from function **sc_main** during elaboration or while simulation is paused, that is, before or after the call to function **sc_start**.

Signals are typically read and written during the evaluation phase, but the value of the signal is only modified during the subsequent update phase. If and only if the value of the signal actually changes as a result of being written, an event (the *value-changed event*) shall be notified in the delta notification phase that immediately follows.

If a given signal is written on multiple occasions within a particular evaluation phase, or during elaboration, or from function **sc_main**, the value to which the signal changes in the immediately following update phase shall be determined by the most recent write; that is, *the last write wins*.

NOTE 1—The specialized ports **sc_inout** and **sc_out** have a member function **initialize** for the purpose of initializing the value of a signal during elaboration.

NOTE 2—If the value of a signal is read during elaboration, the value returned will be the initial value of the signal as created by the default constructor for type **T**.

NOTE 3—If a given signal is written and read during the same evaluation phase, the old value will be read. The value written will not be available to be read until the subsequent evaluation phase.

6.4.5 Constructors

sc_signal();

This constructor shall call the base class constructor from its initializer list as follows:

sc_prim_channel(sc_gen_unique_name("signal"))

This constructor shall initialize the value of the signal by calling the default constructor for type **T** from their initializer lists.

explicit **sc_signal**(const char* name_);

This constructor shall call the base class constructor from its initializer list as follows:

sc_prim_channel(name_)

This constructor shall initialize the value of the signal by calling the default constructor for type **T** from their initializer lists.

sc_signal(const char* name_, const T& initial_value_);

This constructor shall call the base class constructor from its initializer list as follows:

sc_prim_channel(name_)

This constructor shall initialize the sc_signal to the value initial_value_ without triggering an event and a subsequent delta cycle.

sc_signal(const char* name_, sc_dt::sc_logic initial_value_);

This constructor shall call the base class constructor from its initializer list as follows:

sc_prim_channel(name_)

This constructor shall initialize the sc_signal to the value initial_value_ without triggering an event and a subsequent delta cycle.

6.4.6 register_port

virtual void **register_port**(sc_port_base&, const char*);

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Member function **register_port** of class **sc_interface** shall be overridden in class **sc_signal** and shall perform the following error check. If the template argument WRITER_POLICY has the value SC_ONE_WRITER, it shall be an error to bind more than one port of type **sc_signal_inout_if** to a given signal. If the WRITER_POLICY has the value SC_MANY_WRITERS, one or more ports of type **sc_signal_inout_if** may be bound to a given signal.

6.4.7 Member functions for reading

virtual const T& **read**() const;

Member function **read** shall return a reference to the current value of the signal but shall not modify the state of the signal.

operator const T& () const;

operator const T& () shall return a reference to the current value of the signal (as returned by member function **read**).

6.4.8 Member functions for writing

virtual sc_writer_policy **get_writer_policy**() const;

Member function **get_writer_policy** shall return the value of the WRITER_POLICY template parameter for the channel instance.

virtual void **write**(const T&);

Member function **write** shall modify the value of the signal such that the signal appears to have the new value (as returned by member function **read**) in the next delta cycle but not before then. This shall be accomplished using the update request mechanism of the primitive channel. The new value is passed as an argument to member function **write**. Member function **write** may be called during elaboration, in which case the update request shall be executed during the initialization phase.

operator=

The behavior of **operator=** shall be equivalent to the following definitions:

sc_signal<T,WRITER_POLICY>& **operator=** (const T& arg) { write(arg); return *this; }
sc_signal<T,WRITER_POLICY>& **operator=** (const sc_signal<T,WRITER_POLICY>& arg) { write(arg.read()); return *this; }

virtual void **update**();

Member function **update** of class **sc_prim_channel** shall be overridden by the implementation in class **sc_signal** to implement the updating of the signal value that occurs as a result of the signal being written. Member function **update** shall modify the current value of the signal such that it gets the new value (as passed as an argument to member function **write**), and it shall cause the value-changed event to be notified in the immediately following delta notification phase if the value of the signal has changed.

NOTE—Member function **update** is called by the scheduler but typically is not called by an application. However, member function **update** of class **sc_signal** may be called from member function **update** of a class derived from class **sc_signal**.

6.4.9 Member functions for events

virtual const sc_event& **default_event**() const;
virtual const sc_event& **value_changed_event**() const;

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Member functions **default_event** and **value_changed_event** shall both return a reference to the value-changed event.

virtual bool **event**() const;

Member function **event** shall return the value **true** if and only if the value of the signal changed in the update phase of the immediately preceding delta cycle and at the current simulation time; that is, a member function **write** or **operator=** was called in the immediately preceding evaluation phase, and the value written or assigned was different from the previous value of the signal.

NOTE—Member function **event** returns **true** when called from a process that was executed as a direct result of the value-changed event of that same signal instance being notified.

6.4.10 Diagnostic member functions

virtual void **print**(std::ostream& = std::cout) const;

Member function **print** shall print the current value of the signal to the stream passed as an argument by calling **operator<< (std::ostream&, T&)**. No additional characters shall be printed.

virtual void **dump**(std::ostream& = std::cout) const;

Member function **dump** shall print at least the hierarchical name, the current value, and the new value of the signal to the stream passed as an argument. The formatting shall be implementation-defined.

virtual const char* **kind**() const;

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

6.4.11 operator<<

template <class T, sc_writer_policy WRITER_POLICY>
inline std::ostream& **operator<<** (std::ostream& , const sc_ signal<T,WRITER_POLICY>&);

operator<< shall print the current value of the signal passed as the second argument to the stream passed as the first argument by calling **operator<<** (std::ostream& , T&).

Example:

```
SC_MODULE(M) {
    sc_core::sc_signal<int> sig;
    SC_CTOR(M) {
        SC_THREAD(writer);
        SC_THREAD(reader);
        SC_METHOD(writer2);
        sensitive << sig;                    // Sensitive to the default event
    }

    void writer() {
        using namespace sc_core;
        wait(50.0, SC_NS);
        sig.write(1);
        sig.write(2);
        wait(50.0, SC_NS);
        sig = 3;                             // Calls operator= ( const T& )
    }
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    void reader() {
        wait(sig.value_changed_event());
        int i = sig.read();                    // Reads a value of 2
        wait(sig.value_changed_event());
        i = sig;                               // Calls operator const T& (), which returns a value of 3
    }

    void writer2() {
        sig.write(sig + 1);                    // An error. A signal shall not have multiple writers
    }
};
```

NOTE—The following classes are related to class **sc_signal**:

- The classes **sc_signal<bool**,WRITER_POLICY**>** and **sc_signal<sc_dt::sc_logic**,WRITER_POLICY**>** provide additional member functions appropriate for two-valued signals.
- The class **sc_buffer** is derived from **sc_signal** but differs in that the value-changed event is notified whenever the buffer is written whether or not the value of the buffer has changed.
- The class **sc_clock** is derived from **sc_signal** and generates a periodic clock signal.
- The class **sc_signal_resolved** allows multiple writers.
- The classes **sc_in**, **sc_out**, and **sc_inout** are specialized ports that may be bound to signals, and which provide functions to access the member functions of the signal conveniently through the port.

6.5 sc_signal<bool,WRITER_POLICY> and sc_signal<sc_dt::sc_logic,WRITER_POLICY>

6.5.1 Description

Classes **sc_signal<bool,WRITER_POLICY>** and **sc_signal<sc_dt::sc_logic,WRITER_POLICY>** are predefined primitive channels that provide additional member functions appropriate for two-valued signals.

6.5.2 Class definition

```
namespace sc_core {

template <sc_writer_policy WRITER_POLICY>
class sc_signal<bool,WRITER_POLICY>
: public sc_signal_inout_if<bool>, public sc_prim_channel
{
    public:
        sc_signal();
        explicit sc_signal( const char* );
        virtual ~sc_signal();

        virtual void register_port( sc_port_base&, const char* );

        virtual const bool& read() const;
        operator const bool& () const;

        virtual sc_writer_policy get_writer_policy() const;
        virtual void write( const bool& );
        sc_signal<bool,WRITER_POLICY>& operator= ( const bool& );
        sc_signal<bool,WRITER_POLICY>& operator= ( const sc_signal<bool,WRITER_POLICY>& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        virtual const sc_event& default_event() const;

        virtual const sc_event& value_changed_event() const;
        virtual const sc_event& posedge_event() const;
        virtual const sc_event& negedge_event() const;

        virtual bool event() const;
        virtual bool posedge() const;
        virtual bool negedge() const;

        virtual void print( std::ostream& = std::cout ) const;
        virtual void dump( std::ostream& = std::cout ) const;
        virtual const char* kind() const;

    protected:
        virtual void update();

    private:
        // Disabled
        sc_signal( const sc_signal<bool,WRITER_POLICY>& );
};


template <sc_writer_policy WRITER_POLICY>
class sc_signal<sc_dt::sc_logic,WRITER_POLICY>
: public sc_signal_inout_if<sc_dt::sc_logic,WRITER_POLICY>, public sc_prim_channel
{
    public:
        sc_signal();
        explicit sc_signal( const char* );
        virtual ~sc_signal();

        virtual void register_port( sc_port_base&, const char* );

        virtual const sc_dt::sc_logic& read() const;
        operator const sc_dt::sc_logic& () const;

        virtual void write( const sc_dt::sc_logic& );
        sc_signal<sc_dt::sc_logic,WRITER_POLICY>& operator= ( const sc_dt::sc_logic& );
        sc_signal<sc_dt::sc_logic,WRITER_POLICY>&
                                    operator= ( const sc_signal<sc_dt::sc_logic,WRITER_POLICY>& );

        virtual const sc_event& default_event() const;

        virtual const sc_event& value_changed_event() const;
        virtual const sc_event& posedge_event() const;
        virtual const sc_event& negedge_event() const;

        virtual bool event() const;
        virtual bool posedge() const;
        virtual bool negedge() const;

        virtual void print( std::ostream& = std::cout ) const;
        virtual void dump( std::ostream& = std::cout ) const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        virtual const char* kind() const;

    protected:
        virtual void update();

    private:
        // Disabled
        sc_signal( const sc_signal<sc_dt::sc_logic,WRITER_POLICY>& );
};

}        // namespace sc_core
```

6.5.3 Member functions

The following list is incomplete. For the remaining member functions, refer to the definitions of the member functions for class **sc_signal** (see 6.4).

virtual const sc_event& **posedge_event** () const;

Member function **posedge_event** shall return a reference to an event that is notified whenever the value of the signal (as returned by member function **read**) changes and the new value of the signal is **true** or **'1'**.

virtual const sc_event& **negedge_event**() const;

Member function **negedge_event** shall return a reference to an event that is notified whenever the value of the signal (as returned by member function **read**) changes and the new value of the signal is **false** or **'0'**.

virtual bool **posedge** () const;

Member function **posedge** shall return the value **true** if and only if the value of the signal changed in the update phase of the immediately preceding delta cycle and at the current simulation time, and the new value of the signal is **true** or **'1'**.

virtual bool **negedge**() const;

Member function **negedge** shall return the value **true** if and only if the value of the signal changed in the update phase of the immediately preceding delta cycle and at the current simulation time, and the new value of the signal is **false** or **'0'**.

Example:

```
SC_MODULE(M) {
    sc_core::sc_signal<bool> clk;
    ...

    void thread_process() {
        for (;;) {
            if (clk.posedge())
                wait(clk.negedge_event());
                ...
        }
    }
};
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.6 sc_buffer

6.6.1 Description

Class **sc_buffer** is a predefined primitive channel derived from class **sc_signal**. Class **sc_buffer** differs from class **sc_signal** in that a value-changed event is notified whenever the buffer is written rather than only when the value of the signal is changed. A *buffer* is an object of the class **sc_buffer**.

6.6.2 Class definition

```
namespace sc_core {

template <class T, sc_writer_policy WRITER_POLICY = SC_DEFAULT_WRITER_POLICY>
class sc_buffer
: public sc_signal<T,WRITER_POLICY>
{
    public:
        sc_buffer();
        explicit sc_buffer( const char* );
        sc_buffer( const char* name_, const T& initial_value_ );

        virtual void write( const T& );

        sc_buffer<T,WRITER_POLICY>& operator= ( const T& );
        sc_buffer<T,WRITER_POLICY>& operator= ( const sc_signal<T,WRITER_POLICY>& );
        sc_buffer<T,WRITER_POLICY>& operator= ( const sc_buffer<T,WRITER_POLICY>& );

        virtual const char* kind() const;

    protected:
        virtual void update();

    private:
        // Disabled
        sc_buffer( const sc_buffer<T,WRITER_POLICY>& );
};

}       // namespace sc_core
```

6.6.3 Constructors

sc_buffer();

This constructor shall call the base class constructor from its initializer list as follows:

sc_signal(sc_gen_unique_name("buffer"))

explicit **sc_buffer**(const char* name_);

This constructor shall call the base class constructor from its initializer list as follows:

sc_signal(name_)

sc_buffer(const char* name_, const T& initial_value_);

This constructor shall call the base class constructor from its initializer list as follows:

sc_signal(name_)

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

This constructor shall initialize the **sc_buffer** to the value initial_value_ without triggering an event and a subsequent delta cycle.

6.6.4 Member functions

virtual void **write**(const T&);

Member function **write** shall modify the value of the buffer such that the buffer appears to have the new value (as returned by member function **read**) in the next delta cycle but not before then. This shall be accomplished using the update request mechanism of the primitive channel. The new value is passed as an argument to member function **write**.

operator=

The behavior of **operator=** shall be equivalent to the following definitions:

```
sc_buffer<T,WRITER_POLICY>& operator= ( const T& arg ) { write( arg ); return *this; }

sc_buffer<T,WRITER_POLICY>&
    operator= ( const sc_signal<T,WRITER_POLICY>& arg ) { write( arg.read() ); return *this; }

sc_buffer<T,WRITER_POLICY>&
    operator= ( const sc_buffer<T,WRITER_POLICY>& arg ) { write( arg.read() ); return *this; }
```

virtual void **update**();

Member function **update** of class **sc_signal** shall be overridden by the implementation in class **sc_buffer** to implement the updating of the buffer value that occurs as a result of the buffer being written. Member function **update** shall modify the current value of the buffer such that it gets the new value (as passed as an argument to member function **write**) and shall cause the value-changed event to be notified in the immediately following delta notification phase, regardless of whether the value of the buffer has changed (see 6.4.4 and 6.4.8). (This is in contrast to member function **update** of the base class **sc_signal**, which only causes the value-changed event to be notified if the new value is different from the old value.)

In other words, suppose the current value of the buffer is V, and member function **write** is called with argument value V. Function **write** will store the new value V (in some implementation-defined storage area distinct from the current value of the buffer) and will call **request_update**. Member function **update** will be called back during the update phase and will set the current value of the buffer to the new value V. The current value of the buffer will not change, because the old value is equal to the new value but the value-changed event will be notified nonetheless.

virtual const char* **kind**() const;

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

Example:

```
SC_MODULE(M) {
    sc_core::sc_buffer<int> buf;

    SC_CTOR(M) {
        SC_THREAD(writer);
        SC_METHOD(reader);
        sensitive << buf;
    }
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    void writer() {
        buf.write(1);
        wait(sc_core::SC_ZERO_TIME);
        buf.write(1);
    }

    void reader() {                    // Executed during initialization and then twice more with buf = 0, 1, 1
        std::cout << buf << std::endl;
    }
};
```

6.7 sc_clock

6.7.1 Description

Class **sc_clock** is a predefined primitive channel derived from the class **sc_signal** and intended to model the behavior of a digital clock signal. A *clock* is an object of the class **sc_clock**. The value and events associated with the clock are accessed through the interface **sc_signal_in_if<bool>**.

6.7.2 Class definition

```
namespace sc_core {

class sc_clock
: public sc_signal<bool>
{
    public:
        sc_clock();
        explicit sc_clock( const char* name_ );

        sc_clock( const char* name_,
                  const sc_time& period_,
                  double duty_cycle_ = 0.5,
                  const sc_time& start_time_ = SC_ZERO_TIME,
                  bool posedge_first_ = true );

        sc_clock( const char* name_,
                  double period_v_,
                  sc_time_unit period_tu_,
                  double duty_cycle_ = 0.5 );

        sc_clock( const char* name_,
                  double period_v_,
                  sc_time_unit period_tu_,
                  double duty_cycle_,
                  double start_time_v_,
                  sc_time_unit start_time_tu_,
                  bool posedge_first_ = true );

        virtual ~sc_clock();

        virtual void write( const bool& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        const sc_time& period() const;
        double duty_cycle() const;
        const sc_time& start_time() const;
        bool posedge_first() const;

        virtual const char* kind() const;

    protected:
        virtual void before_end_of_elaboration();

    private:
        // Disabled
        sc_clock( const sc_clock& );
        sc_clock& operator= ( const sc_clock& );
};

typedef sc_in<bool> sc_in_clk ;

}          // namespace sc_core
```

6.7.3 Characteristic properties

A clock is characterized by the following properties:

a) *Period*—The time interval between two consecutive transitions from value **false** to value **true**, which shall be equal to the time interval between two consecutive transitions from value **true** to value **false**. The period shall be greater than zero. The default period is 1 nanosecond.

b) *Duty cycle*—The proportion of the period during which the clock has the value **true**. The duty cycle shall lie between the limits 0.0 and 1.0, exclusive. The default duty cycle is 0.5.

c) *Start time*—The absolute time of the first transition of the value of the clock (**false** to **true** or **true** to **false**). The default start time is zero.

d) *Posedge_first*—If posedge_first is **true**, the clock is initialized to the value **false**, and changes from **false** to **true** at the start time. If posedge_first is **false**, the clock is initialized to the value **true**, and changes from **true** to **false** at the start time. The default value of posedge_first is **true**.

NOTE—A clock does not have a stop time but will stop in any case when function **sc_stop** is called.

6.7.4 Constructors

The constructors shall set the characteristic properties of the clock as defined by the constructor arguments. Any characteristic property not defined by the constructor arguments shall take a default value as defined in 6.7.3.

The default constructor shall call the base class constructor from its initializer list as follows:

```
sc_signal( sc_gen_unique_name( "clock" ) )
```

6.7.5 write

virtual void **write**(const bool&);

It shall be an error for an application to call member function **write**. The member function **write** of the base class **sc_signal** is not applicable to clocks.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.7.6 Diagnostic member functions

const sc_time& **period**() const;

Member function **period** shall return the period of the clock.

double **duty_cycle**() const;

Member function **duty_cycle** shall return the duty cycle of the clock.

const sc_time& **start_time**() const;

Member function **start_time** shall return the start time of the clock.

bool **posedge_first**() const;

Member function **posedge_first** shall return the value of the posedge_first property of the clock.

virtual const char* **kind**() const;

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

6.7.7 before_end_of_elaboration

virtual void **before_end_of_elaboration()**;

Member function **before_end_of_elaboration**, which is defined in the class **sc_prim_channel**, shall be overridden by the implementation in the current class with a behavior that is implementation-defined.

NOTE 1—An implementation may use **before_end_of_elaboration** to spawn one or more static processes to generate the clock.

NOTE 2—If this member function is overridden in a class derived from the current class, function **before_end_of_elaboration** as overridden in the current class should be called explicitly from the overridden member function of the derived class in order to invoke the implementation-defined behavior.

6.7.8 sc_in_clk

typedef sc_in<bool> **sc_in_clk** ;

The typedef **sc_in_clk** is provided for convenience when adding clock inputs to a module and for backward compatibility with earlier versions of SystemC. An application may use **sc_in_clk** or **sc_in<bool>** interchangeably.

6.8 sc_in

6.8.1 Description

Class **sc_in** is a specialized port class for use with signals. It provides functions to access conveniently certain member functions of the channel to which the port is bound. It may be used to model an input pin on a module.

6.8.2 Class definition

namespace sc_core {

template <class T>
class **sc_in**

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
: public sc_port<sc_signal_in_if<T>,1>
{
    public:
        sc_in();
        explicit sc_in( const char* );
        virtual ~sc_in();

        virtual void bind ( const sc_signal_in_if<T>& );
        void operator() ( const sc_signal_in_if<T>& );

        virtual void bind ( sc_port<sc_signal_in_if<T>, 1>& );
        void operator() ( sc_port<sc_signal_in_if<T>, 1>& );

        virtual void bind ( sc_port<sc_signal_inout_if<T>, 1>& );
        void operator() ( sc_port<sc_signal_inout_if<T>, 1>& );

        virtual void end_of_elaboration();

        const T& read() const;
        operator const T& () const;

        const sc_event& default_event() const;
        const sc_event& value_changed_event() const;
        bool event() const;
        sc_event_finder& value_changed() const;

        virtual const char* kind() const;

    private:
        // Disabled
        sc_in( const sc_in<T>& );
        sc_in<T>& operator= ( const sc_in<T>& );
};

template <class T>
inline void sc_trace( sc_trace_file*, const sc_in<T>&, const std::string& );

}       // namespace sc_core
```

6.8.3 Member functions

The constructors shall pass their arguments to the corresponding constructor for the base class **sc_port**.

The implementation of **operator()** shall achieve its effect by calling the virtual member function **bind**. Member function **bind** shall call member function **bind** of the base class **sc_port**, passing through their parameters as arguments to function **bind**, in order to bind the object of class **sc_in** to the channel or port instance passed as an argument.

Member function **read** and **operator const T&()** shall each call member function **read** of the object to which the port is bound using **operator->** of class **sc_port**, that is:

```
(*this)->read()
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Member functions **default_event**, **value_changed_event**, and **event** shall each call the corresponding member function of the object to which the port is bound using **operator->** of class **sc_port**, for example:

```
(*this)->event()
```

Member function **value_changed** shall return a reference to class **sc_event_finder**, where the event finder object itself shall be constructed using the member function **value_changed_event** (see 5.7).

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

6.8.4 sc_trace

template <class T>
inline void **sc_trace**(sc_trace_file*, const sc_in<T>&, const std::string&);

Function **sc_trace** shall trace the channel to which the port passed as the second argument is bound (see 8.1.1) by calling function **sc_trace** with a second argument of type **const T&** (see 6.4.3). The port need not have been bound at the point during elaboration when function **sc_trace** is called. In this case, the implementation shall defer the call to trace the signal until after the port has been bound and the identity of the signal is known.

6.8.5 end_of_elaboration

virtual void **end_of_elaboration()**;

Member function **end_of_elaboration**, which is defined in the class **sc_port**, shall be overridden by the implementation in the current class with a behavior that is implementation-defined.

NOTE 1—An implementation may use **end_of_elaboration** to implement the deferred call to **sc_trace**.

NOTE 2—If this member function is overridden in a class derived from the current class, function **end_of_elaboration** as overridden in the current class should be called explicitly from the overridden member function of the derived class in order to invoke the implementation-defined behavior.

6.9 sc_in<bool> and sc_in<sc_dt::sc_logic>

6.9.1 Description

Class **sc_in<bool>** and **sc_in<sc_dt::sc_logic>** are specialized port classes that provide additional member functions for two-valued signals.

6.9.2 Class definition

```
namespace sc_core {

template <>
class sc_in<bool>
: public sc_port<sc_signal_in_if<bool>,1>
{
    public:
        sc_in();
        explicit sc_in( const char* );
        virtual ~sc_in();

        virtual void bind ( const sc_signal_in_if<bool>& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        void operator() ( const sc_signal_in_if<bool>& );

        virtual void bind ( sc_port<sc_signal_in_if<bool>, 1>& );
        void operator() ( sc_port<sc_signal_in_if<bool>, 1>& );

        virtual void bind ( sc_port<sc_signal_inout_if<bool>, 1>& );
        void operator() ( sc_port<sc_signal_inout_if<bool>, 1>& );

        virtual void end_of_elaboration();

        const bool& read() const;
        operator const bool& () const;

        const sc_event& default_event() const;
        const sc_event& value_changed_event() const;
        const sc_event& posedge_event() const;
        const sc_event& negedge_event() const;

        bool event() const;
        bool posedge() const;
        bool negedge() const;

        sc_event_finder& value_changed() const;
        sc_event_finder& pos() const;
        sc_event_finder& neg() const;

        virtual const char* kind() const;

    private:
        // Disabled
        sc_in( const sc_in<bool>& );
        sc_in<bool>& operator= ( const sc_in<bool>& );
};

template <>
inline void sc_trace<bool>( sc_trace_file*, const sc_in<bool>&, const std::string& );


template <>
class sc_in<sc_dt::sc_logic>
: public sc_port<sc_signal_in_if<sc_dt::sc_logic>,1>
{
    public:
        sc_in();
        explicit sc_in( const char* );
        virtual ~sc_in();

        virtual void bind ( const sc_signal_in_if<sc_dt::sc_logic>& );
        void operator() ( const sc_signal_in_if<sc_dt::sc_logic>& );

        virtual void bind ( sc_port<sc_signal_in_if<sc_dt::sc_logic>, 1>& );
        void operator() ( sc_port<sc_signal_in_if<sc_dt::sc_logic>, 1>& );

        virtual void bind ( sc_port<sc_signal_inout_if<sc_dt::sc_logic>, 1>& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        void operator() ( sc_port<sc_signal_inout_if<sc_dt::sc_logic>, 1>& );

        virtual void end_of_elaboration();

        const sc_dt::sc_logic& read() const;
        operator const sc_dt::sc_logic& () const;

        const sc_event& default_event() const;
        const sc_event& value_changed_event() const;
        const sc_event& posedge_event() const;
        const sc_event& negedge_event() const;

        bool event() const;
        bool posedge() const;
        bool negedge() const;

        sc_event_finder& value_changed() const;
        sc_event_finder& pos() const;
        sc_event_finder& neg() const;

        virtual const char* kind() const;

    private:
        // Disabled
        sc_in( const sc_in<sc_dt::sc_logic>& );
        sc_in<sc_dt::sc_logic>& operator= ( const sc_in<sc_dt::sc_logic>& );
};

template <>
inline void
sc_trace<sc_dt::sc_logic>( sc_trace_file*, const sc_in<sc_dt::sc_logic>&, const std::string& );

}       // namespace sc_core
```

6.9.3 Member functions

The following list is incomplete. For the remaining member functions and for the function **sc_trace**, refer to the definitions of the member functions for class **sc_in** (see 6.8.3).

Member functions **posedge_event**, **negedge_event**, **posedge**, and **negedge** shall each call the corresponding member function of the object to which the port is bound using **operator->** of class **sc_port**, for example:

```
        (*this)->negedge()
```

Member functions **pos** and **neg** shall return a reference to class **sc_event_finder**, where the event finder object itself shall be constructed using the member function **posedge_event** or **negedge_event**, respectively (see 5.7).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.10 sc_inout

6.10.1 Description

Class **sc_inout** is a specialized port class for use with signals. It provides functions to access conveniently certain member functions of the channel to which the port is bound. It may be used to model an output pin or a bidirectional pin on a module.

6.10.2 Class definition

```
namespace sc_core {

template <class T>
class sc_inout
: public sc_port<sc_signal_inout_if<T>,1>
{
    public:
        sc_inout();
        explicit sc_inout( const char* );
        virtual ~sc_inout();

        void initialize( const T& );
        void initialize( const sc_signal_in_if<T>& );

        virtual void end_of_elaboration();

        const T& read() const;
        operator const T& () const;

        void write( const T& );
        sc_inout<T>& operator= ( const T& );
        sc_inout<T>& operator= ( const sc_signal_in_if<T>& );
        sc_inout<T>& operator= ( const sc_port< sc_signal_in_if<T>, 1>& );
        sc_inout<T>& operator= ( const sc_port< sc_signal_inout_if<T>, 1>& );
        sc_inout<T>& operator= ( const sc_inout<T>& );

        const sc_event& default_event() const;
        const sc_event& value_changed_event() const;
        bool event() const;
        sc_event_finder& value_changed() const;

        virtual const char* kind() const;

    private:
        // Disabled
        sc_inout( const sc_inout<T>& );
};

template <class T>
inline void sc_trace( sc_trace_file*, const sc_inout<T>&, const std::string& );

}          // namespace sc_core
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.10.3 Member functions

The constructors shall pass their arguments to the corresponding constructor for the base class **sc_port**.

Member function **read** and **operator const T&()** shall each call member function **read** of the object to which the port is bound using **operator->** of class **sc_port**, that is:

```
(*this)->read()
```

Member function **write** and **operator=** shall each call the member function **write** of the object to which the port is bound using **operator->** of class **sc_port**, calling member function **read** to get the value of the parameter, where the parameter is an interface or a port, for example:

```
sc_inout<T>& operator= ( const sc_inout<T>& port_ )
{ (*this)->write( port_->read() ); return *this; }
```

Member function **write** shall not be called during elaboration before the port has been bound (see 6.10.4).

Member functions **default_event**, **value_changed_event**, and **event** shall each call the corresponding member function of the object to which the port is bound using **operator->** of class **sc_port**, for example:

```
(*this)->event()
```

Member function **value_changed** shall return a reference to class **sc_event_finder**, where the event finder object itself shall be constructed using the member function **value_changed_event** (see 5.7).

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

6.10.4 initialize

Member function **initialize** shall set the initial value of the signal to which the port is bound by calling member function **write** of that signal using the value passed as an argument to member function **initialize**. If the actual argument is a channel, the initial value shall be determined by reading the value of the channel. The port need not have been bound at the point during elaboration when member function **initialize** is called. In this case, the implementation shall defer the call to **write** until after the port has been bound and the identity of the signal is known.

NOTE 1—A port of class **sc_in** will be bound to exactly one signal, but the binding may be performed indirectly through a port of the parent module.

NOTE 2—The purpose of member function **initialize** is to allow the value of a port to be initialized during elaboration before the port being bound. However, member function **initialize** may be called during elaboration or simulation.

6.10.5 sc_trace

```
template <class T>
inline void sc_trace( sc_trace_file*, const sc_inout<T>&, const std::string& );
```

Function **sc_trace** shall trace the channel to which the port passed as the second argument is bound (see 8.1.1) by calling function **sc_trace** with a second argument of type **const T&** (see 6.4.3). The port need not have been bound at the point during elaboration when function **sc_trace** is called. In this case, the implementation shall defer the call to trace the signal until after the port has been bound and the identity of the signal is known.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.10.6 end_of_elaboration

virtual void **end_of_elaboration()**;

Member function **end_of_elaboration**, which is defined in the class **sc_port**, shall be overridden by the implementation in the current class with a behavior that is implementation-defined.

NOTE 1—An implementation may use **end_of_elaboration** to implement the deferred calls for **initialize** and **sc_trace**.

NOTE 2—If this member function is overridden in a class derived from the current class, function **end_of_elaboration** as overridden in the current class should be called explicitly from the overridden member function of the derived class in order to invoke the implementation-defined behavior.

6.10.7 Binding

Because interface **sc_signal_inout_if** is derived from interface **sc_signal_in_if**, a port of class **sc_in** of a child module may be bound to a port of class **sc_inout** of a parent module but a port of class **sc_inout** of a child module cannot be bound to a port of class **sc_in** of a parent module.

6.11 sc_inout<bool> and sc_inout<sc_dt::sc_logic>

6.11.1 Description

Class **sc_inout<bool>** and **sc_inout<sc_dt::sc_logic>** are specialized port classes that provide additional member functions for two-valued signals.

6.11.2 Class definition

```
namespace sc_core {

template <>
class sc_inout<bool>
: public sc_port<sc_signal_inout_if<bool>,1>
{
    public:
        sc_inout();
        explicit sc_inout( const char* );
        virtual ~sc_inout();

        void initialize( const bool& );
        void initialize( const sc_signal_in_if<bool>& );

        virtual void end_of_elaboration();

        const bool& read() const;
        operator const bool& () const;

        void write( const bool& );
        sc_inout<bool>& operator= ( const bool& );
        sc_inout<bool>& operator= ( const sc_signal_in_if<bool>& );
        sc_inout<bool>& operator= ( const sc_port< sc_signal_in_if<bool>, 1>& );
        sc_inout<bool>& operator= ( const sc_port< sc_signal_inout_if<bool>, 1>& );
        sc_inout<bool>& operator= ( const sc_inout<bool>& );

        const sc_event& default_event() const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        const sc_event& value_changed_event() const;
        const sc_event& posedge_event() const;
        const sc_event& negedge_event() const;

        bool event() const;
        bool posedge() const;
        bool negedge() const;

        sc_event_finder& value_changed() const;
        sc_event_finder& pos() const;
        sc_event_finder& neg() const;

        virtual const char* kind() const;

    private:
        // Disabled
        sc_inout( const sc_inout<bool>& );
};

template <>
inline void sc_trace<bool>( sc_trace_file*, const sc_inout<bool>&, const std::string& );


template <>
class sc_inout<sc_dt::sc_logic>
: public sc_port<sc_signal_inout_if<sc_dt::sc_logic>,1>
{
    public:
        sc_inout();
        explicit sc_inout( const char* );
        virtual ~sc_inout();

        void initialize( const sc_dt::sc_logic& );
        void initialize( const sc_signal_in_if<sc_dt::sc_logic>& );

        virtual void end_of_elaboration();

        const sc_dt::sc_logic& read() const;
        operator const sc_dt::sc_logic& () const;

        void write( const sc_dt::sc_logic& );
        sc_inout<sc_dt::sc_logic>& operator= ( const sc_dt::sc_logic& );
        sc_inout<sc_dt::sc_logic>& operator= ( const sc_signal_in_if<sc_dt::sc_logic>& );
        sc_inout<sc_dt::sc_logic>& operator= ( const sc_port< sc_signal_in_if<sc_dt::sc_logic>, 1>& );
        sc_inout<sc_dt::sc_logic>& operator= ( const sc_port< sc_signal_inout_if<sc_dt::sc_logic>,
1>&);
        sc_inout<sc_dt::sc_logic>& operator= ( const sc_inout<sc_dt::sc_logic>& );

        const sc_event& default_event() const;
        const sc_event& value_changed_event() const;
        const sc_event& posedge_event() const;
        const sc_event& negedge_event() const;

        bool event() const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        bool posedge() const;
        bool negedge() const;

        sc_event_finder& value_changed() const;
        sc_event_finder& pos() const;
        sc_event_finder& neg() const;

        virtual const char* kind() const;

    private:
        // Disabled
        sc_inout( const sc_inout<sc_dt::sc_logic>& );
};

template <>
inline void
sc_trace<sc_dt::sc_logic>( sc_trace_file*, const sc_inout<sc_dt::sc_logic>&, const std::string& );

}       // namespace sc_core
```

6.11.3 Member functions

The following list is incomplete. For the remaining member functions and for the function **sc_trace**, refer to the definitions of the member functions for class **sc_inout**.

Member functions **posedge_event**, **negedge_event**, **posedge**, and **negedge** shall each call the corresponding member function of the object to which the port is bound using **operator->** of class **sc_port**, for example:

```
(*this)->negedge()
```

Member functions **pos** and **neg** shall return a reference to class **sc_event_finder**, where the event finder object itself shall be constructed using the member function **posedge_event** or **negedge_event**, respectively (see 5.7).

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

6.12 sc_out

6.12.1 Description

Class **sc_out** is derived from class **sc_inout** and is identical to class **sc_inout** except for differences inherent in it being a derived class, for example, constructors and assignment operators. The purpose of having both classes is to allow users to express their intent, that is, **sc_out** for output pins and **sc_inout** for bidirectional pins.

6.12.2 Class definition

```
namespace sc_core {

template <class T>
class sc_out
: public sc_inout<T>
{
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    public:
        sc_out();
        explicit sc_out( const char* );
        virtual ~sc_out();

        sc_out<T>& operator= ( const T& );
        sc_out<T>& operator= ( const sc_signal_in_if<T>& );
        sc_out<T>& operator= ( const sc_port< sc_signal_in_if<T>, 1>& );
        sc_out<T>& operator= ( const sc_port< sc_signal_inout_if<T>, 1>& );
        sc_out<T>& operator= ( const sc_out<T>& );

        virtual const char* kind() const;

    private:
        // Disabled
        sc_out( const sc_out<T>& );
};

}       // namespace sc_core
```

6.12.3 Member functions

The constructors shall pass their arguments to the corresponding constructors for the base class **sc_inout<T>**.

The behavior of the assignment operators shall be identical to that of class **sc_inout** but with the class name **sc_out** substituted in place of the class name **sc_inout** wherever appropriate.

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

6.13 sc_signal_resolved

6.13.1 Description

Class **sc_signal_resolved** is a predefined primitive channel derived from class **sc_signal**. A resolved signal is an object of class **sc_signal_resolved** or class **sc_signal_rv**. Class **sc_signal_resolved** differs from class **sc_signal** in that a resolved signal may be written by multiple processes, conflicting values being resolved within the channel.

6.13.2 Class definition

```
namespace sc_core {

class sc_signal_resolved
: public sc_signal<sc_dt::sc_logic,SC_MANY_WRITERS>
{
    public:
        sc_signal_resolved();
        explicit sc_signal_resolved( const char* );
        sc_signal_resolved( const char* name_, const data_type & initial_value_ );
        virtual ~sc_signal_resolved();

        virtual void register_port( sc_port_base&, const char* );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        virtual void write( const sc_dt::sc_logic& );
        sc_signal_resolved& operator= ( const sc_dt::sc_logic& );
        sc_signal_resolved& operator= ( const sc_signal_resolved& );

        virtual const char* kind() const;

    protected:
        virtual void update();

    private:
        // Disabled
        sc_signal_resolved( const sc_signal_resolved& );
};

}       // namespace sc_core
```

6.13.3 Constructors

sc_signal_resolved();

This constructor shall call the base class constructor from its initializer list as follows:

sc_signal(sc_gen_unique_name("signal_resolved"))

explicit **sc_signal_resolved**(const char* name_);

This constructor shall call the base class constructor from its initializer list as follows:

sc_signal(name_)

sc_signal_resolved(const char* name_, const data_type & initial_value_);

This constructor shall call the base class constructor from its initializer list as follows:

sc_signal(name_)

This constructor shall initialize the sc_signal to the value initial_value_ without triggering an event and a subsequent delta cycle.

6.13.4 Resolution semantics

A resolved signal is written by calling member function **write** or **operator=** of the given signal object. Like class **sc_signal**, **operator=** shall call member function **write**.

Each resolved signal shall maintain a *list of written values* containing one value for each distinct process instance that writes to the resolved signal object. This list shall store the value most recently written to the resolved signal object by each such process instance.

If and only if the written value is different from the previous written value or this is the first occasion on which the particular process instance has written to the particular signal object, the member function **write** shall then call the member function **request_update**.

During the update phase, member function **update** shall first use the list of written values to calculate a single *resolved value* for the resolved signal, and then perform update semantics similar to class **sc_signal** but using the resolved value just calculated.

A value shall be added to the list of written values on the first occasion that each particular process instance writes to the resolved signal object. Values shall not be removed from the list of written values. Before the

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

first occasion on which a given process instance writes to a given resolved signal, that process instance shall not contribute to the calculation of the resolved value for that signal.

The resolved value shall be calculated from the list of written values using the following algorithm:

1) Take a copy of the list.
2) Take any two values from the copy of the list and replace them with one value according to the truth table shown in Table 3.
3) Repeat step 2) until only a single value remains. This is the resolved value.

Table 3—Resolution table for sc_signal_resolved

	'0'	**'1'**	**'Z'**	**'X'**
'0'	'0'	'X'	'0'	'X'
'1'	'X'	'1'	'1'	'X'
'Z'	'0'	'1'	'Z'	'X'
'X'	'X'	'X'	'X'	'X'

Before the first occasion on which a given process instance writes to a given resolved signal, the value written by that process instance is effectively **'Z'** in terms of its effect on the resolution calculation. On the other hand, the default initial value for a resolved signal (as would be returned by member function **read** before the first **write**) is **'X'**. Thus, it is strongly recommended that each process instance that writes to a given resolved signal perform a write to that signal at time zero.

NOTE 1—The order in which values are passed to the function defined by the truth table in Table 3 does not affect the result of the calculation.

NOTE 2—The calculation of the resolved value is performed using the value most recently written by each and every process that writes to that particular signal object, regardless of whether the most recent write occurred in the current delta cycle, in a previous delta cycle, or at an earlier time.

NOTE 3—These same resolution semantics apply, whether the resolved signal is accessed directly by a process or is accessed indirectly through a port bound to the resolved signal.

6.13.5 Member functions

Member function **register_port** of class **sc_signal** shall be overridden in class **sc_signal_resolved**, such that the error check for multiple output ports performed by **sc_signal::register_port** is disabled for channel objects of class **sc_signal_resolved**.

Member function **write**, **operator=**, and member function **update** shall have the same behavior as the corresponding members of class **sc_signal**, except where the behavior differs for multiple writers as defined in 6.13.4.

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

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Example:

```
SC_MODULE(M) {
    sc_core::sc_signal_resolved sig;

    SC_CTOR(M) {
        SC_THREAD(T1);
        SC_THREAD(T2);
        SC_THREAD(T3);
    }

    void T1() {
                                        // Time=0 ns, no written values      sig=X
        wait(10.0, sc_core::SC_NS);
        sig = sc_dt::SC_LOGIC_0;        // Time=10 ns, written values=0      sig=0
        wait(20.0, sc_core::SC_NS);
        sig = sc_dt::SC_LOGIC_Z;        // Time=30 ns, written values=Z,Z    sig=Z
    }

    void T2() {
        wait(20.0, sc_core::SC_NS);
        sig = sc_dt::SC_LOGIC_Z;        // Time=20 ns, written values=0,Z    sig=0
        wait(30.0, sc_core::SC_NS);
        sig = sc_dt::SC_LOGIC_0;        // Time=50 ns, written values=Z,0,1  sig=X
}

    void T3() {
        wait(40.0, sc_core::SC_NS);
        sig = sc_dt::SC_LOGIC_1;        // Time=40 ns, written values=Z,Z,1  sig=1
    }
};
```

6.14 sc_in_resolved

6.14.1 Description

Class **sc_in_resolved** is a specialized port class for use with resolved signals. It is similar in behavior to port class **sc_in<sc_dt::sc_logic>** from which it is derived. The only difference is that a port of class **sc_in_resolved** shall be bound to a channel of class **sc_signal_resolved**, whereas a port of class **sc_in<sc_dt::sc_logic>** may be bound to a channel of class **sc_signal<sc_dt::sc_logic,WRITER_POLICY>** or class **sc_signal_resolved**.

6.14.2 Class definition

```
namespace sc_core {

class sc_in_resolved
: public sc_in<sc_dt::sc_logic>
{
    public:
        sc_in_resolved();
        explicit sc_in_resolved( const char* );
        virtual ~sc_in_resolved();
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        virtual void end_of_elaboration();

        virtual const char* kind() const;

    private:
        // Disabled
        sc_in_resolved( const sc_in_resolved& );
        sc_in_resolved& operator= (const sc_in_resolved& );
};

}       // namespace sc_core
```

6.14.3 Member functions

The constructors shall pass their arguments to the corresponding constructors for the base class **sc_in<sc_dt::sc_logic>**.

Member function **end_of_elaboration** shall perform an error check. It is an error if the port is not bound to a channel of class **sc_signal_resolved**.

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

NOTE—As always, the port may be bound indirectly through a port of a parent module.

6.15 sc_inout_resolved

6.15.1 Description

Class **sc_inout_resolved** is a specialized port class for use with resolved signals. It is similar in behavior to port class **sc_inout<sc_dt::sc_logic>** from which it is derived. The only difference is that a port of class **sc_inout_resolved** shall be bound to a channel of class **sc_signal_resolved**, whereas a port of class **sc_inout<sc_dt::sc_logic>** may be bound to a channel of class **sc_signal<sc_dt::sc_logic,WRITER_POLICY>** or class **sc_signal_resolved**.

6.15.2 Class definition

```
namespace sc_core {

class sc_inout_resolved
: public sc_inout<sc_dt::sc_logic>
{
    public:
        sc_inout_resolved();
        explicit sc_inout_resolved( const char* );
        virtual ~sc_inout_resolved();

        virtual void end_of_elaboration();

        sc_inout_resolved& operator= ( const sc_dt::sc_logic& );
        sc_inout_resolved& operator= ( const sc_signal_in_if<sc_dt::sc_logic>& );
        sc_inout_resolved& operator= ( const sc_port<sc_signal_in_if<sc_dt::sc_logic>, 1>& );
        sc_inout_resolved& operator= ( const sc_port<sc_signal_inout_if<sc_dt::sc_logic>, 1>& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_inout_resolved& operator= ( const sc_inout_resolved& );

        virtual const char* kind() const;

    private:
        // Disabled
        sc_inout_resolved( const sc_inout_resolved& );
};

}        // namespace sc_core
```

6.15.3 Member functions

The constructors shall pass their arguments to the corresponding constructors for the base class **sc_inout<sc_dt::sc_logic>**.

Member function **end_of_elaboration** shall perform an error check. It is an error if the port is not bound to a channel of class **sc_signal_resolved**.

The behavior of the assignment operators shall be identical to that of class **sc_inout<sc_dt::sc_logic>** but with the class name **sc_inout_resolved** substituted in place of the class name **sc_inout<sc_dt::sc_logic>** wherever appropriate.

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

NOTE—As always, the port may be bound indirectly through a port of a parent module.

6.16 sc_out_resolved

6.16.1 Description

Class **sc_out_resolved** is derived from class **sc_inout_resolved**, and it is identical to class **sc_inout_resolved** except for differences inherent in it being a derived class, for example, constructors and assignment operators. The purpose of having both classes is to allow users to express their intent, that is, **sc_out_resolved** for output pins connected to resolved signals and **sc_inout_resolved** for bidirectional pins connected to resolved signals.

6.16.2 Class definition

```
namespace sc_core {

class sc_out_resolved
: public sc_inout_resolved
{
    public:
        sc_out_resolved();
        explicit sc_out_resolved( const char* );
        virtual ~sc_out_resolved();

        sc_out_resolved& operator= ( const sc_dt::sc_logic& );
        sc_out_resolved& operator= ( const sc_signal_in_if<sc_dt::sc_logic>& );
        sc_out_resolved& operator= ( const sc_port<sc_signal_in_if<sc_dt::sc_logic>, 1>& );
        sc_out_resolved& operator= ( const sc_port<sc_signal_inout_if<sc_dt::sc_logic>, 1>& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_out_resolved& operator= ( const sc_out_resolved& );

        virtual const char* kind() const;

    private:
        // Disabled
        sc_out_resolved( const sc_out_resolved& );
};

}       // namespace sc_core
```

6.16.3 Member functions

The constructors shall pass their arguments to the corresponding constructors for the base class **sc_inout_resolved**.

The behavior of the assignment operators shall be identical to that of class **sc_inout_resolved** but with the class name **sc_out_resolved** substituted in place of the class name **sc_inout_resolved** wherever appropriate.

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

6.17 sc_signal_rv

6.17.1 Description

Class **sc_signal_rv** is a predefined primitive channel derived from class **sc_signal**. Class **sc_signal_rv** is similar to class **sc_signal_resolved**. The difference is that the argument to the base class template **sc_signal** is type **sc_dt::sc_lv<W>** instead of type **sc_dt::sc_logic**.

6.17.2 Class definition

```
namespace sc_core {

template <int W>
class sc_signal_rv
: public sc_signal<sc_dt::sc_lv<W>,SC_MANY_WRITERS>
{
    public:
        sc_signal_rv();
        explicit sc_signal_rv( const char* );
        virtual ~sc_signal_rv();

        virtual void register_port( sc_port_base&, const char* );

        virtual void write( const sc_dt::sc_lv<W>& );
        sc_signal_rv<W>& operator= ( const sc_dt::sc_lv<W>& );
        sc_signal_rv<W>& operator= ( const sc_signal_rv<W>& );

        virtual const char* kind() const;

    protected:
        virtual void update();
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    private:
        // Disabled
        sc_signal_rv( const sc_signal_rv<W>& );
};

}       // namespace sc_core
```

6.17.3 Semantics and member functions

The semantics of class **sc_signal_rv** shall be identical to the semantics of class **sc_signal_resolved** except for differences due to the fact that the value to be resolved is of type **sc_dt::sc_lv** (see 6.13.4).

The value shall be propagated through the resolved signal as an atomic value; that is, an event shall be notified, and the entire value of the vector shall be resolved and updated whenever any bit of the vector written by any process changes.

The *list of written values* shall contain values of type **sc_dt::sc_lv**, and each value of type **sc_dt::sc_lv** shall be treated atomically for the purpose of building and updating the list.

If and only if the written value differs from the previous written value (in one or more bit positions) or this is the first occasion on which the particular process has written to the particular signal object, the member function **write** shall then call the member function **request_update**.

The resolved value shall be calculated for the entire vector by applying the rule described in 6.13.4 to each bit position within the vector in turn.

The default constructor shall call the base class constructor from its initializer list as follows:

```
    sc_signal( sc_gen_unique_name( "signal_rv" ) )
```

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

6.18 sc_in_rv

6.18.1 Description

Class **sc_in_rv** is a specialized port class for use with resolved signals. It is similar in behavior to port class **sc_in<sc_dt::sc_lv<W>>** from which it is derived. The only difference is that a port of class **sc_in_rv** shall be bound to a channel of class **sc_signal_rv**, whereas a port of class **sc_in<sc_dt::sc_lv<W>>** may be bound to a channel of class **sc_signal<sc_dt::sc_lv<W>,WRITER_POLICY>** or class **sc_signal_rv**.

6.18.2 Class definition

```
namespace sc_core {

template <int W>
class sc_in_rv
: public sc_in<sc_dt::sc_lv<W>>
{
    public:
        sc_in_rv();
        explicit sc_in_rv( const char* );
        virtual ~sc_in_rv();
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        virtual void end_of_elaboration();

        virtual const char* kind() const;

    private:
        // Disabled
        sc_in_rv( const sc_in_rv<W>& );
        sc_in_rv<W>& operator= ( const sc_in_rv<W>& );
};

}       // namespace sc_core
```

6.18.3 Member functions

The constructors shall pass their arguments to the corresponding constructors for the base class **sc_in<sc_dt::sc_lv<W>>**.

Member function **end_of_elaboration** shall perform an error check. It is an error if the port is not bound to a channel of class **sc_signal_rv**.

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

NOTE—As always, the port may be bound indirectly through a port of a parent module.

6.19 sc_inout_rv

6.19.1 Description

Class **sc_inout_rv** is a specialized port class for use with resolved signals. It is similar in behavior to port class **sc_inout<sc_dt::sc_lv<W>>** from which it is derived. The only difference is that a port of class **sc_inout_rv** shall be bound to a channel of class **sc_signal_rv**, whereas a port of class **sc_inout<sc_dt::sc_lv<W>>** may be bound to a channel of class **sc_signal<sc_dt::sc_lv<W>,WRITER_POLICY>** or class **sc_signal_rv**.

6.19.2 Class definition

```
namespace sc_core {

template <int W>
class sc_inout_rv
: public sc_inout<sc_dt::sc_lv<W>>
{
    public:
        sc_inout_rv();
        explicit sc_inout_rv( const char* );
        virtual ~sc_inout_rv();

        sc_inout_rv<W>& operator= ( const sc_dt::sc_lv<W>& );
        sc_inout_rv<W>& operator= ( const sc_signal_in_if<sc_dt::sc_lv<W>>& );
        sc_inout_rv<W>& operator= ( const sc_port<sc_signal_in_if<sc_dt::sc_lv<W>>, 1>& );
        sc_inout_rv<W>& operator= ( const sc_port<sc_signal_inout_if<sc_dt::sc_lv<W>>, 1>& );
        sc_inout_rv<W>& operator= ( const sc_inout_rv<W>& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        virtual void end_of_elaboration();

        virtual const char* kind() const;

    private:
        // Disabled
        sc_inout_rv( const sc_inout_rv<W>& );
};

}        // namespace sc_core
```

6.19.3 Member functions

The constructors shall pass their arguments to the corresponding constructors for the base class **sc_inout<sc_dt::sc_lv<W>>**.

Member function **end_of_elaboration** shall perform an error check. It is an error if the port is not bound to a channel of class **sc_signal_rv**.

The behavior of the assignment operators shall be identical to that of class **sc_inout<sc_dt::sc_lv<W>>** but with the class name **sc_inout_rv** substituted in place of the class name **sc_inout<sc_dt::sc_lv<W>>** wherever appropriate.

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

NOTE—The port may be bound indirectly through a port of a parent module.

6.20 sc_out_rv

6.20.1 Description

Class **sc_out_rv** is derived from class **sc_inout_rv**, and it is identical to class **sc_inout_rv** except for differences inherent in it being a derived class, for example, constructors and assignment operators. The purpose of having both classes is to allow users to express their intent, that is, **sc_out_rv** for output pins connected to resolved vectors and **sc_inout_rv** for bidirectional pins connected to resolved vectors.

6.20.2 Class definition

```
namespace sc_core {

template <int W>
class sc_out_rv
: public sc_inout_rv<W>
{
    public:
        sc_out_rv();
        explicit sc_out_rv( const char* );
        virtual ~sc_out_rv();

        sc_out_rv<W>& operator= ( const sc_dt::sc_lv<W>& );
        sc_out_rv<W>& operator= ( const sc_signal_in_if<sc_dt::sc_lv<W>>& );
        sc_out_rv<W>& operator= ( const sc_port<sc_signal_in_if<sc_dt::sc_lv<W>>, 1>& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_out_rv<W>& operator= ( const sc_port<sc_signal_inout_if<sc_dt::sc_lv<W>>, 1>& );
        sc_out_rv<W>& operator= ( const sc_out_rv<W>& );

        virtual const char* kind() const;

    private:
        // Disabled
        sc_out_rv( const sc_out_rv<W>& );
};

}         // namespace sc_core
```

6.20.3 Member functions

The constructors shall pass their arguments to the corresponding constructors for the base class **sc_inout_rv<W>**.

The behavior of the assignment operators shall be identical to that of class **sc_inout_rv<W>** but with the class name **sc_out_rv<W>** substituted in place of the class name **sc_inout_rv<W>** wherever appropriate.

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

6.21 sc_fifo_in_if

6.21.1 Description

Class **sc_fifo_in_if** is an interface proper and is implemented by the predefined channel **sc_fifo**. Interface **sc_fifo_in_if** gives read access to a fifo channel, and it is derived from two further interfaces proper, **sc_fifo_nonblocking_in_if** and **sc_fifo_blocking_in_if**.

6.21.2 Class definition

```
namespace sc_core {

template <class T>
class sc_fifo_nonblocking_in_if
: virtual public sc_interface
{
    public:
        virtual bool nb_read( T& ) = 0;
        virtual const sc_event& data_written_event() const = 0;
};

template <class T>
class sc_fifo_blocking_in_if
: virtual public sc_interface
{
    public:
        virtual void read( T& ) = 0;
        virtual T read() = 0;
};

template <class T>
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
class sc_fifo_in_if : public sc_fifo_nonblocking_in_if<T>, public sc_fifo_blocking_in_if<T>
{
    public:
        virtual int num_available() const = 0;

    protected:
        sc_fifo_in_if();

    private:
        // Disabled
        sc_fifo_in_if( const sc_fifo_in_if<T>& );
        sc_fifo_in_if<T>& operator= ( const sc_fifo_in_if<T>& );
};

}          // namespace sc_core
```

6.21.3 Member functions

The following member functions are all pure virtual functions. The descriptions refer to the expected definitions of the functions when overridden in a channel that implements this interface. The precise semantics will be channel-specific.

Member functions **read** and **nb_read** shall return the value least recently written into the fifo and shall remove that value from the fifo such that it cannot be read again. If the fifo is empty, member function **read** shall suspend until a value has been written to the fifo, whereas member function **nb_read** shall return immediately. The return value of the function **nb_read** shall indicate whether a value was read.

When calling member function **void read(T&)** of class **sc_fifo_blocking_in_if**, the application shall be obliged to ensure that the lifetime of the actual argument extends from the time the function is called to the time the function call reaches completion. Moreover, the application shall not modify the value of the actual argument during that period.

Member function **data_written_event** shall return a reference to an event that is notified whenever a value is written into the fifo.

Member function **num_available** shall return the number of values currently available in the fifo to be read.

6.22 sc_fifo_out_if

6.22.1 Description

Class **sc_fifo_out_if** is an interface proper and is implemented by the predefined channel **sc_fifo**. Interface **sc_fifo_out_if** gives write access to a fifo channel and is derived from two further interfaces proper, **sc_fifo_nonblocking_out_if** and **sc_fifo_blocking_out_if**.

6.22.2 Class definition

```
namespace sc_core {

template <class T>
class sc_fifo_nonblocking_out_if
: virtual public sc_interface
{
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    public:
        virtual bool nb_write( const T& ) = 0;
        virtual const sc_event& data_read_event() const = 0;
};

template <class T>
class sc_fifo_blocking_out_if
: virtual public sc_interface
{
    public:
        virtual void write( const T& ) = 0;
};

template <class T>
class sc_fifo_out_if : public sc_fifo_nonblocking_out_if<T>, public sc_fifo_blocking_out_if<T>
{
    public:
        virtual int num_free() const = 0;

    protected:
        sc_fifo_out_if();

    private:
        // Disabled
        sc_fifo_out_if( const sc_fifo_out_if<T>& );
        sc_fifo_out_if<T>& operator= ( const sc_fifo_out_if<T>& );
};

}          // namespace sc_core
```

6.22.3 Member functions

The following member functions are all pure virtual functions. The descriptions refer to the expected definitions of the functions when overridden in a channel that implements this interface. The precise semantics will be channel-specific.

Member functions **write** and **nb_write** shall write the value passed as an argument into the fifo. If the fifo is full, member function **write** shall suspend until a value has been read from the fifo, whereas member function **nb_write** shall return immediately. The return value of the function **nb_write** shall indicate whether a value was written into an empty slot.

When calling member function **void write(const T&)** of class **sc_fifo_blocking_out_if**, the application shall be obliged to ensure that the lifetime of the actual argument extends from the time the function is called to the time the function call reaches completion, and moreover, the application shall not modify the value of the actual argument during that period.

Member function **data_read_event** shall return a reference to an event that is notified whenever a value is read from the fifo.

Member function **num_free** shall return the number of unoccupied slots in the fifo available to accept written values.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.23 sc_fifo

6.23.1 Description

Class **sc_fifo** is a predefined primitive channel intended to model the behavior of a fifo, that is, a first-in-first-out buffer. In this clause, *fifo* refers to an object of class **sc_fifo**. Each fifo has a number of *slots* for storing values. The number of slots is fixed when the object is constructed.

6.23.2 Class definition

```
namespace sc_core {

template <class T>
class sc_fifo
: public sc_fifo_in_if<T>, public sc_fifo_out_if<T>, public sc_prim_channel
{
    public:
        explicit sc_fifo( int size_ = 16 );
        explicit sc_fifo( const char* name_, int size_ = 16);
        virtual ~sc_fifo();

        virtual void register_port( sc_port_base&, const char* );

        virtual void read( T& );
        virtual T read();
        virtual bool nb_read( T& );
        operator T ();

        virtual void write( const T& );
        virtual bool nb_write( const T& );
        sc_fifo<T>& operator= ( const T& );

        virtual const sc_event& data_written_event() const;
        virtual const sc_event& data_read_event() const;

        virtual int num_available() const;
        virtual int num_free() const;

        virtual void print( std::ostream& = std::cout ) const;
        virtual void dump( std::ostream& = std::cout ) const;
        virtual const char* kind() const;

    protected:
        virtual void update();

    private:
        // Disabled
        sc_fifo( const sc_fifo<T>& );
        sc_fifo& operator= ( const sc_fifo<T>& );
};

template <class T>
inline std::ostream& operator<< ( std::ostream&, const sc_fifo<T>& );

}          // namespace sc_core
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.23.3 Template parameter T

The argument passed to template **sc_fifo** shall be either a C++ type for which the predefined semantics for assignment are adequate (for example, a fundamental type or a pointer) or a type **T** that obeys each of the following rules:

a) The following stream operator shall be defined and should copy the state of the object given as the second argument to the stream given as the first argument. The way in which the state information is formatted is undefined by this standard. The implementation shall use this operator in implementing the behavior of the member functions **print** and **dump**.

   std::ostream& **operator<<** (std::ostream&, const T&);

b) If the default assignment semantics are inadequate to assign the state of the object, the following assignment operator should be defined for the type **T**. The implementation shall use this operator to copy the value being written into a fifo slot or the value being read out of a fifo slot.

   const T& **operator=** (const T&);

c) If any constructor for type **T** exists, a default constructor for type **T** shall be defined.

NOTE 1—The assignment operator is not obliged to assign the complete state of the object, although it should typically do so. For example, diagnostic information may be associated with an object that is not to be propagated through the fifo.

NOTE 2—The SystemC data types proper (**sc_dt::sc_int, sc_dt::sc_logic**, and so forth) all conform to the above rule set.

NOTE 3—It is legal to pass type **sc_module*** through a fifo, although this would be regarded as an abuse of the module hierarchy and thus bad practice.

6.23.4 Constructors

explicit **sc_fifo**(int size_ = 16);

This constructor shall call the base class constructor from its initializer list as follows:

sc_prim_channel(sc_gen_unique_name("fifo"))

explicit **sc_fifo**(const char* name_, int size_ = 16);

This constructor shall call the base class constructor from its initializer list as follows:

sc_prim_channel(name_)

Both constructors shall initialize the number of slots in the fifo to the value given by the parameter **size_**. The number of slots shall be greater than zero.

6.23.5 register_port

virtual void **register_port**(sc_port_base&, const char*);

Member function **register_port** of class **sc_interface** shall be overridden in class **sc_fifo** and shall perform an error check. It is an error if more than one port of type **sc_fifo_in_if** is bound to a given fifo, and it is an error if more than one port of type **sc_fifo_out_if** is bound to a given fifo.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.23.6 Member functions for reading

virtual void **read**(T&);
virtual T **read**();
virtual bool **nb_read**(T&);

Member functions **read** and **nb_read** shall return the value least recently written into the fifo and shall remove that value from the fifo such that it cannot be read again. Multiple values may be read within a single delta cycle. The order in which values are read from the fifo shall precisely match the order in which values were written into the fifo. Values written into the fifo during the current delta cycle are not available for reading in that delta cycle but become available for reading in the immediately following delta cycle.

The value read from the fifo shall be returned as the value of the member function or as an argument passed by reference, as appropriate.

If the fifo is empty (that is, no values are available for reading), member function **read** shall suspend until the *data-written event* is notified. At that point, it shall resume (in the immediately following evaluation phase) and complete the reading of the value least recently written into the fifo before returning.

If the fifo is empty, member function **nb_read** shall return immediately without modifying the state of the fifo, without calling **request_update**, and with a return value of **false**. Otherwise, if a value is available for reading, the return value of member function **nb_read** shall be **true**.

operator T ();

The behavior of **operator T()** shall be equivalent to the following definition:

```
operator T (){ return read(); }
```

6.23.7 Member functions for writing

virtual void **write**(const T&);
virtual bool **nb_write**(const T&);

Member functions **write** and **nb_write** shall write the value passed as an argument into the fifo. Multiple values may be written within a single delta cycle. If values are read from the fifo during the current delta cycle, the empty slots in the fifo so created do not become free for the purposes of writing until the immediately following delta cycle.

If the fifo is full (that is, no free slots exist for the purposes of writing), member function **write** shall suspend until the *data-read event* is notified. At which point, it shall resume (in the immediately following evaluation phase) and complete the writing of the argument value into the fifo before returning.

If the fifo is full, member function **nb_write** shall return immediately without modifying the state of the fifo, without calling **request_update**, and with a return value of **false**. Otherwise, if a slot is free, the return value of member function **nb_write** shall be **true**.

operator=

The behavior of **operator=** shall be equivalent to the following definition:

```
sc_fifo<T>& operator= ( const T& a ) { write( a ); return *this; }
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.23.8 The update phase

Member functions **read**, **nb_read**, **write**, and **nb_write** shall complete the act of reading or writing the fifo by calling member function **request_update** of class **sc_prim_channel**.

virtual void **update**();

Member function **update** of class **sc_prim_channel** shall be overridden in class **sc_fifo** to update the number of values available for reading and the number of free slots for writing and shall cause the *data-written event* or the *data-read event* to be notified in the immediately following delta notification phase as necessary.

NOTE—If a fifo is empty and member functions **write** and **read** are both called (from the same process or from two different processes) during the evaluation phase of the same delta cycle, the write will complete in that delta cycle, but the read will suspend because the fifo is empty. The number of values available for reading will be incremented to one during the update phase, and the read will complete in the following delta cycle, returning the value just written.

6.23.9 Member functions for events

virtual const sc_event& **data_written_event**() const;

Member function **data_written_event** shall return a reference to an event, the *data-written event*, that is notified in the delta notification phase that occurs at the end of the delta cycle in which a value is written into the fifo.

virtual const sc_event& **data_read_event**() const;

Member function **data_read_event** shall return a reference to an event, the *data-read event*, that is notified in the delta notification phase that occurs at the end of the delta cycle in which a value is read from the fifo.

6.23.10 Member functions for available values and free slots

virtual int **num_available**() const;

Member function **num_available** shall return the number of values that are available for reading in the current delta cycle. The calculation shall deduct any values read during the current delta cycle but shall not add any values written during the current delta cycle.

virtual int **num_free**() const;

Member function **num_free** shall return the number of empty slots that are free for writing in the current delta cycle. The calculation shall deduct any slots written during the current delta cycle but shall not add any slots made free by reading in the current delta cycle.

6.23.11 Diagnostic member functions

virtual void **print**(std::ostream& = std::cout) const;

Member function **print** shall print a list of the values stored in the fifo and that are available for reading. They will be printed in the order they were written to the fifo and are printed to the stream passed as an argument by calling **operator<< (std::ostream&, T&)**. The formatting shall be implementation-defined.

virtual void **dump**(std::ostream& = std::cout) const;

Member function **dump** shall print at least the hierarchical name of the fifo and a list of the values stored in the fifo that are available for reading. They are printed to the stream passed as an argument. The formatting shall be implementation-defined.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

virtual const char* **kind**() const;

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

6.23.12 operator<<

template <class T>
inline std::ostream& **operator<<** (std::ostream&, const sc_fifo<T>&);

operator<< shall call member function **print** to print the contents of the fifo passed as the second argument to the stream passed as the first argument by calling operator **operator<< (std::ostream&, T&)**.

Example:

```
SC_MODULE(M) {
    sc_core::sc_fifo<int> fifo;

    SC_CTOR(M) : fifo(4) {
        SC_THREAD(T);
    }

    void T() {
        int d;
        fifo.write(1);
        fifo.print(std::cout);                  // 1
        fifo.write(2);
        fifo.print(std::cout);                  // 1 2
        fifo.write(3);
        fifo.print(std::cout);                  // 1 2 3
        std::cout << fifo.num_available();      // 0 values available to read
        std::cout << fifo.num_free();           // 1 free slot
        fifo.read(d);                           // read suspends and returns in the next delta cycle
        fifo.print(std::cout);                  // 2 3
        std::cout << fifo.num_available();      // 2 values available to read
        std::cout << fifo.num_free();           // 1 free slot
        fifo.read(d);
        fifo.print(std::cout);                  // 3
        fifo.read(d);
        fifo.print(std::cout);                  // Empty
        std::cout << fifo.num_available();      // 0 values available to read
        std::cout << fifo.num_free();           // 1 free slot
        wait(sc_core::SC_ZERO_TIME);
        std::cout << fifo.num_free();           // 4 free slots
    }
};
```

6.24 sc_fifo_in

6.24.1 Description

Class **sc_fifo_in** is a specialized port class for use when reading from a fifo. It provides functions to access conveniently certain member functions of the fifo to which the port is bound.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.24.2 Class definition

```
namespace sc_core {

template <class T>
class sc_fifo_in
: public sc_port<sc_fifo_in_if<T>,0>
{
    public:
        sc_fifo_in();
        explicit sc_fifo_in( const char* );
        virtual ~sc_fifo_in();

        void read( T& );
        T read();
        bool nb_read( T& );
        const sc_event& data_written_event() const;
        sc_event_finder& data_written() const;
        int num_available() const;
        virtual const char* kind() const;

    private:
        // Disabled
        sc_fifo_in( const sc_fifo_in<T>& );
        sc_fifo_in<T>& operator= ( const sc_fifo_in<T>& );
};

}          // namespace sc_core
```

6.24.3 Member functions

The constructors shall pass their arguments to the corresponding constructor for the base class **sc_port**.

Member functions **read**, **nb_read**, **data_written_event**, and **num_available** shall each call the corresponding member function of the object to which the port is bound using **operator->** of class **sc_port**, for example:

```
T read() { return (*this)->read(); }
```

Member function **data_written** shall return a reference to class **sc_event_finder**, where the event finder object itself shall be constructed using the member function **data_written_event** (see 5.7).

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

6.25 sc_fifo_out

6.25.1 Description

Class **sc_fifo_out** is a specialized port class for use when writing to a fifo. It provides functions to access conveniently certain member functions of the fifo to which the port is bound.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.25.2 Class definition

```
namespace sc_core {

template <class T>
class sc_fifo_out
: public sc_port<sc_fifo_out_if<T>,0>
{
    public:
        sc_fifo_out();
        explicit sc_fifo_out( const char* );
        virtual ~sc_fifo_out();

        void write( const T& );
        bool nb_write( const T& );
        const sc_event& data_read_event() const;
        sc_event_finder& data_read() const;
        int num_free() const;
        virtual const char* kind() const;

    private:
        // Disabled
        sc_fifo_out( const sc_fifo_out<T>& );
        sc_fifo_out<T>& operator= ( const sc_fifo_out<T>& );
};


}          // namespace sc_core
```

6.25.3 Member functions

The constructors shall pass their arguments to the corresponding constructor for the base class **sc_port**.

Member functions **write**, **nb_write**, **data_read_event**, and **num_free** shall each call the corresponding member function of the object to which the port is bound using **operator->** of class **sc_port**, for example:

```
    void write( const T& a ) { (*this)->write( a ); }
```

Member function **data_read** shall return a reference to class **sc_event_finder**, where the event finder object itself shall be constructed using the member function **data_read_event** (see 5.7).

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

Example:

```
// Type passed as template argument to sc_fifo<>
class U {
 public:
    U(int val = 0) {                          // If any constructor exists, a default constructor is required.
        ptr = new int;
        *ptr = val;
    }

    int get() const { return *ptr; }
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    void set(int i) { *ptr = i; }

    // Default assignment semantics are inadequate
    const U &operator=(const U &arg) {
        *(this->ptr) = *(arg.ptr);
        return *this;
    }

private:
    int *ptr;
};

// operator<< required
std::ostream &operator<<(std::ostream &os, const U &arg) { return (os << arg.get()); }

SC_MODULE(M1) {
    sc_core::sc_fifo_out<U> fifo_out;

    SC_CTOR(M1) {
        SC_THREAD(producer);
    }

    void producer() {
        U u;
        for (int i = 0; i < 4; i++) {
            u.set(i);
            bool status;
            do {
                wait(1.0, sc_core::SC_NS);
                status = fifo_out.nb_write(u);          // Non-blocking write
            } while (!status);
        }
    }
};

SC_MODULE(M2) {
    sc_core::sc_fifo_in<U> fifo_in;

    SC_CTOR(M2) {
        SC_THREAD(consumer);
        sensitive << fifo_in.data_written();
    }

    void consumer() {
        for (;;) {
            wait(fifo_in.data_written_event());
            U u;
            bool status = fifo_in.nb_read(u);
            std::cout << u << " ";                      // 0 1 2 3
        }
    }
};

SC_MODULE(Top) {
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    sc_core::sc_fifo<U> fifo;
    M1 m1;
    M2 m2;

    SC_CTOR(Top)
    : m1("m1"), m2("m2") {
        m1.fifo_out(fifo);
        m2.fifo_in(fifo);
    }
};
```

6.26 sc_mutex_if

6.26.1 Description

Class **sc_mutex_if** is an interface proper and is implemented by the predefined channel **sc_mutex**.

6.26.2 Class definition

```
namespace sc_core {

class sc_mutex_if
: virtual public sc_interface
{
    public:
        virtual int lock() = 0;
        virtual int trylock() = 0;
        virtual int unlock() = 0;

    protected:
        sc_mutex_if();

    private:
        // Disabled
        sc_mutex_if( const sc_mutex_if& );
        sc_mutex_if& operator= ( const sc_mutex_if& );
};

}          // namespace sc_core
```

6.26.3 Member functions

The behavior of the member functions of class **sc_mutex_if** is defined in class **sc_mutex**.

6.27 sc_mutex

6.27.1 Description

Class **sc_mutex** is a predefined channel intended to model the behavior of a mutual exclusion lock used to control access to a resource shared by concurrent processes. A *mutex* is an object of class **sc_mutex**. A mutex shall be in one of two exclusive states: *unlocked* or *locked*. Only one process can lock a given mutex at one time. A mutex can only be unlocked by the particular process instance that locked the mutex but may be locked subsequently by a different process.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.27.2 Class definition

```
namespace sc_core {

class sc_mutex
: public sc_mutex_if, public sc_object
{
    public:
        sc_mutex();
        explicit sc_mutex( const char* );

        virtual int lock();
        virtual int trylock();
        virtual int unlock();

        virtual const char* kind() const;

    private:
        // Disabled
        sc_mutex( const sc_mutex& );
        sc_mutex& operator= ( const sc_mutex& );
};

}          // namespace sc_core
```

6.27.3 Constructors

sc_mutex();

This constructor shall call the base class constructor from its initializer list as follows:

```
sc_object( sc_gen_unique_name( "mutex" ) )
```

explicit **sc_mutex**(const char* name_);

This constructor shall call the base class constructor from its initializer list as follows:

```
sc_object( name_ )
```

Both constructors shall unlock the mutex.

6.27.4 Member functions

virtual int **lock**();

If the mutex is unlocked, member function **lock** shall lock the mutex and return.

If the mutex is locked, member function **lock** shall suspend until the mutex is unlocked (by another process). At that point, it shall resume and attempt to lock the mutex by applying these same rules again.

Member function **lock** shall unconditionally return the value **0**.

If multiple processes attempt to lock the mutex in the same delta cycle, the choice of which process instance is given the lock in that delta cycle shall be non-deterministic; that is, it will rely on the order in which processes are resumed within the evaluation phase.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

virtual int **trylock**();

If the mutex is unlocked, member function **trylock** shall lock the mutex and shall return the value **0**.

If the mutex is locked, member function **trylock** shall immediately return the value –**1**. The mutex shall remain locked.

virtual int **unlock**();

If the mutex is unlocked, member function **unlock** shall return the value –1. The mutex shall remain unlocked.

If the mutex was locked by a process instance other than the calling process, member function **unlock** shall return the value –1. The mutex shall remain locked.

If the mutex was locked by the calling process, member function **unlock** shall unlock the mutex and shall return the value **0**. If processes are suspended and are waiting for the mutex to be unlocked, the lock shall be given to exactly one of these processes (the choice of process instance being non-deterministic) while the remaining processes shall suspend again. This shall be accomplished within a single evaluation phase; that is, an implementation shall use immediate notification to signal the act of unlocking a mutex to other processes.

virtual const char* **kind**() const;

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

6.28 sc_semaphore_if

6.28.1 Description

Class **sc_semaphore_if** is an interface proper and is implemented by the predefined channel **sc_semaphore**.

6.28.2 Class definition

```
namespace sc_core {

class sc_semaphore_if
: virtual public sc_interface
{
    public:
        virtual int wait() = 0;
        virtual int trywait() = 0;
        virtual int post() = 0;
        virtual int get_value() const = 0;

    protected:
        sc_semaphore_if();

    private:
        // Disabled
        sc_semaphore_if( const sc_semaphore_if& );
        sc_semaphore_if& operator= ( const sc_semaphore_if& );
};

}          // namespace sc_core
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.28.3 Member functions

The behavior of the member functions of class **sc_semaphore_if** is defined in class **sc_semaphore**.

6.29 sc_semaphore

6.29.1 Description

Class **sc_semaphore** is a predefined channel intended to model the behavior of a software semaphore used to provide limited concurrent access to a shared resource. A semaphore has an integer value, the *semaphore value*, which is set to the permitted number of concurrent accesses when the semaphore is constructed.

6.29.2 Class definition

```
namespace sc_core {

class sc_semaphore
: public sc_semaphore_if, public sc_object
{
    public:
        explicit sc_semaphore( int );
        sc_semaphore( const char*, int );

        virtual int wait();
        virtual int trywait();
        virtual int post();
        virtual int get_value() const;

        virtual const char* kind() const;

    private:
        // Disabled
        sc_semaphore( const sc_semaphore& );
        sc_semaphore& operator= ( const sc_semaphore& );
};

}        // namespace sc_core
```

6.29.3 Constructors

explicit **sc_semaphore**(int);

This constructor shall call the base class constructor from its initializer list as follows:

```
sc_object( sc_gen_unique_name( "semaphore" ) )
```

sc_semaphore(const char* name_, int);

This constructor shall call the base class constructor from its initializer list as follows:

```
sc_object( name_ )
```

Both constructors shall set the *semaphore value* to the value of the **int** parameter, which shall be non-negative.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.29.4 Member functions

virtual int **wait**();

If the semaphore value is greater than **0**, member function **wait** shall decrement the semaphore value and return.

If the semaphore value is equal to **0**, member function **wait** shall suspend until the semaphore value is incremented (by another process). At that point, it shall resume and attempt to decrement the semaphore value by applying these same rules again.

Member function **wait** shall unconditionally return the value **0**.

The semaphore value shall not become negative. If multiple processes attempt to decrement the semaphore value in the same delta cycle, the choice of which process instance decrements the semaphore value and which processes suspend shall be non-deterministic; that is, it will rely on the order in which processes are resumed within the evaluation phase.

virtual int **trywait**();

If the semaphore value is greater than **0**, member function **trywait** shall decrement the semaphore value and shall return the value **0**.

If the semaphore value is equal to **0**, member function **trywait** shall immediately return the value –1 without modifying the semaphore value.

virtual int **post**();

Member function **post** shall increment the semaphore value. If processes exist that are suspended and are waiting for the semaphore value to be incremented, exactly one of these processes shall be permitted to decrement the semaphore value (the choice of process instance being non-deterministic) while the remaining processes shall suspend again. This shall be accomplished within a single evaluation phase; that is, an implementation shall use immediate notification to signal the act of incrementing the semaphore value to any waiting processes.

Member function **post** shall unconditionally return the value **0**.

virtual int **get_value**() const;

Member function **get_value** shall return the semaphore value.

virtual const char* **kind**() const;

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

NOTE 1—The semaphore value may be decremented and incremented by different processes.

NOTE 2—The semaphore value may exceed the value set by the constructor.

6.30 sc_event_queue

6.30.1 Description

Class **sc_event_queue** represents an event queue. Like class **sc_event**, an event queue has a member function **notify**. Unlike an **sc_event**, an event queue is a hierarchical channel and can have multiple notifications pending.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.30.2 Class definition

```
namespace sc_core {

class sc_event_queue_if
: public virtual sc_interface
{
    public:
        virtual void notify( double , sc_time_unit ) = 0;
        virtual void notify( const sc_time& ) = 0;
        virtual void cancel_all() = 0;
};

class sc_event_queue
: public sc_event_queue_if , public sc_module
{
     public:
        sc_event_queue( sc_module_name name_ =
            sc_module_name(sc_gen_unique_name("event_queue")));
        ~sc_event_queue();

        virtual const char* kind() const;

        virtual void notify( double , sc_time_unit );
        virtual void notify( const sc_time& );
        virtual void cancel_all();

        virtual const sc_event& default_event() const;
};

}          // namespace sc_core
```

6.30.3 Constraints on usage

Class **sc_event_queue** is a hierarchical channel, and thus, **sc_event_queue** objects can only be constructed during elaboration.

NOTE—An object of class **sc_event_queue** cannot be used in most contexts requiring an **sc_event** but can be used to create static sensitivity because it implements member function **sc_interface::default_event**.

6.30.4 Constructors

sc_event_queue(sc_module_name name_= sc_module_name(sc_gen_unique_name("event_queue")));

This constructor shall pass the module name argument through to the constructor for the base class **sc_module**.

6.30.5 kind

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

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

6.30.6 Member functions

virtual void **notify**(double , sc_time_unit);
virtual void **notify**(const sc_time&);

A call to member function **notify** with an argument that represents a zero time shall cause a delta notification on the default event.

A call to function **notify** with an argument that represents a non-zero time shall cause a timed notification on the default event at the given time, expressed relative to the simulation time when function **notify** is called. In other words, the value of the time argument is added to the current simulation time to determine the time at which the event will be notified.

If function **notify** is called when there is a already one or more notifications pending, the new notification shall be queued in addition to the pending notifications. Each queued notification shall occur at the time determined by the semantics of function **notify**, irrespective of the order in which the calls to **notify** are made.

The default event shall not be notified more than once in any one delta cycle. If multiple notifications are pending for the same delta cycle, those notifications shall occur in successive delta cycles. If multiple timed notification are pending for the same simulation time, those notifications shall occur in successive delta cycles starting with the first delta cycle at that simulation time step and with no gaps in the delta cycle sequence.

virtual void **cancel_all**();

Member function **cancel_all** shall immediately delete every pending notification for this event queue object including both delta and timed notifications, but it shall have no effect on other event queue objects.

virtual const sc_event& **default_event**() const;

Member function **default_event** shall return a reference to the default event.

The mechanism used to queue notifications shall be implementation-defined, with the proviso that an event queue object must provide a single default event that is notified once for every call to member function **notify**.

NOTE—Event queue notifications are anonymous in the sense that the only information carried by the default event is the time of notification. A process instance sensitive to the default event cannot tell which call to function **notify** caused the notification.

Example:

```
SC_MODULE(Mod) {
    sc_core::sc_event_queue EQ;
    ...

    SC_CTOR(Mod) {
        SC_THREAD(T);
        SC_METHOD(M);
        sensitive << EQ;
        dont_initialize();
    }

    void T() {
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        EQ.notify(2.0, sc_core::SC_NS);          // M runs at time 2ns
        EQ.notify(1.0, sc_core::SC_NS);          // M runs at time 1ns, 1st or 2nd delta cycle
        EQ.notify(sc_core::SC_ZERO_TIME);        // M runs at time 0ns
        EQ.notify(1.0, sc_core::SC_NS);          // M runs at time 1ns, 2nd or 1st delta cycle
    }
};
```

6.31 sc_stub, sc_unbound, sc_tie

```
namespace sc_core
{

class sc_stub implementation-defined

implementation-defined sc_unbound;

template <typename T>
implementation-defined sc_tie::value(const T&);

} // namespace sc_core
```

Class **sc_stub** shall define a predefined channel that acts as a stub. The class shall be used for the definition of **sc_unbound** and **sc_tie**.

The static object **sc_unbound** shall define a predefined channel that represents an unbound connection. Each time **sc_unbound** is used in an application, a new predefined channel shall be created with a name with prefix “sc_unbound”, followed by an underscore and series of one or more decimal digits from the character set 0–9.

It shall be error to bind **sc_unbound** to a port that is not of type **sc_port** <**sc_signal_inout_if**<T>>. It shall be allowed to read and write to this channel. The value read from the channel is implementation-defined. Values written to the channel shall be ignored. The channel shall not notify an event when values are written.

NOTE—An application cannot not use **sc_unbound** for input ports of type **sc_port**<**sc_signal_in_if**<T>> because its value is undefined.

The function **sc_tie**::**value** shall return a predefined channel to tie the port to the specified value of type **T**. The type of the value shall be compatible with the type of the associated channel to which the port is bound. Each time **sc_tie** is used in an application, a new predefined channel shall be created with a name with prefix “sc_tie”, followed by an underscore and series of one or more decimal digits from the character set 0–9.

It shall be error to bind **sc_tie** to a port that is not of type **sc_port**<**sc_signal_in_if**<T>> or **sc_port**<**sc_signal_inout_if**<T>>. It shall be allowed to read and write to this channel. The value read from the channel shall correspond to the specified predefined types. Values written to the channel shall be ignored. The channel shall not notify an event when values are written.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7. SystemC data types

7.1 Introduction

All native C++ types are supported within a SystemC application. SystemC provides additional data type classes within the **sc_dt** namespace to represent values with application-specific word lengths applicable to digital hardware. These data types are referred to as *SystemC data types*.

The SystemC data type classes consist of the following:

- *Limited-precision integers*, which are classes derived from class **sc_int_base**, class **sc_uint_base**, or instances of such classes. A limited-precision integer shall represent a signed or unsigned integer value at a precision limited by its underlying native C++ representation and its specified word length.
- *Finite-precision integers*, which are classes derived from class **sc_signed**, class **sc_unsigned**, or instances of such classes. A finite-precision integer shall represent a signed or unsigned integer value at a precision limited only by its specified word length.
- *Finite-precision fixed-point types*, which are classes derived from class **sc_fxnum** or instances of such classes. A finite-precision fixed-point type shall represent a signed or unsigned fixed-point value at a precision limited only by its specified word length, integer word length, quantization mode, and overflow mode.
- *Limited-precision fixed-point types*, which are classes derived from class **sc_fxnum_fast** or instances of such classes. A limited-precision fixed-point type shall represent a signed or unsigned fixed-point value at a precision limited by its underlying native C++ floating-point representation and its specified word length, integer word length, quantization mode, and overflow mode.
- *Variable-precision fixed-point type*, which is the class **sc_fxval**. A variable-precision fixed-point type shall represent a fixed-point value with a precision that may vary over time and is not subject to quantization or overflow.
- *Limited variable-precision fixed-point type*, which is the class **sc_fxval_fast**. A limited variable-precision fixed-point type shall represent a fixed-point value with a precision that is limited by its underlying C++ floating-point representation and that may vary over time and is not subject to quantization or overflow.
- *Single-bit logic types* implement a four-valued logic data type with states *logic 0*, *logic 1*, *high-impedance*, and *unknown* and shall be represented by the symbols **'0'**, **'1'**, **'X'**, and **'Z'**, respectively. The lowercase symbols **'x'** and **'z'** are acceptable alternatives for **'X'** and **'Z'**, respectively, as character literals assigned to single-bit logic types.
- *Bit vectors*, which are classes derived from class **sc_bv_base**, or instances of such classes. A bit vector shall implement a multiple bit data type, where each bit has a state of *logic 0* or *logic 1* and is represented by the symbols **'0'** or **'1'**, respectively.
- *Logic vectors*, which are classes derived from class **sc_lv_base**, or instances of such classes. A logic vector shall implement a multiple-bit data type, where each bit has a state of *logic 0*, *logic 1*, *high-impedance,* or *unknown* and is represented by the symbols **'0'**, **'1'**, **'X'**, or **'Z'**. The lowercase symbols **'x'** and **'z'** are acceptable alternatives for **'X'** and **'Z'**, respectively, within string literals assigned to logic vectors.

Apart from the single-bit logic types, the variable-precision fixed-point types, and the limited variable-precision fixed-point types, the classes within each category are organized as an object-oriented hierarchy with common behavior defined in base classes. A class template shall be derived from each base class by the implementation such that applications can specify word lengths as template arguments.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The term *fixed-point type* is used in this standard to refer to any finite-precision fixed-point type or limited-precision fixed-point type. The variable-precision and limited variable-precision fixed-point types are fixed-point types only in the restricted sense that they store a representation of a fixed-point value and can be mixed with other fixed-point types in expressions, but they are not fixed-point types in the sense that they do not model quantization or overflow effects and are not intended to be used directly by an application.

The term *numeric type* is used in this standard to refer to any limited-precision integer, finite-precision integer, finite-precision fixed-point type, or limited-precision fixed-point type. The term *vector* is used to refer to any bit vector or logic vector. The word length of a numeric type or vector object shall be set when the object is initialized and shall not subsequently be altered. Each bit within a word shall have an index. The right-hand bit shall have index **0** and is the least-significant bit for numeric types. The index of the left-hand bit shall be the word length minus **1**.

The limited-precision signed integer base class is **sc_int_base**. The limited-precision unsigned integer base class is **sc_uint_base**. The corresponding class templates are **sc_int** and **sc_uint**, respectively.

The finite-precision signed integer base class is **sc_signed**. The finite-precision unsigned integer base class is **sc_unsigned**. The corresponding class templates are **sc_bigint** and **sc_biguint**, respectively.

The signed finite-precision fixed-point base class is **sc_fix**. The unsigned finite-precision fixed-point base class is **sc_ufix**. Both base classes are derived from class **sc_fxnum**. The corresponding class templates are **sc_fixed** and **sc_ufixed**, respectively.

The signed limited-precision fixed-point base class is **sc_fix_fast**. The unsigned limited-precision fixed-point base class is **sc_ufix_fast**. Both base classes are derived from class **sc_fxnum_fast**. The corresponding class templates are **sc_fixed_fast** and **sc_ufixed_fast**, respectively.

The variable-precision fixed-point class is **sc_fxval**. The limited variable-precision fixed-point class is **sc_fxval_fast**. These two classes are used as the operand types and return types of many fixed-point operations.

The bit vector base class is **sc_bv_base**. The corresponding class template is **sc_bv**.

The logic vector base class is **sc_lv_base**. The corresponding class template is **sc_lv**.

The single-bit logic type is **sc_logic**.

It is recommended that applications create SystemC data type objects using the class templates given in this clause (for example, **sc_int**) rather than the untemplated base classes (for example, **sc_int_base**).

The relationships between the SystemC data type classes are shown in Table 4.

Table 4—SystemC data types

Class template	Base class	Generic base class	Representation	Precision
sc_int	sc_int_base	sc_value_base	signed integer	limited
sc_uint	sc_uint_base	sc_value_base	unsigned integer	limited
sc_bigint	sc_signed	sc_value_base	signed integer	finite
sc_biguint	sc_unsigned	sc_value_base	unsigned integer	finite
sc_fixed	sc_fix	sc_fxnum	signed fixed-point	finite

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 4—SystemC data types *(continued)*

Class template	Base class	Generic base class	Representation	Precision
sc_ufixed	sc_ufix	sc_fxnum	unsigned fixed-point	finite
sc_fixed_fast	sc_fix_fast	sc_fxnum_fast	signed fixed-point	limited
sc_ufixed_fast	sc_ufix_fast	sc_fxnum_fast	unsigned fixed-point	limited
		sc_fxval	fixed-point	variable
		sc_fxval_fast	fixed-point	limited-variable
	sc_logic		single bit	
sc_bv	sc_bv_base		bit vector	
sc_lv	sc_lv_base		logic vector	

7.2 Common characteristics

7.2.1 Overview

Subclause 7.2 specifies some common characteristics of the SystemC data types, such as common operators and functions, and should be taken as specifying a set of obligations on the implementation to provide operators and functions with the given behavior. In some cases the implementation has some flexibility with regard to how the given behavior is implemented. The remainder of Clause 7 gives a detailed definition of the SystemC data type classes.

An underlying principle is that native C++ integer and floating-point types, C++ string types, and SystemC data types may be mixed in expressions.

Equality and bitwise operators can be used for all SystemC data types. Arithmetic and relational operators can be used with the numeric types only. The semantics of the equality operators, bitwise operators, arithmetic operators, and relational operators are the same in SystemC as in C++.

User-defined conversions supplied by the implementation support translation from SystemC types to C++ native types and other SystemC types.

Bit-select, part-select, and concatenation operators return an instance of a proxy class. The term *proxy class* is used in this standard to refer to a class whose purpose is to represent a SystemC data type object within an expression and which provides additional operators or features not otherwise present in the represented object. An example is a proxy class that allows an **sc_int** variable to be used as if it were a C++ array of **bool** and to distinguish between its use as an rvalue or an lvalue within an expression. Instances of proxy classes are only intended to be used within the expressions that create them. An application should not call a proxy class constructor to create a named object and should not declare a pointer or reference to a proxy class. It is strongly recommended that an application avoid the use of a proxy class as the return type of a function because the lifetime of the object to which the proxy class refers may not extend beyond the function return statement.

NOTE 1—The bitwise shift left or shift right operation has no meaning for a single-bit logic type and is undefined.

NOTE 2—The term *user-defined conversions* in this context has the same meaning as in the C++ standard. It applies to type conversions of class objects by calling constructors and conversion functions that are used for implicit type conversions and explicit type conversions.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

NOTE 3—Care should be taken when mixing signed and unsigned numeric types in expressions that use implicit type conversions since an implementation is not required to issue a warning if the polarity of a converted value is changed.

7.2.2 Initialization and assignment operators

Overloaded constructors shall be provided by the implementation for all integer (limited-precision integer and finite-precision integer) class templates that allow initialization with an object of any SystemC data type.

Overloaded constructors shall be provided for all vector (bit vector and logic vector) class templates that allow initialization with an object of any SystemC integer or vector data type.

Overloaded constructors shall be provided for all finite-precision fixed-point and limited precision fixed-point class templates that allow initialization with an object of any SystemC integer data type.

All SystemC data type classes shall define a copy constructor that creates a copy of the specified object with the same value and the same word length.

Overloaded assignment operators and constructors shall perform direct or indirect conversion between types. The data type base classes may define a restricted set of constructors and assignment operators that only permit direct initialization from a subset of the SystemC data types. As a general principle, data type class template constructors may be called implicitly by an application to perform conversion from other types since their word length is specified by a template argument. On the other hand, the data type base class constructors with a single parameter of a different type should only be called explicitly since the required word length is not specified.

If the target of an assignment operation has a word length that is insufficient to hold the value assigned to it, the left-hand bits of the value stored shall be truncated to fit the target word length. If truncation occurs, an implementation may generate a warning but is not obliged to do so, and an application can in any case disable such a warning (see 3.3.6).

If a data type object or string literal is assigned to a target having a greater word length, the value shall be extended with additional bits at its left-hand side to match the target word length. Extension of a signed numeric type shall preserve both its sign and magnitude and is referred to as *sign extension*. Extension of all other types shall insert bits with a value of logic 0 and is referred to as *zero extension*.

Assignment of a fixed-point type to an integer type shall use the integer component only; any fractional component is discarded.

Assignment of a value with a word length greater than 1 to a single-bit logic type shall be an error.

NOTE—An integer literal is always treated as unsigned unless prefixed by a minus symbol. An unsigned integer literal will always be extended with leading zeros when assigned to a data type object having a larger word length, regardless of whether the object itself is signed or unsigned.

7.2.3 Precision of arithmetic expressions

The type of the value returned by any arithmetic expression containing only limited-precision integers or limited-precision integers and native C++ integer types shall be an implementation-defined C++ integer type with a maximum word length of 64 bits. The action taken by an implementation if the precision required by the return value exceeds 64 bits is undefined and the value is implementation-dependent.

The value returned by any arithmetic expression containing only finite-precision integers or finite-precision integers and any combination of limited-precision or native C++ integer types shall be a finite-precision integer with a word-length sufficient to contain the value with no loss of accuracy.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The value returned by any arithmetic expression containing any fixed-point type shall be a variable-precision or limited variable-precision fixed-point type (see 7.10.5).

Applications should use explicit type casts within expressions combining multiple types where an implementation does not provide overloaded operators with signatures exactly matching the operand types.

Example:

```
int i = 10;
sc_dt::int64 i64 = 100;                          // long long int
sc_dt::sc_int<16> sci = 2;
sc_dt::sc_bigint<16> bi = 20;
float f = 2.5;
sc_dt::sc_fixed<16, 8> scf = 2.5;
(i * sci);                                       // Ambiguous
(i * static_cast<sc_dt::int_type>(sci));         // Implementation-defined C++ integer
(i * bi);                                        // 48-bit finite-precision integer (assumes int = 32 bits)
(i64 * bi);                                      // 80-bit finite-precision integer
(f * bi);                                        // Ambiguous
(static_cast<int>(f) * bi);                      // 48-bit finite-precision integer (assumes int = 32 bits)
(scf * sci);                                     // Variable-precision fixed-point type
```

7.2.4 Base class default word length

```
namespace sc_dt {
    enum { SC_NOW, SC_LATER };
}
```

The default word length of a data type base class shall be used where its default constructor is called (implicitly or explicitly). The default word length shall be set by the *length parameter* in context at the point of construction. A length parameter may be brought into context by creating a length context object. Length contexts shall have local scope and by default be activated immediately. Once activated, they shall remain in effect for as long as they are in scope, or until another length context is activated. Activation of a length context shall be deferred if its second constructor argument is SC_LATER (the default value is SC_NOW). A deferred length context can be activated by calling its member function **begin**.

Length contexts shall be managed by a global length context stack. When a length context is activated, it shall be placed at the top of the stack. A length context may be deactivated and removed from the top of the stack by calling its member function **end**. The member function **end** shall only be called for the length context currently at the top of the context stack. A length context is implicitly deactivated and removed from the stack when it goes out of scope. A deferred length context that has been activated by calling its member function **begin** should be explicitly deactivated and removed from the stack by calling its member function **end**. The current context shall always be the length context at the top of the stack.

A length context shall only be activated once. An active length context shall only be deactivated once.

The classes **sc_length_param** and **sc_length_context** shall be used to create length parameters and length contexts, respectively, for SystemC integers and vectors.

In addition to the word length, the fixed-point types shall have default integer word length and mode attributes. These shall be set by the *fixed-point type parameter* in context at the point of construction. A fixed-point type parameter shall be brought into context by creating a *fixed-point type context object*. The use of a fixed-point type context shall follow the same rules as a length context. A stack for fixed-point type contexts with the same characteristics as the length context stack shall exist.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The classes **sc_fxtype_params** and **sc_fxtype_context** shall be used to create fixed-point type parameters and fixed-point type contexts, respectively.

Example:

```
sc_dt::sc_length_param length10(10);
sc_dt::sc_length_context cntxt10(length10);                    // length10 now in context
sc_dt::sc_int_base int_array[2];                               // Array of 10-bit integers
sc_core::sc_signal<sc_dt::sc_int_base> S1;                     // Signal of 10-bit integer
{
    sc_dt::sc_length_param length12(12);
    sc_dt::sc_length_context cntxt12(length12, sc_dt::SC_LATER);       // cntxt12 deferred
    sc_dt::sc_length_param length14(14);
    sc_dt::sc_length_context cntxt14(length14, sc_dt::SC_LATER);       // cntxt14 deferred
    sc_dt::sc_uint_base var1;                                  // length 10
    cntxt12.begin();                                           // Bring length12 into context
    sc_dt::sc_uint_base var2;                                  // length 12
    cntxt14.begin();                                           // Bring length14 into context
    sc_dt::sc_uint_base var3;                                  // length 14
    cntxt14.end();                                             // end cntx14, cntx12 restored
    sc_dt::sc_bv_base var4;                                    // length 12
}                                                              // cntxt12 out of scope, cntx10 restored
sc_dt::sc_bv_base var5; // length 10
```

NOTE 1—The context stacks allow a default context to be locally replaced by an alternative context and subsequently restored.

NOTE 2—An activated context remains active for the lifetime of the context object or until it is explicitly deactivated. A context can therefore affect the default parameters of data type objects created outside of the function in which it is activated. An application should ensure that any contexts created or activated within functions whose execution order is non-deterministic do not result in temporal ordering dependencies in other parts of the application. Failure to meet this condition could result in behavior that is implementation-dependent.

7.2.5 Word length

The word length (a positive integer indicating the number of bits) of a SystemC integer, vector, part-select, or concatenation shall be returned by the member function **length**.

7.2.6 Bit-select

Bit-selects are instances of a proxy class that reference the bit at the specified position within an associated object that is a SystemC numeric type or vector.

The C++ subscript operator (**operator[]**) shall be overloaded by the implementation to create a bit-select when called with a single non-negative integer argument specifying the bit position. It shall be an error if the specified bit position is outside the bounds of its numeric type or vector object.

User-defined conversions shall allow bit-selects to be used in expressions where a **bool** object operand is expected. A bit-select of an lvalue may be used as an rvalue or an lvalue. A bit-select of an rvalue shall only be used as an rvalue.

A bit-select or a **bool** value may be assigned to an lvalue bit-select. The assignment shall modify the state of the selected bit within the associated numeric type or vector object represented by the lvalue. An application shall not assign a value to an rvalue bit-select.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Bit-selects for integer, bit vector, and logic vector types shall have an explicit **to_bool** conversion function that returns the state of the selected bit.

Example:

```
sc_dt::sc_int<4> I1;                // 4 bit signed integer
I1[1] = true;                       // Selected bit used as lvalue
bool b0 = I1[0].to_bool();          // Selected bit used as rvalue
```

NOTE 1—Bit-selects corresponding to lvalues and rvalues of a particular type are themselves objects of two distinct classes.

NOTE 2—A bit-select class can contain user-defined conversions for both implicit and explicit conversion of the selected bit value to **bool**.

7.2.7 Part-select

Part-selects are instances of a proxy class that provide access to a contiguous subset of bits within an associated object that is a numeric type or vector.

The member function **range(int , int)** of a numeric type, bit vector, or logic vector shall create a part-select. The two non-negative integer arguments specify the left- and right-hand index positions. A part-select shall provide a reference to a word within its associated object, starting at the left-hand index position and extending to, and including, the right-hand index position. It shall be an error if the left-hand index position or right-hand index position lies outside the bounds of the object. It shall be an error for the left-hand index to be less than the right-hand index.

The C++ function call operator (**operator()**) shall be overloaded by the implementation to create a part-select and may be used as a direct replacement for the **range** function.

User-defined conversions shall allow a part-select to be used in expressions where the expected operand is an object of the numeric type or vector type associated with the part-select, subject to certain constraints (see 7.5.7.3, 7.6.8.3, and 7.9.8.3). A part-select of an lvalue may be used as an rvalue or as an lvalue. A part-select of an rvalue shall only be used as an rvalue.

Integer part-selects may be directly assigned to an object of any other SystemC data type, with the exception of bit-selects. Fixed-point part-selects may be directly assigned to any SystemC integer or vector, any part-select or any concatenation. Vector part-selects may only be directly assigned to a vector, vector part-select, or vector concatenation (assignments to other types are ambiguous or require an explicit conversion).

The bits within a part-select do not reflect the sign of their associated object and shall be taken as representing an unsigned binary number when converted to a numeric value. Assignments of part-selects to a target having a greater word length shall be zero extended, regardless of the type of their associated object.

Example:

```
sc_dt::sc_int<8> I2 = 2;                // "0b00000010"
I2.range(3,2) = I2.range(1,0);          // "0b00001010"
sc_dt::sc_int<8> I3 = I2.range(3,0);    // "0b00001010"
                                        // Zero-extended to 8 bits
sc_dt::sc_bv<8> b1 = "0b11110000";
```

NOTE 1—A part-select cannot be used to reverse the bit-order of a limited-precision integer type.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

NOTE 2—Part-selects corresponding to lvalues and rvalues of a particular type are themselves objects of two distinct classes.

NOTE 3—A part-select is not required to be an acceptable replacement where an object reference operand is expected. If an implementation provides a mechanism to allow such replacements (for example, by defining the appropriate overloaded member functions), it is not required to do so for all data types.

7.2.8 Concatenation

Concatenations are instances of a proxy class that reference the bits within multiple objects as if they were part of a single aggregate object.

The **concat**(arg0 , arg1) function shall create a concatenation. The *concatenation arguments* (arg0 and arg1) may be two SystemC integer, vector, bit-select, part-select, or concatenation objects. The C++ comma operator (**operator,**) shall also be overloaded to create a concatenation and may be used as a direct replacement for the **concat** function.

The type of a concatenation argument shall be a *concatenation base type*, or it shall be derived from a concatenation base type. An implementation shall provide a common concatenation base type for all SystemC integers and a common concatenation base type for all vectors. The concatenation base type of bit-select and part-select concatenation arguments is the same as their associated integer or vector objects. The concatenation arguments may be any combination of two objects having the same concatenation base type. A concatenation object shall have the same concatenation base type as the concatenation arguments passed to the function that created the object. The set of permissible concatenation arguments for a given concatenation base type consists of the following:

a) Objects whose base class or concatenation base type matches the given concatenation base type
b) Bit-selects of item a)
c) Part-selects of item a)
d) Concatenations of item a) and/or item b) and/or item c) in any combination

When both concatenation arguments are lvalues, the concatenation shall be an lvalue. If any concatenation argument is an rvalue, the concatenation shall be an rvalue.

A single concatenation argument may be a **bool** value when the other argument is a SystemC integer, vector, bit-select, part-select, or concatenation object. The resulting concatenation shall be an rvalue.

An expression may be assigned to an lvalue concatenation if the base type of the expression return value is the same as the base type of the lvalue concatenation. If the word length of a value assigned to a concatenation with a signed base type is smaller than the word length of the concatenation, the value shall be sign-extended to match the word length of the concatenation. Assignments to concatenations of all other numeric types and vectors shall be zero-extended (if required). Assignment to a concatenation shall update the values of the objects specified by its concatenation arguments.

A concatenation may be assigned to an object whose base class is the same as the concatenation base type. Where a concatenation is assigned to a target having a greater word length than the concatenation, it is zero-extended to the target length. When a concatenation is assigned to a target having a shorter word length than the concatenation, the left-hand bits of the value shall be truncated to fit the target word length. If truncation occurs, an implementation may generate a warning but is not obliged to do so, and an application can in any case disable such a warning (see 3.3.6).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Examples:

The following concatenations are well-formed:

```
sc_dt::sc_uint<8> U1 = 2;                                   // "0b00000010"
sc_dt::sc_uint<2> U2 = 1;                                   // "0b01"
sc_dt::sc_uint<8> U3 = (true, U1.range(3, 0), U2, U2[0]);   // U3 = "0b10010011"
                                                            // Base class same as concatenation base type
(U2[0], U1[0], U1.range(7, 1)) = (U1[7], U1);               // Copies U1[7] to U2[0], U1 rotated left
concat(U2[0], concat(U1[0], U1.range(7, 1))) = concat(U1[7], U1);
                                                            // Same as previous example but using concat
```

The following concatenations are ill-formed:

```
sc_dt::sc_bv<8> Bv1;
(Bv1, U1) = "0xffff";                                       // Bv1 and U1 do not share common base type

bool C1 = true; bool C2 = false;
U2 = (C1, C1);                                              // Cannot concatenate 2 bool objects
(C1, I1) = "0x1ff";                                         // Bool concatenation argument creates rvalue
```

NOTE 1—Parentheses are required around the concatenation arguments when using the C++ comma operator because of its low operator precedence.

NOTE 2—An implementation is not required to support bit-selects and part-selects of concatenations.

NOTE 3—Concatenations corresponding to lvalues and rvalues of a particular type are themselves objects of two distinct classes.

7.2.9 Reduction operators

The reduction operators shall perform a sequence of bitwise operations on a SystemC integer or vector to produce a **bool** result. The first step shall be a boolean operation applied to the first and second bits of the object. The boolean operation shall then be re-applied using the previous result and the next bit of the object. This process shall be repeated until every bit of the object has been processed. The value returned shall be the result of the final boolean operation. The following reduction operators shall be provided:

a) **and_reduce** performs a bitwise AND between all bits.
b) **nand_reduce** performs a bitwise NAND between all bits.
c) **or_reduce** performs a bitwise OR between all bits.
d) **nor_reduce** performs a bitwise NOR between all bits.
e) **xor_reduce** performs a bitwise XOR between all bits.
f) **xnor_reduce** performs a bitwise XNOR between all bits.

7.2.10 Integer conversion

All SystemC data types shall provide an assignment operator that can accept a C++ integer value. A signed value shall be sign-extended to match the length of the SystemC data type target.

SystemC data types shall provide member functions for explicit type conversion to C++ integer types as follows:

a) **to_int** converts to native C++ **int** type.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

b) **to_uint** converts to native C++ **unsigned** type.
c) **to_long** converts to native C++ **long** type.
d) **to_ulong** converts to native C++ **unsigned long** type.
e) **to_uint64()** converts to a native C++ unsigned integer type having a word length of 64 bits.
f) **to_int64()** converts to native C++ integer type having a word length of 64 bits.

These member functions shall interpret the bits within a SystemC integer, fixed-point type or vector, or any part-select or concatenation thereof, as representing an unsigned binary value, with the exception of signed integers and signed fixed-point types.

Truncation shall be performed where necessary for the value to be represented as a C++ integer.

Attempting to convert a logic vector containing **'X'** or **'Z'** values to an integer shall be an error.

7.2.11 String input and output

void **scan**(std::istream& is = std::cin);
void **print**(std::ostream& os = std::cout) const;

All SystemC data types shall provide a member function **scan** that allows an object value to be set by reading a string from the specified C++ input stream. The string content may use any of the representations permitted by 7.3.

All SystemC data types shall provide a member function **print** that allows an object value to be written to a C++ output stream.

SystemC numeric types shall be printed as signed or unsigned decimal values. SystemC vector types shall be printed as a string of bit values.

All SystemC data types shall support the output stream inserter (**operator<<**) for formatted printing to a C++ stream. The format shall be the same as for the member function **print**.

The C++ ostream manipulators **dec**, **oct**, and **hex** shall have the same effect for limited-precision and finite-precision integers and vector types as they do for standard C++ integers: that is, they shall cause the values of such objects to be printed in decimal, octal, or hexadecimal formats, respectively. The formats used shall be those described in 7.3 with the exception that vectors shall be printed as a bit-pattern string when the **dec** manipulator is active.

All SystemC data types shall support the input stream inserter (**operator>>**) for formatted input from a C++ input stream. The permitted formats shall be the same as those permitted for the member function **scan**.

void **dump** (std::ostream& os = std::cout) const;

All fixed-point types shall additionally provide a member function **dump** that shall print at least the type name and value to the stream passed as an argument. The purpose of **dump** is to allow an implementation to dump out diagnostic information to help the user debug an application.

7.2.12 Conversion of application-defined types in integer expressions

The generic base proxy class template **sc_generic_base** shall be provided by the implementation and may be used as a base class for application-defined classes.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

All SystemC integer, integer part-select, and integer concatenation classes shall provide an assignment operator that accepts an object derived from the generic base proxy class template. All SystemC integer classes shall additionally provide an overloaded constructor with a single argument that is a constant reference to a generic base proxy object.

NOTE—The generic base proxy class is not included in the collection of classes described by the term “SystemC data types” as used in this standard.

7.3 String literals

A string literal representation may be used as the value of a SystemC numeric or vector type object. It shall consist of a standard prefix followed by a magnitude expressed as one or more digits.

The magnitude representation for SystemC integer types shall be based on that of C++ integer literals.

The magnitude representation for SystemC vector types shall be based on that of C++ unsigned integer literals.

The magnitude representation for SystemC fixed-point types shall be based on that of C++ floating literals but without the optional floating suffix.

Where alphabetic characters appear in the prefix or magnitude representations, each individual character may be a lowercase letter or an uppercase letter. A string literal representation shall not be case sensitive.

The permitted representations are identified with a symbol from the enumerated type **sc_numrep** as specified in Table 5.

Table 5—String literal representation

sc_numrep	Prefix (not case sensitive)	Magnitude format
SC_NOBASE	implementation-defined	implementation-defined
SC_DEC	0d	decimal
SC_BIN	0b	binary
SC_BIN_US	0bus	binary unsigned
SC_BIN_SM	0bsm	binary sign & magnitude
SC_OCT	0o	octal
SC_OCT_US	0ous	octal unsigned
SC_OCT_SM	0osm	octal sign & magnitude
SC_HEX	0x	hexadecimal
SC_HEX_US	0xus	hexadecimal unsigned
SC_HEX_SM	0xsm	hexadecimal sign & magnitude
SC_CSD	0csd	canonical signed digit

An implementation shall provide overloaded constructors and assignment operators that permit the value of any SystemC numeric type or vector to be set by a character string having one of the prefixes specified in Table 5. The character ‘+’ or ‘-’ may optionally be placed before the prefix for decimal and “sign &

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

magnitude” formats to indicate polarity. The prefix shall be followed by an unsigned integer value, except in the cases of the binary, octal, and hexadecimal formats, where the prefix shall be followed by a two’s complement value expressed as a binary, octal, or hexadecimal integer, respectively. An implementation shall sign-extend any integer string literal used to set the value of an object having a longer word length.

The canonical signed digit representation shall use the character ‘**-**’ to represent the bit value –1.

A bit-pattern string (containing bit or logic character values with no prefix) may be assigned to a vector. If the number of characters in the bit-pattern string is less than the vector word length, the string shall be zero extended at its left-hand side to the vector word length. The result of assigning such a string to a numeric type is undefined.

An instance of a SystemC numeric type, vector, part-select, or concatenation may be converted to a C++ **std::string** object by calling its member function **to_string**. The signature of **to_string** shall be as follows:

```
std::string to_string( sc_numrep numrep , bool with_prefix );
```

The **numrep** argument shall be one of the **sc_numrep** values given in Table 5. The magnitude representation in a string created from an unsigned integer or vector shall be prefixed by a single zero, except where **numrep** is SC_DEC. If the **with_prefix** argument is **true**, the prefix corresponding to the **numrep** value in Table 5 shall be appended to the left-hand side of the resulting string. The default value of **with_prefix** shall be **true**.

It shall be an error to call the member function **to_string** of a logic-vector object if any of its elements have the value **'X'** or **'Z'**.

The value of an instance of a single-bit logic type may be converted to a single character by calling its member function **to_char**.

Example:

```
sc_dt::sc_int<4> I1;                                  // 4-bit signed integer
I1 = "0b10100";                                       // 5-bit signed binary literal truncated to 4 bits
std::string S1 = I1.to_string(sc_dt::SC_BIN, true);   // The contents of S1 will be the string "0b0100"
sc_dt::sc_int<10> I2;                                 // 10-bit integer
I2 = "0d478";                                         // Decimal equivalent of "0b0111011110"
std::string S2 = I2.to_string(sc_dt::SC_CSD, false); // The contents of S2 will be the string "1000-000-0"
sc_dt::sc_uint<8> I3;                                 // 8-bit unsigned integer
I3 = "0x7";                                           // Zero-extended to 8-bit value "0x07"
std::string S3 = I3.to_string(sc_dt::SC_HEX);         // The contents of S3 will be the string "0x007"
sc_dt::sc_lv<16> lv;                                  // 16-bit logic vector
lv = "0xff";                                          // Sign-extended to 16-bit value "0xffff"
std::string S4 = lv.to_string(sc_dt::SC_HEX);         // The contents of S4 will be the string "0x0ffff"
sc_dt::sc_bv<8> bv;                                   // 8-bit bit vector
bv = "11110000";                                      // Bit-pattern string
std::string S5 = bv.to_string(sc_dt::SC_BIN);         // The contents of S5 will be the string "0b011110000"
```

NOTE—SystemC data types may provide additional overloaded **to_string** functions that require a different number of arguments.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.4 *sc_value_base*†

7.4.1 Description

Class *sc_value_base*† provides a common base class for all SystemC limited-precision integers and finite-precision integers. It provides a set of virtual member functions that may be called by an implementation to perform concatenation operations.

7.4.2 Class definition

```
namespace sc_dt {

class sc_value_base†
{
    friend class sc_concatref†;
    private:
        virtual void concat_clear_data( bool to_ones=false );
         virtual bool concat_get_ctrl( implementation-defined* dst_p , int low_i ) const;
         virtual bool concat_get_data( implementation-defined* dst_p , int low_i ) const;
        virtual uint64 concat_get_uint64() const;
        virtual int concat_length( bool* xz_present_p=0 ) const;
        virtual void concat_set( int64 src , int low_i );
        virtual void concat_set( const sc_signed& src , int low_i );
        virtual void concat_set( const sc_unsigned& src , int low_i );
        virtual void concat_set( uint64 src , int low_i );
};

}       // namespace sc_dt
```

7.4.3 Constraints on usage

An application should not create an object of type *sc_value_base*† and should not directly call any member function inherited by a derived class from an *sc_value_base*† parent.

If an application-defined class derived from the generic base proxy class template **sc_generic_base** is also derived from *sc_value_base*†, objects of this class may be used as arguments to an integer concatenation. Such a class shall override the virtual member functions of *sc_value_base*† as private members to provide the concatenation operations permitted for objects of that type.

It shall be an error for any member function of *sc_value_base*† that is not overriden in a derived class to be called for an object of the derived class.

7.4.4 Member functions

virtual void **concat_clear_data**(bool to_ones=false);

Member function **concat_clear_data** shall set every bit in the *sc_value_base*† object to the state provided by the argument.

virtual bool **concat_get_ctrl**(*implementation-defined** dst_p , int low_i) const;

Member function **concat_get_ctrl** shall copy control data to the packed-array given as the first argument, starting at the bit position within the packed-array given by the second argument. The return value shall always be **false**. The type of the first argument shall be a pointer to an unsigned integral type.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

virtual bool **concat_get_data**(*implementation-defined** dst_p , int low_i) const;

Member function **concat_get_data** shall copy data to the packed-array given as the first argument, starting at the bit position within the packed-array given by the second argument. The return value shall be **true** if the data is non-zero; otherwise, it shall be **false**. The type of the first argument shall be a pointer to an unsigned integral type.

virtual uint64 **concat_get_uint64**() const;

Member function **concat_get_uint64** shall return the value of the *sc_value_base*† object as a C++ unsigned integer having a word length of exactly 64-bits.

virtual int **concat_length**(bool* xz_present_p=0) const;

Member function **concat_length** shall return the number of bits in the *sc_value_base*† object. The value of the object associated with the optional argument shall be set to **true** if any bits have the value '**X**' or '**Z**'.

virtual void **concat_set**(int64 src , int low_i);
virtual void **concat_set**(const sc_signed& src , int low_i);
virtual void **concat_set**(const sc_unsigned& src , int low_i);
virtual void **concat_set**(uint64 src , int low_i);

Member function **concat_set** shall set the value of the *sc_value_base*† object to the bit-pattern of the integer given by the first argument. The bit-pattern shall be read as a contiguous sequence of bits starting at the position given by the second argument.

7.5 Limited-precision integer types

7.5.1 Type definitions

The following type definitions are used in the limited-precision integer type classes:

```
namespace sc_dt {

typedef implementation-defined int_type;
typedef implementation-defined uint_type;
typedef implementation-defined int64;
typedef implementation-defined uint64;

}          // namespace sc_dt
```

int_type is an implementation-dependent native C++ integer type. An implementation shall provide a minimum representation size of 64 bits.

uint_type is an implementation-dependent native C++ unsigned integer type. An implementation shall provide a minimum representation size of 64 bits.

int64 is a native C++ integer type having a word length of exactly 64 bits.

uint64 is a native C++ unsigned integer type having a word length of exactly 64 bits.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.5.2 sc_int_base

7.5.2.1 Description

Class **sc_int_base** represents a limited word-length integer. The word length is specified by a constructor argument or, by default, by the **sc_length_context** object currently in scope. The word length of an **sc_int_base** object shall be fixed during instantiation and shall not subsequently be changed.

The integer value shall be held in an implementation-dependent native C++ integer type. A minimum representation size of 64 bits is required.

sc_int_base is the base class for the **sc_int** class template.

7.5.2.2 Class definition

```
namespace sc_dt {

class sc_int_base
: public sc_value_base†
{
    friend class sc_uint_bitref_r†;
    friend class sc_uint_bitref†;
    friend class sc_uint_subref_r†;
    friend class sc_uint_subref†;

    public:
        // Constructors
        explicit sc_int_base( int w = sc_length_param().len() );
        sc_int_base( int_type v , int w );
        sc_int_base( const sc_int_base& a );

        template<typename T>
        explicit sc_int_base( const sc_generic_base<T>& a );
        explicit sc_int_base( const sc_int_subref_r†& a );
        explicit sc_int_base( const sc_signed& a );
        explicit sc_int_base( const sc_unsigned& a );
        explicit sc_int_base( const sc_bv_base& v );
        explicit sc_int_base( const sc_lv_base& v );
        explicit sc_int_base( const sc_uint_subref_r†& v );
        explicit sc_int_base( const sc_signed_subref_r†& v );
        explicit sc_int_base( const sc_unsigned_subref_r†& v );

        // Destructor
        ~sc_int_base();

        // Assignment operators
        sc_int_base& operator= ( int_type v );
        sc_int_base& operator= ( const sc_int_base& a );
        sc_int_base& operator= ( const sc_int_subref_r†& a );
        template<class T>
        sc_int_base& operator= ( const sc_generic_base<T>& a );
        sc_int_base& operator= ( const sc_signed& a );
        sc_int_base& operator= ( const sc_unsigned& a );
        sc_int_base& operator= ( const sc_fxval& a );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
sc_int_base& operator= ( const sc_fxval_fast& a );
sc_int_base& operator= ( const sc_fxnum& a );
sc_int_base& operator= ( const sc_fxnum_fast& a );
sc_int_base& operator= ( const sc_bv_base& a );
sc_int_base& operator= ( const sc_lv_base& a );
sc_int_base& operator= ( const char* a );
sc_int_base& operator= ( unsigned long a );
sc_int_base& operator= ( long a );
sc_int_base& operator= ( unsigned int a );
sc_int_base& operator= ( int a );
sc_int_base& operator= ( uint64 a );
sc_int_base& operator= ( double a );

// Prefix and postfix increment and decrement operators
sc_int_base& operator++ ();                  // Prefix
sc_int_base operator++ ( int );              // Postfix
sc_int_base& operator-- ();                  // Prefix
sc_int_base operator-- ( int );              // Postfix

// Bit selection
sc_int_bitref† operator[] ( int i );
sc_int_bitref_r† operator[] ( int i ) const;

// Part selection
sc_int_subref† operator() ( int left , int right );
sc_int_subref_r† operator() ( int left , int right ) const;
sc_int_subref† range( int left , int right );
sc_int_subref_r† range( int left , int right ) const;

// Capacity
int length() const;

// Reduce member functions
bool and_reduce() const;
bool nand_reduce() const;
bool or_reduce() const;
bool nor_reduce() const;
bool xor_reduce() const;
bool xnor_reduce() const;

// Implicit conversion to int_type
operator int_type() const;

// Explicit conversions
int to_int() const;
unsigned int to_uint() const;
long to_long() const;
unsigned long to_ulong() const;
int64 to_int64() const;
uint64 to_uint64() const;
double to_double() const;

// Explicit conversion to character string
std::string to_string( sc_numrep numrep = SC_DEC ) const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    std::string to_string( sc_numrep numrep , bool w_prefix ) const;

    // Other member functions
    void print( std::ostream& os = std::cout ) const;
    void scan( std::istream& is = std::cin );

};

}    // namespace sc_dt
```

7.5.2.3 Constraints on usage

The word length of an **sc_int_base** object shall not be greater than the maximum size of the integer representation used to hold its value.

7.5.2.4 Constructors

explicit **sc_int_base**(int w = sc_length_param().len());

Constructor **sc_int_base** shall create an object of word length specified by **w**. It is the default constructor when **w** is not specified (in which case its value shall be set by the current length context). The initial value of the object shall be **0**.

sc_int_base(int_type v , int w);

Constructor **sc_int_base** shall create an object of word length specified by **w** with initial value specified by **v**. Truncation of most significant bits shall occur if the value cannot be represented in the specified word length.

template< class T >
sc_int_base(const sc_generic_base<T>& a);

Constructor **sc_int_base** shall create an **sc_int_base** object with a word length matching the constructor argument. The constructor shall set the initial value of the object to the value returned from the member function **to_int64** of the constructor argument.

The other constructors shall create an **sc_int_base** object whose size and value matches that of the argument. The size of the argument shall not be greater than the maximum word length of an **sc_int_base** object.

7.5.2.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ integer representation to **sc_int_base**, using truncation or sign-extension as described in 7.2.2.

7.5.2.6 Implicit type conversion

operator int_type() const;

Operator **int_type** can be used for implicit type conversion from **sc_int_base** to the native C++ integer representation.

NOTE 1—This operator enables the use of standard C++ bitwise logical and arithmetic operators with **sc_int_base** objects.

NOTE 2—This operator is used by the C++ output stream operator and by the member functions of other data type classes that are not explicitly overload for **sc_int_base**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.5.2.7 Explicit type conversion

std::string **to_string**(sc_numrep numrep = SC_DEC) const;
std::string **to_string**(sc_numrep numrep, bool w_prefix) const;

Member function **to_string** shall perform the conversion to an **std::string**, as described in 7.2.12. Calling the **to_string** function with a single argument is equivalent to calling the **to_string** function with two arguments where the second argument is **true**. Calling the **to_string** function with no arguments is equivalent to calling the **to_string** function with two arguments, where the first argument is SC_DEC and the second argument is **true**.

7.5.2.8 Arithmetic, bitwise, and comparison operators

Operations specified in Table 6 are permitted. The following applies:

— **n** represents an object of type **sc_int_base**.
— **i** represents an object of integer type **int_type**.

The arguments of the comparison operators may also be of any other class that is derived from **sc_int_base**.

Table 6—sc_int_base arithmetic, bitwise, and comparison operations

Expression	Return type	Operation
n += i	sc_int_base&	sc_int_base assign sum
n -= i	sc_int_base&	sc_int_base assign difference
n *= i	sc_int_base&	sc_int_base assign product
n /= i	sc_int_base&	sc_int_base assign quotient
n %= i	sc_int_base&	sc_int_base assign remainder
n &= i	sc_int_base&	sc_int_base assign bitwise and
n |= i	sc_int_base&	sc_int_base assign bitwise or
n ^= i	sc_int_base&	sc_int_base assign bitwise exclusive or
n<<= i	sc_int_base&	sc_int_base assign left-shift
n >>= i	sc_int_base&	sc_int_base assign right-shift
n == n	bool	test equal
n != n	bool	test not equal
n < n	bool	test less than
n <= n	bool	test less than or equal
n > n	bool	test greater than
n >= n	bool	test greater than or equal

Arithmetic and bitwise operations permitted for C++ integer types shall be permitted for **sc_int_base** objects using implicit type conversions. The return type of these operations is an implementation-dependent C++ integer type.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

NOTE—An implementation is required to supply overloaded operators on **sc_int_base** objects to satisfy the requirements of this subclause. It is unspecified whether these operators are members of **sc_int_base**, global operators, or provided in some other way.

7.5.2.9 Other member functions

void **scan**(std::istream& is = std::cin);

Member function **scan** shall set the value by reading the next formatted character string from the specified input stream (see 7.2.11).

void **print**(std::ostream& os = std::cout) const;

Member function **print** shall write the value as a formatted character string to the specified output stream (see 7.2.11).

int **length**() const;

Member function **length** shall return the word length (see 7.2.5).

7.5.3 sc_uint_base

7.5.3.1 Description

Class **sc_uint_base** represents a limited word-length unsigned integer. The word length shall be specified by a constructor argument or, by default, by the **sc_length_context** object currently in scope. The word length of an **sc_uint_base** object shall be fixed during instantiation and shall not subsequently be changed.

The integer value shall be held in an implementation-dependent native C++ unsigned integer type. A minimum representation size of 64 bits is required.

sc_uint_base is the base class for the **sc_uint** class template.

7.5.3.2 Class definition

```
namespace sc_dt {

class sc_uint_base
: public sc_value_base†
{
    friend class sc_uint_bitref_r†;
    friend class sc_uint_bitref‡;
    friend class sc_uint_subref_r†;
    friend class sc_uint_subref‡;

    public:
        // Constructors
        explicit sc_uint_base( int w = sc_length_param().len() );
        sc_uint_base( uint_type v , int w );
        sc_uint_base( const sc_uint_base& a );
        explicit sc_uint_base( const sc_uint_subref_r† & a );

        template <class T>
        explicit sc_uint_base( const sc_generic_base<T>& a );
        explicit sc_uint_base( const sc_bv_base& v );
        explicit sc_uint_base( const sc_lv_base& v );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
explicit sc_uint_base( const sc_int_subref_r† & v );
explicit sc_uint_base( const sc_signed_subref_r† & v );
explicit sc_uint_base( const sc_unsigned_subref_r† & v );
explicit sc_uint_base( const sc_signed& a );
explicit sc_uint_base( const sc_unsigned& a );

// Destructor
~sc_uint_base();

// Assignment operators
sc_uint_base& operator= ( uint_type v );
sc_uint_base& operator= ( const sc_uint_base& a );
sc_uint_base& operator= ( const sc_uint_subref_r† & a );
template <class T>
sc_uint_base& operator= ( const sc_generic_base<T>& a );
sc_uint_base& operator= ( const sc_signed& a );
sc_uint_base& operator= ( const sc_unsigned& a );
sc_uint_base& operator= ( const sc_fxval& a );
sc_uint_base& operator= ( const sc_fxval_fast& a );
sc_uint_base& operator= ( const sc_fxnum& a );
sc_uint_base& operator= ( const sc_fxnum_fast& a );
sc_uint_base& operator= ( const sc_bv_base& a );
sc_uint_base& operator= ( const sc_lv_base& a );
sc_uint_base& operator= ( const char* a );
sc_uint_base& operator= ( unsigned long a );
sc_uint_base& operator= ( long a );
sc_uint_base& operator= ( unsigned int a );
sc_uint_base& operator= ( int a );
sc_uint_base& operator= ( int64 a );
sc_uint_base& operator= ( double a );

// Prefix and postfix increment and decrement operators
sc_uint_base& operator++ ();                // Prefix
sc_uint_base operator++ ( int );            // Postfix
sc_uint_base& operator-- ();                // Prefix
sc_uint_base operator-- ( int );            // Postfix

// Bit selection
sc_uint_bitref† operator[] ( int i );
sc_uint_bitref_r† operator[] ( int i ) const;

// Part selection
sc_uint_subref† operator() ( int left, int right );
sc_uint_subref_r† operator() ( int left, int right ) const;
sc_uint_subref† range( int left, int right );
sc_uint_subref_r† range( int left, int right ) const;

// Capacity
int length() const;

// Reduce member functions
bool and_reduce() const;
bool nand_reduce() const;
bool or_reduce() const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        bool nor_reduce() const;
        bool xor_reduce() const;
        bool xnor_reduce() const;

        // Implicit conversion to uint_type
        operator uint_type() const;

        // Explicit conversions
        int to_int() const;
        unsigned int to_uint() const;
        long to_long() const;
        unsigned long to_ulong() const;
        int64 to_int64() const;
        uint64 to_uint64() const;
        double to_double() const;

        // Explicit conversion to character string
        std::string to_string( sc_numrep numrep = SC_DEC ) const;
        std::string to_string( sc_numrep numrep , bool w_prefix ) const;

        // Other member functions
        void print( std::ostream& os = std::cout ) const;
        void scan( std::istream& is = std::cin );
};

}       // namespace sc_dt
```

7.5.3.3 Constraints on usage

The word length of an **sc_uint_base** object shall not be greater than the maximum size of the unsigned integer representation used to hold its value.

7.5.3.4 Constructors

explicit **sc_uint_base**(int w = sc_length_param().len());

Constructor **sc_uint_base** shall create an object of word length specified by **w**. This is the default constructor when **w** is not specified (in which case its value is set by the current length context). The initial value of the object shall be **0**.

sc_uint_base(uint_type v , int w);

Constructor **sc_uint_base** shall create an object of word length specified by **w** with initial value specified by **v**. Truncation of most significant bits shall occur if the value cannot be represented in the specified word length.

template< class T >
sc_uint_base(const sc_generic_base<T>& a);

Constructor **sc_uint_base** shall create an **sc_uint_base** object with a word length matching the constructor argument. The constructor shall set the initial value of the object to the value returned from the member function **to_uint64** of the constructor argument.

The other constructors shall create an **sc_uint_base** object whose size and value matches that of the argument. The size of the argument shall not be greater than the maximum word length of an **sc_uint_base** object.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.5.3.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ integer representation to **sc_uint_base**, using truncation or sign-extension as described in 7.2.2.

7.5.3.6 Implicit type conversion

operator **uint_type()** const;

Operator **uint_type** can be used for implicit type conversion from **sc_uint_base** to the native C++ unsigned integer representation.

NOTE 1—This operator enables the use of standard C++ bitwise logical and arithmetic operators with **sc_uint_base** objects.

NOTE 2—This operator is used by the C++ output stream operator and by the member functions of other data type classes that are not explicitly overload for **sc_uint_base**.

7.5.3.7 Explicit type conversion

std::string **to_string**(sc_numrep numrep = SC_DEC) const;
std::string **to_string**(sc_numrep numrep , bool w_prefix) const;

Member function **to_string** shall perform the conversion to an **std::string**, as described in 7.2.12. Calling the **to_string** function with a single argument is equivalent to calling the **to_string** function with two arguments, where the second argument is **true**. Calling the **to_string** function with no arguments is equivalent to calling the **to_string** function with two arguments, where the first argument is SC_DEC and the second argument is **true**.

7.5.3.8 Arithmetic, bitwise, and comparison operators

Operations specified in Table 7 are permitted. The following applies:

— **U** represents an object of type **sc_uint_base**.
— **u** represents an object of integer type **uint_type**.

The arguments of the comparison operators may also be of any other class that is derived from **sc_uint_base**.

Table 7—sc_uint_base arithmetic, bitwise, and comparison operations

Expression	Return type	Operation
U += u	sc_uint_base&	sc_uint_base assign sum
U -= u	sc_uint_base&	sc_uint_base assign difference
U *= u	sc_uint_base&	sc_uint_base assign product
U /= u	sc_uint_base&	sc_uint_base assign quotient
U %= u	sc_uint_base&	sc_uint_base assign remainder
U &= u	sc_uint_base&	sc_uint_base assign bitwise and
U |= u	sc_uint_base&	sc_uint_base assign bitwise or
U ^= u	sc_uint_base&	sc_uint_base assign bitwise exclusive or

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 7—sc_uint_base arithmetic, bitwise, and comparison operations *(continued)*

Expression	Return type	Operation
U <<= u	sc_uint_base&	sc_uint_base assign left-shift
U >>= u	sc_uint_base&	sc_uint_base assign right-shift
U == U	bool	test equal
U != U	bool	test not equal
U < U	bool	test less than
U <= U	bool	test less than or equal
U > U	bool	test greater than
U >= U	bool	test greater than or equal

Arithmetic and bitwise operations permitted for C++ integer types shall be permitted for **sc_uint_base** objects using implicit type conversions. The return type of these operations is an implementation-dependent C++ integer type.

NOTE—An implementation is required to supply overloaded operators on **sc_uint_base** objects to satisfy the requirements of this subclause. It is unspecified whether these operators are members of **sc_uint_base**, global operators, or provided in some other way.

7.5.3.9 Other member functions

void **scan**(std::istream& is = std::cin);

Member function **scan** shall set the value by reading the next formatted character string from the specified input stream (see 7.2.11).

void **print**(std::ostream& os = std::cout) const;

Member function **print** shall write the value as a formatted character string to the specified output stream (see 7.2.11).

int **length**() const;

Member function **length** shall return the word length (see 7.2.5).

7.5.4 sc_int

7.5.4.1 Description

Class template **sc_int** represents a limited word-length signed integer. The word length shall be specified by a template argument.

Any public member functions of the base class **sc_int_base** that are overridden in class **sc_int** shall have the same behavior in the two classes. Any public member functions of the base class not overridden in this way shall be publicly inherited by class **sc_int**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.5.4.2 Class definition

```
namespace sc_dt {

template <int W>
class sc_int
: public sc_int_base
{
    public:
        // Constructors
        sc_int();
        sc_int( int_type v );
        sc_int( const sc_int<W>& a );
        sc_int( const sc_int_base& a );
        sc_int( const sc_int_subref_r† & a );

        template <class T>
        sc_int( const sc_generic_base<T>& a );
        sc_int( const sc_signed& a );
        sc_int( const sc_unsigned& a );
        explicit sc_int( const sc_fxval& a );
        explicit sc_int( const sc_fxval_fast& a );
        explicit sc_int( const sc_fxnum& a );
        explicit sc_int( const sc_fxnum_fast& a );
        sc_int( const sc_bv_base& a );
        sc_int( const sc_lv_base& a );
        sc_int( const char* a );
        sc_int( unsigned long a );
        sc_int( long a );
        sc_int( unsigned int a );
        sc_int( int a );
        sc_int( uint64 a );
        sc_int( double a );

        // Assignment operators
        sc_int<W>& operator= ( int_type v );
        sc_int<W>& operator= ( const sc_int_base& a );
        sc_int<W>& operator= ( const sc_int_subref_r† & a );
        sc_int<W>& operator= ( const sc_int<W>& a );
        template <class T>
        sc_int<W>& operator= ( const sc_generic_base<T>& a );
        sc_int<W>& operator= ( const sc_signed& a );
        sc_int<W>& operator= ( const sc_unsigned& a );
        sc_int<W>& operator= ( const sc_fxval& a );
        sc_int<W>& operator= ( const sc_fxval_fast& a );
        sc_int<W>& operator= ( const sc_fxnum& a );
        sc_int<W>& operator= ( const sc_fxnum_fast& a );
        sc_int<W>& operator= ( const sc_bv_base& a );
        sc_int<W>& operator= ( const sc_lv_base& a );
        sc_int<W>& operator= ( const char* a );
        sc_int<W>& operator= ( unsigned long a );
        sc_int<W>& operator= ( long a );
        sc_int<W>& operator= ( unsigned int a );
        sc_int<W>& operator= ( int a );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_int<W>& operator= ( uint64 a );
        sc_int<W>& operator= ( double a );

        // Prefix and postfix increment and decrement operators
        sc_int<W>& operator++ ();                // Prefix
        sc_int<W> operator++ ( int );            // Postfix
        sc_int<W>& operator-- ();                // Prefix
        sc_int<W> operator-- ( int );            // Postfix
};

}       // namespace sc_dt
```

7.5.4.3 Constraints on usage

The word length of an **sc_int** object shall not be greater than the maximum word length of an **sc_int_base**.

7.5.4.4 Constructors

sc_int();

Default constructor **sc_int** shall create an **sc_int** object of word length specified by the template argument **W**. The initial value of the object shall be **0**.

template< class T >
sc_int(const sc_generic_base<T>& a);

Constructor **sc_int** shall create an **sc_int** object of word length specified by the template argument. The constructor shall set the initial value of the object to the value returned from the member function **to_int64** of the constructor argument.

The other constructors shall create an **sc_int** object of word length specified by the template argument **W** and value corresponding to the integer magnitude of the constructor argument. If the word length of the specified initial value differs from the template argument, truncation or sign-extension shall be used as described in 7.2.2.

7.5.4.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ integer representation to **sc_int**, using truncation or sign-extension as described in 7.2.2.

7.5.4.6 Arithmetic and bitwise operators

Operations specified in Table 8 are permitted. The following applies:

- **n** represents an object of type **sc_int**.
- **i** represents an object of integer type **int_type**.

Arithmetic and bitwise operations permitted for C++ integer types shall be permitted for **sc_int** objects using implicit type conversions. The return type of these operations is an implementation-dependent C++ integer type.

NOTE—An implementation is required to supply overloaded operators on **sc_int** objects to satisfy the requirements of this subclause. It is unspecified whether these operators are members of **sc_int**, global operators, or provided in some other way.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 8—sc_int arithmetic and bitwise operations

Expression	Return type	Operation
n += i	sc_int<W>&	sc_int assign sum
n -= i	sc_int<W>&	sc_int assign difference
n *= i	sc_int<W>&	sc_int assign product
n /= i	sc_int<W>&	sc_int assign quotient
n %= i	sc_int<W>&	sc_int assign remainder
n &= i	sc_int<W>&	sc_int assign bitwise and
n |= i	sc_int<W>&	sc_int assign bitwise or
n ^= i	sc_int<W>&	sc_int assign bitwise exclusive or
n <<= i	sc_int<W>&	sc_int assign left-shift
n >>= i	sc_int<W>&	sc_int assign right-shift

7.5.5 sc_uint

7.5.5.1 Description

Class template **sc_uint** represents a limited word-length unsigned integer. The word length shall be specified by a template argument. Any public member functions of the base class **sc_uint_base** that are overridden in class **sc_uint** shall have the same behavior in the two classes. Any public member functions of the base class not overridden in this way shall be publicly inherited by class **sc_uint**.

7.5.5.2 Class definition

```
namespace sc_dt {

template <int W>
class sc_uint
: public sc_uint_base
{
    public:
        // Constructors

        sc_uint();
        sc_uint( uint_type v );
        sc_uint( const sc_uint<W>& a );
        sc_uint( const sc_uint_base& a );
        sc_uint( const sc_uint_subref_r† & a );
        template <class T>
        sc_uint( const sc_generic_base<T>& a );
        sc_uint( const sc_signed& a );
        sc_uint( const sc_unsigned& a );
        explicit sc_uint( const sc_fxval& a );
        explicit sc_uint( const sc_fxval_fast& a );
        explicit sc_uint( const sc_fxnum& a );
        explicit sc_uint( const sc_fxnum_fast& a );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    sc_uint( const sc_bv_base& a );
    sc_uint( const sc_lv_base& a );
    sc_uint( const char* a );
    sc_uint( unsigned long a );
    sc_uint( long a );
    sc_uint( unsigned int a );
    sc_uint( int a );
    sc_uint( int64 a );
    sc_uint( double a );

    // Assignment operators
    sc_uint<W>& operator= ( uint_type v );
    sc_uint<W>& operator= ( const sc_uint_base& a );
    sc_uint<W>& operator= ( const sc_uint_subref_r† & a );
    sc_uint<W>& operator= ( const sc_uint<W>& a );
    template <class T>
    sc_uint<W>& operator= ( const sc_generic_base<T>& a );
    sc_uint<W>& operator= ( const sc_signed& a );
    sc_uint<W>& operator= ( const sc_unsigned& a );
    sc_uint<W>& operator= ( const sc_fxval& a );
    sc_uint<W>& operator= ( const sc_fxval_fast& a );
    sc_uint<W>& operator= ( const sc_fxnum& a );
    sc_uint<W>& operator= ( const sc_fxnum_fast& a );
    sc_uint<W>& operator= ( const sc_bv_base& a );
    sc_uint<W>& operator= ( const sc_lv_base& a );
    sc_uint<W>& operator= ( const char* a );
    sc_uint<W>& operator= ( unsigned long a );
    sc_uint<W>& operator= ( long a );
    sc_uint<W>& operator= ( unsigned int a );
    sc_uint<W>& operator= ( int a );
    sc_uint<W>& operator= ( int64 a );
    sc_uint<W>& operator= ( double a );

    // Prefix and postfix increment and decrement operators
    sc_uint<W>& operator++ ();              // Prefix
    sc_uint<W> operator++ ( int );          // Postfix
    sc_uint<W>& operator-- ();              // Prefix
    sc_uint<W> operator-- ( int );          // Postfix
};

}    // namespace sc_dt
```

7.5.5.3 Constraints on usage

The word length of an **sc_uint** object shall not be greater than the maximum word length of an **sc_uint_base**.

7.5.5.4 Constructors

sc_uint();

Default constructor **sc_uint** shall create an **sc_uint** object of word length specified by the template argument **W**. The initial value of the object shall be **0**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

template< class T >
sc_uint(const sc_generic_base<T>& a);

Constructor **sc_uint** shall create an **sc_uint** object of word length specified by the template argument. The constructor shall set the initial value of the object to the value returned from the member function **to_uint64** of the constructor argument.

The other constructors shall create an **sc_uint** object of word length specified by the template argument **W** and value corresponding to the integer magnitude of the constructor argument. If the word length of the specified initial value differs from the template argument, truncation or sign-extension shall be used as described in 7.2.2.

7.5.5.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ integer representation to **sc_uint**. If the size of a data type or string literal operand differs from the **sc_uint** word length, truncation or sign-extension shall be used as described in 7.2.2.

7.5.5.6 Arithmetic and bitwise operators

Operations specified in Table 9 are permitted. The following applies:

— **U** represents an object of type **sc_uint**.
— **u** represents an object of integer type **uint_type**.

Table 9—sc_uint arithmetic and bitwise operations

Expression	Return type	Operation
U += u	sc_uint<W>&	sc_uint assign sum
U -= u	sc_uint<W>&	sc_uint assign difference
U *= u	sc_uint<W>&	sc_uint assign product
U /= u	sc_uint<W>&	sc_uint assign quotient
U %= u	sc_uint<W>&	sc_uint assign remainder
U &= u	sc_uint<W>&	sc_uint assign bitwise and
U |= u	sc_uint<W>&	sc_uint assign bitwise or
U ^= u	sc_uint<W>&	sc_uint assign bitwise exclusive or
U <<= u	sc_uint<W>&	sc_uint assign left-shift
U >>= u	sc_uint<W>&	sc_uint assign right-shift

Arithmetic and bitwise operations permitted for C++ integer types shall be permitted for **sc_uint** objects using implicit type conversions. The return type of these operations is an implementation-dependent C++ integer.

NOTE—An implementation is required to supply overloaded operators on **sc_uint** objects to satisfy the requirements of this subclause. It is unspecified whether these operators are members of **sc_uint**, global operators, or provided in some other way.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.5.6 Bit-selects

7.5.6.1 Description

Class *sc_int_bitref_r*† represents a bit selected from an **sc_int_base** used as an rvalue.

Class *sc_int_bitref*† represents a bit selected from an **sc_int_base** used as an lvalue.

Class *sc_uint_bitref_r*† represents a bit selected from an **sc_uint_base** used as an rvalue.

Class *sc_uint_bitref*† represents a bit selected from an **sc_uint_base** used as an lvalue.

7.5.6.2 Class definition

```
namespace sc_dt {

class sc_int_bitref_r†
: public sc_value_base†
{
    friend class sc_int_base;

    public:
        // Copy constructor
        sc_int_bitref_r†( const sc_int_bitref_r†& a );

        // Destructor
        virtual ~sc_int_bitref_r†();

        // Capacity
        int length() const;

        // Implicit conversion to uint64
        operator uint64 () const;
        bool operator! () const;
        bool operator~ () const;

        // Explicit conversions
        bool to_bool() const;

        // Other member functions
        void print( std::ostream& os = std::cout ) const;

    protected:
        sc_int_bitref_r†();

    private:
        // Disabled
        sc_int_bitref_r†& operator= ( const sc_int_bitref_r†& );
};

// ----------------------------------------------------------------

class sc_int_bitref†
: public sc_int_bitref_r†
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
{
    friend class sc_int_base;

    public:
        // Copy constructor
        sc_int_bitref_r( const sc_int_bitref_r& a );

        // Assignment operators
        sc_int_bitref_r& operator= ( const sc_int_bitref_r& b );
        sc_int_bitref_r& operator= ( const sc_int_bitref& b );
        sc_int_bitref_r& operator= ( bool b );
        sc_int_bitref_r& operator&= ( bool b );
        sc_int_bitref_r& operator|= ( bool b );
        sc_int_bitref_r& operator^= ( bool b );

        // Other member functions
        void scan( std::istream& is = std::cin );

    private:
        sc_int_bitref_r();
};

// ----------------------------------------------------------------

class sc_uint_bitref_r
: public sc_value_base
{
    friend class sc_uint_base;

    public:
        // Copy constructor
        sc_uint_bitref_r( const sc_uint_bitref_r& a );

        // Destructor
        virtual ~sc_uint_bitref_r();

        // Capacity
        int length() const;

        // Implicit conversion to uint64
        operator uint64 () const;
        bool operator! () const;
        bool operator~ () const;

        // Explicit conversions
        bool to_bool() const;

        // Other member functions
        void print( std::ostream& os = std::cout ) const;

    protected:
        sc_uint_bitref_r();
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    private:
        // Disabled
        sc_uint_bitref_r† & operator= ( const sc_uint_bitref_r† & );
};

// ----------------------------------------------------------------

class sc_uint_bitref†
: public sc_uint_bitref_r†
{
    friend class sc_uint_base;

    public:
        // Copy constructor
        sc_uint_bitref† ( const sc_uint_bitref† & a );

        // Assignment operators
        sc_uint_bitref† & operator= ( const sc_uint_bitref_r† & b );
        sc_uint_bitref† & operator= ( const sc_uint_bitref† & b );
        sc_uint_bitref† & operator= ( bool b );
        sc_uint_bitref† & operator&= ( bool b );
        sc_uint_bitref† & operator|= ( bool b );
        sc_uint_bitref† & operator^= ( bool b );

        // Other member functions
        void scan( std::istream& is = std::cin );

    private:
        sc_uint_bitref†();
};

}       // namespace sc_dt
```

7.5.6.3 Constraints on usage

Bit-select objects shall only be created using the bit-select operators of an **sc_int_base** or **sc_uint_base** object (or an instance of a class derived from **sc_int_base** or **sc_uint_base**).

An application shall not explicitly create an instance of any bit-select class.

An application should not declare a reference or pointer to any bit-select object.

It is strongly recommended that an application avoid the use of a bit-select as the return type of a function because the lifetime of the object to which the bit-select refers may not extend beyond the function return statement.

Example:

```
sc_dt::sc_int_bitref get_bit_n(sc_dt::sc_int_base i, int n) {
    return i[n]; // Unsafe: returned bit-select references local variable
}
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.5.6.4 Assignment operators

Overloaded assignment operators for the lvalue bit-selects shall provide conversion from **bool** values. Assignment operators for rvalue bit-selects shall be declared as private so that they cannot be used by an application.

7.5.6.5 Implicit type conversion

operator uint64() const;

> Operator **uint64** can be used for implicit type conversion from a bit-select to the native C++ unsigned integer having exactly 64 bits. If the selected bit has the value **'1'** (true), the conversion shall return the value 1; otherwise, it shall return 0.

bool **operator!** () const;
bool **operator~** () const;

> **operator!** and **operator~** shall return a C++ **bool** value that is the inverse of the selected bit.

7.5.6.6 Other member functions

void **scan**(std::istream& is = std::cin);

> Member function **scan** shall set the value of the bit referenced by an lvalue bit-select. The value shall correspond to the C++ bool value obtained by reading the next formatted character string from the specified input stream (see 7.2.11).

void **print**(std::ostream& os = std::cout) const;

> Member function **print** shall print the value of the bit referenced by the bit-select to the specified output stream (see 7.2.11). The formatting shall be implementation-defined but shall be equivalent to printing the value returned by member function **to_bool**.

int **length**() const;

> Member function **length** shall unconditionally return a word length of 1 (see 7.2.5).

7.5.7 Part-selects

7.5.7.1 Description

Class *sc_int_subref_r*† represents a signed integer part-select from an **sc_int_base** used as an rvalue.

Class *sc_int_subref*† represents a signed integer part-select from an **sc_int_base** used as an lvalue.

Class *sc_uint_subref_r*† represents an unsigned integer part-select from an **sc_uint_base** used as an rvalue.

Class *sc_uint_subref*† represents an unsigned integer part-select from an **sc_uint_base** used as an lvalue.

7.5.7.2 Class definition

```
namespace sc_dt {

class sc_int_subref_r†
{
    friend class sc_int_base;
    friend class sc_int_subref†;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    public:
        // Copy constructor
        sc_int_subref_r†( const sc_int_subref_r†& a );

        // Destructor
        virtual ~sc_int_subref_r†();

        // Capacity
        int length() const;

        // Reduce member functions
        bool and_reduce() const;
        bool nand_reduce() const;
        bool or_reduce() const;
        bool nor_reduce() const;
        bool xor_reduce() const;
        bool xnor_reduce() const;

        // Implicit conversion to uint_type
        operator uint_type() const;

        // Explicit conversions
        int to_int() const;
        unsigned int to_uint() const;
        long to_long() const;
        unsigned long to_ulong() const;
        int64 to_int64() const;
        uint64 to_uint64() const;
        double to_double() const;

        // Explicit conversion to character string
        std::string to_string( sc_numrep numrep = SC_DEC ) const;
        std::string to_string( sc_numrep numrep , bool w_prefix ) const;

        // Other member functions
        void print( std::ostream& os = std::cout ) const;

protected:
    sc_int_subref_r†();

private:
        // Disabled
        sc_int_subref_r†& operator= ( const sc_int_subref_r†& );
};

// ----------------------------------------------------------------

class sc_int_subref†
: public sc_int_subref_r†
{
    friend class sc_int_base;

    public:
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        // Copy constructor
        sc_int_subref_r( const sc_int_subref_r& a );

        // Assignment operators
        sc_int_subref_r& operator= ( int_type v );
        sc_int_subref_r& operator= ( const sc_int_base& a );
        sc_int_subref_r& operator= ( const sc_int_subref_r& a );
        sc_int_subref_r& operator= ( const sc_int_subref_r& a );
        template< class T >
        sc_int_subref_r& operator= ( const sc_generic_base<T>& a );
        sc_int_subref_r& operator= ( const char* a );
        sc_int_subref_r& operator= ( unsigned long a );
        sc_int_subref_r& operator= ( long a );
        sc_int_subref_r& operator= ( unsigned int a );
        sc_int_subref_r& operator= ( int a );
        sc_int_subref_r& operator= ( uint64 a );
        sc_int_subref_r& operator= ( double a );
        sc_int_subref_r& operator= ( const sc_signed& );
        sc_int_subref_r& operator= ( const sc_unsigned& );
        sc_int_subref_r& operator= ( const sc_bv_base& );
        sc_int_subref_r& operator= ( const sc_lv_base& );

        // Other member functions
        void scan( std::istream& is = std::cin );

    protected:
        sc_int_subref_r();

};

// ----------------------------------------------------------------

class sc_uint_subref_r
{
    friend class sc_uint_base;
    friend class sc_uint_subref_r;

    public:
        // Copy constructor
        sc_uint_subref_r( const sc_uint_subref_r& a );

        // Destructor
        virtual ~sc_uint_subref_r();

        // Capacity
        int length() const;

        // Reduce member functions
        bool and_reduce() const;
        bool nand_reduce() const;
        bool or_reduce() const;
        bool nor_reduce() const;
        bool xor_reduce() const;
        bool xnor_reduce() const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        // Implicit conversion to uint_type
        operator uint_type() const;

        // Explicit conversions
        int to_int() const;
        unsigned int to_uint() const;
        long to_long() const;
        unsigned long to_ulong() const;
        int64 to_int64() const;
        uint64 to_uint64() const;
        double to_double() const;

        // Explicit conversion to character string
        std::string to_string( sc_numrep numrep = SC_DEC ) const;
        std::string to_string( sc_numrep numrep , bool w_prefix ) const;

        // Other member functions
        void print( std::ostream& os = std::cout ) const;

    protected:
        sc_uint_subref_r†();

    private:
        // Disabled
        sc_uint_subref_r& operator= ( const sc_uint_subref_r& );
};

// ----------------------------------------------------------

class sc_uint_subref†
: public sc_uint_subref_r†
{
    friend class sc_uint_base;

    public:
        // Copy constructor
        sc_uint_subref†( const sc_uint_subref† & a );

        // Assignment operators
        sc_uint_subref† & operator= ( uint_type v );
        sc_uint_subref† & operator= ( const sc_uint_base& a );
        sc_uint_subref† & operator= ( const sc_uint_subref_r& a );
        sc_uint_subref† & operator= ( const sc_uint_subref& a );
        template<class T>
        sc_uint_subref† & operator= ( const sc_generic_base<T>& a );
        sc_uint_subref† & operator= ( const char* a );
        sc_uint_subref† & operator= ( unsigned long a );
        sc_uint_subref† & operator= ( long a );
        sc_uint_subref† & operator= ( unsigned int a );
        sc_uint_subref† & operator= ( int a );
        sc_uint_subref† & operator= ( int64 a );
        sc_uint_subref† & operator= ( double a );
        sc_uint_subref† & operator= ( const sc_signed& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_uint_subref†& operator= ( const sc_unsigned& );
        sc_uint_subref†& operator= ( const sc_bv_base& );
        sc_uint_subref†& operator= ( const sc_lv_base& );

        // Other member functions
        void scan( std::istream& is = std::cin );

    protected:
        sc_uint_subref†();
};

}       // namespace sc_dt
```

7.5.7.3 Constraints on usage

Integer part-select objects shall only be created using the part-select operators of an **sc_int_base** or **sc_uint_base** object (or an instance of a class derived from **sc_int_base** or **sc_uint_base**), as described in 7.2.7.

An application shall not explicitly create an instance of any integer part-select class.

An application should not declare a reference or pointer to any integer part-select object.

It shall be an error if the left-hand index of a limited-precision integer part-select is less than the right-hand index.

It is strongly recommended that an application avoid the use of a part-select as the return type of a function because the lifetime of the object to which the part-select refers may not extend beyond the function return statement.

Example:

```
sc_dt::sc_int_subref get_byte(sc_dt::sc_int_base ib, int pos) {
    return ib(pos+7, pos); // Unsafe: returned part-select references local variable
}
```

7.5.7.4 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ integer representation to lvalue integer part-selects. If the size of a data type or string literal operand differs from the integer part-select word length, truncation, zero-extension, or sign-extension shall be used as described in 7.2.2.

Assignment operators for rvalue integer part-selects shall be declared as private so that they cannot be used by an application.

7.5.7.5 Implicit type conversion

sc_int_subref_r†::**operator uint_type**() const;
sc_uint_subref_r†::**operator uint_type**() const;

operator int_type and **operator uint_type** can be used for implicit type conversion from integer part-selects to the native C++ unsigned integer representation.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

NOTE 1—These operators enable the use of standard C++ bitwise logical and arithmetic operators with integer part-select objects.

NOTE 2—These operators are used by the C++ output stream operator and by member functions of other data type classes that are not explicitly overload for integer part-selects.

7.5.7.6 Explicit type conversion

std::string **to_string**(sc_numrep numrep = SC_DEC) const;
std::string **to_string**(sc_numrep numrep , bool w_prefix) const;

Member function **to_string** shall perform the conversion to an **std::string**, as described in 7.2.12. Calling the **to_string** function with a single argument is equivalent to calling the **to_string** function with two arguments, where the second argument is **true**. Calling the **to_string** function with no arguments is equivalent to calling the **to_string** function with two arguments, where the first argument is SC_DEC and the second argument is **true**.

7.5.7.7 Other member functions

void **scan**(std::istream& is = std::cin);

Member function **scan** shall set the values of the bits referenced by an lvalue part-select by reading the next formatted character string from the specified input stream (see 7.2.11).

void **print**(std::ostream& os = std::cout) const;

Member function **print** shall print the values of the bits referenced by the part-select to the specified output stream (see 7.2.11).

int **length**() const;

Member function **length** shall return the word length of the part-select (see 7.2.5).

7.6 Finite-precision integer types

7.6.1 Type definitions

The following type definitions are used in the finite-precision integer type classes:

```
namespace sc_dt{

typedef implementation-defined int64;
typedef implementation-defined uint64;

}          // namespace sc_dt
```

int64 is a native C++ integer type having a word length of exactly 64 bits.

uint64 is a native C++ unsigned integer type having a word length of exactly 64 bits.

7.6.2 Constraints on usage

Overloaded arithmetic and comparison operators allow finite-precision integer objects to be used in expressions following similar but not identical rules to standard C++ integer types. The differences from the standard C++ integer operator behavior are as follows:

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

a) Where one operand is unsigned and the other is signed, the unsigned operand shall be converted to signed and the return type shall be signed.
b) The return type of a subtraction shall always be signed.
c) The word length of the return type of an arithmetic operator shall depend only on the nature of the operation and on the word length of its operands.
d) A floating-point variable or literal shall not be directly used as an operand. It should first be converted to an appropriate signed or unsigned integer type.

7.6.3 sc_signed

7.6.3.1 Description

Class **sc_signed** represents a finite word-length integer. The word length shall be specified by a constructor argument or, by default, by the length context object currently in scope. The word length of an **sc_signed** object shall be fixed during instantiation and shall not subsequently be changed.

The integer value shall be stored with a finite precision determined by the specified word length. The precision shall not depend on the limited resolution of any standard C++ integer type.

sc_signed is the base class for the **sc_bigint** class template.

7.6.3.2 Class definition

```
namespace sc_dt {

class sc_signed
: public sc_value_base†
{
    friend class sc_concatref†;
    friend class sc_signed_bitref_r†;
    friend class sc_signed_bitref‡;
    friend class sc_signed_subref_r†;
    friend class sc_signed_subref‡;
    friend class sc_unsigned;
    friend class sc_unsigned_subref;

    public:
        // Constructors
        explicit sc_signed( int nb = sc_length_param().len() );
        sc_signed( const sc_signed&   v );
        sc_signed( const sc_unsigned& v );
        template<class T>
        explicit sc_signed( const sc_generic_base<T>& v );
        explicit sc_signed( const sc_bv_base& v );
        explicit sc_signed( const sc_lv_base& v );
        explicit sc_signed( const sc_int_subref_r& v );
        explicit sc_signed( const sc_uint_subref_r& v );
        explicit sc_signed( const sc_signed_subref_r& v );
        explicit sc_signed( const sc_unsigned_subref_r& v );

        // Assignment operators
        sc_signed& operator= ( const sc_signed& v );
        sc_signed& operator= ( const sc_signed_subref_r†& a );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
template< class T >
sc_signed& operator= ( const sc_generic_base<T>& a );
sc_signed& operator= ( const sc_unsigned& v );
sc_signed& operator= ( const sc_unsigned_subref_r† & a );
sc_signed& operator= ( const char* v );
sc_signed& operator= ( int64 v );
sc_signed& operator= ( uint64 v );
sc_signed& operator= ( long v );
sc_signed& operator= ( unsigned long v );
sc_signed& operator= ( int v );
sc_signed& operator= ( unsigned int v );
sc_signed& operator= ( double v );
sc_signed& operator= ( const sc_int_base& v );
sc_signed& operator= ( const sc_uint_base& v );
sc_signed& operator= ( const sc_bv_base& );
sc_signed& operator= ( const sc_lv_base& );
sc_signed& operator= ( const sc_fxval& );
sc_signed& operator= ( const sc_fxval_fast& );
sc_signed& operator= ( const sc_fxnum& );
sc_signed& operator= ( const sc_fxnum_fast& );

// Destructor
~sc_signed();

// Increment operators.
sc_signed& operator++ ();
sc_signed operator++ ( int );

// Decrement operators.
sc_signed& operator-- ();
sc_signed operator-- ( int );

// Bit selection
sc_signed_bitref† operator[] ( int i );
sc_signed_bitref_r† operator[] ( int i ) const;

// Part selection
sc_signed_subref† range( int i , int j );
sc_signed_subref_r† range( int i , int j ) const;
sc_signed_subref† operator() ( int i , int j );
sc_signed_subref_r† operator() ( int i , int j ) const;

// Explicit conversions
int to_int() const;
unsigned int to_uint() const;
long to_long() const;
unsigned long to_ulong() const;
int64 to_int64() const;
uint64 to_uint64() const;
double to_double() const;

// Explicit conversion to character string
std::string to_string( sc_numrep numrep = SC_DEC ) const;
std::string to_string( sc_numrep numrep, bool w_prefix ) const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        // Print functions
        void print( std::ostream& os = std::cout ) const;
        void scan( std::istream& is = std::cin );

        // Capacity
        int length() const;

        // Reduce member functions
        bool and_reduce() const;
        bool nand_reduce() const;
        bool or_reduce() const;
        bool nor_reduce() const;
        bool xor_reduce() const;
        bool xnor_reduce() const;

    // Overloaded operators

};

}          // namespace sc_dt
```

7.6.3.3 Constraints on usage

An object of type **sc_signed** shall not be used as a direct replacement for a C++ integer type since no implicit type conversion member functions are provided. An explicit type conversion is required to pass the value of an **sc_signed** object as an argument to a function expecting a C++ integer value argument.

7.6.3.4 Constructors

explicit **sc_signed**(int nb = sc_length_param().len());

Constructor **sc_signed** shall create an **sc_signed** object of word length specified by **nb**. This is the default constructor when **nb** is not specified (in which case its value is set by the current length context). The initial value of the object shall be **0**.

template< class T >
sc_signed(const sc_generic_base<T>& a);

Constructor **sc_signed** shall create an **sc_signed** object with a word length matching the constructor argument. The constructor shall set the initial value of the object to the value returned from the member function **to_sc_signed** of the constructor argument.

The other constructors create an **sc_signed** object with the same word length and value as the constructor argument.

7.6.3.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ integer representation to **sc_signed**, using truncation or sign-extension as described in 7.2.2.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.6.3.6 Explicit type conversion

std::string **to_string**(sc_numrep numrep = SC_DEC) const;
std::string **to_string**(sc_numrep numrep, bool w_prefix) const;

Member function **to_string** shall perform conversion to an **std::string** representation, as described in 7.2.12. Calling the **to_string** function with a single argument is equivalent to calling the **to_string** function with two arguments, where the second argument is **true**. Calling the **to_string** function with no arguments is equivalent to calling the **to_string** function with two arguments, where the first argument is SC_DEC and the second argument is **true**.

7.6.3.7 Arithmetic, bitwise, and comparison operators

Operations specified in Table 10, Table 11, and Table 12 are permitted. The following applies:

— **S** represents an object of type **sc_signed**.
— **U** represents an object of type **sc_unsigned**.
— **i** represents an object of integer type **int**, **long**, **unsigned int**, **unsigned long**, **sc_signed**, **sc_unsigned**, **sc_int_base**, or **sc_uint_base**.
— **s** represents an object of signed integer type **int**, **long**, **sc_signed**, or **sc_int_base**.

The operands may also be of any other class that is derived from those just given.

Table 10—sc_signed arithmetic operations

Expression	Return type	Operation
S + i	sc_signed	sc_signed addition
i + S	sc_signed	sc_signed addition
U + s	sc_signed	addition of sc_unsigned and signed
s + U	sc_signed	addition of signed and sc_unsigned
S += i	sc_signed&	sc_signed assign sum
S - i	sc_signed	sc_signed subtraction
i - S	sc_signed	sc_signed subtraction
U - i	sc_signed	sc_unsigned subtraction
i - U	sc_signed	sc_unsigned subtraction
S -= i	sc_signed&	sc_signed assign difference
S * i	sc_signed	sc_signed multiplication
i * S	sc_signed	sc_signed multiplication
U * s	sc_signed	multiplication of sc_unsigned by signed
s * U	sc_signed	multiplication of signed by sc_unsigned
S *= i	sc_signed&	sc_signed assign product
S / i	sc_signed	sc_signed division
i / S	sc_signed	sc_signed division
U / s	sc_signed	division of sc_unsigned by signed
s / U	sc_signed	division of signed by sc_unsigned
S /= i	sc_signed&	sc_signed assign quotient

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 10—sc_signed arithmetic operations *(continued)*

Expression	Return type	Operation
S % i	sc_signed	sc_signed remainder
i % S	sc_signed	sc_signed remainder
U % s	sc_signed	remainder of sc_unsigned with signed
s % U	sc_signed	remainder of signed with sc_unsigned
S %= i	sc_signed&	sc_signed assign remainder
+S	sc_signed	sc_signed unary plus
-S	sc_signed	sc_signed unary minus
-U	sc_signed	sc_unsigned unary minus

Table 11—sc_signed bitwise operations

Expression	Return type	Operation
S & i	sc_signed	sc_signed bitwise and
i & S	sc_signed	sc_signed bitwise and
U & s	sc_signed	sc_unsigned bitwise and signed
s & U	sc_signed	signed bitwise and sc_unsigned
S &= i	sc_signed&	sc_signed assign bitwise and
S | i	sc_signed	sc_signed bitwise or
i | S	sc_signed	sc_signed bitwise or
U | s	sc_signed	sc_unsigned bitwise or signed
s | U	sc_signed	signed bitwise or sc_unsigned
S |= i	sc_signed&	sc_signed assign bitwise or
S ^ i	sc_signed	sc_signed bitwise exclusive or
i ^ S	sc_signed	sc_signed bitwise exclusive or
U ^ s	sc_signed	sc_unsigned bitwise exclusive or signed
s ^ U	sc_signed	sc_unsigned bitwise exclusive or signed
S ^= i	sc_signed&	sc_signed assign bitwise exclusive or
S << i	sc_signed	sc_signed left-shift
U << S	sc_unsigned	sc_unsigned left-shift
S <<= i	sc_signed&	sc_signed assign left-shift
S >> i	sc_signed	sc_signed right-shift
U >> S	sc_unsigned	sc_unsigned right-shift
S >>= i	sc_signed&	sc_signed assign right-shift
~S	sc_signed	sc_signed bitwise complement

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 12—sc_signed comparison operations

Expression	Return type	Operation
S == i	bool	test equal
i == S	bool	test equal
S != i	bool	test not equal
i != S	bool	test not equal
S < i	bool	test less than
i < S	bool	test less than
S <= i	bool	test less than or equal
i <= S	bool	test less than or equal
S > i	bool	test greater than
i > S	bool	test greater than
S >= i	bool	test greater than or equal
i >= S	bool	test greater than or equal

If the result of any arithmetic operation is zero, the word length of the return value shall be set by the **sc_length_context** in scope. Otherwise, the following rules apply:

— Addition shall return a result with a word length that is equal to the word length of the longest operand plus one.
— Multiplication shall return a result with a word length that is equal to the sum of the word lengths of the two operands.
— Remainder shall return a result with a word length that is equal to the word length of the shortest operand.
— All other arithmetic operators shall return a result with a word length that is equal to the word length of the longest operand.

Binary bitwise operators shall return a result with a word length that is equal to the word length of the longest operand.

The left shift operator shall return a result with a word length that is equal to the word length of its **sc_signed** operand plus the right (integer) operand. Bits added on the right-hand side of the result shall be set to zero.

The right shift operator shall return a result with a word length that is equal to the word length of its **sc_signed** operand. Bits added on the left-hand side of the result shall be set to the same value as the left-hand bit of the **sc_signed** operand (a right-shift preserves the sign).

The behavior of a shift operator is undefined if the right operand is negative.

NOTE—An implementation is required to supply overloaded operators on **sc_signed** objects to satisfy the requirements of this subclause. It is unspecified whether these operators are members of **sc_signed**, global operators, or provided in some other way.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.6.3.8 Other member functions

void **scan**(std::istream& is = std::cin);

Member function **scan** shall set the value by reading the next formatted character string from the specified input stream (see 7.2.11).

void **print**(std::ostream& os = std::cout) const;

Member function **print** shall write the value as a formatted character string to the specified output stream (see 7.2.11).

int **length**() const;

Member function **length** shall return the word length (see 7.2.5).

7.6.4 sc_unsigned

7.6.4.1 Description

Class **sc_unsigned** represents a finite word-length unsigned integer. The word length shall be specified by a constructor argument or, by default, by the length context currently in scope. The word length of an **sc_unsigned** object is fixed during instantiation and shall not be subsequently changed.

The integer value shall be stored with a finite precision determined by the specified word length. The precision shall not depend on the limited resolution of any standard C++ integer type.

sc_unsigned is the base class for the **sc_biguint** class template.

7.6.4.2 Class definition

```
namespace sc_dt {

class sc_unsigned
: public sc_value_base†
{
    friend class sc_concatref†;
    friend class sc_unsigned_bitref_r†;
    friend class sc_unsigned_bitref†;
    friend class sc_unsigned_subref_r†;
    friend class sc_unsigned_subref†;
    friend class sc_signed;
    friend class sc_signed_subref†;

    public:
        // Constructors
        explicit sc_unsigned( int nb = sc_length_param().len() );
        sc_unsigned( const sc_unsigned& v );
        sc_unsigned( const sc_signed& v );
        template<class T>
        explicit sc_unsigned( const sc_generic_base<T>& v );
        explicit sc_unsigned( const sc_bv_base& v );
        explicit sc_unsigned( const sc_lv_base& v );
        explicit sc_unsigned( const sc_int_subref_r& v );
        explicit sc_unsigned( const sc_uint_subref_r& v );
        explicit sc_unsigned( const sc_signed_subref_r& v );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
explicit sc_unsigned( const sc_unsigned_subref_r& v );

// Assignment operators
sc_unsigned& operator= ( const sc_unsigned& v);
sc_unsigned& operator= ( const sc_unsigned_subref_r† & a );
template<class T>
sc_unsigned& operator= ( const sc_generic_base<T>& a );
sc_unsigned& operator= ( const sc_signed& v );
sc_unsigned& operator= ( const sc_signed_subref_r† & a );
sc_unsigned& operator= ( const char* v);
sc_unsigned& operator= ( int64 v );
sc_unsigned& operator= ( uint64 v );
sc_unsigned& operator= ( long v );
sc_unsigned& operator= ( unsigned long v );
sc_unsigned& operator= ( int v );
sc_unsigned& operator= ( unsigned int v );
sc_unsigned& operator= ( double v );
sc_unsigned& operator= ( const sc_int_base& v );
sc_unsigned& operator= ( const sc_uint_base& v );
sc_unsigned& operator= ( const sc_bv_base& );
sc_unsigned& operator= ( const sc_lv_base& );
sc_unsigned& operator= ( const sc_fxval& );
sc_unsigned& operator= ( const sc_fxval_fast& );
sc_unsigned& operator= ( const sc_fxnum& );
sc_unsigned& operator= ( const sc_fxnum_fast& );

// Destructor
 ~sc_unsigned();

// Increment operators
sc_unsigned& operator++ ();
sc_unsigned operator++ ( int );

// Decrement operators
sc_unsigned& operator-- ();
sc_unsigned operator-- ( int) ;

// Bit selection
sc_unsigned_bitref† operator[] ( int i );
sc_unsigned_bitref_r† operator[] ( int i ) const;

// Part selection
sc_unsigned_subref† range ( int i , int j );
sc_unsigned_subref_r† range( int i , int j ) const;
sc_unsigned_subref† operator() ( int i , int j );
sc_unsigned_subref_r† operator() ( int i , int j ) const;

// Explicit conversions
int to_int() const;
unsigned int to_uint() const;
long to_long() const;
unsigned long to_ulong() const;
int64 to_int64() const;
uint64 to_uint64() const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        double to_double() const;

        // Explicit conversion to character string
        std::string to_string( sc_numrep numrep = SC_DEC ) const;
        std::string to_string( sc_numrep numrep, bool w_prefix ) const;

        // Print functions
        void print( std::ostream& os = std::cout ) const;
        void scan( std::istream& is = std::cin );

        // Capacity
         int length() const;                                    // Bit width

    // Reduce member functions
        bool and_reduce() const;
        bool nand_reduce() const;
        bool or_reduce() const;
        bool nor_reduce() const;
        bool xor_reduce() const;
        bool xnor_reduce() const;

        // Overloaded operators

};

}       // namespace sc_dt
```

7.6.4.3 Constraints on usage

An object of type **sc_unsigned** may not be used as a direct replacement for a C++ integer type since no implicit type conversion member functions are provided. An explicit type conversion is required to pass the value of an **sc_unsigned** object as an argument to a function expecting a C++ integer value argument.

7.6.4.4 Constructors

explicit **sc_unsigned**(int nb = sc_length_param().len());

Constructor **sc_unsigned** shall create an **sc_unsigned** object of word length specified by **nb**. This is the default constructor when **nb** is not specified (in which case its value is set by the current length context). The initial value shall be **0**.

template< class T >
sc_unsigned(const sc_generic_base<T>& a);

Constructor **sc_unsigned** shall create an **sc_unsigned** object with a word length matching the constructor argument. The constructor shall set the initial value of the object to the value returned from the member function **to_sc_unsigned** of the constructor argument.

The other constructors create an **sc_unsigned** object with the same word length and value as the constructor argument.

7.6.4.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ integer representation to **sc_unsigned**, using truncation or sign-extension as described in 7.2.2.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.6.4.6 Explicit type conversion

std::string **to_string**(sc_numrep numrep = SC_DEC) const;
std::string **to_string**(sc_numrep numrep, bool w_prefix) const;

Member function **to_string** shall perform the conversion to an **std::string**, as described in 7.2.12. Calling the **to_string** function with a single argument is equivalent to calling the **to_string** function with two arguments, where the second argument is **true**. Calling the **to_string** function with no arguments is equivalent to calling the **to_string** function with two arguments, where the first argument is SC_DEC and the second argument is **true**.

7.6.4.7 Arithmetic, bitwise, and comparison operators

Operations specified in Table 13, Table 14, and Table 15 are permitted. The following applies:

- **S** represents an object of type **sc_signed**.
- **U** represents an object of type **sc_unsigned**.
- **i** represents an object of integer type **int**, **long**, **unsigned int**, **unsigned long**, **sc_signed**, **sc_unsigned**, **sc_int_base**, or **sc_uint_base**.
- **s** represents an object of signed integer type **int**, **long**, **sc_signed**, or **sc_int_base**.
- **u** represents an object of unsigned integer type **unsigned int**, **unsigned long**, **sc_unsigned**, or **sc_uint_base**.

The operands may also be of any other class that is derived from those just given.

Table 13—sc_unsigned arithmetic operations

Expression	Return type	Operation
U + u	sc_unsigned	sc_unsigned addition
u + U	sc_unsigned	sc_unsigned addition
U + s	sc_signed	addition of sc_unsigned and signed
s + U	sc_signed	addition of signed and sc_unsigned
U += i	sc_unsigned&	sc_unsigned assign sum
U - i	sc_signed	sc_unsigned subtraction
i - U	sc_signed	sc_unsigned subtraction
U -= i	sc_unsigned&	sc_unsigned assign difference
U * u	sc_unsigned	sc_unsigned multiplication
u * U	sc_unsigned	sc_unsigned multiplication
U * s	sc_signed	multiplication of sc_unsigned by signed
s * U	sc_signed	multiplication of signed by sc_unsigned
U *= i	sc_unsigned&	sc_unsigned assign product
U / u	sc_unsigned	sc_unsigned division
u / U	sc_unsigned	sc_unsigned division
U / s	sc_signed	division of sc_unsigned by signed

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 13—sc_unsigned arithmetic operations *(continued)*

Expression	Return type	Operation
s / U	sc_signed	division of signed by sc_unsigned
U /= i	sc_unsigned&	sc_unsigned assign quotient
U % u	sc_unsigned	sc_unsigned remainder
u % U	sc_unsigned	sc_unsigned remainder
U % s	sc_signed	remainder of sc_unsigned with signed
s % U	sc_signed	remainder of signed with sc_unsigned
U %= i	sc_unsigned&	sc_unsigned assign remainder
+U	sc_unsigned	sc_unsigned unary plus
-U	sc_signed	sc_unsigned unary minus

Table 14—sc_unsigned bitwise operations

Expression	Return type	Operation
U & u	sc_unsigned	sc_unsigned bitwise and
u & U	sc_unsigned	sc_unsigned bitwise and
U & s	sc_signed	sc_unsigned bitwise and signed
s & U	sc_signed	signed bitwise and sc_unsigned
U &= i	sc_unsigned&	sc_unsigned assign bitwise and
U | u	sc_unsigned	sc_unsigned bitwise or
u | U	sc_unsigned	sc_unsigned bitwise or
U | s	sc_signed	sc_unsigned bitwise or signed
s | U	sc_signed	signed bitwise or sc_unsigned
U |= i	sc_unsigned&	sc_unsigned assign bitwise or
U ^ u	sc_unsigned	sc_unsigned bitwise exclusive or
u ^ U	sc_unsigned	sc_unsigned bitwise exclusive or
U ^ s	sc_signed	sc_unsigned bitwise exclusive or signed
s ^ U	sc_signed	sc_unsigned bitwise exclusive or signed
U ^= i	sc_unsigned&	sc_unsigned assign bitwise exclusive or
U << i	sc_unsigned	sc_unsigned left-shift
S << U	sc_signed	sc_signed left-shift
U <<= i	sc_unsigned&	sc_unsigned assign left-shift
U >> i	sc_unsigned	sc_unsigned right-shift

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 14—sc_unsigned bitwise operations *(continued)*

Expression	Return type	Operation
S >> U	sc_signed	sc_signed right-shift
U >>= i	sc_unsigned&	sc_unsigned assign right-shift
~U	sc_unsigned	sc_unsigned bitwise complement

Table 15—sc_unsigned comparison operations

Expression	Return type	Operation
U == i	bool	test equal
i == U	bool	test equal
U != i	bool	test not equal
i != U	bool	test not equal
U < i	bool	test less than
i < U	bool	test less than
U <= i	bool	test less than or equal
i <= U	bool	test less than or equal
U > i	bool	test greater than
i > U	bool	test greater than
U >= i	bool	test greater than or equal
i >= U	bool	test greater than or equal

If the result of any arithmetic operation is zero, the word length of the return value shall be set by the **sc_length_context** in scope. Otherwise, the following rules apply:

— Addition shall return a result with a word length that is equal to the word length of the longest operand plus one.
— Multiplication shall return a result with a word length that is equal to the sum of the word lengths of the two operands.
— Remainder shall return a result with a word length that is equal to the word length of the shortest operand.
— All other arithmetic operators shall return a result with a word length that is equal to the word length of the longest operand.

Binary bitwise operators shall return a result with a word length that is equal to the word length of the longest operand.

The left shift operator shall return a result with a word length that is equal to the word length of its **sc_unsigned** operand plus the right (integer) operand. Bits added on the right-hand side of the result shall be set to zero.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The right shift operator shall return a result with a word length that is equal to the word length of its **sc_unsigned** operand. Bits added on the left-hand side of the result shall be set to zero.

NOTE—An implementation is required to supply overloaded operators on **sc_unsigned** objects to satisfy the requirements of this subclause. It is unspecified whether these operators are members of **sc_unsigned**, global operators, or provided in some other way.

7.6.4.8 Other member functions

void **scan**(std::istream& is = std::cin);

Member function **scan** shall set the value by reading the next formatted character string from the specified input stream (see 7.2.11).

void **print**(std::ostream& os = std::cout) const;

Member function **print** shall write the value as a formatted character string to the specified output stream (see 7.2.11).

int **length**() const;

Member function **length** shall return the word length (see 7.2.5).

7.6.5 sc_bigint

7.6.5.1 Description

Class template **sc_bigint** represents a finite word-length signed integer. The word length shall be specified by a template argument. The integer value shall be stored with a finite precision determined by the specified word length. The precision shall not depend on the limited resolution of any standard C++ integer type.

Any public member functions of the base class **sc_signed** that are overridden in class **sc_bigint** shall have the same behavior in the two classes. Any public member functions of the base class not overridden in this way shall be publicly inherited by class **sc_bigint**. The operations specified in 7.6.3.7 are permitted for objects of type **sc_bigint**.

7.6.5.2 Class definition

```
namespace sc_dt {

template< int W >
class sc_bigint
: public sc_signed
{
    public:
        // Constructors
        sc_bigint();
        sc_bigint( const sc_bigint<W>& v );
        sc_bigint( const sc_signed& v );
        sc_bigint( const sc_signed_subref† & v );
        template< class T >
        sc_bigint( const sc_generic_base<T>& a );
        sc_bigint( const sc_unsigned& v );
        sc_bigint( const sc_unsigned_subref† & v );
        sc_bigint( const char* v );
        sc_bigint( int64 v );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    sc_bigint( uint64 v );
    sc_bigint( long v );
    sc_bigint( unsigned long v );
    sc_bigint( int v );
    sc_bigint( unsigned int v );
    sc_bigint( double v );
    sc_bigint( const sc_bv_base& v );
    sc_bigint( const sc_lv_base& v );
    explicit sc_bigint( const sc_fxval& v );
    explicit sc_bigint( const sc_fxval_fast& v );
    explicit sc_bigint( const sc_fxnum& v );
    explicit sc_bigint( const sc_fxnum_fast& v );

    // Destructor
    ~sc_bigint();

    // Assignment operators
    sc_bigint<W>& operator= ( const sc_bigint<W>& v );
    sc_bigint<W>& operator= ( const sc_signed& v );
    sc_bigint<W>& operator= (const sc_signed_subref† & v );
    template< class T >
    sc_bigint<W>& operator= ( const sc_generic_base<T>& a );
    sc_bigint<W>& operator= ( const sc_unsigned& v );
    sc_bigint<W>& operator= ( const sc_unsigned_subref† & v );
    sc_bigint<W>& operator= ( const char* v );
    sc_bigint<W>& operator= ( int64 v );
    sc_bigint<W>& operator= ( uint64 v );
    sc_bigint<W>& operator= ( long v );
    sc_bigint<W>& operator= ( unsigned long v );
    sc_bigint<W>& operator= ( int v );
    sc_bigint<W>& operator= ( unsigned int v );
    sc_bigint<W>& operator= ( double v );
    sc_bigint<W>& operator= ( const sc_bv_base& v );
    sc_bigint<W>& operator= ( const sc_lv_base& v );
    sc_bigint<W>& operator= ( const sc_int_base& v );
    sc_bigint<W>& operator= ( const sc_uint_base& v );
    sc_bigint<W>& operator= ( const sc_fxval& v );
    sc_bigint<W>& operator= ( const sc_fxval_fast& v );
    sc_bigint<W>& operator= ( const sc_fxnum& v );
    sc_bigint<W>& operator= ( const sc_fxnum_fast& v );

};

}       // namespace sc_dt
```

7.6.5.3 Constraints on usage

An object of type **sc_bigint** may not be used as a direct replacement for a C++ integer type since no implicit type conversion member functions are provided. An explicit type conversion is required to pass the value of an **sc_bigint** object as an argument to a function expecting a C++ integer value argument.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.6.5.4 Constructors

sc_bigint();

Default constructor **sc_bigint** shall create an **sc_bigint** object of word length specified by the template argument **W** and shall set the initial value to **0**.

template< class T >
sc_bigint(const sc_generic_base<T>& a);

Constructor **sc_bigint** shall create an **sc_bigint** object of word length specified by the template argument. The constructor shall set the initial value of the object to the value returned from the member function **to_sc_signed** of the constructor argument.

Other constructors shall create an **sc_bigint** object of word length specified by the template argument **W** and value corresponding to the integer magnitude of the constructor argument. If the word length of the specified initial value differs from the template argument, truncation or sign-extension shall be used as described in 7.2.2.

NOTE—Most constructors can be used as implicit conversions from fundamental types or SystemC data types to **sc_bigint**. Hence, a function having an **sc_bigint** parameter can be passed a floating-point argument, for example, and the argument will be implicitly converted. The exceptions are the conversions from fixed-point types to **sc_bigint**, which must be called explicitly.

7.6.5.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ integer representation to **sc_bigint**, using truncation or sign-extension as described in 7.2.2.

7.6.6 sc_biguint

7.6.6.1 Description

Class template **sc_biguint** represents a finite word-length unsigned integer. The word length shall be specified by a template argument. The integer value shall be stored with a finite precision determined by the specified word length. The precision shall not depend on the limited resolution of any standard C++ integer type.

Any public member functions of the base class **sc_unsigned** that are overridden in class **sc_biguint** shall have the same behavior in the two classes. Any public member functions of the base class not overridden in this way shall be publicly inherited by class **sc_biguint**. The operations specified in 7.6.4.7 are permitted for objects of type **sc_biguint**.

7.6.6.2 Class definition

```
namespace sc_dt {

template< int W >
class sc_biguint
: public sc_unsigned
{
    public:
        // Constructors
        sc_biguint();
        sc_biguint( const sc_biguint<W>& v );
        sc_biguint( const sc_unsigned& v );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_biguint( const sc_unsigned_subref† & v );
        template< class T >
        sc_biguint( const sc_generic_base<T>& a );
        sc_biguint( const sc_signed& v );
        sc_biguint( const sc_signed_subref† & v );
        sc_biguint( const char* v );
        sc_biguint( int64 v );
        sc_biguint( uint64 v );
        sc_biguint( long v );
        sc_biguint( unsigned long v );
        sc_biguint( int v );
        sc_biguint( unsigned int v );
        sc_biguint( double v );
        sc_biguint( const sc_bv_base& v );
        sc_biguint( const sc_lv_base& v );
        explicit sc_biguint( const sc_fxval& v );
        explicit sc_biguint( const sc_fxval_fast& v );
        explicit sc_biguint( const sc_fxnum& v );
        explicit sc_biguint( const sc_fxnum_fast& v );

        // Destructor
        ~sc_biguint();

        // Assignment operators
        sc_biguint<W>& operator= ( const sc_biguint<W>& v );
        sc_biguint<W>& operator= ( const sc_unsigned& v );
        sc_biguint<W>& operator= ( const sc_unsigned_subref† & v );
        template< class T >
        sc_biguint<W>& operator= ( const sc_generic_base<T>& a );
        sc_biguint<W>& operator= ( const sc_signed& v );
        sc_biguint<W>& operator= ( const sc_signed_subref† & v );
        sc_biguint<W>& operator= ( const char* v );
        sc_biguint<W>& operator= ( int64 v );
        sc_biguint<W>& operator= ( uint64 v );
        sc_biguint<W>& operator= ( long v );
        sc_biguint<W>& operator= ( unsigned long v );
        sc_biguint<W>& operator= ( int v );
        sc_biguint<W>& operator= ( unsigned int v );
        sc_biguint<W>& operator= ( double v );
        sc_biguint<W>& operator= ( const sc_bv_base& v );
        sc_biguint<W>& operator= ( const sc_lv_base& v );
        sc_biguint<W>& operator= ( const sc_int_base& v );
        sc_biguint<W>& operator= ( const sc_uint_base& v );
        sc_biguint<W>& operator= ( const sc_fxval& v );
        sc_biguint<W>& operator= ( const sc_fxval_fast& v );
        sc_biguint<W>& operator= ( const sc_fxnum& v );
        sc_biguint<W>& operator= ( const sc_fxnum_fast& v );
};

}          // namespace sc_dt
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.6.6.3 Constraints on usage

An object of type **sc_biguint** may not be used as a direct replacement for a C++ integer type since no implicit type conversion member functions are provided. An explicit type conversion is required to pass the value of an **sc_biguint** object as an argument to a function expecting a C++ integer value argument.

7.6.6.4 Constructors

sc_biguint();

Default constructor **sc_biguint** shall create an **sc_biguint** object of word length specified by the template argument **W** and shall set the initial value to **0**.

template< class T >
sc_biguint(const sc_generic_base<T>& a);

Constructor shall create an **sc_biguint** object of word length specified by the template argument. The constructor shall set the initial value of the object to the value returned from the member function **to_sc_unsigned** of the constructor argument.

The other constructors shall create an **sc_biguint** object of word length specified by the template argument **W** and value corresponding to the integer magnitude of the constructor argument. If the word length of the specified initial value differs from the template argument, truncation or sign-extension shall be used as described in 7.2.2.

NOTE—Most constructors can be used as implicit conversions from fundamental types or SystemC data types to **sc_biguint**. Hence, a function having an **sc_biguint** parameter can be passed a floating-point argument, for example, and the argument will be implicitly converted. The exceptions are the conversions from fixed-point types to **sc_biguint**, which must be called explicitly.

7.6.6.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ integer representation to **sc_biguint**, using truncation or sign-extension, as described in 7.2.2.

7.6.7 Bit-selects

7.6.7.1 Description

Class *sc_signed_bitref_r*† represents a bit selected from an **sc_signed** used as an rvalue.

Class *sc_signed_bitref*† represents a bit selected from an **sc_signed** used as an lvalue.

Class *sc_unsigned_bitref_r*† represents a bit selected from an **sc_unsigned** used as an rvalue.

Class *sc_unsigned_bitref*† represents a bit selected from an **sc_unsigned** used as an lvalue.

7.6.7.2 Class definition

```
namespace sc_dt {

class sc_signed_bitref_r†
: public sc_value_base†
{
    friend class sc_signed;
    friend class sc_signed_bitref†;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    public:
        // Copy constructor
        sc_signed_bitref_r†( const sc_signed_bitref_r† & a );

        // Destructor
        virtual ~sc_signed_bitref_r†();

        // Capacity
        int length() const;

        // Implicit conversion to uint64
        operator uint64 () const;
        bool operator! () const;
        bool operator~ () const;

        // Explicit conversions
        bool to_bool() const;

        // Other member functions
        void print( std::ostream& os = std::cout ) const;

    protected:
        sc_signed_bitref_r†();

    private:
        // Disabled
        sc_signed_bitref_r† & operator= ( const sc_signed_bitref_r† & );
};

// ----------------------------------------------------------------------

class sc_signed_bitref†
: public sc_signed_bitref_r†
{
    friend class sc_signed;

    public:
        // Copy constructor
        sc_signed_bitref†( const sc_signed_bitref† & a );

        // Assignment operators
        sc_signed_bitref† & operator= ( const sc_signed_bitref_r† & );
        sc_signed_bitref† & operator= ( const sc_signed_bitref† & );
        sc_signed_bitref† & operator= ( bool );

        sc_signed_bitref† & operator&= ( bool );
        sc_signed_bitref† & operator|= ( bool );
        sc_signed_bitref† & operator^= ( bool );

        // Other member functions
        void scan( std::istream& is = std::cin );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    protected:
        sc_signed_bitref_r();
};

// ----------------------------------------------------------------------

class sc_unsigned_bitref_r
: public sc_value_base
{
    friend class sc_unsigned;

    public:
        // Copy constructor
        sc_unsigned_bitref_r( const sc_unsigned_bitref_r& a );

        // Destructor
        virtual ~sc_unsigned_bitref_r();

        // Capacity
        int length() const;

        // Implicit conversion to uint64
        operator uint64 () const;
        bool operator! () const;
        bool operator~ () const;

        // Explicit conversions
        bool to_bool() const;

        // Other member functions
        void print( std::ostream& os = std::cout ) const;

    protected:
        sc_unsigned_bitref_r();

    private:
        // Disabled
        sc_unsigned_bitref_r& operator= ( const sc_unsigned_bitref_r& );
};

// ----------------------------------------------------------------------

class sc_unsigned_bitref
: public sc_unsigned_bitref_r
{
    friend class sc_unsigned;

    public:
        // Copy constructor
        sc_unsigned_bitref( const sc_unsigned_bitref& a );

        // Assignment operators
        sc_unsigned_bitref& operator= ( const sc_unsigned_bitref_r& );
        sc_unsigned_bitref& operator= ( const sc_unsigned_bitref& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_unsigned_bitref† & operator= ( bool );

        sc_unsigned_bitref† & operator&= ( bool );
        sc_unsigned_bitref† & operator|= ( bool );
        sc_unsigned_bitref† & operator^= ( bool );

        // Other member functions
        void scan( std::istream& is = std::cin );

    protected:
        sc_unsigned_bitref†();
};

}       // namespace sc_dt
```

7.6.7.3 Constraints on usage

Bit-select objects shall only be created using the bit-select operators of an **sc_signed** or **sc_unsigned** object (or an instance of a class derived from **sc_signed** or **sc_unsigned**).

An application shall not explicitly create an instance of any bit-select class.

An application should not declare a reference or pointer to any bit-select object.

It is strongly recommended that an application avoid the use of a bit-select as the return type of a function because the lifetime of the object to which the bit-select refers may not extend beyond the function return statement.

Example:

```
sc_dt::sc_signed_bitref get_bit_n(sc_dt::sc_signed iv, int n) {
    return iv[n]; // Unsafe: returned bit-select references local variable
}
```

7.6.7.4 Assignment operators

Overloaded assignment operators for the lvalue bit-selects shall provide conversion from **bool** values. Assignment operators for rvalue bit-selects shall be declared as private so that they cannot be used by an application.

7.6.7.5 Implicit type conversion

operator uint64 () const;

> **operator uint64** can be used for implicit type conversion from a bit-select to a native C++ unsigned integer having exactly 64 bits. If the selected bit has the value **'1'** (true), the conversion shall return the value 1; otherwise, it shall return 0.

bool **operator!** () const;
bool **operator~** () const;

> **operator!** and **operator~** shall return a C++ bool value that is the inverse of the selected bit.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.6.7.6 Other member functions

void **scan**(std::istream& is = std::cin);

Member function **scan** shall set the value of the bit referenced by an lvalue bit-select. The value shall correspond to the C++ bool value obtained by reading the next formatted character string from the specified input stream (see 7.2.11).

void **print**(std::ostream& os = std::cout) const;

Member function **print** shall print the value of the bit referenced by the bit-select to the specified output stream (see 7.2.11). The formatting shall be implementation-defined but shall be equivalent to printing the value returned by member function **to_bool**.

int **length**() const;

Member function **length** shall unconditionally return a word length of 1 (see 7.2.5).

7.6.8 Part-selects

7.6.8.1 Description

Class *sc_signed_subref_r*† represents a signed integer part-select from an **sc_signed** used as an rvalue.

Class *sc_signed_subref*† represents a signed integer part-select from an **sc_signed** used as an lvalue.

Class *sc_unsigned_subref_r*† represents an unsigned integer part-select from an **sc_unsigned** used as an rvalue.

Class *sc_unsigned_subref*† represents an unsigned integer part-select from an **sc_unsigned** used as an lvalue.

7.6.8.2 Class definition

```
namespace sc_dt {

class sc_signed_subref_r†
: public sc_value_base†
{
    friend class sc_signed;
    friend class sc_unsigned;

    public:
        // Copy constructor
        sc_signed_subref_r†( const sc_signed_subref_r†& a );

        // Destructor
        virtual ~sc_unsigned_subref_r†();

        // Capacity
        int length() const;

        // Implicit conversion to sc_unsigned
        operator sc_unsigned () const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        // Explicit conversions
        int to_int() const;
        unsigned int to_uint() const;
        long to_long() const;
        unsigned long to_ulong() const;
        int64 to_int64() const;
        uint64 to_uint64() const;
        double to_double() const;

        // Explicit conversion to character string
        std::string to_string( sc_numrep numrep = SC_DEC ) const;
        std::string to_string( sc_numrep numrep, bool w_prefix ) const;

        // Reduce member functions
        bool and_reduce() const;
        bool nand_reduce() const;
        bool or_reduce() const;
        bool nor_reduce() const;
        bool xor_reduce() const;
        bool xnor_reduce() const;

        // Other member functions
        void print( std::ostream& os = std::cout ) const;

    protected:
        sc_signed_subref_r†();

    private:
        // Disabled
        sc_signed_subref_r† & operator= ( const sc_signed_subref_r† & );
};

// ----------------------------------------------------------------

class sc_signed_subref†
: public sc_signed_subref_r†
{
    friend class sc_signed;

    public:
        // Copy constructor
        sc_signed_subref†( const sc_signed_subref† & a );

        // Assignment operators
        sc_signed_subref† & operator= ( const sc_signed_subref_r† & a );
        sc_signed_subref† & operator= ( const sc_signed_subref† & a );
        sc_signed_subref† & operator= ( const sc_signed& a );
        template< class T >
        sc_signed_subref† & operator= ( const sc_generic_base<T>& a );
        sc_signed_subref† & operator= ( const sc_unsigned_subref_r† & a );
        sc_signed_subref† & operator= ( const sc_unsigned& a );
        sc_signed_subref† & operator= ( const char* a );
        sc_signed_subref† & operator= ( unsigned long a );
        sc_signed_subref† & operator= ( long a );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_signed_subref_d& operator= ( unsigned int a );
        sc_signed_subref_d& operator= ( int a );
        sc_signed_subref_d& operator= ( uint64 a );
        sc_signed_subref_d& operator= ( int64 a );
        sc_signed_subref_d& operator= ( double a );
        sc_signed_subref_d& operator= ( const sc_int_base& a );
        sc_signed_subref_d& operator= ( const sc_uint_base& a );

        // Other member functions
        void scan( std::istream& is = std::cin );

    private:
        // Disabled
        sc_signed_subref_d();
};

// ----------------------------------------------------------------

class sc_unsigned_subref_r_d
: public sc_value_base_d
{
    friend class sc_signed;
    friend class sc_unsigned;

    public:
        // Copy constructor
        sc_unsigned_subref_r_d( const sc_unsigned_subref_r_d& a );

        // Destructor
        virtual ~sc_unsigned_subref_r_d();

        // Capacity
        int length() const;

        // Implicit conversion to sc_unsigned
        operator sc_unsigned () const;

        // Explicit conversions
        int to_int() const;
        unsigned int to_uint() const;
        long to_long() const;
        unsigned long to_ulong() const;
        int64 to_int64() const;
        uint64 to_uint64() const;
        double to_double() const;

        // Explicit conversion to character string
        std::string to_string( sc_numrep numrep = SC_DEC ) const;
        std::string to_string( sc_numrep numrep , bool w_prefix ) const;

        // Reduce member functions
        bool and_reduce() const;
        bool nand_reduce() const;
        bool or_reduce() const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        bool nor_reduce() const;
        bool xor_reduce() const;
        bool xnor_reduce() const;

        // Other member functions
        void print( std::ostream& os = std::cout ) const;

    protected:
        sc_unsigned_subref_r†();

    private:
        // Disabled
        sc_unsigned_subref_r& operator= ( const sc_unsigned_subref_r† & );
};

// ----------------------------------------------------------------

class sc_unsigned_subref†
: public sc_unsigned_subref_r†
{
    friend class sc_unsigned;

    public:
        // Copy constructor
        sc_unsigned_subref†( const sc_unsigned_subref† & a );

        // Assignment operators
        sc_unsigned_subref† & operator= ( const sc_unsigned_subref_r† & a );
        sc_unsigned_subref† & operator= ( const sc_unsigned_subref† & a );
        sc_unsigned_subref† & operator= ( const sc_unsigned& a );
        template<class T>
        sc_unsigned_subref† & operator= ( const sc_generic_base<T>& a );
        sc_unsigned_subref† & operator= ( const sc_signed_subref_r& a );
        sc_unsigned_subref† & operator= ( const sc_signed& a );
        sc_unsigned_subref† & operator= ( const char* a );
        sc_unsigned_subref† & operator= ( unsigned long a );
        sc_unsigned_subref† & operator= ( long a );
        sc_unsigned_subref† & operator= ( unsigned int a );
        sc_unsigned_subref† & operator= ( int a );
        sc_unsigned_subref† & operator= ( uint64 a );
        sc_unsigned_subref† & operator= ( int64 a );
        sc_unsigned_subref† & operator= ( double a );
        sc_unsigned_subref† & operator= ( const sc_int_base& a );
        sc_unsigned_subref† & operator= ( const sc_uint_base& a );

        // Other member functions
        void scan( std::istream& is = std::cin );

    protected:
        sc_unsigned_subref†();
};

}       // namespace sc_dt
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.6.8.3 Constraints on usage

Integer part-select objects shall only be created using the part-select operators of an **sc_signed** or **sc_unsigned** object (or an instance of a class derived from **sc_signed** or **sc_unsigned**), as described in 7.2.7.

An application shall not explicitly create an instance of any integer part-select class.

An application should not declare a reference or pointer to any integer part-select object.

It is strongly recommended that an application avoid the use of a part-select as the return type of a function because the lifetime of the object to which the part-select refers may not extend beyond the function return statement.

Example:

```
sc_dt::sc_signed_subref get_byte(sc_dt::sc_signed s, int pos) {
    return s(pos+7, pos); // Unsafe: returned part-select references local variable
}
```

The left-hand index of a finite-precision integer part-select shall not be less than the right-hand index. The bit order in the part-select cannot be reversed.

7.6.8.4 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ integer representation to lvalue integer part-selects. If the size of a data type or string literal operand differs from the integer part-select word length, truncation, zero-extension, or sign-extension shall be used, as described in 7.2.2.

Assignment operators for rvalue integer part-selects shall be declared as private so that they cannot be used by an application.

7.6.8.5 Implicit type conversion

sc_signed_subref_r† :: **operator sc_unsigned** () const;
s*c_unsigned_subref_r*† :: **operator sc_unsigned** () const;

> **operator sc_unsigned** can be used for implicit type conversion from integer part-selects to **sc_unsigned**.
>
> NOTE—These operators are used by the output stream operator and by member functions of other data type classes that are not explicitly overloaded for finite-precision integer part-selects.

7.6.8.6 Explicit type conversion

std::string **to_string**(sc_numrep numrep = SC_DEC) const;
std::string **to_string**(sc_numrep numrep , bool w_prefix) const;

> Member function **to_string** shall perform a conversion to an **std::string** representation, as described in 7.2.12. Calling the **to_string** function with a single argument is equivalent to calling the **to_string** function with two arguments, where the second argument is **true**. Calling the **to_string** function with no arguments is equivalent to calling the **to_string** function with two arguments where the first argument is SC_DEC and the second argument is **true**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.6.8.7 Other member functions

void **scan**(std::istream& is = std::cin);

Member function **scan** shall set the values of the bits referenced by an lvalue part-select by reading the next formatted character string from the specified input stream (see 7.2.11).

void **print**(std::ostream& os = std::cout) const;

Member function **print** shall print the values of the bits referenced by the part-select to the specified output stream (see 7.2.11).

int **length**() const;

Member function **length** shall return the word length of the part-select (see 7.2.5).

7.7 Integer concatenations

7.7.1 Description

Class *sc_concatref*[†] represents a concatenation of bits from one or more objects whose concatenation base types are SystemC integers.

7.7.2 Class definition

```
namespace sc_dt {

class sc_concatref†
: public sc_generic_base<sc_concatref†>, public sc_value_base†
{
    public:
        // Destructor
        virtual ~sc_concatref†();

        // Capacity
        unsigned int length() const;

        // Explicit conversions
        int to_int() const;
        unsigned int to_uint() const;
        long to_long() const;
        unsigned long to_ulong() const;
        int64 to_int64() const;
        uint64 to_uint64() const;
        double to_double() const;
        void to_sc_signed( sc_signed& target ) const;
        void to_sc_unsigned( sc_unsigned& target ) const;

        // Implicit conversions
        operator uint64() const;
        operator const sc_unsigned&() const;

        // Unary operators
        sc_unsigned operator+ () const;
        sc_unsigned operator- () const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_unsigned operator~ () const;

        // Explicit conversion to character string
        std::string to_string( sc_numrep numrep = SC_DEC ) const;
        std::string to_string( sc_numrep numrep , bool w_prefix ) const;

        // Assignment operators
        const sc_concatref†& operator= ( int v );
        const sc_concatref†& operator= ( unsigned int v );
        const sc_concatref†& operator= ( long v );
        const sc_concatref†& operator= ( unsigned long v );
        const sc_concatref†& operator= ( int64 v );
        const sc_concatref†& operator= ( uint64 v );
        const sc_concatref†& operator= ( const sc_concatref†& v );
        const sc_concatref†& operator= ( const sc_signed& v );
        const sc_concatref†& operator= ( const sc_unsigned& v );
        const sc_concatref†& operator= ( const char* v_p );
        const sc_concatref†& operator= ( const sc_bv_base& v );
        const sc_concatref†& operator= ( const sc_lv_base& v );

        // Reduce member functions
        bool and_reduce() const;
        bool nand_reduce() const;
        bool or_reduce() const;
        bool nor_reduce() const;
        bool xor_reduce() const;
        bool xnor_reduce() const;

        // Other member functions
        void print( std::ostream& os = std::cout ) const;
        void scan( std::istream& is );

    private:
        sc_concatref†( const sc_concatref†& );
        ~sc_concatref†();
};

sc_concatref†& concat( sc_value_base†& a , sc_value_base†& b );
const sc_concatref†& concat( const sc_value_base†& a , const sc_value_base†& b );
const sc_concatref†& concat( const sc_value_base†& a, bool b );
const sc_concatref†& concat( bool a , const sc_value_base†& b );
sc_concatref†& operator, ( sc_value_base†& a , sc_value_base†& b );
const sc_concatref†& operator, ( const sc_value_base†& a , const sc_value_base†& b );
const sc_concatref†& operator, ( const sc_value_base†& a , bool b );
const sc_concatref†& operator, ( bool a , const sc_value_base†& b );

}       // namespace sc_dt
```

7.7.3 Constraints on usage

Integer concatenation objects shall only be created using the **concat** function (or **operator,**) according to the rules in 7.2.8.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

At least one of the concatenation arguments shall be an object with a SystemC integer concatenation base type, that is, an instance of a class derived directly or indirectly from class *sc_value_base*[†].

A single concatenation argument (that is, one of the two arguments to the **concat** function or **operator,**) may be a **bool** value, a reference to a **sc_core::sc_signal<bool,WRITER_POLICY>** channel, or a reference to a **sc_core::sc_in<bool>**, **sc_core::sc_inout<bool>**, or **sc_core::sc_out<bool>** port.

An application shall not explicitly create an instance of any integer concatenation class. An application shall not implicitly create an instance of any integer concatenation class by using it as a function argument or as a function return value.

An application should not declare a reference or pointer to any integer concatenation object.

7.7.4 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ integer representation to lvalue integer concatenations. If the size of a data type or string literal operand differs from the integer concatenation word length, truncation, zero-extension, or sign-extension shall be used, as described in 7.2.2.

Assignment operators for rvalue integer concatenations shall not be called by an application.

7.7.5 Implicit type conversion

operator uint64 () const;
operator const sc_unsigned& () const;

> Operators **uint64** and **sc_unsigned** shall provide implicit unsigned type conversion from an integer concatenation to a native C++ unsigned integer having exactly 64 bits or a an **sc_unsigned** object with a length equal to the total number of bits contained within the objects referenced by the concatenation.

NOTE—Enables the use of standard C++ and SystemC bitwise logical and arithmetic operators with integer concatenation objects.

7.7.6 Explicit type conversion

std::string **to_string**(sc_numrep numrep = SC_DEC) const;
std::string **to_string**(sc_numrep numrep , bool w_prefix) const;

> Member function **to_string** shall convert the object to an **std::string** representation, as described in 7.2.12. Calling the **to_string** function with a single argument is equivalent to calling the **to_string** function with two arguments, where the second argument is true. Calling the **to_string** function with no arguments is equivalent to calling the **to_string** function with two arguments, where the first argument is SC_DEC and the second argument is true.

7.7.7 Other member functions

void **scan**(std::istream& is = std::cin);

> Member function **scan** shall set the values of the bits referenced by an lvalue concatenation by reading the next formatted character string from the specified input stream (see 7.2.11).

void **print**(std::ostream& os = std::cout) const;

> Member function **print** shall print the values of the bits referenced by the concatenation to the specified output stream (see 7.2.11).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

int **length**() const;

Member function **length** shall return the word length of the concatenation (see 7.2.5).

7.8 Generic base proxy class

7.8.1 Description

Class template **sc_generic_base** provides a common proxy base class for application-defined data types that are required to be converted to a SystemC integer.

7.8.2 Class definition

```
namespace sc_dt {

template< class T >
class sc_generic_base
{
    public:
        inline const T* operator-> () const;
        inline T* operator-> ();
};

}       // namespace sc_dt
```

7.8.3 Constraints on usage

An application shall not explicitly create an instance of **sc_generic_base**.

Any application-defined type derived from **sc_generic_base** shall provide the following public const member functions:

int **length**() const;

Member function **length** shall return the number of bits required to hold the integer value.

uint64 **to_uint64**() const;

Member function **to_uint64** shall return the value as a native C++ unsigned integer having exactly 64 bits.

int64 **to_int64**() const;

Member function **to_int64** shall return the value as a native C++ signed integer having exactly 64 bits.

void **to_sc_unsigned**(sc_unsigned&) const;

Member function **to_sc_unsigned** shall return the value as an unsigned integer using the sc_unsigned argument passed by reference.

void **to_sc_signed**(sc_signed&) const;

Member function **to_sc_signed** shall return the value as a signed integer using the sc_signed argument passed by reference.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.9 Logic and vector types

7.9.1 Type definitions

The following enumerated type definition is used by the logic and vector type classes. Its literal values represent (in numerical order) the four possible logic states: *logic 0*, *logic 1*, *high-impedance*, and *unknown*, respectively. This type is not intended to be used directly by an application, which should instead use the character literals **'0'**, **'1'**, **'Z'**, and **'X'** to represent the logic states, or the application may use the constants SC_LOGIC_0, SC_LOGIC_1, SC_LOGIC_Z, and SC_LOGIC_X in contexts where the character literals would be ambiguous.

```
namespace sc_dt {

enum sc_logic_value_t
{
    Log_0 = 0,
    Log_1,
    Log_Z,
    Log_X
};

}        // namespace sc_dt
```

7.9.2 sc_logic

7.9.2.1 Description

Class **sc_logic** represents a single bit with a value corresponding to any one of the four logic states. Applications should use the character literals **'0'**, **'1'**, **'Z'**, and **'X'** to represent the states logic 0, logic 1, high-impedance, and unknown, respectively. The lowercase character literals **'z'** and **'x'** are acceptable alternatives to **'Z'** and **'X'**, respectively. Any other character used as an **sc_logic** literal shall be interpreted as the unknown state.

The C++ bool values **false** and **true** may be used as arguments to **sc_logic** constructors and operators. They shall be interpreted as logic 0 and logic 1, respectively.

Logic operations shall be permitted for **sc_logic** values following the truth tables shown in Table 16, Table 17, Table 18, and Table 19.

Table 16—sc_logic AND truth table

	'0'	**'1'**	**'Z'**	**'X'**
'0'	'0'	'0'	'0'	'0'
'1'	'0'	'1'	'X'	'X'
'Z'	'0'	'X'	'X'	'X'
'X'	'0'	'X'	'X'	'X'

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 17—sc_logic OR truth table

	'0'	**'1'**	**'Z'**	**'X'**
'0'	'0'	'1'	'X'	'X'
'1'	'1'	'1'	'1'	'1'
'Z'	'X'	'1'	'X'	'X'
'X'	'X'	'1'	'X'	'X'

Table 18—sc_logic exclusive or truth table

	'0'	**'1'**	**'Z'**	**'X'**
'0'	'0'	'1'	'X'	'X'
'1'	'1'	'0'	'X'	'X'
'Z'	'X'	'X'	'X'	'X'
'X'	'X'	'X'	'X'	'X'

Table 19—sc_logic complement truth table

	'0'	**'1'**	**'Z'**	**'X'**
	'1'	'0'	'X'	'X'

7.9.2.2 Class definition

```
namespace sc_dt {

class sc_logic
{
    public:
        // Constructors
        sc_logic();
        sc_logic( const sc_logic& a );
        sc_logic( sc_logic_value_t v );
        explicit sc_logic( bool a );
        explicit sc_logic( char a );
        explicit sc_logic( int a );

        // Destructor
        ~sc_logic();

        // Assignment operators
        sc_logic& operator= ( const sc_logic& a );
        sc_logic& operator= ( sc_logic_value_t v );
        sc_logic& operator= ( bool a );
        sc_logic& operator= ( char a );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_logic& operator= ( int a );

        // Explicit conversions
        sc_logic_value_t value() const;
        char to_char() const;
        bool to_bool() const;
        bool is_01() const;

        void print( std::ostream& os = std::cout ) const;
        void scan( std::istream& is = std::cin );

    private:
        // Disabled
        explicit sc_logic( const char* );
        sc_logic& operator= ( const char* );
};

}       // namespace sc_dt
```

7.9.2.3 Constraints on usage

An integer argument to an **sc_logic** constructor or operator shall be equivalent to the corresponding **sc_logic_value_t** enumerated value. It shall be an error if any such integer argument is outside the range of 0 to 3.

A literal value assigned to an **sc_logic** object or used to initialize an **sc_logic** object may be a character literal but not a string literal.

7.9.2.4 Constructors

sc_logic();

Default constructor **sc_logic** shall create an **sc_logic** object with a value of unknown.

sc_logic(const sc_logic& a);
sc_logic(sc_logic_value_t v);
explicit **sc_logic**(bool a);
explicit **sc_logic**(char a);
explicit **sc_logic**(int a);

Constructor **sc_logic** shall create an **sc_logic** object with the value specified by the argument.

7.9.2.5 Explicit type conversion

sc_logic_value_t **value**() const;

Member function **value** shall convert the **sc_logic** value to the **sc_logic_value_t** equivalent.

char **to_char**() const;

Member function **to_char** shall convert the **sc_logic** value to the **char** equivalent.

bool **to_bool**() const;

Member function **to_bool** shall convert the **sc_logic** value to **false** or **true**. It shall be an error to call this function if the **sc_logic** value is not logic 0 or logic 1.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

bool **is_01**() const;

Member function **is_01** shall return **true** if the **sc_logic** value is logic 0 or logic 1; otherwise, the return value shall be **false**.

7.9.2.6 Bitwise and comparison operators

The operations specified in Table 20 shall be permitted. The following applies:

— **L** represents an object of type **sc_logic**.
— **n** represents an object of type **int**, **sc_logic**, **sc_logic_value_t**, **bool**, **char**, or **int**.

NOTE—An implementation is required to supply overloaded operators on **sc_logic** objects to satisfy the requirements of this subclause. It is unspecified whether these operators are members of **sc_logic**, global operators, or provided in some other way.

Table 20—sc_logic bitwise and comparison operations

Expression	Return type	Operation
~L	const sc_logic	sc_logic bitwise complement
L & n	const sc_logic	sc_logic bitwise and
n & L	const sc_logic	sc_logic bitwise and
L &= n	sc_logic&	sc_logic assign bitwise and
L | n	const sc_logic	sc_logic bitwise or
n | L	const sc_logic	sc_logic bitwise or
L |= n	sc_logic&	sc_logic assign bitwise or
L ^ n	const sc_logic	sc_logic bitwise exclusive or
n^ L	const sc_logic	sc_logic bitwise exclusive or
L ^= n	sc_logic&	sc_logic assign bitwise exclusive or
L == n	bool	test equal
n == L	bool	test equal
L != n	bool	test not equal
n != L	bool	test not equal

7.9.2.7 Other member functions

void **scan**(std::istream& is = std::cin);

Member function **scan** shall set the value by reading the next non-white-space character from the specified input stream (see 7.2.11).

void **print**(std::ostream& os = std::cout) const;

Member function **print** shall write the value as the character literal **'0'**, **'1'**, **'X'**, or **'Z'** to the specified output stream (see 7.2.11).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.9.2.8 sc_logic constant definitions

A constant of type **sc_logic** shall be defined for each of the four possible **sc_logic_value_t** states. These constants should be used by applications to assign values to, or compare values with, other **sc_logic** objects, particularly in those cases where an implicit conversion from a C++ char value would be ambiguous.

```
namespace sc_dt {

const sc_logic SC_LOGIC_0( Log_0 );
const sc_logic SC_LOGIC_1( Log_1 );
const sc_logic SC_LOGIC_Z( Log_Z );
const sc_logic SC_LOGIC_X( Log_X );

}       // namespace sc_dt
```

Example:

```
sc_core::sc_signal<sc_dt::sc_logic> A;
A = sc_dt::SC_LOGIC_0;                  // Recommended representation of logic 0
A = static_cast<sc_dt::sc_logic>('0');  // Correct but not recommended
A = '0';                                // Error: ambiguous conversion
```

7.9.3 sc_bv_base

7.9.3.1 Description

Class **sc_bv_base** represents a finite word-length bit vector. It can be treated as an array of **bool** or as an array of **sc_logic_value_t** (with the restriction that only the states logic 0 and logic 1 are legal). The word length shall be specified by a constructor argument or, by default, by the length context object currently in scope. The word length of an **sc_bv_base** object shall be fixed during instantiation and shall not subsequently be changed.

sc_bv_base is the base class for the **sc_bv** class template.

7.9.3.2 Class definition

```
namespace sc_dt {

class sc_bv_base
{
    friend class sc_lv_base;

    public:
        // Constructors
        explicit sc_bv_base( int nb = sc_length_param().len() );
        explicit sc_bv_base( bool a, int nb = sc_length_param().len() );
        sc_bv_base( const char* a );
        sc_bv_base( const char* a , int nb );
        template <class X>
        sc_bv_base( const sc_subref_r†<X>& a );
        template <class T1, class T2>
        sc_bv_base( const sc_concref_r†<T1,T2>& a );
        sc_bv_base( const sc_lv_base& a );
        sc_bv_base( const sc_bv_base& a );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
// Destructor
virtual ~sc_bv_base();

// Assignment operators
template <class X>
sc_bv_base& operator= ( const sc_subref_r†<X>& a );
template <class T1, class T2>
sc_bv_base& operator= ( const sc_concref_r†<T1,T2>& a );
sc_bv_base& operator= ( const sc_bv_base& a );
sc_bv_base& operator= ( const sc_lv_base& a );
sc_bv_base& operator= ( const char* a );
sc_bv_base& operator= ( const bool* a );
sc_bv_base& operator= ( const sc_logic* a );
sc_bv_base& operator= ( const sc_unsigned& a );
sc_bv_base& operator= ( const sc_signed& a );
sc_bv_base& operator= ( const sc_uint_base& a );
sc_bv_base& operator= ( const sc_int_base& a );
sc_bv_base& operator= ( unsigned long a );
sc_bv_base& operator= ( long a );
sc_bv_base& operator= ( unsigned int a );
sc_bv_base& operator= ( int a );
sc_bv_base& operator= ( uint64 a );
sc_bv_base& operator= ( int64 a );

// Bitwise rotations
sc_bv_base& lrotate( int n );
sc_bv_base& rrotate( int n );

// Bitwise reverse
sc_bv_base& reverse();

// Bit selection
sc_bitref†<sc_bv_base> operator[] ( int i );
sc_bitref_r†<sc_bv_base> operator[] ( int i ) const;

// Part selection
sc_subref†<sc_bv_base> operator() ( int hi , int lo );
sc_subref_r†<sc_bv_base> operator() ( int hi , int lo ) const;

sc_subref†<sc_bv_base> range( int hi , int lo );
sc_subref_r†<sc_bv_base> range( int hi , int lo ) const;

// Reduce functions
sc_logic_value_t and_reduce() const;
sc_logic_value_t nand_reduce() const;
sc_logic_value_t or_reduce() const;
sc_logic_value_t nor_reduce() const;
sc_logic_value_t xor_reduce() const;
sc_logic_value_t xnor_reduce() const;

// Common member functions
int length() const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        // Explicit conversions to character string
        std::string to_string() const;
        std::string to_string( sc_numrep ) const;
        std::string to_string( sc_numrep , bool ) const;

        // Explicit conversions
        int to_int() const;
        unsigned int to_uint() const;
        long to_long() const;
        unsigned long to_ulong() const;
        int64 to_int64() const;
        uint64 to_uint64() const;
        bool is_01() const;

        // Other member functions
        void print( std::ostream& os = std::cout ) const;
        void scan( std::istream& is = std::cin );
};

}       // namespace sc_dt
```

7.9.3.3 Constraints on usage

Attempting to assign the **sc_logic_value_t** values high-impedance or unknown to any element of an **sc_bv_base** object shall be an error.

The result of assigning an array of bool or an array of **sc_logic** to an **sc_bv_base** object having a greater word length than the number of array elements is undefined.

7.9.3.4 Constructors

explicit **sc_bv_base**(int nb = sc_length_param().len());

Default constructor **sc_bv_base** shall create an **sc_bv_base** object of word length specified by **nb** and shall set the initial value of each element to logic **0.** This is the default constructor when **nb** is not specified (in which case its value is set by the current length context).

explicit **sc_bv_base**(bool a , int nb = sc_length_param().len());

Constructor **sc_bv_base** shall create an **sc_bv_base** object of word length specified by **nb**. If **nb** is not specified, the length shall be set by the current length context. The constructor shall set the initial value of each element to the value of **a**.

sc_bv_base(const char* a);

Constructor **sc_bv_base** shall create an **sc_bv_base** object with an initial value set by the string **a**. The word length shall be set to the number of characters in the string.

sc_bv_base(const char* a , int nb);

Constructor **sc_bv_base** shall create an **sc_bv_base** object with an initial value set by the string and word length **nb**. If the number of characters in the string does not match the value of **nb**, the initial value shall be truncated or zero extended to match the word length.

template <class X> **sc_bv_base**(const *sc_subref_r*†*<X>*& a);
template <class T1, class T2> **sc_bv_base**(const *sc_concref_r*†*<T1,T2>*& a);
sc_bv_base(const sc_lv_base& a);

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

sc_bv_base(const sc_bv_base& a);

Constructor **sc_bv_base** shall create an **sc_bv_base** object with the same word length and value as **a**.

NOTE—An implementation may provide a different set of constructors to create an **sc_bv_base** object from an *sc_subref_r*†*<T>*, *sc_concref_r*†*<T1,T2>*, or **sc_lv_base** object, for example, by providing a class template that is used as a common base class for all these types.

7.9.3.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ integer representation to **sc_bv_base**, using truncation or zero-extension, as described in 7.2.2.

7.9.3.6 Explicit type conversion

std::string **to_string**() const;
std::string **to_string**(sc_numrep) const;
std::string **to_string**(sc_numrep , bool) const;

Member function **to_string** shall perform the conversion to an **std::string** representation, as described in 7.2.12. Calling the **to_string** function with a single argument is equivalent to calling the **to_string** function with two arguments, where the second argument is **true**.

Calling the **to_string** function with no arguments shall create a binary string with a single **'1'** or **'0'** corresponding to each bit. This string shall not be prefixed by **"0b"** or a leading zero.

Example:

```
sc_dt::sc_bv_base B(4);                          // 4-bit vector
B = "0xf";                                       // Each bit set to logic 1
std::string S1 = B.to_string(sc_dt::SC_BIN, false); // The contents of S1 will be the string "01111"
std::string S2 = B.to_string(sc_dt::SC_BIN);     // The contents of S2 will be the string "0b01111"
std::string S3 = B.to_string();                  // The contents of S3 will be the string "1111"
```

bool **is_01**() const;

Member function **is_01** shall always return **true** since an **sc_bv_base** object can only contain elements with a value of logic 0 or logic 1.

Member functions that return the integer equivalent of the bit representation shall be provided to satisfy the requirements of 7.2.10.

7.9.3.7 Bitwise and comparison operators

Operations specified in Table 21 and Table 22 are permitted. The following applies:

- **B** represents an object of type **sc_bv_base**.
- **Vi** represents an object of logic vector type **sc_bv_base**, **sc_lv_base**, *sc_subref_r*†*<T>,* or *sc_concref_r*†*<T1,T2>* or integer type **int**, **long**, **unsigned int**, **unsigned long**, **sc_signed**, **sc_unsigned**, **sc_int_base**, or **sc_uint_base**.
- **i** represents an object of integer type **int**.
- **A** represents an array object with elements of type **char**, **bool**, or **sc_logic**.

The operands may also be of any other class that is derived from those just given.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 21—sc_bv_base bitwise operations

Expression	Return type	Operation
B & Vi	const sc_lv_base	sc_bv_base bitwise and
Vi & B	const sc_lv_base	sc_bv_base bitwise and
B & A	const sc_lv_base	sc_bv_base bitwise and
A & B	const sc_lv_base	sc_bv_base bitwise and
B &= Vi	sc_bv_base&	sc_bv_base assign bitwise and
B &= A	sc_bv_base&	sc_bv_base assign bitwise and
B | Vi	const sc_lv_base	sc_bv_base bitwise or
Vi | B	const sc_lv_base	sc_bv_base bitwise or
B | A	const sc_lv_base	sc_bv_base bitwise or
A | B	const sc_lv_base	sc_bv_base bitwise or
B |= Vi	sc_bv_base&	sc_bv_base assign bitwise or
B |= A	sc_bv_base&	sc_bv_base assign bitwise or
B ^ Vi	const sc_lv_base	sc_bv_base bitwise exclusive or
Vi ^ B	const sc_lv_base	sc_bv_base bitwise exclusive or
B ^ A	const sc_lv_base	sc_bv_base bitwise exclusive or
A ^ B	const sc_lv_base	sc_bv_base bitwise exclusive or
B ^= Vi	sc_bv_base&	sc_bv_base assign bitwise exclusive or
B ^= A	sc_bv_base&	sc_bv_base assign bitwise exclusive or
B << i	const sc_lv_base	sc_bv_base left-shift
B <<= i	sc_bv_base&	sc_bv_base assign left-shift
B >> i	const sc_lv_base	sc_bv_base right-shift
B >>= i	sc_bv_base&	sc_bv_base assign right-shift
~B	const sc_lv_base	sc_bv_base bitwise complement

Table 22—sc_bv_base comparison operations

Expression	Return type	Operation
B == Vi	bool	test equal
Vi == B	bool	test equal
B == A	bool	test equal
A == B	bool	test equal

Binary bitwise operators shall return a result with a word length that is equal to the word length of the longest operand.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The left shift operator shall return a result with a word length that is equal to the word length of its **sc_bv_base** operand plus the right (integer) operand. Bits added on the right-hand side of the result shall be set to zero.

The right shift operator returns a result with a word length that is equal to the word length of its **sc_bv_base** operand. Bits added on the left-hand side of the result shall be set to zero.

It is an error if the right operand of a shift operator is negative.

sc_bv_base& **lrotate**(int n);

Member function **lrotate** shall rotate an **sc_bv_base** object **n** places to the left.

sc_bv_base& **rrotate**(int n);

Member function **rrotate** shall rotate an **sc_bv_base** object **n** places to the right.

sc_bv_base& **reverse**();

Member function **reverse** shall reverse the bit order in an **sc_bv_base** object.

NOTE—An implementation is required to supply overloaded operators on **sc_bv_base** objects to satisfy the requirements of this subclause. It is unspecified whether these operators are members of **sc_bv_base**, global operators, or provided in some other way.

7.9.3.8 Other member functions

void **scan**(std::istream& is = std::cin);

Member function **scan** shall set the value by reading the next formatted character string from the specified input stream (see 7.2.11).

void **print**(std::ostream& os = std::cout) const;

Member function **print** shall write the value as a formatted character string to the specified output stream (see 7.2.11).

int **length**() const;

Member function **length** shall return the word length (see 7.2.5).

7.9.4 sc_lv_base

7.9.4.1 Description

Class **sc_lv_base** represents a finite word-length bit vector. It can be treated as an array of **sc_logic_value_t** values. The word length shall be specified by a constructor argument or, by default, by the length context object currently in scope. The word length of an **sc_lv_base** object shall be fixed during instantiation and shall not subsequently be changed.

sc_lv_base is the base class for the **sc_lv** class template.

7.9.4.2 Class definition

```
namespace sc_dt {

class sc_lv_base
{
    friend class sc_bv_base;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
public:
    // Constructors
    explicit sc_lv_base( int length_ = sc_length_param().len() );
    explicit sc_lv_base( const sc_logic& a, int length_ = sc_length_param().len() );
    sc_lv_base( const char* a );
    sc_lv_base( const char* a , int length_ );
    template <class X>
    sc_lv_base( const sc_subref_r†<X>& a );
    template <class T1, class T2>
    sc_lv_base( const sc_concref_r†<T1,T2>& a );
    sc_lv_base( const sc_bv_base& a );
    sc_lv_base( const sc_lv_base& a );

    // Destructor
    virtual ~sc_lv_base();

    // Assignment operators
    template <class X>
    sc_lv_base& operator= ( const sc_subref_r†<X>& a );
    template <class T1, class T2>
    sc_lv_base& operator= ( const sc_concref_r†<T1,T2>& a );
    sc_lv_base& operator= ( const sc_bv_base& a );
    sc_lv_base& operator= ( const sc_lv_base& a );
    sc_lv_base& operator= ( const char* a );
    sc_lv_base& operator= ( const bool* a );
    sc_lv_base& operator= ( const sc_logic* a );
    sc_lv_base& operator= ( const sc_unsigned& a );
    sc_lv_base& operator= ( const sc_signed& a );
    sc_lv_base& operator= ( const sc_uint_base& a );
    sc_lv_base& operator= ( const sc_int_base& a );
    sc_lv_base& operator= ( unsigned long a );
    sc_lv_base& operator= ( long a );
    sc_lv_base& operator= ( unsigned int a );
    sc_lv_base& operator= ( int a );
    sc_lv_base& operator= ( uint64 a );
    sc_lv_base& operator= ( int64 a );

    // Bitwise rotations
    sc_lv_base& lrotate( int n );
    sc_lv_base& rrotate( int n );

    // Bitwise reverse
    sc_lv_base& reverse();

    // Bit selection
    sc_bitref†<sc_bv_base> operator[] ( int i );
    sc_bitref_r†<sc_bv_base> operator[] ( int i ) const;

    // Part selection
    sc_subref†<sc_lv_base> operator() ( int hi , int lo );
    sc_subref_r†<sc_lv_base> operator() ( int hi , int lo ) const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    sc_subref†<sc_lv_base> range( int h i, int lo );
    sc_subref_r†<sc_lv_base> range( int hi , int lo ) const;

    // Reduce functions
    sc_logic_value_t and_reduce() const;
    sc_logic_value_t nand_reduce() const;
    sc_logic_value_t or_reduce() const;
    sc_logic_value_t nor_reduce() const;
    sc_logic_value_t xor_reduce() const;
    sc_logic_value_t xnor_reduce() const;

    // Common member functions
    int length() const;

    // Explicit conversions to character string
    std::string to_string() const;
    std::string to_string( sc_numrep ) const;
    std::string to_string( sc_numrep , bool ) const;

    // Explicit conversions
    int to_int() const;
    unsigned int to_uint() const;
    long to_long() const;
    unsigned long to_ulong() const;
    int64 to_int64() const;
    uint64 to_uint64() const;
    bool is_01() const;

    // Other member functions
    void print( std::ostream& os = std::cout ) const;
    void scan( std::istream& is = std::cin );
};

}    // namespace sc_dt
```

7.9.4.3 Constraints on usage

The result of assigning an array of **bool** or an array of **sc_logic** to an **sc_lv_base** object having a greater word length than the number of array elements is undefined.

7.9.4.4 Constructors

explicit **sc_lv_base**(int nb = sc_length_param().len());

Constructor **sc_lv_base** shall create an **sc_lv_base** object of word length specified by **nb** and shall set the initial value of each element to logic 0. This is the default constructor when **nb** is not specified (in which case its value shall be set by the current length context).

explicit **sc_lv_base**(bool a, int nb = sc_length_param().len());

Constructor **sc_lv_base** shall create an **sc_lv_base** object of word length specified by **nb** and shall set the initial value of each element to the value of **a.** If nb is not specified, the length shall be set by the current length context.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

sc_lv_base(const char* a);

Constructor **sc_lv_base** shall create an **sc_lv_base** object with an initial value set by the string literal **a**. The word length shall be set to the number of characters in the string literal.

sc_lv_base(const char* a , int nb);

Constructor **sc_lv_base** shall create an **sc_lv_base** object with an initial value set by the string literal and word length **nb**. If the number of characters in the string literal does not match the value of **nb**, the initial value shall be truncated or zero extended to match the word length.

template <class X> **sc_lv_base**(const *sc_subref_r*†*<X>*& a);
template <class T1, class T2> **sc_lv_base**(const *sc_concref_r*†*<T1,T2>*& a);

sc_lv_base(const **sc_bv_base**& a);

Constructor **sc_lv_base** shall create an **sc_lv_base** object with the same word length and value as **a**.

sc_lv_base(const **sc_lv_base**& a);

Constructor **sc_lv_base** shall create an **sc_lv_base** object with the same word length and value as **a**.

NOTE—An implementation may provide a different set of constructors to create an **sc_lv_base** object from an *sc_subref_r*†*<T>*, *sc_concref_r*†, or **sc_bv_base** object, for example, by providing a class template that is used as a common base class for all these types.

7.9.4.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ integer representation to **sc_lv_base**, using truncation or zero-extension, as described in 7.2.2.

7.9.4.6 Explicit type conversion

std::string **to_string**() const;
std::string **to_string**(sc_numrep) const;
std::string **to_string**(sc_numrep , bool) const;

Member function **to_string** shall perform a conversion to an **std::string** representation, as described in 7.2.12. Calling the **to_string** function with a single argument is equivalent to calling the **to_string** function with two arguments, where the second argument is **true**. Attempting to call the single or double argument **to_string** function for an **sc_lv_base** object with one or more elements set to the high-impedance or unknown state shall be an error.

Calling the **to_string** function with no arguments shall create a logic value string with a single **'1'**, **'0'**, **'Z'**, or **'X'** corresponding to each bit. This string shall not be prefixed by **"0b"** or a leading zero.

Example:

```
sc_dt::sc_lv_base L(4);                                  // 4-bit vector
L = "0xf";                                               // Each bit set to logic 1
std::string S1 = L.to_string(sc_dt::SC_BIN, false);  // The contents of S1 will be the string "01111"
std::string S2 = L.to_string(sc_dt::SC_BIN);         // The contents of S2 will be the string "0b01111"
std::string S3 = L.to_string();                          // The contents of S3 will be the string "1111"
```

bool **is_01**() const;

Member function **is_01** shall return **true** only when every element of an **sc_lv_base** object has a value of logic 0 or logic 1. If any element has the value high-impedance or unknown, it shall return **false**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Member functions that return the integer equivalent of the bit representation shall be provided to satisfy the requirements of 7.2.10. Calling any such integer conversion function for an object having one or more bits set to the high-impedance or unknown state shall be an error.

7.9.4.7 Bitwise and comparison operators

Operations specified in Table 23 and Table 24 are permitted. The following applies:

— **L** represents an object of type **sc_lv_base**.
— **Vi** represents an object of logic vector type **sc_bv_base**, **sc_lv_base**, *sc_subref_r*†*<T>,* or *sc_concref_r*†*<T1,T2>*, or integer type **int**, **long**, **unsigned int**, **unsigned long**, **sc_signed**, **sc_unsigned**, **sc_int_base**, or **sc_uint_base**.
— **i** represents an object of integer type **int**.
— **A** represents an array object with elements of type **char**, **bool**, or **sc_logic**.

The operands may also be of any other class that is derived from those just given.

Table 23—sc_lv_base bitwise operations

Expression	Return type	Operation
L & Vi	const sc_lv_base	sc_lv_base bitwise and
Vi & L	const sc_lv_base	sc_lv_base bitwise and
L & A	const sc_lv_base	sc_lv_base bitwise and
A & L	const sc_lv_base	sc_lv_base bitwise and
L &= Vi	sc_lv_base&	sc_lv_base assign bitwise and
L &= A	sc_lv_base&	sc_lv_base assign bitwise and
L | Vi	const sc_lv_base	sc_lv_base bitwise or
Vi | L	const sc_lv_base	sc_lv_base bitwise or
L | A	const sc_lv_base	sc_lv_base bitwise or
A | L	const sc_lv_base	sc_lv_base bitwise or
L |= Vi	sc_lv_base&	sc_lv_base assign bitwise or
L |= A	sc_lv_base&	sc_lv_base assign bitwise or
L ^ Vi	const sc_lv_base	sc_lv_base bitwise exclusive or
Vi ^ L	const sc_lv_base	sc_lv_base bitwise exclusive or
L ^ A	const sc_lv_base	sc_lv_base bitwise exclusive or
A ^ L	const sc_lv_base	sc_lv_base bitwise exclusive or
L ^= Vi	sc_lv_base&	sc_lv_base assign bitwise exclusive or
L ^= A	sc_lv_base&	sc_lv_base assign bitwise exclusive or
L << i	const sc_lv_base	sc_lv_base left-shift
L <<= i	sc_lv_base&	sc_lv_base assign left-shift

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 23—sc_lv_base bitwise operations *(continued)*

Expression	Return type	Operation
L >> i	const sc_lv_base	sc_lv_base right-shift
L >>= i	sc_lv_base&	sc_lv_base assign right-shift
~L	const sc_lv_base	sc_lv_base bitwise complement

Table 24—sc_lv_base comparison operations

Expression	Return type	Operation
L == Vi	bool	test equal
Vi == L	bool	test equal
L == A	bool	test equal
A == L	bool	test equal

Binary bitwise operators shall return a result with a word length that is equal to the word length of the longest operand.

The left shift operator shall return a result with a word length that is equal to the word length of its **sc_lv_base** operand plus the right (integer) operand. Bits added on the right-hand side of the result shall be set to zero.

The right shift operator shall return a result with a word length that is equal to the word length of its **sc_lv_base** operand. Bits added on the left-hand side of the result shall be set to zero.

It is an error if the right operand of a shift operator is negative.

sc_lv_base& **lrotate**(int n);

Member function **lrotate** shall rotate an **sc_lv_base** object **n** places to the left.

sc_lv_base& **rrotate**(int n);

Member function **rrotate** shall rotate an **sc_lv_base** object **n** places to the right.

sc_lv_base& **reverse**();

Member function **reverse** shall reverse the bit order in an **sc_lv_base** object.

NOTE—An implementation is required to supply overloaded operators on **sc_lv_base** objects to satisfy the requirements of this subclause. It is unspecified whether these operators are members of **sc_lv_base**, global operators, or provided in some other way.

7.9.4.8 Other member functions

void **scan**(std::istream& is = std::cin);

Member function **scan** shall set the value by reading the next formatted character string from the specified input stream (see 7.2.11).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

void **print**(std::ostream& os = std::cout) const;

Member function **print** shall write the value as a formatted character string to the specified output stream (see 7.2.11).

int **length**() const;

Member function **length** shall return the word length (see 7.2.5).

7.9.5 sc_bv

7.9.5.1 Description

Class template **sc_bv** represents a finite word-length bit vector. It can be treated as an array of **bool** or as an array of **sc_logic_value_t** values (with the restriction that only the states logic 0 and logic 1 are legal). The word length shall be specified by a template argument.

Any public member functions of the base class **sc_bv_base** that are overridden in class **sc_bv** shall have the same behavior in the two classes. Any public member functions of the base class not overridden in this way shall be publicly inherited by class **sc_bv**.

7.9.5.2 Class definition

```
namespace sc_dt {

template <int W>
class sc_bv
: public sc_bv_base
{
    public:
        // Constructors
        sc_bv();
        explicit sc_bv( bool init_value );
        explicit sc_bv( char init_value );
        sc_bv( const char* a );
        sc_bv( const bool* a );
        sc_bv( const sc_logic* a );
        sc_bv( const sc_unsigned& a );
        sc_bv( const sc_signed& a );
        sc_bv( const sc_uint_base& a );
        sc_bv( const sc_int_base& a );
        sc_bv( unsigned long a );
        sc_bv( long a );
        sc_bv( unsigned int a );
        sc_bv( int a );
        sc_bv( uint64 a );
        sc_bv( int64 a );
        template <class X>
        sc_bv( const sc_subref_r†<X>& a );
        template <class T1, class T2>
        sc_bv( const sc_concref_r†<T1,T2>& a );
        sc_bv( const sc_bv_base& a );
        sc_bv( const sc_lv_base& a );
        sc_bv( const sc_bv<W>& a );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        // Assignment operators
        template <class X>
        sc_bv<W>& operator= ( const sc_subref_r†<X>& a );
        template <class T1, class T2>
        sc_bv<W>& operator= ( const sc_concref_r†<T1,T2>& a );
        sc_bv<W>& operator= ( const sc_bv_base& a );
        sc_bv<W>& operator= ( const sc_lv_base& a );
        sc_bv<W>& operator= ( const sc_bv<W>& a );
        sc_bv<W>& operator= ( const char* a );
        sc_bv<W>& operator= ( const bool* a );
        sc_bv<W>& operator= ( const sc_logic* a );
        sc_bv<W>& operator= ( const sc_unsigned& a );
        sc_bv<W>& operator= ( const sc_signed& a );
        sc_bv<W>& operator= ( const sc_uint_base& a );
        sc_bv<W>& operator= ( const sc_int_base& a );
        sc_bv<W>& operator= ( unsigned long a );
        sc_bv<W>& operator= ( long a );
        sc_bv<W>& operator= ( unsigned int a );
        sc_bv<W>& operator= ( int a );
        sc_bv<W>& operator= ( uint64 a );
        sc_bv<W>& operator= ( int64 a );
};

}       // namespace sc_dt
```

7.9.5.3 Constraints on usage

Attempting to assign the **sc_logic_value_t** values high-impedance or unknown to any element of an **sc_bv** object shall be an error.

The result of assigning an array of **bool** or an array of **sc_logic** to an **sc_bv** object having a greater word length than the number of array elements is undefined.

7.9.5.4 Constructors

sc_bv();

The default constructor **sc_bv** shall create an **sc_bv** object of word length specified by the template argument **W**, and it shall set the initial value of every element to logic 0.

The other constructors shall create an **sc_bv** object of word length specified by the template argument **W** and value corresponding to the constructor argument. If the word length of a data type or string literal argument differs from the template argument, truncation or zero-extension shall be applied, as described in 7.2.2. If the number of elements in an array of **bool** or array of **sc_logic** used as the constructor argument is less than the word length, the initial value of all elements shall be undefined.

NOTE—An implementation may provide a different set of constructors to create an **sc_bv** object from an *sc_subref_r†<T>*, *sc_concref_r†<T1,T2>*, **sc_bv_base**, or **sc_lv_base** object, for example, by providing a class template that is used as a common base class for all these types.

7.9.5.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ integer representation to **sc_bv**, using truncation or zero-extension, as described in 7.2.2. The exception is assignment of an array of **bool** or an array of **sc_logic** to an **sc_bv** object, as described in 7.9.5.4.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.9.6 sc_lv

7.9.6.1 Description

Class template **sc_lv** represents a finite word-length bit vector. It can be treated as an array of **sc_logic_value_t** values. The word length shall be specified by a template argument.

Any public member functions of the base class **sc_lv_base** that are overridden in class **sc_lv** shall have the same behavior in the two classes. Any public member functions of the base class not overridden in this way shall be publicly inherited by class **sc_lv**.

7.9.6.2 Class definition

```
namespace sc_dt {

template <int W>
class sc_lv
: public sc_lv_base
{
    public:
        // Constructors
        sc_lv();
        explicit sc_lv( const sc_logic& init_value );
        explicit sc_lv( bool init_value );
        explicit sc_lv( char init_value );
        sc_lv( const char* a );
        sc_lv( const bool* a );
        sc_lv( const sc_logic* a );
        sc_lv( const sc_unsigned& a );
        sc_lv( const sc_signed& a );
        sc_lv( const sc_uint_base& a );
        sc_lv( const sc_int_base& a );
        sc_lv( unsigned long a );
        sc_lv( long a );
        sc_lv( unsigned int a );
        sc_lv( int a );
        sc_lv( uint64 a );
        sc_lv( int64 a );
        template <class X>
        sc_lv( const sc_subref_r†<X>& a );
        template <class T1, class T2>
        sc_lv( const sc_concref_r†<T1,T2>& a );
        sc_lv( const sc_bv_base& a );
        sc_lv( const sc_lv_base& a );
        sc_lv( const sc_lv<W>& a );

        // Assignment operators
        template <class X>
        sc_lv<W>& operator= ( const sc_subref_r†<X>& a );
        template <class T1, class T2>
        sc_lv<W>& operator= ( const sc_concref_r†<T1,T2>& a );
        sc_lv<W>& operator= ( const sc_bv_base& a );
        sc_lv<W>& operator= ( const sc_lv_base& a );
        sc_lv<W>& operator= ( const sc_lv<W>& a );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_lv<W>& operator= ( const char* a );
        sc_lv<W>& operator= ( const bool* a );
        sc_lv<W>& operator= ( const sc_logic* a );
        sc_lv<W>& operator= ( const sc_unsigned& a );
        sc_lv<W>& operator= ( const sc_signed& a );
        sc_lv<W>& operator= ( const sc_uint_base& a );
        sc_lv<W>& operator= ( const sc_int_base& a );
        sc_lv<W>& operator= ( unsigned long a );
        sc_lv<W>& operator= ( long a );
        sc_lv<W>& operator= ( unsigned int a );
        sc_lv<W>& operator= ( int a );
        sc_lv<W>& operator= ( uint64 a );
        sc_lv<W>& operator= ( int64 a );
};

}       // namespace sc_dt
```

7.9.6.3 Constraints on usage

The result of assigning an array of **bool** or an array of **sc_logic** to an **sc_lv** object having a greater word length than the number of array elements is undefined.

7.9.6.4 Constructors

sc_lv();

Default constructor **sc_lv** shall create an **sc_lv** object of word length specified by the template argument **W** and shall set the initial value of every element to unknown.

The other constructors shall create an **sc_lv** object of word length specified by the template argument **W** and value corresponding to the constructor argument. If the word length of a data type or string literal argument differs from the template argument, truncation or zero-extension shall be applied, as described in 7.2.2. If the number of elements in an array of **bool** or array of **sc_logic** used as the constructor argument is less than the word length, the initial value of all elements shall be undefined.

NOTE—An implementation may provide a different set of constructors to create an **sc_lv** object from an *sc_subref_r*†*<T>*, *sc_concref_r*†*<T1,T2>*, **sc_bv_base**, or **sc_lv_base** object, for example, by providing a class template that is used as a common base class for all these types.

7.9.6.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ integer representation to **sc_lv**, using truncation or zero-extension, as described in 7.2.2. The exception is assignment from an array of **bool** or an array of **sc_logic** to an **sc_lv** object, as described in 7.9.6.4.

7.9.7 Bit-selects

7.9.7.1 Description

Class template *sc_bitref_r*†*<T>* represents a bit selected from a vector used as an rvalue.

Class template *sc_bitref*†*<T>* represents a bit selected from a vector used as an lvalue.

The use of the term “vector” here includes part-selects and concatenations of bit vectors and logic vectors. The template parameter is the name of the class accessed by the bit-select.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.9.7.2 Class definition

```
namespace sc_dt {

template <class T>
class sc_bitref_r†
{
    friend class sc_bv_base;
    friend class sc_lv_base;

    public:
        // Copy constructor
        sc_bitref_r†( const sc_bitref_r†<T>& a );

        // Bitwise complement
        sc_logic operator~ () const;

        // Implicit conversion to sc_logic
        operator sc_logic() const;

        // Explicit conversions
        bool is_01() const;
        bool to_bool() const;
        char to_char() const;
        explicit operator bool() const;
        bool operator!() const;

        // Common member functions
        int length() const;

        // Other member functions
        void print( std::ostream& os = std::cout ) const;

    private:
        // Disabled
        sc_bitref_r†();
        sc_bitref_r†<T>& operator= ( const sc_bitref_r†<T>& );
};

// ----------------------------------------------------------------

template <class T>
class sc_bitref†
: public sc_bitref_r†<T>
{
    friend class sc_bv_base;
    friend class sc_lv_base;

    public:
        // Copy constructor
        sc_bitref†( const sc_bitref†<T>& a );

        // Assignment operators
        sc_bitref†<T>& operator= ( const sc_bitref_r†<T>& a );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_bitref<T>& operator= ( const sc_bitref<T>& a );
        sc_bitref<T>& operator= ( const sc_logic& a );
        sc_bitref<T>& operator= ( sc_logic_value_t v );
        sc_bitref<T>& operator= ( bool a );
        sc_bitref<T>& operator= ( char a );
        sc_bitref<T>& operator= ( int a );

        // Bitwise assignment operators
        sc_bitref<T>& operator&= ( const sc_bitref_r<T>& a );
        sc_bitref<T>& operator&= ( const sc_logic& a );
        sc_bitref<T>& operator&= ( sc_logic_value_t v );
        sc_bitref<T>& operator&= ( bool a );
        sc_bitref<T>& operator&= ( char a );
        sc_bitref<T>& operator&= ( int a );

        sc_bitref<T>& operator|= ( const sc_bitref_r<T>& a );
        sc_bitref<T>& operator|= ( const sc_logic& a );
        sc_bitref<T>& operator|= ( sc_logic_value_t v );
        sc_bitref<T>& operator|= ( bool a );
        sc_bitref<T>& operator|= ( char a );
        sc_bitref<T>& operator|= ( int a );

        sc_bitref<T>& operator^= ( const sc_bitref_r<T>& a );
        sc_bitref<T>& operator^= ( const sc_logic& a );
        sc_bitref<T>& operator^= ( sc_logic_value_t v );
        sc_bitref<T>& operator^= ( bool a );
        sc_bitref<T>& operator^= ( char a );
        sc_bitref<T>& operator^= ( int a );

        // Other member functions
        void scan( std::istream& is = std::cin );

    private:
        // Disabled
        sc_bitref();
};

}       // namespace sc_dt
```

7.9.7.3 Constraints on usage

Bit-select objects shall only be created using the bit-select operators of an **sc_bv_base** or **sc_lv_base** object (or an instance of a class derived from **sc_bv_base** or **sc_lv_base**) or a part-select or concatenation thereof, as described in 7.2.7.

An application shall not explicitly create an instance of any bit-select class.

An application should not declare a reference or pointer to any bit-select object.

It is strongly recommended that an application avoid the use of a bit-select as the return type of a function because the lifetime of the object to which the bit-select refers may not extend beyond the function return statement.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Example:

```
sc_dt::sc_bitref<sc_dt::sc_bv_base> get_bit_n(sc_dt::sc_bv_base bv, int n) {
    return bv[n]; // Unsafe: returned bit-select references local variable
}
```

7.9.7.4 Assignment operators

Overloaded assignment operators for the lvalue bit-select shall provide conversion to **sc_logic_value_t** values. The assignment operator for the rvalue bit-select shall be declared as private so that it cannot be used by an application.

7.9.7.5 Implicit type conversion

operator sc_logic() const;

Operator **sc_logic** shall create an **sc_logic** object with the same value as the bit-select.

7.9.7.6 Explicit type conversion

char **to_char**() const;

Member function **to_char** shall convert the bit-select value to the **char** equivalent.

bool **to_bool**() const;

Member function **to_bool** shall convert the bit-select value to **false** or **true**. It shall be an error to call this function if the **sc_logic** value is not logic 0 or logic 1.

bool **is_01**() const;

Member function **is_01** shall return **true** if the **sc_logic** value is logic 0 or logic 1; otherwise, the return value shall be **false**.

explicit **operator bool**() const;

Operator bool shall provide explicit conversion to a value of type bool.

bool **operator!**() const;

Operator! shall provide explicit negating conversion to a value of type bool.

7.9.7.7 Bitwise and comparison operators

Operations specified in Table 25 are permitted. The following applies:

B represents an object of type *sc_bitref_r*†<*T*> (or any derived class).

NOTE—An implementation is required to supply overloaded operators on *sc_bitref_r*†<*T*> objects to satisfy the requirements of this subclause. It is unspecified whether these operators are members of *sc_bitref_r*†<*T*>, global operators, or provided in some other way.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 25—sc_bitref_r†<T> bitwise and comparison operations

Expression	Return type	Operation
B & B	const sc_logic	*sc_bitref_r†<T>* bitwise and
B | B	const sc_logic	*sc_bitref_r†<T>* bitwise or
B ^ B	const sc_logic	*sc_bitref_r†<T>* bitwise exclusive or
B == B	bool	test equal
B != B	bool	test not equal

7.9.7.8 Other member functions

void **scan**(std::istream& is = std::cin);

Member function **scan** shall set the value of the bit referenced by an lvalue bit-select. The value shall correspond to the C++ bool value obtained by reading the next formatted character string from the specified input stream (see 7.2.11).

void **print**(std::ostream& os = std::cout) const;

Member function **print** shall print the value of the bit referenced by the bit-select to the specified output stream (see 7.2.11). The formatting shall be implementation-defined but shall be equivalent to printing the value returned by member function **to_bool**.

int **length**() const;

Member function **length** shall unconditionally return a word length of 1 (see 7.2.5).

7.9.8 Part-selects

7.9.8.1 Description

Class template *sc_subref_r†<T>* represents a part-select from a vector used as an rvalue.

Class template *sc_subref†<T>* represents a part-select from a vector used as an lvalue.

The use of the term “vector” here includes part-selects and concatenations of bit vectors and logic vectors. The template parameter is the name of the class accessed by the part-select.

The set of operations that can be performed on a part-select shall be identical to that of its associated vector (subject to the constraints that apply to rvalue objects).

7.9.8.2 Class definition

```
namespace sc_dt {

template <class T>
class sc_subref_r†
{
    public:
        // Copy constructor
        sc_subref_r†( const sc_subref_r†<T>& a );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        // Bit selection
        sc_bitref_r†<sc_subref_r†<T>> operator[] ( int i ) const;

        // Part selection
        sc_subref_r†<sc_subref_r†<T>> operator() ( int hi , int lo ) const;

        sc_subref_r†<sc_subref_r†<T>> range( int hi , int lo ) const;

        // Reduce functions
        sc_logic_value_t and_reduce() const;
        sc_logic_value_t nand_reduce() const;
        sc_logic_value_t or_reduce() const;
        sc_logic_value_t nor_reduce() const;
        sc_logic_value_t xor_reduce() const;
        sc_logic_value_t xnor_reduce() const;

        // Common member functions
        int length() const;

        // Explicit conversions to character string
        std::string to_string() const;
        std::string to_string( sc_numrep ) const;
        std::string to_string( sc_numrep , bool ) const;

        // Explicit conversions
        int to_int() const;
        unsigned int to_uint() const;
        long to_long() const;
        unsigned long to_ulong() const;
        int64 to_int64() const;
        uint64 to_uint64() const;
        bool is_01() const;

        // Other member functions
        void print( std::ostream& os = std::cout ) const;
        bool reversed() const;

    private:
        // Disabled
        sc_subref_r†();
        sc_subref_r†<T>& operator= ( const sc_subref_r†<T>& );
};

// ----------------------------------------------------------------

template <class T>
class sc_subref†
: public sc_subref_r†<T>
{
    public:
        // Copy constructor
        sc_subref†( const sc_subref†<T>& a );

        // Assignment operators
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        template <class T>
        sc_subref†<T>& operator= ( const sc_subref_r†<T>& a );
        template <class T1, class T2>
        sc_subref†<T>& operator= ( const sc_concref_r†<T1,T2>& a );
        sc_subref†<T>& operator= ( const sc_bv_base& a );
        sc_subref†<T>& operator= ( const sc_lv_base& a );
        sc_subref†<T>& operator= ( const sc_subref_r†<T>& a );
        sc_subref†<T>& operator= ( const sc_subref†<T>& a );
        sc_subref†<T>& operator= ( const char* a );
        sc_subref†<T>& operator= ( const bool* a );
        sc_subref†<T>& operator= ( const sc_logic* a );
        sc_subref†<T>& operator= ( const sc_unsigned& a );
        sc_subref†<T>& operator= ( const sc_signed& a );
        sc_subref†<T>& operator= ( const sc_uint_base& a );
        sc_subref†<T>& operator= ( const sc_int_base& a );
        sc_subref†<T>& operator= ( unsigned long a );
        sc_subref†<T>& operator= ( long a );
        sc_subref†<T>& operator= ( unsigned int a );
        sc_subref†<T>& operator= ( int a );
        sc_subref†<T>& operator= ( uint64 a );
        sc_subref†<T>& operator= ( int64 a );

        // Bitwise rotations
        sc_subref†<T>& lrotate( int n );
        sc_subref†<T>& rrotate( int n );

        // Bitwise reverse
        sc_subref†<T>& reverse();

        // Bit selection
        sc_bitref†<sc_subref†<T>> operator[] ( int i );

        // Part selection
        sc_subref†<sc_subref†<T>> operator() ( int hi , int lo );

        sc_subref†<sc_subref†<T>> range( int hi , int lo );

        // Other member functions
        void scan( std::istream& = std::cin );

    private:
        // Disabled
        sc_subref†();
};

}       // namespace sc_dt
```

7.9.8.3 Constraints on usage

Part-select objects shall only be created using the part-select operators of an **sc_bv_base** or **sc_lv_base** object (or an instance of a class derived from **sc_bv_base** or **sc_lv_base**) or a part-select or concatenation thereof, as described in 7.2.7.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

An application shall not explicitly create an instance of any part-select class.

An application should not declare a reference or pointer to any part-select object.

An rvalue part-select shall not be used to modify the vector with which it is associated.

It is strongly recommended that an application avoid the use of a part-select as the return type of a function because the lifetime of the object to which the part-select refers may not extend beyond the function return statement.

Example:

```
sc_dt::sc_subref<sc_dt::sc_bv_base> get_byte(sc_dt::sc_bv_base bv, int pos) {
    return bv(pos+7, pos);                          // Unsafe: returned part-select references local variable
}
```

The left-hand index of a vector type part-select shall not be less than the right-hand index. The bit order in the part-select cannot be reversed.

7.9.8.4 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ integer representation to lvalue part-selects. If the size of a data type or string literal operand differs from the part-select word length, truncation or zero-extension shall be used, as described in 7.2.2. If an array of **bool** or array of **sc_logic** is assigned to a part-select and its number of elements is less than the part-select word length, the value of the part-select shall be undefined.

The default assignment operator for an rvalue part-select is private so that it cannot be used by an application.

7.9.8.5 Explicit type conversion

std::string **to_string**() const;
std::string **to_string**(sc_numrep) const;
std::string **to_string**(sc_numrep , bool) const;

> Member function **to_string** shall convert to an **std::string** representation, as described in 7.2.12. Calling the **to_string** function with a single argument is equivalent to calling the **to_string** function with two arguments, where the second argument is **true**. Attempting to call the single or double argument **to_string** function for a part-select with one or more elements set to the high-impedance or unknown state shall be an error.
>
> Calling the **to_string** function with no arguments shall create a logic value string with a single **'1'**, **'0'**, **'Z'** , or **'X'** corresponding to each bit. This string shall not prefixed by **"0b"** or a leading zero.

bool **is_01**() const;

> Member function **is_01** shall return **true** only when every element of a part-select has a value of logic 0 or logic 1. If any element has the value high-impedance or unknown, it shall return **false**.
>
> Member functions that return the integer equivalent of the bit representation shall be provided to satisfy the requirements of 7.2.10. Calling any such integer conversion function for an object having one or more bits set to the high-impedance or unknown state shall be an error.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.9.8.6 Bitwise and comparison operators

Operations specified in Table 26 and Table 28 are permitted for all vector part-selects. Operations specified in Table 27 are permitted for lvalue vector part-selects only. The following applies:

— **P** represents an lvalue or rvalue vector part-select.
— **L** represents an lvalue vector part-select.
— **Vi** represents an object of logic vector type **sc_bv_base**, **sc_lv_base**, *sc_subref_r*†<*T*>, or *sc_concref_r*†<*T1,T2*>, or integer type **int**, **long**, **unsigned int**, **unsigned long**, **sc_signed**, **sc_unsigned**, **sc_int_base**, or **sc_uint_base**.
— **i** represents an object of integer type **int**.
— **A** represents an array object with elements of type **char**, **bool**, or **sc_logic**.

The operands may also be of any other class that is derived from those just given.

Table 26—sc_subref_r†<T> bitwise operations

Expression	Return type	Operation
P & Vi	const sc_lv_base	*sc_subref_r*†<*T*> bitwise and
Vi & P	const sc_lv_base	*sc_subref_r*†<*T*> bitwise and
P & A	const sc_lv_base	*sc_subref_r*†<*T*> bitwise and
A & P	const sc_lv_base	*sc_subref_r*†<*T*> bitwise and
P | Vi	const sc_lv_base	*sc_subref_r*†<*T*> bitwise or
Vi | P	const sc_lv_base	*sc_subref_r*†<*T*> bitwise or
P | A	const sc_lv_base	*sc_subref_r*†<*T*> bitwise or
A | P	const sc_lv_base	*sc_subref_r*†<*T*> bitwise or
P ^ Vi	const sc_lv_base	*sc_subref_r*†<*T*> bitwise exclusive or
Vi ^ P	const sc_lv_base	*sc_subref_r*†<*T*> bitwise exclusive or
P ^ A	const sc_lv_base	*sc_subref_r*†<*T*> bitwise exclusive or
A ^ P	const sc_lv_base	*sc_subref_r*†<*T*> bitwise exclusive or
P << i	const sc_lv_base	*sc_subref_r*†<*T*> left-shift
P >> i	const sc_lv_base	*sc_subref_r*†<*T*> right-shift
~P	const sc_lv_base	*sc_subref_r*†<*T*> bitwise complement

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 27—sc_subref†<T> bitwise operations

Expression	Return type	Operation
L &= Vi	*sc_subref_r†<T>*&	*sc_subref_r†<T>* assign bitwise and
L &= A	*sc_subref_r†<T>*&	*sc_subref_r†<T>* assign bitwise and
L |= Vi	*sc_subref_r†<T>*&	*sc_subref_r†<T>* assign bitwise or
L |= A	*sc_subref_r†<T>*&	*sc_subref_r†<T>* assign bitwise or
L ^= Vi	*sc_subref_r†<T>*&	*sc_subref_r†<T>* assign bitwise exclusive or
L ^= A	*sc_subref_r†<T>*&	*sc_subref_r†<T>* assign bitwise exclusive or
L <<= i	*sc_subref_r†<T>*&	*sc_subref_r†<T>* assign left-shift
L >>= i	*sc_subref_r†<T>*&	*sc_subref_r†<T>* assign right-shift

Table 28—sc_subref_r†<T> comparison operations

Expression	Return type	Operation
P == Vi	bool	test equal
Vi == P	bool	test equal
P == A	bool	test equal
A == P	bool	test equal

Binary bitwise operators shall return a result with a word length that is equal to the word length of the longest operand.

The left shift operator shall return a result with a word length that is equal to the word length of its part-select operand plus the right (integer) operand. Bits added on the right-hand side of the result shall be set to zero.

The right shift operator shall return a result with a word length that is equal to the word length of its part-select operand. Bits added on the left-hand side of the result shall be set to zero.

It is an error if the right operand of a shift operator is negative.

sc_subref†<T>& **lrotate**(int n);

Member function **lrotate** shall rotate an lvalue part-select n places to the left.

sc_subref†<T>& **rrotate**(int n);

Member function **rrotate** shall rotate an lvalue part-select n places to the right.

sc_subref†<T>& **reverse**();

Member function **reverse** shall reverse the bit order in an lvalue part-select.

NOTE—An implementation is required to supply overloaded operators on *sc_subref_r†<T>* and *sc_subref†<T>* objects to satisfy the requirements of this subclause. It is unspecified whether these operators are members of *sc_subref†<T>*, members of *sc_subref†<T>*, global operators, or provided in some other way.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.9.8.7 Other member functions

void **scan**(std::istream& is = std::cin);

Member function **scan** shall set the values of the bits referenced by an lvalue part-select by reading the next formatted character string from the specified input stream (see 7.2.11).

void **print**(std::ostream& os = std::cout) const;

Member function **print** shall print the values of the bits referenced by the part-select to the specified output stream (see 7.2.11).

int **length**() const;

Member function **length** shall return the word length of the part-select (see 7.2.5).

bool **reversed**() const;

Member function **reversed** shall return **true** if the elements of a part-select are in the reverse order to those of its associated vector (if the left-hand index used to form the part-select is less than the right-hand index); otherwise, the return value shall be **false**.

7.9.9 Concatenations

7.9.9.1 Description

Class template *sc_concref_r†<T1,T2>* represents a concatenation of bits from one or more vector used as an rvalue.

Class template *sc_concref†<T1,T2>* represents a concatenation of bits from one or more vector used as an lvalue.

The use of the term “vector” here includes part-selects and concatenations of bit vectors and logic vectors. The template parameters are the class names of the two vectors used to create the concatenation.

The set of operations that can be performed on a concatenation shall be identical to that of its associated vectors (subject to the constraints that apply to rvalue objects).

7.9.9.2 Class definition

```
namespace sc_dt {

template <class T1, class T2>
class sc_concref_r†
{
    public:
        // Copy constructor
        sc_concref_r†( const sc_concref_r†<T1,T2>& a );

        // Destructor
        virtual ~sc_concref_r†();

        // Bit selection
        sc_bitref_r†<sc_concref_r†<T1,T2>> operator[] ( int i ) const;

        // Part selection
        sc_subref_r†<sc_concref_r†<T1,T2>> operator() ( int hi , int lo ) const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_subref_r†<sc_concref_r†<T1,T2>> range( int hi , int lo ) const;

        // Reduce functions
        sc_logic_value_t and_reduce() const;
        sc_logic_value_t nand_reduce() const;
        sc_logic_value_t or_reduce() const;
        sc_logic_value_t nor_reduce() const;
        sc_logic_value_t xor_reduce() const;
        sc_logic_value_t xnor_reduce() const;

        // Common member functions
        int length() const;

        // Explicit conversions to character string
        std::string to_string() const;
        std::string to_string( sc_numrep ) const;
        std::string to_string( sc_numrep , bool ) const;

        // Explicit conversions
        int to_int() const;
        unsigned int to_uint() const;
        long to_long() const;
        unsigned long to_ulong() const;
        int64 to_int64() const;
        uint64 to_uint64() const;
        bool is_01() const;

        // Other member functions
        void print( std::ostream& os = std::cout ) const;

    private:
        // Disabled
        sc_concref†();
        sc_concref_r†<T1,T2>& operator= ( const sc_concref_r†<T1,T2>& );
};

// ----------------------------------------------------------------

template <class T1, class T2>
class sc_concref†
: public sc_concref_r†<T1,T2>
{
    public:
        // Copy constructor
        sc_concref†( const sc_concref†<T1,T2>& a );

        // Assignment operators
        template <class T>
        sc_concref†<T1,T2>& operator= ( const sc_subref_r†<T>& a );
        template <class T1, class T2>
        sc_concref†<T1,T2>& operator= ( const sc_concref_r†<T1,T2>& a );
        sc_concref†<T1,T2>& operator= ( const sc_bv_base& a );
        sc_concref†<T1,T2>& operator= ( const sc_lv_base& a );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_concref_r<T1,T2>& operator= ( const sc_concref_r<T1,T2>& a );
        sc_concref_r<T1,T2>& operator= ( const char* a );
        sc_concref_r<T1,T2>& operator= ( const bool* a );
        sc_concref_r<T1,T2>& operator= ( const sc_logic* a );
        sc_concref_r<T1,T2>& operator= ( const sc_unsigned& a );
        sc_concref_r<T1,T2>& operator= ( const sc_signed& a );
        sc_concref_r<T1,T2>& operator= ( const sc_uint_base& a );
        sc_concref_r<T1,T2>& operator= ( const sc_int_base& a );
        sc_concref_r<T1,T2>& operator= ( unsigned long a );
        sc_concref_r<T1,T2>& operator= ( long a );
        sc_concref_r<T1,T2>& operator= ( unsigned int a );
        sc_concref_r<T1,T2>& operator= ( int a );
        sc_concref_r<T1,T2>& operator= ( uint64 a );
        sc_concref_r<T1,T2>& operator= ( int64 a );

        // Bitwise rotations
        sc_concref_r<T1,T2>& lrotate( int n );
        sc_concref_r<T1,T2>& rrotate( int n );

        // Bitwise reverse
        sc_concref_r<T1,T2>& reverse();

        // Bit selection
        sc_bitref_r<sc_concref_r<T1,T2>> operator[] ( int i );

        // Part selection
        sc_subref_r<sc_concref_r<T1,T2>> operator() ( int hi , int lo );

        sc_subref_r<sc_concref_r<T1,T2>> range( int hi , int lo );

        // Other member functions
        void scan( std::istream& = std::cin );

    private:
        // Disabled
        sc_concref_r();
};

// r-value concatenation operators and functions

template <typename C1, typename C2>
sc_concref_r<C1,C2> operator, ( C1 , C2 );

template <typename C1, typename C2>
sc_concref_r<C1,C2> concat( C1 , C2 );

// l-value concatenation operators and functions

template <typename C1, typename C2>
sc_concref<C1,C2> operator, ( C1 , C2 );

template <typename C1, typename C2>
sc_concref<C1,C2> concat( C1 , C2 );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
}       // namespace sc_dt
```

7.9.9.3 Constraints on usage

Concatenation objects shall only be created using the **concat** function (or **operator,**) according to the rules in 7.2.8. The concatenation arguments shall be objects with a common concatenation base type of **sc_bv_base** or **sc_lv_base** (or an instance of a class derived from **sc_bv_base** or **sc_lv_base**) or a part-select or concatenation of them.

An application shall not explicitly create an instance of any concatenation class.

An application should not declare a reference or pointer to any concatenation object.

An rvalue concatenation shall be created when any argument to the **concat** function (or **operator,**) is an rvalue. An rvalue concatenation shall not be used to modify any vector with which it is associated.

It is strongly recommended that an application avoid the use of a concatenation as the return type of a function because the lifetime of the objects to which the concatenation refer may not extend beyond the function return statement.

Example:

```
sc_dt::sc_concref_r<sc_dt::sc_bv_base, sc_dt::sc_bv_base> pad(sc_dt::sc_bv_base& bv, char pchar) {
    const sc_dt::sc_bv<4> padword(pchar);                  // Unsafe: returned concatenation references
                                                           // a non-static local variable (padword)
    return concat(bv,padword);
}
```

7.9.9.4 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ integer representation to lvalue concatenations. If the size of a data type or string literal operand differs from the concatenation word length, truncation or zero-extension shall be used, as described in 7.2.2. If an array of **bool** or array of **sc_logic** is assigned to a concatenation and its number of elements is less than the concatenation word length, the value of the concatenation shall be undefined.

The default assignment operator for an rvalue concatenation shall be declared as private so that it cannot be used by an application.

7.9.9.5 Explicit type conversion

std::string **to_string**() const;
std::string **to_string**(sc_numrep) const;
std::string **to_string**(sc_numrep , bool) const;

Member function **to_string** shall perform the conversion to an **std::string** representation, as described in 7.2.12. Calling the **to_string** function with a single argument is equivalent to calling the **to_string** function with two arguments, where the second argument is **true**. Attempting to call the single or double argument **to_string** function for a concatenation with one or more elements set to the high-impedance or unknown state shall be an error.

Calling the **to_string** function with no arguments shall create a logic value string with a single **'1'**, **'0'**, **'Z'**, or **'X'** corresponding to each bit. This string shall not prefixed by **"0b"** or a leading zero.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

bool **is_01**() const;

Member function **is_01** shall return **true** only when every element of a concatenation has a value of logic 0 or logic 1. If any element has the value high-impedance or unknown, it shall return **false**.

Member functions that return the integer equivalent of the bit representation shall be provided to satisfy the requirements of 7.2.10. Calling any such integer conversion function for an object having one or more bits set to the high-impedance or unknown state shall be an error.

7.9.9.6 Bitwise and comparison operators

Operations specified in Table 29 and Table 31 are permitted for all vector concatenations; operations specified in Table 30 are permitted for lvalue vector concatenations only. The following applies:

- **C** represents an lvalue or rvalue vector concatenation.
- **L** represents an lvalue vector concatenation.
- **Vi** represents an object of logic vector type **sc_bv_base**, **sc_lv_base**, *sc_subref_r†<T>*, or *sc_concref_r†<T1,T2>*, or integer type **int**, **long**, **unsigned int**, **unsigned long**, **sc_signed**, **sc_unsigned**, **sc_int_base**, or **sc_uint_base**.
- **i** represents an object of integer type **int**.
- **A** represents an array object with elements of type **char**, **bool**, or **sc_logic**.

The operands may also be of any other class that is derived from those just given.

Table 29—sc_concref_r†<T1,T2> bitwise operations

Expression	Return type	Operation
C & Vi	const sc_lv_base	*sc_concref_r†<T1,T2>* bitwise and
Vi & C	const sc_lv_base	*sc_concref_r†<T1,T2>* bitwise and
C & A	const sc_lv_base	*sc_concref_r†<T1,T2>* bitwise and
A & C	const sc_lv_base	*sc_concref_r†<T1,T2>* bitwise and
C | Vi	const sc_lv_base	*sc_concref_r†<T1,T2>* bitwise or
Vi | C	const sc_lv_base	*sc_concref_r<T1,T2>* bitwise or
C | A	const sc_lv_base	*sc_concref_r<T1,T2>* bitwise or
A | C	const sc_lv_base	*sc_concref_r<T1,T2>* bitwise or
C ^ Vi	const sc_lv_base	*sc_concref_r<T1,T2>* bitwise exclusive or
Vi ^ C	const sc_lv_base	*sc_concref_r<T1,T2>* bitwise exclusive or
C ^ A	const sc_lv_base	*sc_concref_r<T1,T2>* bitwise exclusive or
A ^ C	const sc_lv_base	*sc_concref_r<T1,T2>* bitwise exclusive or
C << i	const sc_lv_base	*sc_concref_r<T1,T2>* left-shift
C >> i	const sc_lv_base	*sc_concref_r<T1,T2>* right-shift
~C	const sc_lv_base	*sc_concref_r<T1,T2>* bitwise complement

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 30—sc_concref†<T1,T2> bitwise operations

Expression	Return type	Operation
L &= Vi	*sc_concref†<T1,T2>*&	*sc_concref†<T1,T2>* assign bitwise and
L &= A	*sc_concref†<T1,T2>*&	*sc_concref†<T1,T2>* assign bitwise and
L |= Vi	*sc_concref†<T1,T2>*&	*sc_concref†<T1,T2>* assign bitwise or
L |= A	*sc_concref†<T1,T2>*&	*sc_concref†<T1,T2>* assign bitwise or
L ^= Vi	*sc_concref†<T1,T2>*&	*sc_concref†<T1,T2>* assign bitwise exclusive or
L ^= A	*sc_concref†<T1,T2>*&	*sc_concref†<T1,T2>* assign bitwise exclusive or
L <<= i	*sc_concref†<T1,T2>*&	*sc_concref†<T1,T2>* assign left-shift
L >>= i	*sc_concref†<T1,T2>*&	*sc_concref†<T1,T2>* assign right-shift

Table 31—sc_concref_r†<T1,T2> comparison operations

Expression	Return type	Operation
C == Vi	bool	test equal
Vi == C	bool	test equal
C == A	bool	test equal
A == C	bool	test equal

Binary bitwise operators shall return a result with a word length that is equal to the word length of the longest operand.

The left shift operator shall return a result with a word length that is equal to the word length of its concatenation operand plus the right (integer) operand. Bits added on the right-hand side of the result shall be set to zero.

The right shift operator shall return a result with a word length that is equal to the word length of its concatenation operand. Bits added on the left-hand side of the result shall be set to zero.

sc_concref†<T1,T2>& **lrotate**(int n);

Member function **lrotate** shall rotate an lvalue part-select n places to the left.

sc_concref†<T1,T2>& **rrotate**(int n);

Member function **rrotate** shall rotate an lvalue part-select n places to the right.

sc_concref†<T1,T2>& **reverse**();

Member function **reverse** shall reverse the bit order in an lvalue part-select.

NOTE—An implementation is required to supply overloaded operators on *sc_concref_r†<T1,T2>* and *sc_concref†<T1,T2>* objects to satisfy the requirements of this subclause. It is unspecified whether these operators are members of *sc_concref_r†<T1,T2>*, members of *sc_concref†<T1,T2>*, global operators, or provided in some other way.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.9.9.7 Other member functions

void **scan**(std::istream& is = std::cin);

Member function **scan** shall set the values of the bits referenced by an lvalue concatenation by reading the next formatted character string from the specified input stream (see 7.2.11).

void **print**(std::ostream& os = std::cout) const;

Member function **print** shall print the values of the bits referenced by the concatenation to the specified output stream (see 7.2.11).

int **length**() const;

Member function **length** shall return the word length of the concatenation (see 7.2.5).

7.9.9.8 concat and operator,

template <typename C1, typename C2>
sc_concref_r†*<C1,C2>* **operator,** (C1 , C2);

template <typename C1, typename C2>
sc_concref_r†*<C1,C2>* **concat**(C1 , C2);

template <typename C1, typename C2>
sc_concref‡*<C1,C2>* **operator,** (C1 , C2);

template <typename C1, typename C2>
sc_concref‡*<C1,C2>* **concat**(C1 , C2);

Explicit template specializations of function **concat** and **operator,** shall be provided for all permitted concatenations. Attempting to concatenate any two objects that do not have an explicit template specialization for function **concat** or **operator,** defined shall be an error.

A template specialization for rvalue concatenations shall be provided for all combinations of concatenation argument types C1 and C2. Argument C1 has a type in the following list:

sc_bitref_r†*<T>*
sc_subref_r†*<T>*
sc_concref_r†*<T1,T2>*
const sc_bv_base&
const sc_lv_base&

Argument C2 has one of the types just given or a type in the following list:

sc_bitref‡*<T>*
sc_subref‡*<T>*
sc_concref‡*<T1,T2>*
sc_bv_base&
sc_lv_base&

Additional template specializations for rvalue concatenations shall be provided for the cases where a single argument has type **bool**, **const char***, or **const sc_logic&**. This argument shall be implicitly converted to an equivalent single-bit **const sc_lv_base** object.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

A template specialization for lvalue concatenations shall be provided for all combinations of concatenation argument types C1 and C2, where each argument has a type in the following list:

sc_bitref†*<T>*
sc_subref†*<T>*
sc_concref†*<T1,T2>*
sc_bv_base&
sc_lv_base&

7.10 Fixed-point types

7.10.1 Overview

Subclause 7.10 describes the fixed-point types and the operations and conventions imposed by these types.

7.10.2 Fixed-point representation

In the SystemC binary fixed-point representation, a number shall be represented by a sequence of bits with a specified position for the binary point. Bits to the left of the binary point shall represent the integer part of the number, and bits to the right of the binary point shall represent the fractional part of the number.

A SystemC fixed-point type shall be characterized by the following:

— The word length (**wl**), which shall be the total number of bits in the number representation.
— The integer word length (**iwl**), which shall be the number of bits in the integer part (the position of the binary point relative to the left-most bit).
— The bit encoding (which shall be signed, two's compliment, or unsigned).

The right-most bit of the number shall be the least significant bit (LSB), and the left-most bit shall be the most significant bit (MSB).

The binary point may be located outside of the data bits. That is, the binary point may be a number of bit positions to the right of the LSB or it may be a number of bit positions to the left of the MSB.

The fixed-point representation can be interpreted according to the following three cases:

— **wl < iwl**

  There are **(iwl-wl)** zeros between the LSB and the binary point. See index 1 in Table 32 for an example of this case.

— **0 <= iwl <= wl**

  The binary point is contained within the bit representation. See index 2, 3, 4, and 5 in Table 32 for examples of this case.

— **iwl < 0**

  There are **(-iwl)** sign-extended bits between the binary point and the MSB. For an unsigned type, the sign-extended bits are zero. For a signed type, the extended bits repeat the MSB. See index 6 and 7 in Table 32 for examples of this case.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 32—Examples of fixed-point formats

Index	wl	iwl	Fixed-point representation[a]	Range signed	Ranged unsigned
1	5	7	xxxxx00.	[–64,60]	[0,124]
2	5	5	xxxxx.	[–16,15]	[0,31]
3	5	3	xxx.xx	[–4,3.75]	[0,7.75]
4	5	1	x.xxxx	[–1,0.9375]	[0,1.9375]
5	5	0	.xxxxx	[–0.5,0.46875]	[0,0.96875]
6	5	–2	.ssxxxxx	[0.125,0.1171875]	[0,0.2421875]
7	1	–1	.sx	[–0.25,0]	[0,0.25]

[a] **x** is an arbitrary binary digit, 0, or 1. **s** is a sign-extended digit, 0, or 1.

The MSB in the fixed-point representation of a signed type shall be the sign bit. The sign bit may be behind the binary point.

The range of values for a signed fixed-point format shall be given by the following:

$$[-2^{(\mathrm{iwl}-1)}f\ ,2^{(\mathrm{iwl}-1)}-2^{-(\mathrm{wl}-\mathrm{iwl})}]$$

The range of values for a unsigned fixed-point format shall be given by the following:

$$[0\ \ ,2^{(\mathrm{iwl})}-2^{-(\mathrm{wl}-\mathrm{iwl})}]$$

7.10.3 Fixed-point type conversion

Fixed-point type conversion (conversion of a value to a specific fixed-point representation) shall be performed whenever a value is assigned to a fixed-point type variable (including initialization).

If the magnitude of the value is outside the range of the fixed-point representation, or the value has greater precision than the fixed-point representation provides, it shall be mapped (converted) to a value that can be represented. This conversion shall be performed in two steps:

a) If the value is within range but has greater precision (it is between representable values), quantization shall be performed to reduce the precision.
b) If the magnitude of the value is outside the range, overflow handling shall be performed to reduce the magnitude.

If the target fixed-point representation has greater precision, the additional least significant bits shall be zero extended. If the target fixed-point representation has a greater range, sign extension or zero extension shall be performed for signed and unsigned fixed-point types, respectively, to extend the representation of their most significant bits.

Multiple quantization modes (distinct quantization characteristics) and multiple overflow modes (distinct overflow characteristics) are defined (see 7.10.10.2 and 7.10.10.10).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.4 Fixed-point data types

7.10.4.1 Overview

Subclause 7.10.4 describes the classes that are provided to represent fixed-point values.

7.10.4.2 Finite-precision fixed-point types

The following finite- and variable-precision fixed-point data types shall be provided:

sc_fixed<wl,iwl,q_mode,o_mode,n_bits>
sc_ufixed<wl,iwl,q_mode,o_mode,n_bits>
sc_fix
sc_ufix
sc_fxval

These types shall be parameterized as to the fixed-point representation (**wl**, **iwl**) and fixed-point conversion modes (**q_mode**, **o_mode**, **n_bits**). The declaration of a variable of one of these types shall specify the values for these parameters. The type parameter values of a variable shall not be modified after the variable declaration. Any data value assigned to the variable shall be converted to specified representation (with the specified word length and binary point location) with the specified quantization and overflow processing (**q_mode**, **o_mode**, **n_bits**) applied if required.

The finite-precision fixed-point types have a common base class **sc_fxnum**. An application or implementation shall not directly create an object of type **sc_fxnum**. A reference or pointer to class **sc_fxnum** may be used to access an object of any type derived from **sc_fxnum**.

The type **sc_fxval** is a variable-precision type. A variable of type **sc_fxval** may store a fixed-point value of arbitrary width and binary point location. A value assigned to a **sc_fxval** variable shall be stored without a loss of precision or magnitude (the value shall not be modified by quantization or overflow handling).

Types **sc_fixed**, **sc_fix**, and **sc_fxval** shall have a signed (two's compliment) representation. Types **sc_ufixed** and **sc_ufix** have an unsigned representation.

A fixed-point variable that is declared without an initial value shall be uninitialized. Uninitialized variables may be used wherever the use of an initialized variable is permitted. The result of an operation on an uninitialized variable shall be undefined.

7.10.4.3 Limited-precision fixed-point types

The following limited-precision versions of the fixed-point types shall be provided:

sc_fixed_fast<wl,iwl,q_mode,o_mode,n_bits>
sc_ufixed_fast<wl,iwl,q_mode,o_mode,n_bits>
sc_fix_fast
sc_ufix_fast
sc_fxval_fast

The limited-precision types shall use the same semantics as the finite-precision fixed-point types. Finite-precision and limited-precision types may be mixed freely in expressions. A variable of a limited-precision type shall be a legal replacement in any expression where a variable of the corresponding finite-precision fixed-point type is expected.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The limited-precision fixed-point value shall be held in an implementation-dependent native C++ floating-point type. An implementation shall provide a minimum length of 53 bits to represent the mantissa.

NOTE—For bit-true behavior with the limited-precision types, the word length of the result of any operation or expression should not exceed 53 bits.

7.10.5 Fixed-point expressions and operations

Fixed-point operations shall be performed using variable-precision fixed-point values; that is, the evaluation of a fixed-point operator shall proceed as follows (except as noted below for specific operators):

— The operands shall be converted (promoted) to variable-precision fixed-point values.
— The operation shall be performed, computing a variable-precision fixed-point result. The result shall be computed so that there is no loss of precision or magnitude (that is, sufficient bits are computed to precisely represent the result).

The right-hand side of a fixed-point assignment shall be evaluated as a variable-precision fixed-point value that is converted to the fixed-point representation specified by the target of the assignment.

If all the operands of a fixed-point operation are limited-precision types, a limited-precision operation shall be performed. This operation shall use limited variable-precision fixed-point values (**sc_fxval_fast**), and the result shall be a limited variable-precision fixed-point value.

The right operand of a fixed-point shift operation (the shift amount) shall be of type **int**. If a fixed-point shift operation is called with a fixed-point value for the right operand, the fractional part of the value shall be truncated (no quantization).

The result of the equality and relational operators shall be type **bool**.

Fixed-point operands of a bitwise operator shall be of a finite- or limited-precision type (they shall not be variable precision). Furthermore, both operands of a binary bitwise operator shall have the same sign representation (both signed or both unsigned). The result of a fixed-point bitwise operation shall be either **sc_fix** or **sc_ufix** (or **sc_fix_fast** or **sc_ufix_fast**), depending on the sign representation of the operands. For binary operators, the two operands shall be aligned at the binary point. The operands shall be temporarily extended (if necessary) to have the same integer word length and fractional word length. The result shall have the same integer and fractional word lengths as the temporarily extended operands.

The remainder operator (%) is not supported for fixed-point types.

The permitted operators are given in Table 33. The following applies:

— **A** represents a fixed-point object.
— **B** and **C** represent appropriate numeric values or objects.
— **s1**, **s2**, **s3** represent signed finite- or limited-precision fixed-point objects.
— **u1**, **u2**, **u3** represent unsigned finite- or limited-precision fixed-point objects.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 33—Fixed-point arithmetic and bitwise functions

Expression	Operation
A = B + C;	Addition with assignment
A = B - C;	Subtraction with assignment
A = B * C;	Multiplication with assignment
A = B / C;	Division with assignment
A = B << i;	Left shift with assignment
A = B >> i;	Right shift with assignment
s1 = s2 & s3;	Bitwise and with assignment for signed operands
s1 = s2 | s3;	Bitwise or with assignment for signed operands
s1 = s2 ^ s3;	Bitwise exclusive-or with assignment for signed operands
u1 = u2 & u3;	Bitwise and with assignment for unsigned operands
u1 = u2 | u3;	Bitwise or with assignment for unsigned operands
u1 = u2 ^ u3;	Bitwise exclusive-or with assignment for unsigned operands

The operands of arithmetic fixed-point operations may be combinations of the types listed in Table 34, Table 35, Table 36, and Table 37.

The addition operations specified in Table 34 are permitted for finite-precision fixed-point objects. The following applies:

— **F**, **F1**, **F2** represent objects derived from type **sc_fxnum**.
— **n** represents an object of numeric type **int**, **long**, **unsigned int**, **unsigned long**, **float**, **double**, **sc_signed**, **sc_unsigned**, **sc_int_base**, **sc_uint_base**, **sc_fxval**, or **sc_fxval_fast**, or an object derived from **sc_fxnum_fast** or a numeric string literal.

The operands may also be of any other class that is derived from those just given.

Table 34—Finite-precision fixed-point addition operations

Expression	Operation
F = F1 + F2;	sc_fxnum addition, sc_fxnum assign
F1 += F2;	sc_fxnum assign addition
F1 = F2 + n;	sc_fxnum addition, sc_fxnum assign
F1 = n + F2;	sc_fxnum addition, sc_fxnum assign
F += n;	sc_fxnum assign addition

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The addition operations specified in Table 35 are permitted for variable-precision fixed-point objects. The following applies:

— **V**, **V1**, **V2** represent objects of type **sc_fxval**.
— **n** represents an object of numeric type **int**, **long**, **unsigned int**, **unsigned long**, **float**, **double**, **sc_signed**, **sc_unsigned**, **sc_int_base**, **sc_uint_base**, or **sc_fxval_fast**, or an object derived from **sc_fxnum_fast** or a numeric string literal.

The operands may also be of any other class that is derived from those just given.

Table 35—Variable-precision fixed-point addition operations

Expression	Operation
V = V1 + V2;	sc_fxval addition, sc_fxval assign
V1 += V2;	sc_fxval assign addition
V1 = V2 + n;	sc_fxval addition, sc_fxval assign
V1 = n + V2;	sc_fxval addition, sc_fxval assign
V += n;	sc_fxval assign addition

The addition operations specified in Table 36 are permitted for limited-precision fixed-point objects. The following applies:

— **F**, **F1**, **F2** represent objects derived from type **sc_fxnum_fast**.
— **n** represents an object of numeric type **int**, **long**, **unsigned int**, **unsigned long**, **float**, **double**, **sc_signed**, **sc_unsigned**, **sc_int_base**, **sc_uint_base**, or **sc_fxval_fast,** or a numeric string literal.

The operands may also be of any other class that is derived from those just given.

Table 36—Limited-precision fixed-point addition operations

Expression	Operation
F = F1 + F2;	sc_fxnum_fast addition, sc_fxnum_fast assign
F1 += F2;	sc_fxnum_fast assign addition
F1 = F2 + n;	sc_fxnum_fast addition, sc_fxnum_fast assign
F1 = n + F2;	sc_fxnum_fast addition, sc_fxnum_fast assign
F += n;	sc_fxnum_fast assign addition

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The addition operations specified in Table 37 are permitted for limited variable-precision fixed-point objects. The following applies:

— **V**, **V1**, **V2** represent objects of type **sc_fxval_fast**.
— **n** represents an object of numeric type **int**, **long**, **unsigned int**, **unsigned long**, **float**, **double**, **sc_signed**, **sc_unsigned**, **sc_int_base**, or **sc_uint_base**, or a numeric string literal.

The operands may also be of any other class that is derived from those just given.

Table 37—Limited variable-precision fixed-point addition operations

Expression	Operation
V = V1 + V2;	sc_fxval_fast addition, sc_fxval_fast assign
V1 += V2;	sc_fxval_fast assign addition
V1 = V2 + n;	sc_fxval_fast addition, sc_fxval_fast assign
V1 = n + V2;	sc_fxval_fast addition, sc_fxval_fast assign
V += n;	sc_fxval_fast assign addition

Subtraction, multiplication, and division operations are also permitted with the same combinations of operand types as listed in Table 34, Table 35, Table 36, and Table 37.

7.10.6 Bit and part selection

Bit and part selection shall be supported for the fixed-point types, as described in 7.2.6 and 7.2.7. They are not supported for the variable-precision fixed-point types **sc_fxval** or **sc_fxval_fast**.

If the left-hand index of a part-select is less than the right-hand index, the bit order of the part-select shall be reversed.

A part-select may be created with an unspecified range (the **range** function or **operator()** is called with no arguments). In this case, the part-select shall have the same word length and same value as its associated fixed-point object.

7.10.7 Variable-precision fixed-point value limits

In some cases, such as division, using variable precision could lead to infinite word lengths. An implementation should provide an appropriate mechanism to define the maximum permitted word length of a variable-precision value and to detect when this maximum word length is reached.

The action taken by an implementation when a variable-precision value reaches its maximum word length is undefined. The result of any operation that causes a variable-precision value to reach its maximum word length shall be the implementation-dependent representable value nearest to the ideal (infinite precision) result.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.8 Fixed-point word length and mode

7.10.8.1 Overview

```
namespace sc_dt {
    enum { SC_ON, SC_OFF };
}
```

The default word length, quantization mode, and saturation mode of a fixed-point type shall be set by the fixed-point type parameter (**sc_fxtype_param**) in context at the point of construction as described in 7.2.4. The fixed-point type parameter shall have a field corresponding to the fixed-point representation (**wl**,**.iwl**) and fixed-point conversion modes (**q_mode**, **o_mode**, **n_bits**). Default values for these fields shall be defined according to Table 38.

Table 38—Built-in default values

Parameter	Value
wl	32
iwl	32
q_mode	SC_TRN
o_mode	SC_WRAP
n_bits	0

The behavior of a fixed-point object in arithmetic operations may be set to emulate that of a floating-point variable by the *floating-point cast switch* in context at its point of construction. A floating-point cast switch shall be brought into context by creating a *floating-point cast context* object. **sc_fxcast_switch** and **sc_fxcast_context** shall be used to create floating-point cast switches and floating-point cast contexts, respectively (see 7.11.6 and 7.11.7).

A global floating-point cast context stack shall manage floating-point cast contexts using the same semantics as the length context stack described in 7.2.4.

A floating-point cast switch may be initialized to the value SC_ON or SC_OFF. These shall cause the arithmetic behavior to be fixed-point or floating-point, respectively. A default floating-point context with the value SC_ON shall be defined.

Example:

```
sc_dt::sc_fxtype_params fxt(32, 16);
sc_dt::sc_fxtype_context fcxt(fxt);
sc_dt::sc_fix A, B, res;                    // wl = 32, iwl = 16
A = 10.0;
B = 0.1;
res = A * B;                                // res = .999908447265625

sc_dt::sc_fxcast_switch fxs(sc_dt::SC_OFF);
sc_dt::sc_fxcast_context fccxt(fxs);
sc_dt::sc_fix C, D;                         // Floating-point behavior
C = 10.0;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
D = 0.1;
res = C * D;                                    // res = 1
```

7.10.8.2 Reading parameter settings

The following functions are defined for every finite-precision fixed-point object and limited-precision fixed-point object and shall return its current parameter settings (at runtime).

const sc_fxcast_switch& **cast_switch()** const;

Member function **cast_switch** shall return the cast switch parameter.

int **iwl()** const;

Member function **iwl** shall return the integer word-length parameter.

int **n_bits()** const;

Member function **n_bits** shall return the number of saturated bits parameter.

sc_o_mode **o_mode()** const;

Member function **o_mode** shall return the overflow mode parameter using the enumerated type **sc_o_mode**, defined as follows:

```
enum sc_o_mode
{
    SC_SAT,             // Saturation
    SC_SAT_ZERO,        // Saturation to zero
    SC_SAT_SYM,         // Symmetrical saturation
    SC_WRAP,            // Wrap-around (*)
    SC_WRAP_SM          // Sign magnitude wrap-around (*)
};
```

sc_q_mode **q_mode()** const;

Member function **q_mode** shall return the quantization mode parameter using the enumerated type **sc_q_mode**, defined as follows:

```
enum sc_q_mode
{
    SC_RND,             // Rounding to plus infinity
    SC_RND_ZERO,        // Rounding to zero
    SC_RND_MIN_INF,     // Rounding to minus infinity
    SC_RND_INF,         // Rounding to infinity
    SC_RND_CONV,        // Convergent rounding
    SC_TRN,             // Truncation
    SC_TRN_ZERO         // Truncation to zero
};
```

const sc_fxtype_params& **type_params()** const;

Member function **type_params** shall return the type parameters.

int **wl()** const;

Member function **wl** shall return the total word-length parameter.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.8.3 Value attributes

The following functions are defined for every fixed-point object and shall return its current value attributes.

bool **is_neg()** const;

Member function **is_neg** shall return **true** if the object holds a negative value; otherwise, the return value shall be **false**.

bool **is_zero()** const;

Member function **is_zero** shall return **true** if the object holds a zero value; otherwise, the return value shall be **false**.

bool **overflow_flag()** const;

Member function **overflow_flag** shall return **true** if the last write action on this objects caused overflow; otherwise, the return value shall be **false**.

bool **quantization_flag()** const;

Member function **quantization_flag** shall return **true** if the last write action on this object caused quantization; otherwise, the return value shall be **false**.

The following function is defined for every finite-precision fixed-point object and shall return its current value:

sc_fxval **value()** const;

The following function is defined for every limited-precision fixed-point object and shall return its current value:

sc_fxval_fast **value()** const;

7.10.9 Conversions to character string

7.10.9.1 Overview

Conversion to character string of the fixed-point types shall be supported by the member function **to_string**, as described in 7.3.

The member function **to_string** for fixed-point types may be called with an additional argument to specify the string format. This argument shall be of enumerated type **sc_fmt** and shall always be at the right-hand side of the argument list.

enum **sc_fmt** { SC_F, SC_E };

The default value for **fmt** shall be SC_F for the finite- and limited-precision fixed-point types. For types **sc_fxval** and **sc_fxval_fast**, the default value for **fmt** shall be SC_E.

The selected format shall give different character strings only when the binary point is not located within the *wl* bits. In that case, either sign extension (MSB side) or zero extension (LSB side) shall be done (SC_F format), or exponents shall be used (SC_E format).

In conversion to SC_DEC number representation or conversion from a variable-precision variable, only those characters necessary to represent the value uniquely shall be generated. In converting the value of a finite- or limited-precision variable to a binary, octal, or hex representation, the number of characters used

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

shall be determined by the integer and fractional widths (iwl, fwl) of the variable (with sign or zero extension as needed).

Example:

```
sc_dt::sc_fixed<7,4> a = –1.5;
a.to_string(sc_dt::SC_DEC);                          // –1.5
a.to_string(sc_dt::SC_BIN);                          // 0b1110.100
sc_dt::sc_fxval b(–1.5);
b.to_string(sc_dt::SC_BIN);                          // 0b10.1
sc_dt::sc_fixed<4,6> c = 20;
c.to_string(sc_dt::SC_BIN, false, sc_dt::SC_F);      // 010100
c.to_string(sc_dt::SC_BIN, false, sc_dt::SC_E);      // 0101e+2
```

7.10.9.2 String shortcut member functions

Four shortcut member functions to the member function **to_string** shall be provided for frequently used combinations of arguments. The shortcut member functions are listed in Table 39.

Table 39—Shortcut member functions

Shortcut member function	Number representation
to_dec()	SC_DEC
to_bin()	SC_BIN
to_oct()	SC_OCT
to_hex()	SC_HEX

The shortcut member functions shall use the default string formatting.

Example:

```
sc_dt::sc_fixed<4,2> a = –1;
a.to_dec();                  // Returns std::string with value "–1"
a.to_bin();                  // Returns std::string with value "0b11.00"
```

7.10.9.3 Bit-pattern string conversion

Bit-pattern strings may be assigned to fixed-point part-selects. The result of assigning a bit-pattern string to a fixed-point object (except using a part-select) is undefined.

If the number of characters in the bit-pattern string is less than the part-select word length, the string shall be zero extended at its left-hand side to the part-select word length.

7.10.10 Finite word-length effects

7.10.10.1 Overview

Subclause 7.10.10 describes the overflow and quantization modes of SystemC.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.10.2 Overflow modes

Overflow shall occur when the magnitude of a value being assigned to a limited-precision variable exceeds the fixed-point representation. In SystemC, specific overflow modes shall be available to control the mapping to a representable value.

The mutually exclusive overflow modes listed in Table 40 shall be provided. The default overflow mode shall be SC_WRAP. When using a wrap-around overflow mode, the number of saturated bits (n_bits) shall by default be set to 0 but can be modified.

Table 40—Overflow modes

Overflow mode	Name
Saturation	SC_SAT
Saturation to zero	SC_SAT_ZERO
Symmetrical saturation	SC_SAT_SYM
Wrap-around[a]	SC_WRAP
Sign magnitude wrap-around[a]	SC_WRAP_SM

[a] With 0 or n_bits saturated bits (n_bits > 0). The default value for n_bits is 0.

In the following subclauses, each overflow mode is explained in more detail. A figure is given to explain the behavior graphically. The x axis shows the input values, and the y axis represents the output values. Together they determine the overflow mode.

To facilitate the explanation of each overflow mode, the concepts MIN and MAX are used:

— In the case of signed representation, MIN is the lowest (negative) number that may be represented; MAX is the highest (positive) number that may be represented with a certain number of bits. A value **x** shall lie in the range:
  -2^{n-1} **(= MIN) <= x <=** $(2^{n-1} - 1)$ **(= MAX)**
  where **n** indicates the number of bits.
— In the case of unsigned representation, MIN shall equal 0 and MAX shall equal $2^n - 1$, where **n** indicates the number of bits.

7.10.10.3 Overflow for signed fixed-point numbers

The following template contains a signed fixed-point number before and after an overflow mode has been applied and a number of flags. The flags are explained below the template. The flags between parentheses indicate the additional optional properties of a bit.

Before	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*
After:						*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*
Flags:	*sD*	*D*	*D*	*D*	*lD*	*sR*	*R(N)*	*R(IN)*	*R*	*R*	*R*	*R*	*R*	*R*	*R*	*R*	*lR*

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The following flags and symbols are used in the template just given and in Table 41:

— *x* represents a binary digit (0 or 1).
— *sD* represents a sign bit before overflow handling.
— *D* represents deleted bits.
— *lD* represents the least significant deleted bit.
— *sR* represents the bit on the MSB position of the result number. For the SC_WRAP_SM, 0 and SC_WRAP_SM, 1 modes, a distinction is made between the original value (*sRo*) and the new value (*sRn*) of this bit.
— *N* represents the saturated bits. Their number is equal to the *n_bits* argument minus 1. They are always taken after the sign bit of the result number. The *n_bits* argument is only taken into account for the SC_WRAP and SC_WRAP_SM overflow modes.
— *lN* represents the least significant saturated bit. This flag is only relevant for the SC_WRAP and SC_WRAP_SM overflow modes. For the other overflow modes, these bits are treated as R-bits. For the SC_WRAP_SM, *n_bits* > 1 mode, *lNo* represents the original value of this bit.
— *R* represents the remaining bits.
— *lR* represents the least significant remaining bit.

Table 41—Overflow handling for signed fixed-point numbers

Overflow mode	Result		
	Sign bit (*sR*)	**Saturated bits (*N*, *lN*)**	**Remaining bits (*R*, *lR*)**
SC_SAT	sD		! sD
	The result number gets the sign bit of the original number. The remaining bits shall get the inverse value of the sign bit.		
SC_SAT_ZERO	0		0
	All bits shall be set to zero.		
SC_SAT_SYM	sD		! sD,
	The result number shall get the sign bit of the original number. The remaining bits shall get the inverse value of the sign bit, except the least significant remaining bit, which shall be set to one.		
SC_WRAP, (n_bits =) 0	sR		x
	All bits except for the deleted bits shall be copied to the result.		
SC_WRAP, (n_bits =) 1	sD		x
	The result number shall get the sign bit of the original number. The remaining bits shall be copied from the original number.		
SC_WRAP, n_bits > 1	sD	! sD	x
	The result number shall get the sign bit of the original number. The saturated bits shall get the inverse value of the sign bit of the original number. The remaining bits shall be copied from the original number.		

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 41—Overflow handling for signed fixed-point numbers *(continued)*

Overflow mode	Result		
	Sign bit (*sR*)	**Saturated bits (*N*, *lN*)**	**Remaining bits (*R*, *lR*)**
SC_WRAP_SM, (n_bits =) 0	lD		x ^ sRo ^ sRn
	The sign bit of the result number shall get the value of the least significant deleted bit. The remaining bits shall be XORed with the original and the new value of the sign bit of the result.		
SC_WRAP_SM, (n_bits =) 1	sD		x ^ sRo ^ sRn
	The result number shall get the sign bit of the original number. The remaining bits shall be XORed with the original and the new value of the sign bit of the result.		
SC_WRAP_SM, n_bits > 1	sD	! sD	x ^INo ^ ! sD
	The result number shall get the sign bit of the original number. The saturated bits shall get the inverse value of the sign bit of the original number. The remaining bits shall be XORed with the original value of the least significant saturated bit and the inverse value of the original sign bit.		

Overflow shall occur when the value of at least one of the deleted bits (*sD*, *D*, *lD*) is not equal to the original value of the bit on the MSB position of the result (*sRo*).

Table 41 shows how a signed fixed-point number shall be cast (in case there is an overflow) for each of the possible overflow modes. The operators used in the table are "!" for a bitwise negation and "^" for a bitwise exclusive-OR.

7.10.10.4 Overflow for unsigned fixed-point numbers

The following template contains an unsigned fixed-point number before and after an overflow mode has been applied and a number of flags. The flags are explained below the template.

Before	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*
After:						*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*
Flags:	*D*	*D*	*D*	*D*	*lD*	*R(N)*	*R(N)*	*R(IN)*	*R*	*R*	*R*	*R*	*R*	*R*	*R*	*R*	*R*

The following flags and symbols are used in the template just given and in Table 42:

- *x* represents an binary digit (0 or 1).
- *D* represents deleted bits.
- *lD* represents the least significant deleted bit.
- *N* represents the saturated bits. Their number is equal to the *n_bits* argument. The *n_bits* argument is only taken into account for the SC_WRAP and SC_WRAP_SM overflow modes.
- *R* represents the remaining bits.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 42—Overflow handling for unsigned fixed-point numbers

Overflow mode	Result	
	Saturated bits (*N*)	**Remaining bits (*R*)**
SC_SAT		1 (overflow) 0 (underflow)
	The remaining bits shall be set to 1 (overflow) or 0 (underflow).	
SC_SAT_ZERO		0
	The remaining bits shall be set to 0.	
SC_SAT_SYM		1 (overflow) 0 (underflow)
	The remaining bits shall be set to 1 (overflow) or 0 (underflow).	
SC_WRAP, (*n_bits* =) 0		x
	All bits except for the deleted bits shall be copied to the result number.	
SC_WRAP, *n_bits* > 0	1	x
	The saturated bits of the result number shall be set to 1. The remaining bits shall be copied to the result.	
SC_WRAP_SM	Not defined for unsigned numbers.	

Table 42 shows how an unsigned fixed-point number shall be cast in case there is an overflow for each of the possible overflow modes.

During the conversion from signed to unsigned, sign extension shall occur before overflow handling, while in the unsigned to signed conversion, zero extension shall occur first.

7.10.10.5 SC_SAT

The SC_SAT overflow mode shall be used to indicate that the output is saturated to MAX in case of overflow, or to MIN in the case of negative overflow. Figure 2 illustrates the SC_SAT overflow mode for a word length of three bits. The x axis represents the word length before rounding; the y axis represents the word length after rounding. The ideal situation is represented by the diagonal dashed line.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

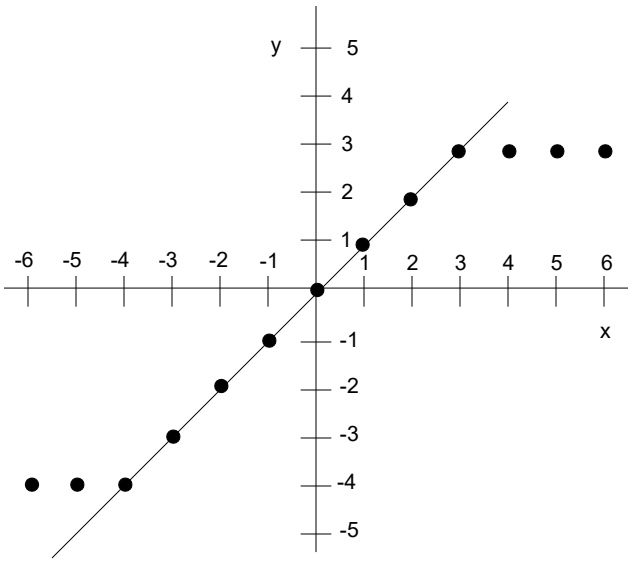

Figure 2—Saturation for signed numbers

Examples (signed, 3-bit number):

before saturation: **0110 (6)**

after saturation: **011 (3)**

There is an overflow because the decimal number 6 is outside the range of values that can be represented exactly by means of three bits. The result is then rounded to the highest positive representable number, which is 3.

before saturation: **1011 (–5)**

after saturation: **100 (–4)**

There is an overflow because the decimal number –5 is outside the range of values that can be represented exactly by means of three bits. The result is then rounded to the lowest negative representable number, which is –4.

Example (unsigned, 3-bit number):

before saturation: **01110 (14)**

after saturation: **111 (7)**

The SC_SAT mode corresponds to the SC_WRAP and SC_WRAP_SM modes with the number of bits to be saturated equal to the number of kept bits.

7.10.10.6 SC_SAT_ZERO

The SC_SAT_ZERO overflow mode shall be used to indicate that the output is forced to zero in case of an overflow, that is, if MAX or MIN is exceeded. Figure 3 illustrates the SC_SAT_ZERO overflow mode for a word length of three bits. The x axis represents the word length before rounding; the y axis represents the word length after rounding.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

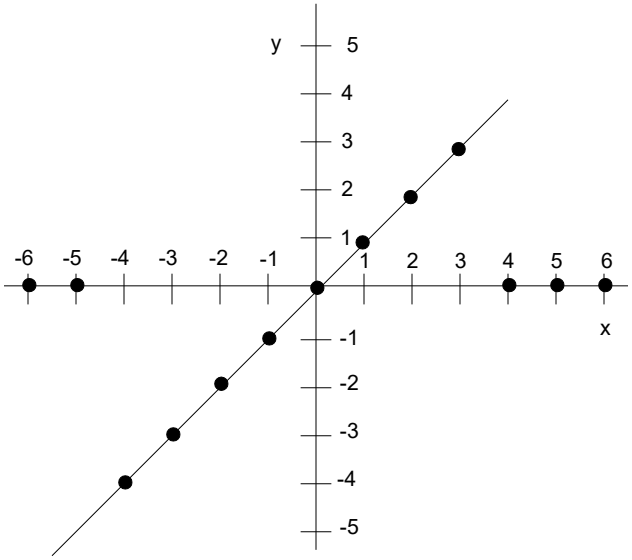

Figure 3—Saturation to zero for signed numbers

Examples (signed, 3-bit number):

before saturation to zero: **0110 (6)**

after saturation to zero: **000 (0)**

There is an overflow because the decimal number 6 is outside the range of values that can be represented exactly by means of three bits. The result is saturated to zero.

before saturation to zero: **1011 (–5)**

after saturation to zero: **000 (0)**

There is an overflow because the decimal number –5 is outside the range of values that can be represented exactly by means of three bits. The result is saturated to zero.

Example (unsigned, 3-bit number):

before saturation to zero: **01110 (14)**

after saturation to zero: **000 (0)**

7.10.10.7 SC_SAT_SYM

The SC_SAT_SYM overflow mode shall be used to indicate that the output is saturated to MAX in case of overflow, to –MAX (signed) or MIN (unsigned) in the case of negative overflow. Figure 4 illustrates the SC_SAT_SYM overflow mode for a word length of three bits. The x axis represents the word length before rounding; the y axis represents the word length after rounding. The ideal situation is represented by the diagonal dashed line.

Examples (signed, 3-bit number):

after symmetrical saturation: **0110 (6)**

after symmetrical saturation: **011 (3)**

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

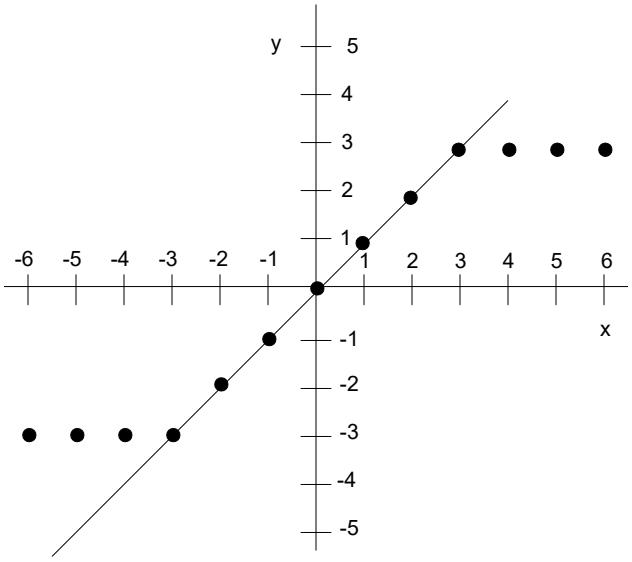

Figure 4—Symmetrical saturation for signed numbers

There is an overflow because the decimal number 6 is outside the range of values that can be represented exactly by means of three bits. The result is then rounded to the highest positive representable number, which is 3.

after symmetrical saturation: **1011 (–5)**

after symmetrical saturation: **101 (–3)**

There is an overflow because the decimal number –5 is outside the range of values that can be represented exactly by means of three bits. The result is then rounded to minus the highest positive representable number, which is –3.

Example (unsigned, 3-bit number):

after symmetrical saturation: **01110 (14)**

after symmetrical saturation: **111 (7)**

7.10.10.8 SC_WRAP

The SC_WRAP overflow mode shall be used to indicate that the output is wrapped around in the case of overflow.

Two different cases are possible:

— SC_WRAP with parameter *n_bits* = 0
— SC_WRAP with parameter *n_bits* > 0

SC_WRAP, 0

This shall be the default overflow mode. All bits except for the deleted bits shall be copied to the result number. Figure 5 illustrates the SC_WRAP overflow mode for a word length of three bits with the *n_bits* parameter set to 0. The x axis represents the word length before rounding; the y axis represents the word length after rounding.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

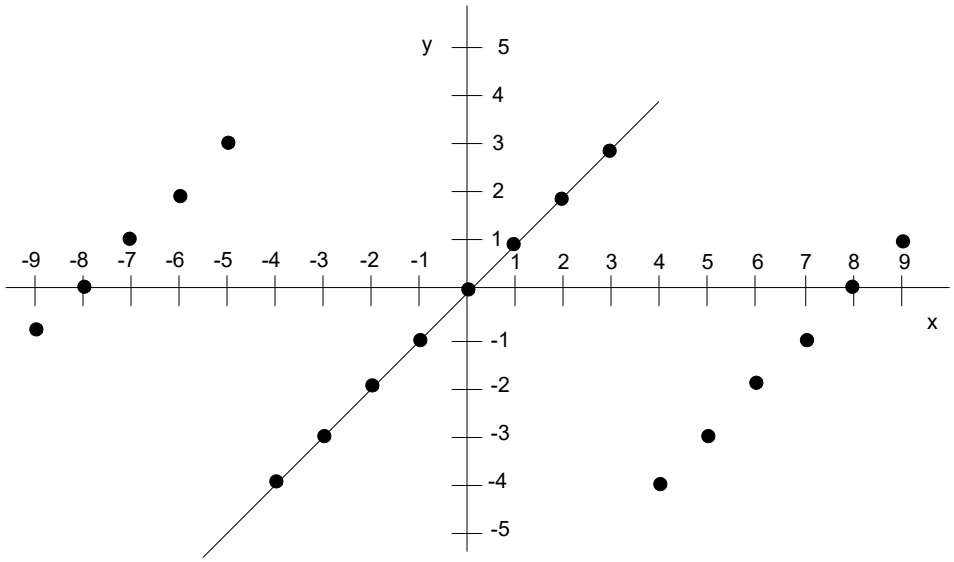

Figure 5—Wrap-around with n_bits = 0 for signed numbers

Examples (signed, 3-bit number):

before wrapping around with 0 bits: **0100 (4)**

after wrapping around with 0 bits: **100 (–4)**

There is an overflow because the decimal number 4 is outside the range of values that can be represented exactly by means of three bits. The MSB is truncated, and the result becomes negative: –4.

before wrapping around with 0 bits: **1011 (–5)**

after wrapping around with 0 bits: **011 (3)**

There is an overflow because the decimal number –5 is outside the range of values that can be represented exactly by means of three bits. The MSB is truncated, and the result becomes positive: 3.

Example (unsigned, 3-bit number):

before wrapping around with 0 bits: **11011 (27)**

after wrapping around with 0 bits: **011 (3)**

SC_WRAP, n_bits > 0: SC_WRAP, 1

Whenever *n_bits* is greater than 0, the specified number of bits on the MSB side of the result shall be saturated with preservation of the original sign; the other bits shall be copied from the original. Positive numbers shall remain positive; negative numbers shall remain negative. Figure 6 illustrates the SC_WRAP overflow mode for a word length of three bits with the *n_bits* parameter set to 1. The x axis represents the word length before rounding; the y axis represents the word length after rounding.

Examples (signed, 3-bit number):

before wrapping around with 1 bit: **0101 (5)**

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

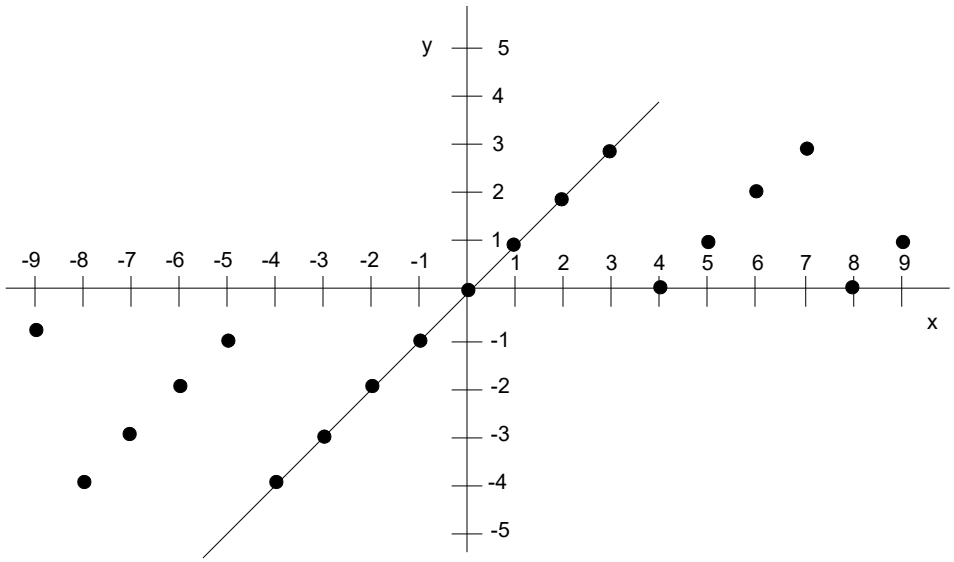

Figure 6—Wrap-around with n_bits = 1 for signed numbers

after wrapping around with 1 bit: **001 (1)**

There is an overflow because the decimal number 5 is outside the range of values that can be represented exactly by means of three bits. The sign bit is kept, so that positive numbers remain positive.

before wrapping around with 1 bit: **1011 (–5)**

after wrapping around with 1 bit: **111 (–1)**

There is an overflow because the decimal number –5 is outside the range of values that can be represented exactly by means of three bits. The MSB is truncated, but the sign bit is kept, so that negative numbers remain negative.

Example (unsigned, 5-bit number):

before wrapping around with 3 bits: **0110010 (50)**

after wrapping around with 3 bits: **11110 (30)**

For this example the SC_WRAP, 3 mode is applied. The result number is five bits wide. The three bits at the MSB side are set to 1; the remaining bits are copied.

7.10.10.9 SC_WRAP_SM

The SC_WRAP_SM overflow mode shall be used to indicate that the output is sign-magnitude wrapped around in the case of overflow. The **n_bits** parameter shall indicate the number of bits (for example, 1) on the MSB side of the cast number that are saturated with preservation of the original sign.

Two different cases are possible:

— SC_WRAP_SM with parameter *n_bits* = 0
— SC_WRAP_SM with parameter *n_bits* > 0

SC_WRAP_SM, 0

The MSBs outside the required word length shall be deleted. The sign bit of the result shall get the value of the least significant of the deleted bits. The other bits shall be inverted in case where the

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

original and the new values of the most significant of the kept bits differ. Otherwise, the other bits shall be copied from the original to the result. See Figure 7.

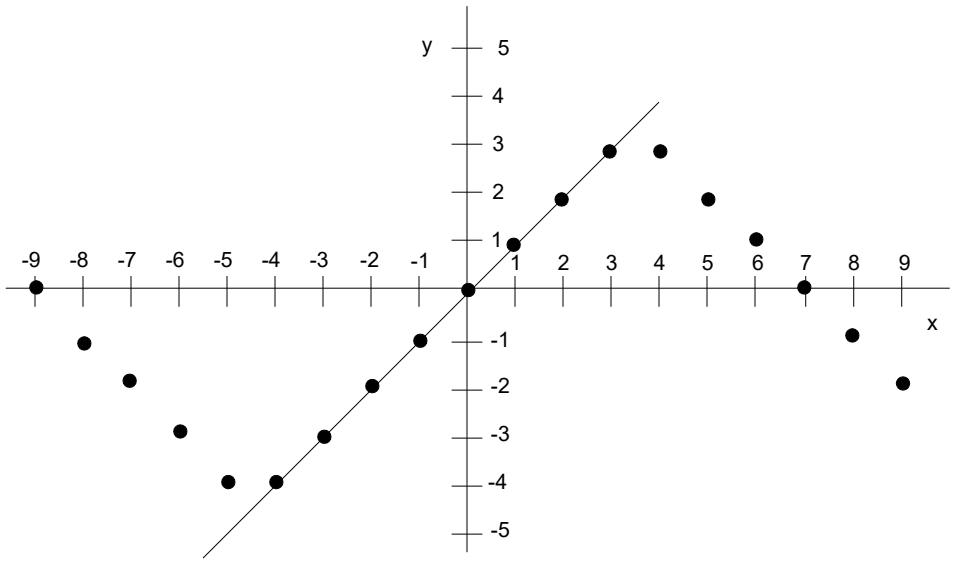

Figure 7—Sign magnitude wrap-around with n_bits = 0

Example:

The sequence of operations to cast a decimal number 4 into three bits and use the overflow mode SC_WRAP_SM, 0, as shown in Figure 7, is as follows:

0100 (4)

The original representation is truncated to be put in a three-bit number:

100 (–4)

The new sign bit is 0. This is the value of least significant deleted bit.

Because the original and the new value of the new sign bit differ, the values of the remaining bits are inverted:

011 (3)

This principle shall be applied to all numbers that cannot be represented exactly by means of three bits, as shown in Table 43.

Table 43—Sign magnitude wrap-around with n_bits = 0 for a three-bit number

Decimal	Binary
8	111
7	000
6	001
5	010
4	011
3	011
2	010

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 43—Sign magnitude wrap-around with n_bits = 0 for a three-bit number *(continued)*

Decimal	Binary
1	001
0	000
–1	111
–2	110
–3	101
–4	100
–5	100
–6	101
–7	110

SC_WRAP_SM, *n_bits* > 0

The first *n_bits* bits on the MSB side of the result number shall be as follows:

— Saturated to MAX in case of a positive number
— Saturated to MIN in case of a negative number

All numbers shall retain their sign.

In case where *n_bits* equals 1, the other bits shall be copied and XORed with the original and the new value of the sign bit of the result. In the case where *n_bits* is greater than 1, the remaining bits shall be XORed with the original value of the least significant saturated bit and the inverse value of the original sign bit. See Figure 8.

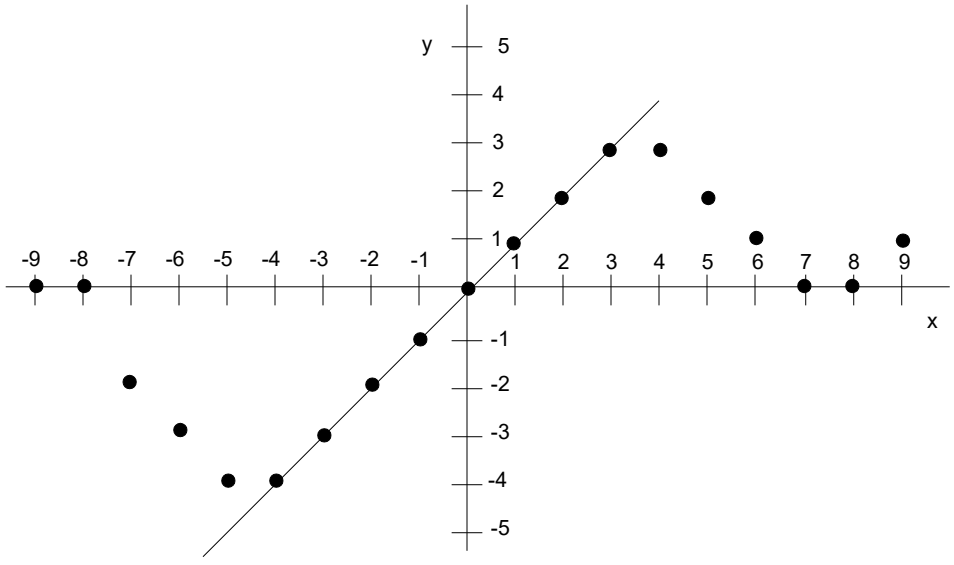

Figure 8—Sign magnitude wrap-around with n_bits = 1

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Example:

SC_WRAP_SM, n_bits > 0: SC_WRAP_SM, 3

The first three bits on the MSB side of the cast number are saturated to MAX or MIN.

If the decimal number 234 is cast into five bits using the overflow mode SC_WRAP_SM, 3, the following happens:

011101010 (234)

The original representation is truncated to five bits:

01010

The original sign bit is copied to the new MSB (bit position 4, starting from bit position 0):

01010

The bits at position 2, 3, and 4 are saturated; they are converted to the maximum value that can be expressed with three bits without changing the sign bit:

01110

The original value of the bit on position 2 was 0. The remaining bits at the LSB side (10) are XORed with this value and with the inverse value of the original sign bit, that is, with 0 and 1, respectively.

01101 (13)

Example:

SC_WRAP_SM, n_bits > 0: SC_WRAP_SM, 1

The first bit on the MSB side of the cast number gets the value of the original sign bit. The other bits are copied and XORed with the original and the new value of the sign bit of the result number.

The sequence of operations to cast the decimal number 12 into three bits using the overflow mode SC_WRAP_SM, 1, as shown in Figure 8, is as follows:

01100 (12)

The original representation is truncated to three bits.

100

The original sign bit is copied to the new MSB (bit position 2, starting from bit position 0).

000

The two remaining bits at the LSB side are XORed with the original (1) and the new value (0) of the new sign bit.

011

This principle shall be applied to all numbers that cannot be represented exactly by means of three bits, as shown in Table 44.

Table 44—Sign-magnitude wrap-around with n_bits=1 for a three-bit number

Decimal	Binary
9	001
8	000
7	000
6	001
5	010

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 44—Sign-magnitude wrap-around with n_bits=1 for a three-bit number *(continued)*

Decimal	Binary
4	011
3	011
2	010
1	001
0	000
–1	111
–2	110
–3	101
–4	100
–5	100
–6	101
–7	110
–8	111
–9	111

7.10.10.10 Quantization modes

Quantization shall be applied when the precision of the value assigned to a fixed-point variable exceeds the precision of the fixed-point variable. In SystemC, specific quantization modes shall be available to control the mapping to a representable value.

The mutually exclusive quantization modes listed in Table 45 shall be provided. The default quantization mode shall be SC_TRN.

Table 45—Quantization modes

Quantization mode	Name
Rounding to plus infinity	SC_RND
Rounding to zero	SC_RND_ZERO
Rounding to minus infinity	SC_RND_MIN_INF
Rounding to infinity	SC_RND_INF
Convergent rounding	SC_RND_CONV
Truncation	SC_TRN
Truncation to zero	SC_TRN_ZERO

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Quantization is the mapping of a value that may not be precisely represented in a specific fixed-point representation to a value that can be represented with arbitrary magnitude. If a value can be precisely represented, quantization shall not change the value. All the rounding modes shall map a value to the nearest value that is representable. When there are two nearest representable values (the value is halfway between them), the rounding modes shall provide different criteria for selection between the two. Both truncate modes shall map a positive value to the nearest representable value that is less than the value. SC_TRN mode shall map a negative value to the nearest representable value that is less than the value, while SC_TRN_ZERO shall map a negative value to the nearest representable value that is greater than the value.

Each of the following quantization modes is followed by a figure. The input values are given on the x axis and the output values on the y axis. Together they determine the quantization mode. In each figure, the quantization mode specified by the respective keyword is combined with the ideal characteristic. This ideal characteristic is represented by the diagonal dashed line.

Before each quantization mode is discussed in detail, an overview is given of how the different quantization modes deal with quantization for signed and unsigned fixed-point numbers.

7.10.10.11 Quantization for signed fixed-point numbers

The following template contains a signed fixed-point number in two's complement representation before and after a quantization mode has been applied, and a number of flags. The flags are explained below the template.

Before	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*
After:	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*						
Flags:	*sR*	*R*	*R*	*R*	*R*	*R*	*R*	*R*	*lR*	*mD*	*D*	*D*	*D*	*D*	*D*

The following flags and symbols are used in the template just given and in Table 46:

- *x* represents a binary digit (0 or 1).
- *sR* represents a sign bit.
- *R* represents the remaining bits.
- *lR* represents the least significant remaining bit.
- *mD* represents the most significant deleted bit.
- *D* represents the deleted bits.
- *r* represents the logical or of the deleted bits except for the *mD* bit in the template just given. When there are no remaining bits, *r* is **false**. This means that *r* is **false** when the two nearest numbers are at equal distance.

Table 46 shows how a signed fixed-point number shall be cast for each of the possible quantization modes in cases where there is quantization. If the two nearest representable numbers are not at equal distance, the result shall be the nearest representable number. This shall be found by applying the SC_RND mode, that is, by adding the most significant of the deleted bits to the remaining bits.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The second column in Table 46 contains the expression that shall be added to the remaining bits. It shall evaluate to a one or a zero. The operators used in the table are "!" for a bitwise negation, "|" for a bitwise OR, and "&" for a bitwise AND.

Table 46—Quantization handling for signed fixed-point numbers

Quantization mode	Expression to be added
SC_RND	mD
	Add the most significant deleted bit to the remaining bits.
SC_RND_ZERO	mD & (sR | r)
	If the most significant deleted bit is 1 and either the sign bit or at least one other deleted bit is 1, add 1 to the remaining bits.
SC_RND_MIN_INF	mD & r
	If the most significant deleted bit is 1 and at least one other deleted bit is 1, add 1 to the remaining bits.
SC_RND_INF	mD & (! sR | r)
	If the most significant deleted bit is 1 and either the inverted value of the sign bit or at least one other deleted bit is 1, add 1 to the remaining bits.
SC_RND_CONV	mD & (lR | r)
	If the most significant deleted bit is 1 and either the least significant of the remaining bits or at least one other deleted bit is 1, add 1 to the remaining bits.
SC_TRN	0
	Copy the remaining bits.
SC_TRN_ZERO	sR & (mD | r)
	If the sign bit is 1 and either the most significant deleted bit or at least one other deleted bit is 1, add 1 to the remaining bits.

7.10.10.12 Quantization for unsigned fixed-point numbers

The following template contains an unsigned fixed-point number before and after a quantization mode has been applied, and a number of flags. The flags are explained below the template.

Before	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*
After:	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*						
Flags:	*R*	*R*	*R*	*R*	*R*	*R*	*R*	*R*	*lR*	*mD*	*D*	*D*	*D*	*D*	*D*

The following flags and symbols are used in the template just given and in Table 47:

- *x* represents a binary digit (0 or 1).
- *R* represents the remaining bits.
- *lR* represents the least significant remaining bit.
- *mD* represents the most significant deleted bit.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

— *D* represents the deleted bits.

— *r* represents the logical or of the deleted bits except for the *mD* bit in the template just given. When there are no remaining bits, *r* is **false**. This means that *r* is **false** when the two nearest numbers are at equal distance.

Table 47—Quantization handling for unsigned fixed-point numbers

Quantization mode	Expression to be added
SC_RND	mD
	Add the most significant deleted bit to the left bits.
SC_RND_ZERO	0
	Copy the remaining bits.
SC_RND_MIN_INF	0
	Copy the remaining bits.
SC_RND_INF	mD
	Add the most significant deleted bit to the left bits.
SC_RND_CONV	mD & (lR | r)
	If the most significant deleted bit is 1 and either the least significant of the remaining bits or at least one other deleted bit is 1, add 1 to the remaining bits.
SC_TRN	0
	Copy the remaining bits.
SC_TRN_ZERO	0
	Copy the remaining bits.

Table 47 shows how an unsigned fixed-point number shall be cast for each of the possible quantization modes in cases where there is quantization. If the two nearest representable numbers are not at equal distance, the result shall be the nearest representable number. This shall be found for all the rounding modes by applying the SC_RND mode, that is, by adding the most significant of the deleted bits to the remaining bits.

The second column in Table 47 contains the expression that shall be added to the remaining bits. It shall evaluate to a one or a zero. The "&" operator used in the table represents a bitwise AND and the "|" a bitwise OR.

NOTE—For all rounding modes, overflow can occur. One extra bit on the MSB side is needed to represent the result in full precision.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.10.13 SC_RND

The result shall be rounded to the nearest representable number by adding the most significant of the deleted LSBs to the remaining bits. This rule shall be used for all rounding modes when the two nearest representable numbers are not at equal distance. When the two nearest representable numbers are at equal distance, this rule implies that there is rounding toward plus infinity, as shown in Figure 9.

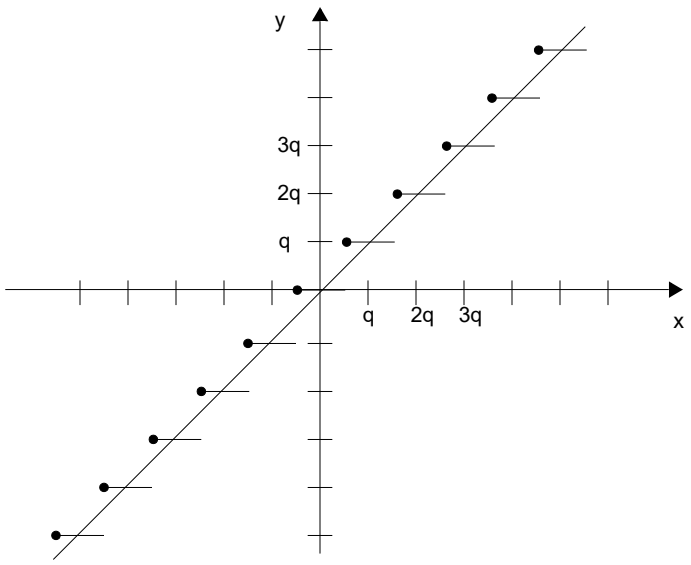

Figure 9—Rounding to plus infinity

In Figure 9, the symbol **q** refers to the quantization step, that is, the resolution of the data type.

Example (signed):

Numbers of type **sc_fixed<4,2>** are assigned to numbers of type **sc_fixed<3,2,SC_RND>**

before rounding to plus infinity: **(1.25)**

after rounding to plus infinity: **01.1 (1.5)**

There is quantization because the decimal number 1.25 is outside the range of values that can be represented exactly by means of a **sc_fixed<3,2,SC_RND>** number. The most significant of the deleted LSBs (1) is added to the new LSB.

before rounding to plus infinity: **10.11 (–1.25)**

after rounding to plus infinity: **11.0 (–1)**

There is quantization because the decimal number –1.25 is outside the range of values that can be represented exactly by means of a **sc_fixed<3,2,SC_RND>** number. The most significant of the deleted LSBs (1) is added to the new LSB.

Example (unsigned):

Numbers of type **sc_ufixed<16,8>** are assigned to numbers of type **sc_ufixed<12,8,SC_RND>**

before rounding to plus infinity: **00100110.01001111 (38.30859375)**

after rounding to plus infinity: **00100110.0101 (38.3125)**

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.10.14 SC_RND_ZERO

If the two nearest representable numbers are not at equal distance, the SC_RND_ZERO mode shall be applied.

If the two nearest representable numbers are at equal distance, the output shall be rounded toward 0, as shown in Figure 10. For positive numbers, the redundant bits on the LSB side shall be deleted. For negative numbers, the most significant of the deleted LSBs shall be added to the remaining bits.

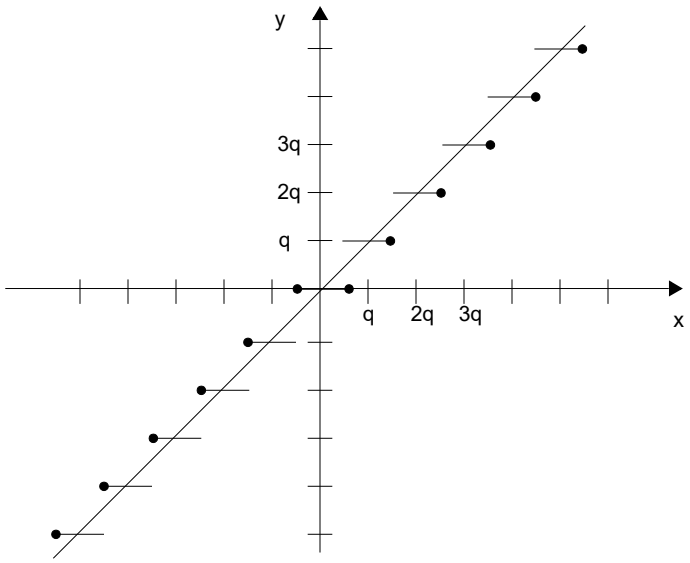

Figure 10—Rounding to zero

Example (signed):

Numbers of type **sc_fixed<4,2>** are assigned to numbers of type **sc_fixed<3,2,SC_RND_ZERO>**

before rounding to zero: **(1.25)**

after rounding to zero: **01.0 (1)**

There is quantization because the decimal number 1.25 is outside the range of values that can be represented exactly by means of a **sc_fixed<3,2,SC_RND_ZERO>** number. The redundant bits are omitted.

before rounding to zero: **10.11 (–1.25)**

after rounding to zero: **11.0 (–1)**

There is quantization because the decimal number –1.25 is outside the range of values that can be represented exactly by means of a **sc_fixed<3,2,SC_RND_ZERO>** number. The most significant of the omitted LSBs (1) is added to the new LSB.

Example (unsigned):

Numbers of type **sc_ufixed<16,8>** are assigned to numbers of type **sc_ufixed<12,8,SC_RND_ZERO>**

before rounding to zero: **000100110.01001 (38.28125)**

after rounding to zero: **000100110.0100 (38.25)**

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.10.15 SC_RND_MIN_INF

If the two nearest representable numbers are not at equal distance, the SC_RND_MIN_INF mode shall be applied.

If the two nearest representable numbers are at equal distance, there shall be rounding toward minus infinity, as shown in Figure 11, by omitting the redundant bits on the LSB side.

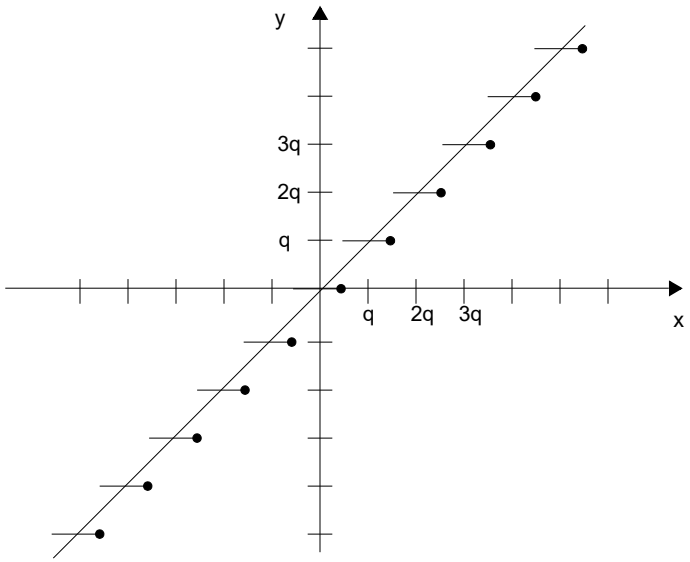

Figure 11—Rounding to minus infinity

Example (signed):

Numbers of type **sc_fixed<4,2>** are assigned to numbers of type **sc_fixed<3,2,SC_RND_MIN_INF>**

before rounding to minus infinity: **01.01 (1.25)**

after rounding to minus infinity: **01.0 (1)**

There is quantization because the decimal number 1.25 is outside the range of values that can be represented exactly by means of a **sc_fixed<3,2,SC_RND_MIN_INF>** number. The surplus bits are truncated.

before rounding to minus infinity: **10.11 (–1.25)**

after rounding to minus infinity: **10.1 (–1.5)**

There is quantization because the decimal number –1.25 is outside the range of values that can be represented exactly by means of a **sc_fixed<3,2,SC_RND_MIN_INF>** number. The surplus bits are truncated.

Example (unsigned):

Numbers of type **sc_ufixed<16,8>** are assigned to numbers of type **sc_ufixed<12,8,SC_RND_MIN_INF>**

before rounding to minus infinity: **000100110.01001 (38.28125)**

after rounding to minus infinity: **000100110.0100 (38.25)**

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.10.16 SC_RND_INF

Rounding shall be performed if the two nearest representable numbers are at equal distance.

For positive numbers, there shall be rounding toward plus infinity if the LSB of the remaining bits is 1 and toward minus infinity if the LSB of the remaining bits is 0, as shown in Figure 12.

For negative numbers, there shall be rounding toward minus infinity if the LSB of the remaining bits is 1 and toward plus infinity if the LSB of the remaining bits is 0, as shown in Figure 12.

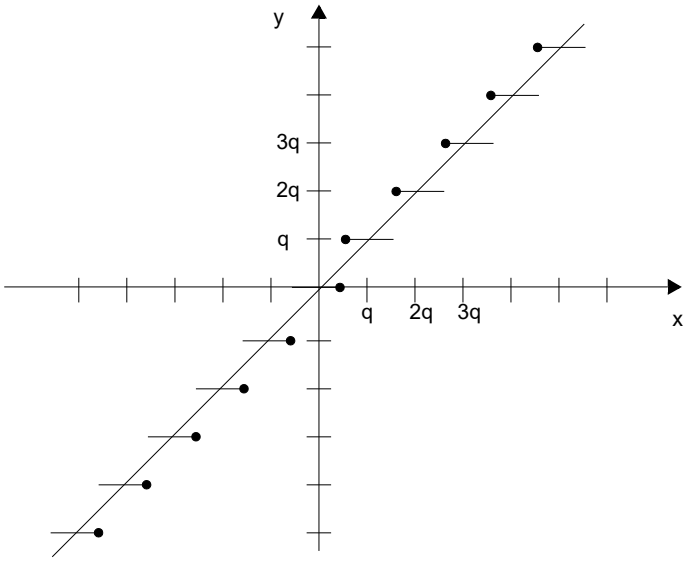

Figure 12—Rounding to infinity

Example (signed):

Numbers of type **sc_fixed<4,2>** are assigned to numbers of type **sc_fixed<3,2,SC_RND_INF>**

before rounding to infinity: **01.01 (1.25)**

after rounding to infinity: **01.1 (1.5)**

There is quantization because the decimal number 1.25 is outside the range of values that can be represented exactly by means of a **sc_fixed<3,2,SC_RND_INF>** number. The most significant of the deleted LSBs (1) is added to the new LSB.

before rounding to infinity: **10.11 (–1.25)**

after rounding to infinity: **10.1 (–1.5)**

There is quantization because the decimal number –1.25 is outside the range of values that can be represented exactly by means of a **sc_fixed<3,2,SC_RND_INF>** number. The surplus bits are truncated.

Example (unsigned):

Numbers of type **sc_ufixed<16,8>** are assigned to numbers of type **sc_ufixed<12,8,SC_RND_INF>**

before rounding to infinity: **000100110.01001 (38.28125)**

after rounding to infinity: **000100110.0101 (38.3125)**

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.10.17 SC_RND_CONV

If the two nearest representable numbers are not at equal distance, the SC_RND_CONV mode shall be applied.

If the two nearest representable numbers are at equal distance, there shall be rounding toward plus infinity if the LSB of the remaining bits is 1. There shall be rounding toward minus infinity if the LSB of the remaining bits is 0. The characteristics are shown in Figure 13.

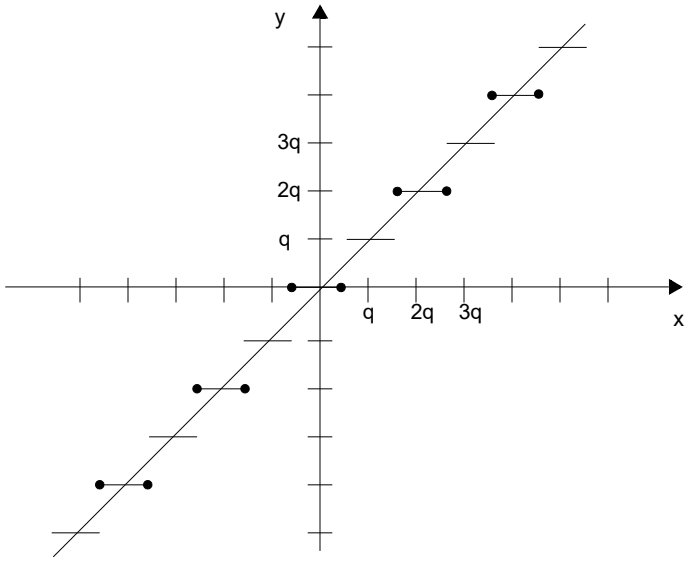

Figure 13—Convergent rounding

Example (signed):

Numbers of type **sc_fixed<4,2>** are assigned to numbers of type **sc_fixed<3,2,SC_RND_CONV>**

before convergent rounding: **00.11 (0.75)**

after convergent rounding: **01.0 (1)**

There is quantization because the decimal number 0.75 is outside the range of values that can be represented exactly by means of a **sc_fixed<3,2,SC_RND_CONV>** number. The surplus bits are truncated, and the result is rounded toward plus infinity.

before convergent rounding: **10.11 (–1.25)**

after convergent rounding: **11.0 (–1)**

There is quantization because the decimal number –1.25 is outside the range of values that can be represented exactly by means of a **sc_fixed<3,2,SC_RND_CONV>** number. The surplus bits are truncated, and the result is rounded toward plus infinity.

Example (unsigned):

Numbers of type **sc_ufixed<16,8>** are assigned to numbers of type **sc_ufixed<12,8,SC_RND_CONV>**

before convergent rounding: 000100110.01001 **(38.28125)**

after convergent rounding: 000100110.0100 **(38.25)**

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

before convergent rounding: **000100110.01011 (38.34375)**

after convergent rounding: **000100110.0110 (38.375)**

7.10.10.18 SC_TRN

SC_TRN shall be the default quantization mode. The result shall be rounded toward minus infinity; that is, the superfluous bits on the LSB side shall be deleted. A quantized number shall be approximated by the first representable number below its original value within the required bit range.

NOTE—In scientific literature, this mode is usually called “value truncation.”

The required characteristics are shown in Figure 14.

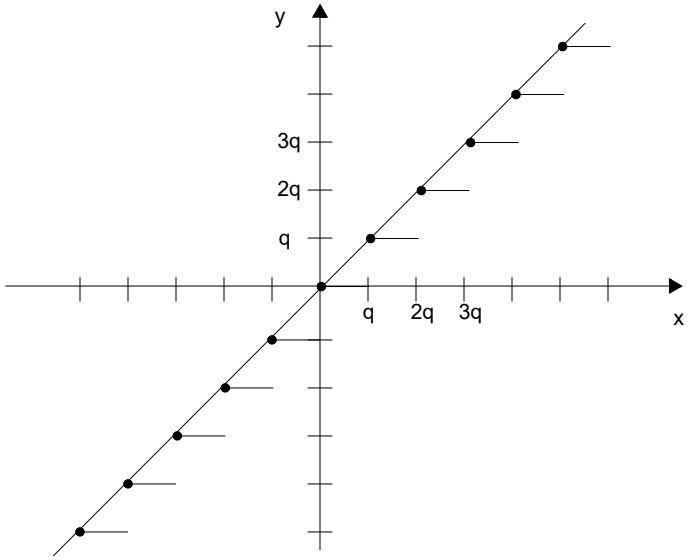

Figure 14—Truncation

Example (signed):

Numbers of type **sc_fixed<4,2>** are assigned to numbers of type **sc_fixed<3,2,SC_TRN>**

before truncation: **01.01 (1.25)**

after truncation: **01.0 (1)**

There is quantization because the decimal number 1.25 is outside the range of values that can be represented exactly by means of a **sc_fixed<3,2,SC_TRN>** number. The LSB is truncated.

before truncation: **10.11 (–1.25)**

after truncation: 1**0.1 (–1.5)**

There is quantization because the decimal number –1.25 is outside the range of values that can be represented exactly by means of a **sc_fixed<3,2,SC_TRN>** number. The LSB is truncated.

Example (unsigned):

Numbers of type **sc_ufixed<16,8>** are assigned to numbers of type **sc_ufixed<12,8,SC_TRN>**

before truncation: **00100110.01001111 (38.30859375)**

after truncation: **00100110.0100 (38.25)**

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.10.19 SC_TRN_ZERO

For positive numbers, this quantization mode shall correspond to SC_TRN. For negative numbers, the result shall be rounded toward zero (SC_RND_ZERO); that is, the superfluous bits on the right-hand side shall be deleted and the sign bit added to the left LSBs, provided at least one of the deleted bits differs from zero. A quantized number shall be approximated by the first representable number that is lower in absolute value.

NOTE—In scientific literature, this mode is usually called “magnitude truncation.”

The required characteristics are shown in Figure 15.

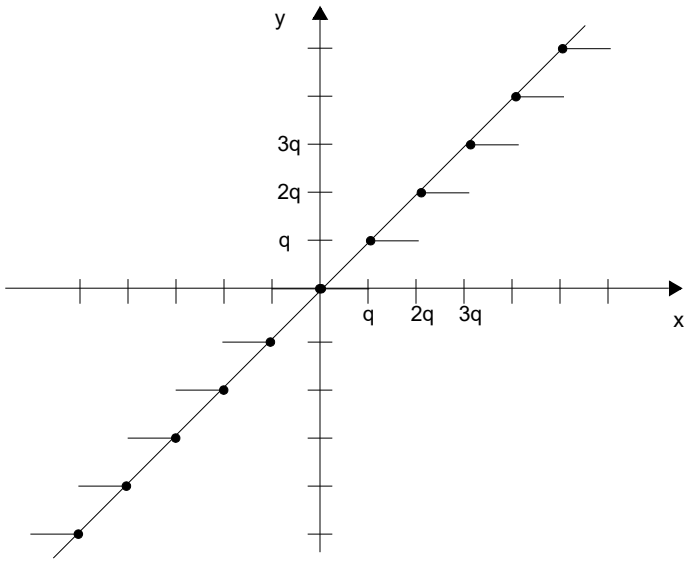

Figure 15—Truncation to zero

Example (signed):

A number of type **sc_fixed<4,2>** is assigned to a number of type **sc_fixed<3,2,SC_TRN_ZERO>**

before truncation to zero: **10.11 (–1.25)**

after truncation to zero: **11.0 (–1)**

There is quantization because the decimal number –1.25 is outside the range of values that can be represented exactly by means of a **sc_fixed<3,2,SC_TRN_ZERO>** number. The LSB is truncated, and then the sign bit (1) is added at the LSB side.

Example (unsigned):

Numbers of type **sc_ufixed<16,8>** are assigned to numbers of type **sc_ufixed<12,8,SC_TRN_ZERO>**

before truncation to zero: **00100110.01001111 (38.30859375)**

after truncation to zero: **00100110.0100 (38.25)**

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.11 sc_fxnum

7.10.11.1 Description

Class **sc_fxnum** is the base class for finite-precision fixed-point types. It shall be provided in order to define functions and overloaded operators that will work with any derived class.

7.10.11.2 Class definition

```
namespace sc_dt {

class sc_fxnum
{
    friend class sc_fxval;

    friend class sc_fxnum_bitref_r;
    friend class sc_fxnum_bitref;
    friend class sc_fxnum_subref_r;
    friend class sc_fxnum_subref;
    friend class sc_fxnum_fast_bitref_r;
    friend class sc_fxnum_fast_bitref;
    friend class sc_fxnum_fast_subref_r;
    friend class sc_fxnum_fast_subref;

    public:
        // Unary operators
        const sc_fxval operator- () const;
        const sc_fxval operator+ () const;

        // Binary operators
        #define DECL_BIN_OP_T( op , tp ) \
            friend sc_fxval operator op ( const sc_fxnum& , tp ); \
            friend sc_fxval operator op ( tp , const sc_fxnum& );
        #define DECL_BIN_OP_OTHER( op ) \
            DECL_BIN_OP_T( op , int64 ) \
            DECL_BIN_OP_T( op , uint64 ) \
            DECL_BIN_OP_T( op , const sc_int_base& ) \
            DECL_BIN_OP_T( op , const sc_uint_base& ) \
            DECL_BIN_OP_T( op , const sc_signed& ) \
            DECL_BIN_OP_T( op, const sc_unsigned& )
        #define DECL_BIN_OP( op , placeholder ) \
            friend sc_fxval operator op ( const sc_fxnum& , const sc_fxnum& ); \
            DECL_BIN_OP_T( op , int ) \
            DECL_BIN_OP_T( op , unsigned int ) \
            DECL_BIN_OP_T( op , long ) \
            DECL_BIN_OP_T( op , unsigned long ) \
            DECL_BIN_OP_T( op , float ) \
            DECL_BIN_OP_T( op , double ) \
            DECL_BIN_OP_T( op, const char*) \
            DECL_BIN_OP_T( op , const sc_fxval&) \
            DECL_BIN_OP_T( op , const sc_fxval_fast& ) \
            DECL_BIN_OP_T( op , const sc_fxnum_fast& ) \
            DECL_BIN_OP_OTHER( op )
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
DECL_BIN_OP( * , mult )
DECL_BIN_OP( + , add )
DECL_BIN_OP( - , sub )
DECL_BIN_OP( / , div )

#undef DECL_BIN_OP_T
#undef DECL_BIN_OP_OTHER
#undef DECL_BIN_OP

friend sc_fxval operator<< ( const sc_fxnum& , int );
friend sc_fxval operator>> ( const sc_fxnum& , int );

// Relational (including equality) operators
#define DECL_REL_OP_T( op , tp ) \
    friend bool operator op ( const sc_fxnum& , tp ); \
    friend bool operator op ( tp , const sc_fxnum& ); \
    DECL_REL_OP_T( op , int64 ) \
    DECL_REL_OP_T( op , uint64 ) \
    DECL_REL_OP_T( op , const sc_int_base& ) \
    DECL_REL_OP_T( op , const sc_uint_base& ) \
    DECL_REL_OP_T( op , const sc_signed& )   \
    DECL_REL_OP_T( op , const sc_unsigned& )
#define DECL_REL_OP( op ) \
    friend bool operator op ( const sc_fxnum& , const sc_fxnum& ); \
    DECL_REL_OP_T( op , int ) \
    DECL_REL_OP_T( op , unsigned int ) \
    DECL_REL_OP_T( op , long  ) \
    DECL_REL_OP_T( op , unsigned long ) \
    DECL_REL_OP_T( op , float ) \
    DECL_REL_OP_T( op , double ) \
    DECL_REL_OP_T( op , const char* )   \
    DECL_REL_OP_T( op , const sc_fxval& ) \
    DECL_REL_OP_T( op , const sc_fxval_fast& ) \
    DECL_REL_OP_T( op , const sc_fxnum_fast& )  \
    DECL_REL_OP_OTHER( op )
DECL_REL_OP( < )
DECL_REL_OP( <= )
DECL_REL_OP( > )
DECL_REL_OP( >= )
DECL_REL_OP( == )
DECL_REL_OP( != )

#undef DECL_REL_OP_T
#undef DECL_REL_OP_OTHER
#undef DECL_REL_OP

// Assignment operators
#define DECL_ASN_OP_T( op , tp ) \
    sc_fxnum& operator op( tp ); \
    DECL_ASN_OP_T( op , int64 ) \
    DECL_ASN_OP_T( op , uint64 ) \
    DECL_ASN_OP_T( op , const sc_int_base& )   \
    DECL_ASN_OP_T( op , const sc_uint_base& )  \
    DECL_ASN_OP_T( op , const sc_signed& ) \
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    DECL_ASN_OP_T( op , const sc_unsigned& )
#define DECL_ASN_OP( op ) \
    DECL_ASN_OP_T( op , int ) \
    DECL_ASN_OP_T( op , unsigned int ) \
    DECL_ASN_OP_T( op , long ) \
    DECL_ASN_OP_T( op , unsigned long ) \
    DECL_ASN_OP_T( op , float ) \
    DECL_ASN_OP_T( op , double ) \
    DECL_ASN_OP_T( op , const char* )  \
    DECL_ASN_OP_T( op , const sc_fxval& )  \
    DECL_ASN_OP_T( op , const sc_fxval_fast& )  \
    DECL_ASN_OP_T( op , const sc_fxnum& ) \
    DECL_ASN_OP_T( op , const sc_fxnum_fast& )  \
    DECL_ASN_OP_OTHER( op )
DECL_ASN_OP( = )
DECL_ASN_OP( *= )
DECL_ASN_OP( /= )
DECL_ASN_OP( += )
DECL_ASN_OP( -= )
DECL_ASN_OP_T( <<= , int )
DECL_ASN_OP_T( >>= , int )

#undef DECL_ASN_OP_T
#undef DECL_ASN_OP_OTHER
#undef DECL_ASN_OP

// Auto-increment and auto-decrement
sc_fxval operator++ ( int );
sc_fxval operator-- ( int );
sc_fxnum& operator++ ();
sc_fxnum& operator-- ();

// Bit selection
sc_fxnum_bitref_r operator[] ( int ) const;
sc_fxnum_bitref operator[] ( int );

// Part selection
sc_fxnum_subref_r operator() ( int , int ) const;
sc_fxnum_subref operator() ( int , int );
sc_fxnum_subref_r range( int , int ) const;
sc_fxnum_subref range( int , int );
sc_fxnum_subref_r operator() () const;
sc_fxnum_subref operator() ();
sc_fxnum_subref_r range() const;
sc_fxnum_subref range();

// Implicit conversion
operator double() const;

// Explicit conversion to primitive types
short to_short() const;
unsigned short to_ushort() const;
int to_int() const;
unsigned int to_uint() const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        long to_long() const;
        unsigned long  to_ulong() const;
        int64 to_int64() const;
        uint64 to_uint64() const;
        float to_float() const;
        double to_double() const;

        // Explicit conversion to character string
        std::string to_string() const;
        std::string to_string( sc_numrep ) const;
        std::string to_string( sc_numrep , bool ) const;
        std::string to_string( sc_fmt ) const;
        std::string to_string( sc_numrep , sc_fmt ) const;
        std::string to_string( sc_numrep , bool , sc_fmt ) const;
        std::string to_dec() const;
        std::string to_bin() const;
        std::string to_oct() const;
        std::string to_hex() const;

        // Query value
        bool is_neg() const;
        bool is_zero() const;
        bool quantization_flag() const;
        bool overflow_flag() const;
        sc_fxval value() const;

        // Query parameters
        int wl() const;
        int iwl() const;
        sc_q_mode q_mode() const;
        sc_o_mode o_mode() const;
        int n_bits() const;
        const sc_fxtype_params& type_params() const;
        const sc_fxcast_switch& cast_switch() const;

        // Print or dump content
        void print( std::ostream& = std::cout ) const;
        void scan( std::istream& = std::cin );
        void dump( std::ostream& = std::cout ) const;

    private:
        // Disabled
        sc_fxnum();
        sc_fxnum( const sc_fxnum& );
};

}       // namespace sc_dt
```

7.10.11.3 Constraints on usage

An application shall not directly create an instance of type **sc_fxnum**. An application may use a pointer to **sc_fxnum** or a reference to **sc_fxnum** to refer to an object of a class derived from **sc_fxnum**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.11.4 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ numeric representation to **sc_fxnum**, using truncation or sign-extension, as described in 7.10.5.

7.10.11.5 Implicit type conversion

operator **double**() const;

Operator **double** shall provide implicit type conversion from **sc_fxnum** to double.

7.10.11.6 Explicit type conversion

short **to_short**() const;
unsigned short **to_ushort**() const;
int **to_int**() const;
unsigned int **to_uint**() const;
long **to_long**() const;
unsigned long **to_ulong**() const;
int64 **to_int64**() const;
uint64 **to_uint64**() const;
float **to_float**() const;
double **to_double**() const;

These member functions shall perform conversion to C++ numeric types.

std::string **to_string**() const;
std::string **to_string**(sc_numrep) const;
std::string **to_string**(sc_numrep , bool) const;
std::string **to_string**(sc_fmt) const;
std::string **to_string**(sc_numrep , sc_fmt) const;
std::string **to_string**(sc_numrep , bool , sc_fmt) const;
std::string **to_dec**() const;
std::string **to_bin**() const;
std::string **to_oct**() const;
std::string **to_hex**() const;

These member functions shall perform the conversion to a **string** representation, as described in 7.2.12, 7.10.9.1, and 7.10.9.2.

7.10.12 sc_fxnum_fast

7.10.12.1 Description

Class **sc_fxnum_fast** is the base class for limited-precision fixed-point types. It shall be provided in order to define functions and overloaded operators that will work with any derived class.

7.10.12.2 Class definition

```
namespace sc_dt {

class sc_fxnum_fast
{
    friend class sc_fxval_fast;

    friend class sc_fxnum_bitref_r;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
friend class sc_fxnum_bitref†;
friend class sc_fxnum_subref_r;
friend class sc_fxnum_subref†;
friend class sc_fxnum_fast_bitref_r;
friend class sc_fxnum_fast_bitref†;
friend class sc_fxnum_fast_subref_r;
friend class sc_fxnum_fast_subref†;

public:
    // Unary operators
    sc_fxval_fast operator- () const;
    sc_fxval_fast operator+ () const;

    // Binary operators
    #define DECL_BIN_OP_T( op , tp ) \
        friend sc_fxval_fast operator op ( const sc_fxnum_fast& , tp ); \
        friend sc_fxval_fast operator op ( tp , const sc_fxnum_fast& );
    #define DECL_BIN_OP_OTHER( op ) \
        DECL_BIN_OP_T( op , int64 ) \
        DECL_BIN_OP_T( op , uint64 ) \
        DECL_BIN_OP_T( op , const sc_int_base& ) \
        DECL_BIN_OP_T( op , const sc_uint_base& ) \
        DECL_BIN_OP_T( op , const sc_signed& ) \
        DECL_BIN_OP_T( op, const sc_unsigned& )
    #define DECL_BIN_OP( op , placeholder ) \
        friend sc_fxval_fast operator op ( const sc_fxnum_fast& , const sc_fxnum_fast& ); \
        DECL_BIN_OP_T( op , int ) \
        DECL_BIN_OP_T( op , unsigned int ) \
        DECL_BIN_OP_T( op , long ) \
        DECL_BIN_OP_T( op , unsigned long ) \
        DECL_BIN_OP_T( op , float ) \
        DECL_BIN_OP_T( op , double ) \
        DECL_BIN_OP_T( op, const char*) \
        DECL_BIN_OP_T( op , const sc_fxval_fast& ) \
        DECL_BIN_OP_OTHER( op )

    DECL_BIN_OP( * , mult )
    DECL_BIN_OP( + , add )
    DECL_BIN_OP( - , sub )
    DECL_BIN_OP( / , div )

    #undef DECL_BIN_OP_T
    #undef DECL_BIN_OP_OTHER
    #undef DECL_BIN_OP

    friend sc_fxval operator<< ( const sc_fxnum_fast& , int );
    friend sc_fxval operator>> ( const sc_fxnum_fast& , int );

    // Relational (including equality) operators
    #define DECL_REL_OP_T( op , tp ) \
        friend bool operator op ( const sc_fxnum_fast& , tp ); \
        friend bool operator op ( tp , const sc_fxnum_fast& );
        DECL_REL_OP_T( op , int64 ) \
        DECL_REL_OP_T( op , uint64 ) \
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    DECL_REL_OP_T( op , const sc_int_base& ) \
    DECL_REL_OP_T( op , const sc_uint_base& ) \
    DECL_REL_OP_T( op , const sc_signed& ) \
    DECL_REL_OP_T( op , const sc_unsigned& )
#define DECL_REL_OP( op ) \
    friend bool operator op ( const sc_fxnum_fast& , const sc_fxnum_fast& ); \
    DECL_REL_OP_T( op , int ) \
    DECL_REL_OP_T( op , unsigned int ) \
    DECL_REL_OP_T( op , long  ) \
    DECL_REL_OP_T( op , unsigned long ) \
    DECL_REL_OP_T( op , float ) \
    DECL_REL_OP_T( op , double ) \
    DECL_REL_OP_T( op , const char* )   \
    DECL_REL_OP_T( op , const sc_fxval_fast& ) \
    DECL_REL_OP_OTHER( op )
DECL_REL_OP( < )
DECL_REL_OP( <= )
DECL_REL_OP( > )
DECL_REL_OP( >= )
DECL_REL_OP( == )
DECL_REL_OP( != )

#undef DECL_REL_OP_T
#undef DECL_REL_OP_OTHER
#undef DECL_REL_OP

// Assignment operators
#define DECL_ASN_OP_T( op , tp ) \
    sc_fxnum& operator op( tp ); \
    DECL_ASN_OP_T( op , int64 ) \
    DECL_ASN_OP_T( op , uint64 ) \
    DECL_ASN_OP_T( op , const sc_int_base& )   \
    DECL_ASN_OP_T( op , const sc_uint_base& )  \
    DECL_ASN_OP_T( op , const sc_signed& ) \
    DECL_ASN_OP_T( op , const sc_unsigned& )
#define DECL_ASN_OP( op ) \
    DECL_ASN_OP_T( op , int ) \
    DECL_ASN_OP_T( op , unsigned int ) \
    DECL_ASN_OP_T( op , long ) \
    DECL_ASN_OP_T( op , unsigned long ) \
    DECL_ASN_OP_T( op , float ) \
    DECL_ASN_OP_T( op , double ) \
    DECL_ASN_OP_T( op , const char* )   \
    DECL_ASN_OP_T( op , const sc_fxval& )  \
    DECL_ASN_OP_T( op , const sc_fxval_fast& )  \
    DECL_ASN_OP_T( op , const sc_fxnum& ) \
    DECL_ASN_OP_T( op , const sc_fxnum_fast& )  \
    DECL_ASN_OP_OTHER( op )
DECL_ASN_OP( = )
DECL_ASN_OP( *= )
DECL_ASN_OP( /= )
DECL_ASN_OP( += )
DECL_ASN_OP( -= )
DECL_ASN_OP_T( <<= , int )
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
DECL_ASN_OP_T( >>= , int )

#undef DECL_ASN_OP_T
#undef DECL_ASN_OP_OTHER
#undef DECL_ASN_OP

// Auto-increment and auto-decrement
sc_fxval_fast operator++ ( int );
sc_fxval_fast operator-- ( int );
sc_fxnum_fast& operator++ ();
sc_fxnum_fast& operator-- ();

// Bit selection
sc_fxnum_fast_bitref_r operator[] ( int ) const;
sc_fxnum_bitref† operator[] ( int );

// Part selection
sc_fxnum_fast_subref_r operator() ( int , int ) const;
sc_fxnum_fast_subref‡ operator() ( int , int );
sc_fxnum_fast_subref_r range( int , int ) const;
sc_fxnum_fast_subref‡ range( int , int );
sc_fxnum_fast_subref_r operator() () const;
sc_fxnum_fast_subref‡ operator() ();
sc_fxnum_fast_subref_r range() const;
sc_fxnum_fast_subref‡ range();

// Implicit conversion
operator double() const;

// Explicit conversion to primitive types
short to_short() const;
unsigned short to_ushort() const;
int to_int() const;
unsigned int to_uint() const;
long to_long() const;
unsigned long  to_ulong() const;
int64 to_int64() const;
uint64 to_uint64() const;
float to_float() const;
double to_double() const;

// Explicit conversion to character string
std::string to_string() const;
std::string to_string( sc_numrep ) const;
std::string to_string( sc_numrep , bool ) const;
std::string to_string( sc_fmt ) const;
std::string to_string( sc_numrep , sc_fmt ) const;
std::string to_string( sc_numrep , bool , sc_fmt ) const;
std::string to_dec() const;
std::string to_bin() const;
std::string to_oct() const;
std::string to_hex() const;

// Query value
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        bool is_neg() const;
        bool is_zero() const;
        bool quantization_flag() const;
        bool overflow_flag() const;
        sc_fxval_fast value() const;

        // Query parameters
        int wl() const;
        int iwl() const;
        sc_q_mode q_mode() const;
        sc_o_mode o_mode() const;
        int n_bits() const;
        const sc_fxtype_params& type_params() const;
        const sc_fxcast_switch& cast_switch() const;

        // Print or dump content
        void print( std::ostream& = std::cout ) const;
        void scan( std::istream& = std::cin );
        void dump( std::ostream& = std::cout ) const;

    private:
        // Disabled
        sc_fxnum_fast();
        sc_fxnum_fast( const sc_fxnum_fast& );
};

}       // namespace sc_dt
```

7.10.12.3 Constraints on usage

An application shall not directly create an instance of type **sc_fxnum_fast**. An application may use a pointer to **sc_fxnum_fast** or a reference to **sc_fxnum_fast** to refer to an object of a class derived from **sc_fxnum_fast**.

7.10.12.4 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ numeric representation to **sc_fxnum_fast**, using truncation or sign-extension, as described in 7.10.5.

7.10.12.5 Implicit type conversion

operator **double**() const;

Operator **double** shall provide implicit type conversion from **sc_fxnum_fast** to double.

7.10.12.6 Explicit type conversion

```
short to_short() const;
unsigned short to_ushort() const;
int to_int() const;
unsigned int to_uint() const;
long to_long() const;
unsigned long  to_ulong() const;
int64 to_int64() const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

uint64 **to_uint64**() const;
float **to_float**() const;
double **to_double**() const;

These member functions shall perform conversion to C++ numeric types.

std::string **to_string**() const;
std::string **to_string**(sc_numrep) const;
std::string **to_string**(sc_numrep , bool) const;
std::string **to_string**(sc_fmt) const;
std::string **to_string**(sc_numrep , sc_fmt) const;
std::string **to_string**(sc_numrep , bool , sc_fmt) const;
std::string **to_dec**() const;
std::string **to_bin**() const;
std::string **to_oct**() const;
std::string **to_hex**() const;

These member functions shall perform the conversion to a **string** representation, as described in 7.2.12, 7.10.9.1, and 7.10.9.2.

7.10.13 sc_fxval

7.10.13.1 Description

Class **sc_fxval** is the variable-precision fixed-point type. It may hold the value of any of the fixed-point types and perform the variable-precision fixed-point arithmetic operations. Type casting shall be performed by the fixed-point types themselves. Limited variable-precision type **sc_fxval_fast** and variable-precision type **sc_fxval** may be mixed freely.

7.10.13.2 Class definition

```
namespace sc_dt {

class sc_fxval
{
    public:
        // Constructors and destructor
        sc_fxval();
        explicit sc_fxval( int );
        explicit sc_fxval( unsigned int );
        explicit sc_fxval( long );
        explicit sc_fxval( unsigned long );
        explicit sc_fxval( float );
        explicit sc_fxval( double );
        explicit sc_fxval( const char* );
        sc_fxval( const sc_fxval& );
        sc_fxval( const sc_fxval_fast& );
        sc_fxval( const sc_fxnum& );
        sc_fxval( const sc_fxnum_fast& );
        explicit sc_fxval( int64 );
        explicit sc_fxval( uint64 );
        explicit sc_fxval( const sc_int_base& );
        explicit sc_fxval( const sc_uint_base& );
        explicit sc_fxval( const sc_signed& );
        explicit sc_fxval( const sc_unsigned& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
 ~sc_fxval();

// Unary operators
sc_fxval operator- () const;
const sc_fxval& operator+ () const;
friend void neg( sc_fxval& , const sc_fxval& );

// Binary operators
#define DECL_BIN_OP_T( op , tp ) \
    friend sc_fxval operator op ( const sc_fxval& , tp ); \
    friend sc_fxval operator op ( tp , const sc_fxval& );
#define DECL_BIN_OP_OTHER( op ) \
    DECL_BIN_OP_T( op , int64 ) \
    DECL_BIN_OP_T( op , uint64 ) \
    DECL_BIN_OP_T( op , const sc_int_base& ) \
    DECL_BIN_OP_T( op , const sc_uint_base& ) \
    DECL_BIN_OP_T( op , const sc_signed& ) \
    DECL_BIN_OP_T( op , const sc_unsigned& )
#define DECL_BIN_OP( op , placeholder ) \
    friend sc_fxval operator op ( const sc_fxval& , const sc_fxval& ); \
    DECL_BIN_OP_T( op , int ) \
    DECL_BIN_OP_T( op , unsigned int ) \
    DECL_BIN_OP_T( op , long ) \
    DECL_BIN_OP_T( op , unsigned long ) \
    DECL_BIN_OP_T( op , float ) \
    DECL_BIN_OP_T( op , double ) \
    DECL_BIN_OP_T( op , const char* ) \
    DECL_BIN_OP_T( op , const sc_fxval_fast& ) \
    DECL_BIN_OP_T( op , const sc_fxnum_fast& ) \
    DECL_BIN_OP_OTHER( op )

 DECL_BIN_OP( * , mult )
 DECL_BIN_OP( + , add )
 DECL_BIN_OP( - , sub )
 DECL_BIN_OP( / , div )

friend sc_fxval operator<< ( const sc_fxval& , int );
friend sc_fxval operator>> ( const sc_fxval& , int );

// Relational (including equality) operators
#define DECL_REL_OP_T( op , tp ) \
    friend bool operator op ( const sc_fxval& , tp ); \
    friend bool operator op ( tp , const sc_fxval& );
#define DECL_REL_OP_OTHER( op ) \
     DECL_REL_OP_T( op , int64 ) \
     DECL_REL_OP_T( op , uint64 ) \
     DECL_REL_OP_T( op , const sc_int_base& ) \
     DECL_REL_OP_T( op , const sc_uint_base& ) \
     DECL_REL_OP_T( op , const sc_signed& ) \
     DECL_REL_OP_T( op , const sc_unsigned& )
#define DECL_REL_OP( op ) \
    friend bool operator op ( const sc_fxval& , const sc_fxval& ); \
    DECL_REL_OP_T( op , int ) \
    DECL_REL_OP_T( op , unsigned int ) \
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    DECL_REL_OP_T( op , long ) \
    DECL_REL_OP_T( op , unsigned long ) \
    DECL_REL_OP_T( op , float ) \
    DECL_REL_OP_T( op , double ) \
    DECL_REL_OP_T( op , const char* ) \
    DECL_REL_OP_T( op , const sc_fxval_fast& ) \
    DECL_REL_OP_T( op , const sc_fxnum_fast& ) \
    DECL_REL_OP_OTHER( op )

DECL_REL_OP( < )
DECL_REL_OP( <= )
DECL_REL_OP( > )
DECL_REL_OP( >= )
DECL_REL_OP( == )
DECL_REL_OP( != )

// Assignment operators
#define DECL_ASN_OP_T( op , tp ) \
    sc_fxval& operator op( tp );
#define DECL_ASN_OP_OTHER( op ) \
    DECL_ASN_OP_T( op , int64 ) \
    DECL_ASN_OP_T( op , uint64 ) \
    DECL_ASN_OP_T( op , const sc_int_base& ) \
    DECL_ASN_OP_T( op , const sc_uint_base& ) \
    DECL_ASN_OP_T( op , const sc_signed& ) \
    DECL_ASN_OP_T( op , const sc_unsigned& )
#define DECL_ASN_OP( op ) \
    DECL_ASN_OP_T( op , int ) \
    DECL_ASN_OP_T( op , unsigned int ) \
    DECL_ASN_OP_T( op , long ) \
    DECL_ASN_OP_T( op , unsigned long ) \
    DECL_ASN_OP_T( op , float ) \
    DECL_ASN_OP_T( op , double ) \
    DECL_ASN_OP_T( op , const char* ) \
    DECL_ASN_OP_T( op , const sc_fxval& ) \
    DECL_ASN_OP_T( op , const sc_fxval_fast& ) \
    DECL_ASN_OP_T( op , const sc_fxnum& ) \
    DECL_ASN_OP_T( op , const sc_fxnum_fast& ) \
    DECL_ASN_OP_OTHER( op )

DECL_ASN_OP( = )
DECL_ASN_OP( *= )
DECL_ASN_OP( /= )
DECL_ASN_OP( += )
DECL_ASN_OP( -= )

DECL_ASN_OP_T( <<= , int )
DECL_ASN_OP_T( >>= , int )

// Auto-increment and auto-decrement
sc_fxval operator++ ( int );
sc_fxval operator-- ( int );
sc_fxval& operator++ ();
sc_fxval& operator-- ();
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        // Implicit conversion
        operator double() const;

        // Explicit conversion to primitive types
        short to_short() const;
        unsigned short to_ushort() const;
        int to_int() const;
        unsigned int to_uint() const;
        long to_long() const;
        unsigned long to_ulong() const;
        int64 to_int64() const;
        uint64 to_uint64() const;
        float to_float() const;
        double to_double() const;

         // Explicit conversion to character string
        std::string to_string() const;
        std::string to_string( sc_numrep ) const;
        std::string to_string( sc_numrep , bool ) const;
        std::string to_string( sc_fmt ) const;
        std::string to_string( sc_numrep , sc_fmt ) const;
        std::string to_string( sc_numrep , bool , sc_fmt ) const;
        std::string to_dec() const;
        std::string to_bin() const;
        std::string to_oct() const;
        std::string to_hex() const;

        // Member functions
        bool is_neg() const;
        bool is_zero() const;
        void print( std::ostream& = std::cout ) const;
        void scan( std::istream& = std::cin );
        void dump( std::ostream& = std::cout ) const;
};

}       // namespace sc_dt
```

7.10.13.3 Constraints on usage

A **sc_fxval** object that is declared without an initial value shall be uninitialized (unless it is declared as static, in which case it shall be initialized to zero). Uninitialized objects may be used wherever an initialized object is permitted. The result of an operation on an uninitialized object is undefined.

7.10.13.4 Public constructors

The constructor argument shall be taken as the initial value of the **sc_fxval** object. The default constructor shall not initialize the value.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.13.5 Operators

The operators that shall be defined for **sc_fxval** are given in Table 48.

Table 48—Operators for sc_fxval

Operator class	Operators in class
Arithmetic	* / + - << >> ++ --
Equality	== !=
Relational	<<= >>=
Assignment	= *= /= += -= <<= >>=

operator<< and **operator>>** define arithmetic shifts that perform sign extension.

The types of the operands shall be as defined in 7.10.5.

7.10.13.6 Implicit type conversion

operator **double**() const;

Operator **double** can be used for implicit type conversion to the C++ type **double**.

7.10.13.7 Explicit type conversion

short **to_short**() const;
unsigned short **to_ushort**() const;
int **to_int**() const;
unsigned int **to_uint**() const;
long **to_long**() const;
unsigned long **to_ulong**() const;
int64 **to_int64**() const;
uint64 **to_uint64**() const;
float **to_float**() const;
double **to_double**() const;

These member functions shall perform the conversion to the respective C++ numeric types.

std::string **to_string**() const;
std::string **to_string**(sc_numrep) const;
std::string **to_string**(sc_numrep , bool) const;
std::string **to_string**(sc_fmt) const;
std::string **to_string**(sc_numrep , sc_fmt) const;
std::string **to_string**(sc_numrep , bool , sc_fmt) const;
std::string **to_dec**() const;
std::string **to_bin**() const;
std::string **to_oct**() const;
std::string **to_hex**() const;

These member functions shall perform the conversion to a **string** representation, as described in 7.2.12, 7.10.9.1, and 7.10.9.2.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.14 sc_fxval_fast

7.10.14.1 Description

Type **sc_fxval_fast** is the limited variable-precision fixed-point type and shall be limited to a mantissa of 53 bits. It may hold the value of any of the fixed-point types and shall be used to perform the limited variable-precision fixed-point arithmetic operations. Limited variable-precision fixed-point type **sc_fxval_fast** and variable-precision fixed-point type **sc_fxval** may be mixed freely.

7.10.14.2 Class definition

```
namespace sc_dt {

class sc_fxval_fast
{
    public:
        sc_fxval_fast();
        explicit sc_fxval_fast( int );
        explicit sc_fxval_fast( unsigned int );
        explicit sc_fxval_fast( long );
        explicit sc_fxval_fast( unsigned long );
        explicit sc_fxval_fast( float );
        explicit sc_fxval_fast( double );
        explicit sc_fxval_fast( const char* );
        sc_fxval_fast( const sc_fxval& );
        sc_fxval_fast( const sc_fxval_fast& );
        sc_fxval_fast( const sc_fxnum& );
        sc_fxval_fast( const sc_fxnum_fast& );
        explicit sc_fxval_fast( int64 );
        explicit sc_fxval_fast( uint64 );
        explicit sc_fxval_fast( const sc_int_base& );
        explicit sc_fxval_fast( const sc_uint_base& );
        explicit sc_fxval_fast( const sc_signed& );
        explicit sc_fxval_fast( const sc_unsigned& );
        ~sc_fxval_fast();

        // Unary operators
        sc_fxval_fast  operator- () const;
        const sc_fxval_fast& operator+ () const;

         // Binary operators
        #define DECL_BIN_OP_T( op , tp ) \
            friend sc_fxval_fast operator op ( const sc_fxval_fast& , tp ); \
            friend sc_fxval_fast operator op ( tp , const sc_fxval_fast& );

        #define DECL_BIN_OP_OTHER( op ) \
            DECL_BIN_OP_T( op , int64 ) \
            DECL_BIN_OP_T( op , uint64 ) \
            DECL_BIN_OP_T( op , const sc_int_base& ) \
            DECL_BIN_OP_T( op , const sc_uint_base& ) \
            DECL_BIN_OP_T( op , const sc_signed& ) \
            DECL_BIN_OP_T( op , const sc_unsigned& )
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
#define DECL_BIN_OP( op , placeholder ) \
    friend sc_fxval_fast operator op ( const sc_fxval_fast& , const sc_fxval_fast& ); \
    DECL_BIN_OP_T( op , int ) \
    DECL_BIN_OP_T( op , unsigned int ) \
    DECL_BIN_OP_T( op , long ) \
    DECL_BIN_OP_T( op , unsigned long ) \
    DECL_BIN_OP_T( op , float ) \
    DECL_BIN_OP_T( op , double ) \
    DECL_BIN_OP_T( op , const char* ) \
    DECL_BIN_OP_OTHER( op )
DECL_BIN_OP( * , mult )
DECL_BIN_OP( + , add )
DECL_BIN_OP( - , sub )
DECL_BIN_OP( / , div )
friend sc_fxval_fast operator<< ( const sc_fxval_fast& , int );
friend sc_fxval_fast operator>> ( const sc_fxval_fast& , int );

// Relational (including equality) operators
#define DECL_REL_OP_T( op , tp ) \
    friend bool operator op ( const sc_fxval_fast& , tp );\
    friend bool operator op ( tp , const sc_fxval_fast& );
#define DECL_REL_OP_OTHER( op ) \
    DECL_REL_OP_T( op , int64 ) \
    DECL_REL_OP_T( op , uint64 ) \
    DECL_REL_OP_T( op , const sc_int_base& ) \
    DECL_REL_OP_T( op , const sc_uint_base& ) \
    DECL_REL_OP_T( op , const sc_signed& ) \
    DECL_REL_OP_T( op , const sc_unsigned& )
#define DECL_REL_OP( op ) \
    friend bool operator op ( const sc_fxval_fast& , const sc_fxval_fast& ); \
    DECL_REL_OP_T( op , int ) \
    DECL_REL_OP_T( op , unsigned int ) \
    DECL_REL_OP_T( op , long ) \
    DECL_REL_OP_T( op , unsigned long ) \
    DECL_REL_OP_T( op , float ) \
    DECL_REL_OP_T( op , double ) \
    DECL_REL_OP_T( op , const char* ) \
    DECL_REL_OP_OTHER( op )

DECL_REL_OP( < )
DECL_REL_OP( <= )
DECL_REL_OP( > )
DECL_REL_OP( >= )
DECL_REL_OP( == )
DECL_REL_OP( != )

// Assignment operators
#define DECL_ASN_OP_T( op , tp ) sc_fxval_fast& operator op( tp );
#define DECL_ASN_OP_OTHER( op ) \
    DECL_ASN_OP_T( op , int64 ) \
    DECL_ASN_OP_T( op , uint64 ) \
    DECL_ASN_OP_T( op , const sc_int_base& ) \
    DECL_ASN_OP_T( op , const sc_uint_base& ) \
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    DECL_ASN_OP_T( op , const sc_signed& ) \
    DECL_ASN_OP_T( op , const sc_unsigned& )
#define DECL_ASN_OP( op ) \
    DECL_ASN_OP_T( op , int ) \
    DECL_ASN_OP_T( op , unsigned int ) \
    DECL_ASN_OP_T( op , long ) \
    DECL_ASN_OP_T( op , unsigned long ) \
    DECL_ASN_OP_T( op , float ) \
    DECL_ASN_OP_T( op , double ) \
    DECL_ASN_OP_T( op , const char* ) \
    DECL_ASN_OP_T( op , const sc_fxval& ) \
    DECL_ASN_OP_T( op , const sc_fxval_fast& ) \
    DECL_ASN_OP_T( op , const sc_fxnum& ) \
    DECL_ASN_OP_T( op , const sc_fxnum_fast& ) \
    DECL_ASN_OP_OTHER( top )

DECL_ASN_OP( = )
DECL_ASN_OP( *= )
DECL_ASN_OP( /= )
DECL_ASN_OP( += )
DECL_ASN_OP( -= )
DECL_ASN_OP_T( <<= , int )
DECL_ASN_OP_T( >>= , int )

// Auto-increment and auto-decrement
sc_fxval_fast operator++ ( int );
sc_fxval_fast operator-- ( int );
sc_fxval_fast& operator++ ();
sc_fxval_fast& operator-- ();

// Implicit conversion
operator double() const;

// Explicit conversion to primitive types
short to_short() const;
unsigned short to_ushort() const;
int to_int() const;
unsigned int to_uint() const;
long to_long() const;
unsigned long to_ulong() const;
int64 to_int64() const;
uint64 to_uint64() const;
float to_float() const;
double to_double() const;

// Explicit conversion to character string
std::string to_string() const;
std::string to_string( sc_numrep ) const;
std::string to_string( sc_numrep , bool ) const;
std::string to_string( sc_fmt ) const;
std::string to_string( sc_numrep , sc_fmt ) const;
std::string to_string( sc_numrep , bool, sc_fmt ) const;
std::string to_dec() const;
std::string to_bin() const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        std::string to_oct() const;
        std::string to_hex() const;

        // Other member functions
        bool is_neg() const;
        bool is_zero() const;
        void print( std::ostream& = std::cout ) const;
        void scan( std::istream& = std::cin );
        void dump( std::ostream& = std::cout ) const;
};

}       // namespace sc_dt
```

7.10.14.3 Constraints on usage

A **sc_fxval_fast** object that is declared without an initial value shall be uninitialized (unless it is declared as static, in which case it shall be initialized to zero). Uninitialized objects may be used wherever an initialized object is permitted. The result of an operation on an uninitialized object is undefined.

7.10.14.4 Public constructors

The constructor argument shall be taken as the initial value of the **sc_fxval_fast** object. The default constructor shall not initialize the value.

7.10.14.5 Operators

The operators that shall be defined for **sc_fxval_fast** are given in Table 49.

Table 49—Operators for sc_fxval_fast

Operator class	Operators in class
Arithmetic	* / + - << >> ++ --
Equality	== !=
Relational	<<= >>=
Assignment	= *= /= += -= <<= >>=

NOTE—**operator<<** and **operator>>** define arithmetic shifts, not bitwise shifts. The difference is that no bits are lost and proper sign extension is done. Hence, these operators are also well defined for signed types, such as **sc_fxval_fast**.

7.10.14.6 Implicit type conversion

operator **double**() const;

Operator **double** can be used for implicit type conversion to the C++ type **double**.

7.10.14.7 Explicit type conversion

short **to_short**() const;
unsigned short **to_ushort**() const;
int **to_int**() const;

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

unsigned int **to_uint**() const;
long **to_long**() const;
unsigned long **to_ulong**() const;
int64 **to_int64**() const;
uint64 **to_uint64**() const;
float **to_float**() const;
double **to_double**() const;

These member functions shall perform the conversion to the respective C++ numeric types.

std::string **to_string**() const;
std::string **to_string**(sc_numrep) const;
std::string **to_string**(sc_numrep , bool) const;
std::string **to_string**(sc_fmt) const;
std::string **to_string**(sc_numrep , sc_fmt) const;
std::string **to_string**(sc_numrep , bool, sc_fmt) const;
std::string **to_dec**() const;
std::string **to_bin**() const;
std::string **to_oct**() const;
std::string **to_hex**() const;

These member functions shall perform the conversion to a **string** representation, as described in 7.2.12, 7.10.9.1, and 7.10.9.2.

7.10.15 sc_fix

7.10.15.1 Description

Class **sc_fix** shall represent a signed (two's complement) finite-precision fixed-point value. The fixed-point type parameters **wl**, **iwl**, **q_mode**, **o_mode**, and **n_bits** may be specified as constructor arguments.

7.10.15.2 Class definition

```
namespace sc_dt {

class sc_fix
: public sc_fxnum
{
    public:
        // Constructors and destructor
        sc_fix();
        sc_fix( int , int );
        sc_fix( sc_q_mode , sc_o_mod e );
        sc_fix( sc_q_mode , sc_o_mode, int );
        sc_fix( int , int , sc_q_mode , sc_o_mode );
        sc_fix( int , int , sc_q_mode, sc_o_mode, int );
        sc_fix( const sc_fxcast_switch& );
        sc_fix( int , int , const sc_fxcast_switch& );
        sc_fix( sc_q_mode , sc_o_mode , const sc_fxcast_switch& );
        sc_fix( sc_q_mode , sc_o_mode , int , const sc_fxcast_switch& );
        sc_fix( int , int , sc_q_mode , sc_o_mode , const sc_fxcast_switch& );
        sc_fix( int , int , sc_q_mode , sc_o_mode , int , const sc_fxcast_switch& );
        sc_fix( const sc_fxtype_params& );
        sc_fix( const sc_fxtype_params& , const sc_fxcast_switch& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
#define DECL_CTORS_T( tp ) \
    sc_fix( tp , int, int ); \
    sc_fix( tp , sc_q_mode , sc_o_mode ); \
    sc_fix( tp , sc_q_mode , sc_o_mode, int ); \
    sc_fix( tp , int , int , sc_q_mode , sc_o_mode ); \
    sc_fix( tp ,  int , int , sc_q_mode , sc_o_mode , int ); \
    sc_fix( tp , const sc_fxcast_switch& ); \
    sc_fix( tp , int , int , const sc_fxcast_switch& ); \
    sc_fix( tp , sc_q_mode , sc_o_mode , const sc_fxcast_switch& ); \
    sc_fix( tp , sc_q_mode , sc_o_mode , in , const sc_fxcast_switch& ); \
    sc_fix( tp , int , int , sc_q_mode , sc_o_mode , const sc_fxcast_switch& ); \
    sc_fix( tp , int, int , sc_q_mode , sc_o_mode , int , const sc_fxcast_switch& ); \
    sc_fix( tp , const sc_fxtype_params& ); \
    sc_fix( tp , const sc_fxtype_params& , const sc_fxcast_switch& );
#define DECL_CTORS_T_A( tp ) \
    sc_fix( tp ); \
    DECL_CTORS_T( tp )
#define DECL_CTORS_T_B( tp ) \
    explicit sc_fix( tp ); \
    DECL_CTORS_T( tp )

DECL_CTORS_T_A( int )
DECL_CTORS_T_A( unsigned int )
DECL_CTORS_T_A( long )
DECL_CTORS_T_A( unsigned long )
DECL_CTORS_T_A( float )
DECL_CTORS_T_A( double )
DECL_CTORS_T_A( const char* )
DECL_CTORS_T_A( const sc_fxval& )
DECL_CTORS_T_A( const sc_fxval_fast& )
DECL_CTORS_T_A( const sc_fxnum& )
DECL_CTORS_T_A( const sc_fxnum_fast& )
DECL_CTORS_T_B( int64 )
DECL_CTORS_T_B( uint64 )
DECL_CTORS_T_B( const sc_int_base& )
DECL_CTORS_T_B( const sc_uint_base& )
DECL_CTORS_T_B( const sc_signed& )
DECL_CTORS_T_B( const sc_unsigned& )
sc_fix( const sc_fix& );

// Unary bitwise operators
sc_fix operator~ () const;

// Binary bitwise operators
friend sc_fix operator& ( const sc_fix& , const sc_fix& );
friend sc_fix operator& ( const sc_fix& , const sc_fix_fast& );
friend sc_fix operator& ( const sc_fix_fast& , const sc_fix& );
friend sc_fix operator| ( const sc_fix& , const sc_fix& );
friend sc_fix operator| ( const sc_fix& , const sc_fix_fast& );
friend sc_fix operator| ( const sc_fix_fast& , const sc_fix& );
friend sc_fix operator^ ( const sc_fix& , const sc_fix& );
friend sc_fix operator^ ( const sc_fix& , const sc_fix_fast& );
friend sc_fix operator^ ( const sc_fix_fast& , const sc_fix& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_fix& operator= ( const sc_fix& );

        #define DECL_ASN_OP_T( op , tp ) \
            sc_fix& operator op ( tp );
        #define DECL_ASN_OP_OTHER( op ) \
            DECL_ASN_OP_T( op , int64 ) \
            DECL_ASN_OP_T( op , uint64 ) \
            DECL_ASN_OP_T( op , const sc_int_base& ) \
            DECL_ASN_OP_T( op , const sc_uint_base& ) \
            DECL_ASN_OP_T( op , const sc_signed& ) \
            DECL_ASN_OP_T( op , const sc_unsigned& )
        #define DECL_ASN_OP( op ) \
            DECL_ASN_OP_T( op , int ) \
            DECL_ASN_OP_T( op , unsigned int ) \
            DECL_ASN_OP_T( op , long ) \
            DECL_ASN_OP_T( op , unsigned long ) \
            DECL_ASN_OP_T( op , float ) \
            DECL_ASN_OP_T( op , double ) \
            DECL_ASN_OP_T( op , const char* )\
            DECL_ASN_OP_T( op , const sc_fxval& )\
            DECL_ASN_OP_T( op , const sc_fxval_fast& )\
            DECL_ASN_OP_T( op , const sc_fxnum& ) \
            DECL_ASN_OP_T( op , const sc_fxnum_fast& ) \
            DECL_ASN_OP_OTHER( op )

        DECL_ASN_OP( = )
        DECL_ASN_OP( *= )
        DECL_ASN_OP( /= )
        DECL_ASN_OP( += )
        DECL_ASN_OP( -= )
        DECL_ASN_OP_T( <<= , int )
        DECL_ASN_OP_T( >>= , int )
        DECL_ASN_OP_T( &= , const sc_fix& )
        DECL_ASN_OP_T( &= , const sc_fix_fast& )
        DECL_ASN_OP_T( |= , const sc_fix& )
        DECL_ASN_OP_T( |= , const sc_fix_fast& )
        DECL_ASN_OP_T( ^= , const sc_fix& )
        DECL_ASN_OP_T( ^= , const sc_fix_fast& )

        sc_fxval operator++ ( int );
        sc_fxval operator-- ( int );
        sc_fix& operator++ ();
        sc_fix& operator-- ();
};

}       // namespace sc_dt
```

7.10.15.3 Constraints on usage

The word length shall be greater than zero. The number of saturated bits, if specified, shall not be less than zero.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.15.4 Public constructors

The constructor arguments may specify the fixed-point type parameters, as described in 7.10.2. The default constructor shall set fixed-point type parameters according to the fixed-point context in scope at the point of construction. An initial value may additionally be specified as a C++ or SystemC numeric object or as a string literal. A fixed-point cast switch may also be passed as a constructor argument to set the fixed-point casting, as described in 7.10.8.

7.10.15.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ numeric representation to **sc_fix**, using truncation or sign-extension, as described in 7.10.5.

7.10.15.6 Bitwise operators

Bitwise operators for all combinations of operands of type **sc_fix** and **sc_fix_fast** shall be defined, as described in 7.10.5.

7.10.16 sc_ufix

7.10.16.1 Description

Class **sc_ufix** shall represent an unsigned finite-precision fixed-point value. The fixed-point type parameters **wl**, **iwl**, **q_mode**, **o_mode**, and **n_bits** may be specified as constructor arguments.

7.10.16.2 Class definition

```
namespace sc_dt {

class sc_ufix
: public sc_fxnum
{
    public:
        // Constructors
        explicit sc_ufix();
        sc_ufix( int , int );
        sc_ufix( sc_q_mode , sc_o_mode );
        sc_ufix( sc_q_mode , sc_o_mode , int );
        sc_ufix( int , int , sc_q_mode , sc_o_mode );
        sc_ufix( int , int , sc_q_mode , sc_o_mode, int );
        explicit sc_ufix( const sc_fxcast_switch& );
        sc_ufix( int , int , const sc_fxcast_switch& );
        sc_ufix( sc_q_mode , sc_o_mode , const sc_fxcast_switch& );
        sc_ufix( sc_q_mode , sc_o_mode , int , const sc_fxcast_switch& );
        sc_ufix( int , int , sc_q_mode , sc_o_mode , const sc_fxcast_switch& );
        sc_ufix( int , int , sc_q_mode , sc_o_mode , int , const sc_fxcast_switch& );
        explicit sc_ufix( const sc_fxtype_params& );
        sc_ufix( const sc_fxtype_params& , const sc_fxcast_switch& );

        #define DECL_CTORS_T( tp ) \
            sc_ufix( tp , int , int ); \
            sc_ufix( tp , sc_q_mode , sc_o_mode ); \
            sc_ufix( tp , sc_q_mode , sc_o_mode , int ); \
            sc_ufix( tp , int , int , sc_q_mode , sc_o_mode ); \
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    sc_ufix( tp , int , int , sc_q_mode , sc_o_mode , int ); \
    sc_ufix( tp , const sc_fxcast_switch& ); \
    sc_ufix( tp , int , int , const sc_fxcast_switch& ); \
    sc_ufix( tp , sc_q_mode , sc_o_mode , const sc_fxcast_switch& ); \
    sc_ufix( tp , sc_q_mode , sc_o_mode , int , const sc_fxcast_switch& ); \
    sc_ufix( tp , int , int , sc_q_mode , sc_o_mode , const sc_fxcast_switch& ); \
    sc_ufix( tp , int , int , sc_q_mode , sc_o_mode , int , const sc_fxcast_switch& ); \
    sc_ufix( tp , const sc_fxtype_params& ); \
    sc_ufix( tp , const sc_fxtype_params& , const sc_fxcast_switch& );
#define DECL_CTORS_T_A( tp ) \
    sc_ufix( tp );  \
    DECL_CTORS_T( tp )
#define DECL_CTORS_T_B( tp ) \
    explicit sc_ufix( tp ); \
    DECL_CTORS_T( tp )

DECL_CTORS_T_A( int )
DECL_CTORS_T_A( unsigned int )
DECL_CTORS_T_A( long )
DECL_CTORS_T_A( unsigned long )
DECL_CTORS_T_A( float )
DECL_CTORS_T_A( double )
DECL_CTORS_T_A( const char* )
DECL_CTORS_T_A( const sc_fxval& )
DECL_CTORS_T_A( const sc_fxval_fast& )
DECL_CTORS_T_A( const sc_fxnum& )
DECL_CTORS_T_A( const sc_fxnum_fast& )
DECL_CTORS_T_B( int64 )
DECL_CTORS_T_B( uint64 )
DECL_CTORS_T_B( const sc_int_base& )
DECL_CTORS_T_B( const sc_uint_base& )
DECL_CTORS_T_B( const sc_signed& )
DECL_CTORS_T_B( const sc_unsigned& )

#undef DECL_CTORS_T
#undef DECL_CTORS_T_A
#undef DECL_CTORS_T_B

// Copy constructor
sc_ufix( const sc_ufix& );

// Unary bitwise operators
sc_ufix operator~ () const;

// Binary bitwise operators
friend sc_ufix operator& ( const sc_ufix& , const sc_ufix& );
friend sc_ufix operator& ( const sc_ufix& , const sc_ufix_fast& );
friend sc_ufix operator& ( const sc_ufix_fast& , const sc_ufix& );
friend sc_ufix operator| ( const sc_ufix& , const sc_ufix& );
friend sc_ufix operator| ( const sc_ufix& , const sc_ufix_fast& );
friend sc_ufix operator| ( const sc_ufix_fast& , const sc_ufix& );
friend sc_ufix operator^ ( const sc_ufix& , const sc_ufix& );
friend sc_ufix operator^ ( const sc_ufix& , const sc_ufix_fast& );
friend sc_ufix operator^ ( const sc_ufix_fast& , const sc_ufix& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    // Assignment operators
    sc_ufix& operator= ( const sc_ufix& );

    #define DECL_ASN_OP_T( op , tp ) \
        sc_ufix& operator op ( tp );
    #define DECL_ASN_OP_OTHER( op ) \
        DECL_ASN_OP_T( op , int64 ) \
        DECL_ASN_OP_T( op , uint64 ) \
        DECL_ASN_OP_T( op , const sc_int_base& )\
        DECL_ASN_OP_T( op , const sc_uint_base& )\
        DECL_ASN_OP_T( op , const sc_signed& ) \
        DECL_ASN_OP_T( op , const sc_unsigned& )
    #define DECL_ASN_OP( op ) \
        DECL_ASN_OP_T( op , int ) \
        DECL_ASN_OP_T( op , unsigned int ) \
        DECL_ASN_OP_T( op , long ) \
        DECL_ASN_OP_T( op , unsigned long ) \
        DECL_ASN_OP_T( op , float ) \
        DECL_ASN_OP_T( op , double ) \
        DECL_ASN_OP_T( op , const char* )\
        DECL_ASN_OP_T( op , const sc_fxval& ) \
        DECL_ASN_OP_T( op , const sc_fxval_fast& ) \
        DECL_ASN_OP_T( op , const sc_fxnum& ) \
        DECL_ASN_OP_T( op , const sc_fxnum_fast& )\
        DECL_ASN_OP_OTHER( op )

    DECL_ASN_OP( = )
    DECL_ASN_OP( *= )
    DECL_ASN_OP( /= )
    DECL_ASN_OP( += )
    DECL_ASN_OP( -= )
    DECL_ASN_OP_T( <<= , int )
    DECL_ASN_OP_T( >>= , int )
    DECL_ASN_OP_T( &= , const sc_ufix& )
    DECL_ASN_OP_T( &= , const sc_ufix_fast& )
    DECL_ASN_OP_T( |= , const sc_ufix& )
    DECL_ASN_OP_T( |= , const sc_ufix_fast& )
    DECL_ASN_OP_T( ^= , const sc_ufix& )
    DECL_ASN_OP_T( ^= , const sc_ufix_fast& )

    #undef DECL_ASN_OP_T
    #undef DECL_ASN_OP_OTHER
    #undef DECL_ASN_OP

    // Auto-increment and auto-decrement
    sc_fxval operator++ ( int );
    sc_fxval operator-- ( int );
    sc_ufix& operator++ ();
    sc_ufix& operator-- ();
};

}    // namespace sc_dt
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.16.3 Constraints on usage

The word length shall be greater than zero. The number of saturated bits, if specified, shall not be less than zero.

7.10.16.4 Public constructors

The constructor arguments may specify the fixed-point type parameters, as described in 7.10.2. The default constructor shall set fixed-point type parameters according to the fixed-point context in scope at the point of construction. An initial value may additionally be specified as a C++ or SystemC numeric object or as a string literal. A fixed-point cast switch may also be passed as a constructor argument to set the fixed-point casting, as described in 7.10.8.

7.10.16.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ numeric representation to **sc_ufix**, using truncation or sign-extension, as described in 7.10.5.

7.10.16.6 Bitwise operators

Bitwise operators for all combinations of operands of type **sc_ufix** and **sc_ufix_fast** shall be defined, as described in 7.10.5.

7.10.17 sc_fix_fast

7.10.17.1 Description

Class **sc_fix_fast** shall represent a signed (two's complement) limited-precision fixed-point value. The fixed-point type parameters **wl**, **iwl**, **q_mode**, **o_mode**, and **n_bits** may be specified as constructor arguments.

7.10.17.2 Class definition

```
namespace sc_dt {

class sc_fix_fast
: public sc_fxnum_fast
{
    public:
        // Constructors
        sc_fix_fast();
        sc_fix_fast( int , int );
        sc_fix_fast( sc_q_mode , sc_o_mode );
        sc_fix_fast( sc_q_mode , sc_o_mode , int );
        sc_fix_fast( int , int , sc_q_mode , sc_o_mode );
        sc_fix_fast( int , int , sc_q_mode , sc_o_mode , int );
        sc_fix_fast( const sc_fxcast_switch& );
        sc_fix_fast( int , int , const sc_fxcast_switch& );
        sc_fix_fast( sc_q_mode , sc_o_mode , const sc_fxcast_switch& );
        sc_fix_fast( sc_q_mode , sc_o_mode , int , const sc_fxcast_switch& );
        sc_fix_fast( int , int , sc_q_mode , sc_o_mode , const sc_fxcast_switch& );
        sc_fix_fast( int , int , sc_q_mode , sc_o_mode , int , const sc_fxcast_switch& );
        sc_fix_fast( const sc_fxtype_params& );
        sc_fix_fast( const sc_fxtype_params& , const sc_fxcast_switch& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
#define DECL_CTORS_T( tp ) \
    sc_fix_fast( tp , int , int ); \
    sc_fix_fast( tp , sc_q_mode , sc_o_mode ); \
    sc_fix_fast( tp ,  sc_q_mode , sc_o_mode , int ); \
    sc_fix_fast( tp , int , int , sc_q_mode , sc_o_mode ); \
    sc_fix_fast( tp , int , int , sc_q_mode , sc_o_mode , int ); \
    sc_fix_fast( tp , const sc_fxcast_switch& ); \
    sc_fix_fast( tp , int , int , const sc_fxcast_switch& ); \
    sc_fix_fast( tp , sc_q_mode , sc_o_mode , const sc_fxcast_switch& ); \
    sc_fix_fast( tp , sc_q_mod e, sc_o_mode , int , const sc_fxcast_switch& ); \
    sc_fix_fast( tp , int , int , sc_q_mode , sc_o_mode , const sc_fxcast_switch& ); \
    sc_fix_fast( tp , int , int , sc_q_mode , sc_o_mode , int , const sc_fxcast_switch& ); \
    sc_fix_fast( tp , const sc_fxtype_params& ); \
    sc_fix_fast( tp , const sc_fxtype_params& , const sc_fxcast_switch& );
#define DECL_CTORS_T_A( tp ) \
    sc_fix_fast( tp ); \
    DECL_CTORS_T( tp )
#define DECL_CTORS_T_B( tp ) \
    explicit sc_fix_fast( tp ); \
    DECL_CTORS_T( tp )

DECL_CTORS_T_A( int )
DECL_CTORS_T_A( unsigned int )
DECL_CTORS_T_A( long )
DECL_CTORS_T_A( unsigned long )
DECL_CTORS_T_A( float )
DECL_CTORS_T_A( double )
DECL_CTORS_T_A( const char* )
DECL_CTORS_T_A( const sc_fxval& )
DECL_CTORS_T_A( const sc_fxval_fast& )
DECL_CTORS_T_A( const sc_fxnum& )
DECL_CTORS_T_A( const sc_fxnum_fast& )
DECL_CTORS_T_B( int64 )
DECL_CTORS_T_B( uint64 )
DECL_CTORS_T_B( const sc_int_base& )
DECL_CTORS_T_B( const sc_uint_base& )
DECL_CTORS_T_B( const sc_signed& )
DECL_CTORS_T_B( const sc_unsigned& )

// Copy constructor
sc_fix_fast( const sc_fix_fast& );

// Operators
sc_fix_fast operator~ () const;
friend sc_fix_fast operator& ( const sc_fix_fast& , const sc_fix_fast& );
friend sc_fix_fast operator^ ( const sc_fix_fast& , const sc_fix_fast& );
friend sc_fix_fast operator| ( const sc_fix_fast& , const sc_fix_fast& );
sc_fix_fast& operator= ( const sc_fix_fast& );

#define DECL_ASN_OP_T( op , tp ) \
    sc_fix_fast& operator op ( tp );
#define DECL_ASN_OP_OTHER( op ) \
    DECL_ASN_OP_T( op , int64 ) \
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        DECL_ASN_OP_T( op , uint64 ) \
        DECL_ASN_OP_T( op , const sc_int_base& )\
        DECL_ASN_OP_T( op , const sc_uint_base& )\
        DECL_ASN_OP_T( op , const sc_signed& )\
        DECL_ASN_OP_T( op , const sc_unsigned& )
    #define DECL_ASN_OP( op ) \
        DECL_ASN_OP_T( op , int ) \
        DECL_ASN_OP_T( op , unsigned int ) \
        DECL_ASN_OP_T( op , long ) \
        DECL_ASN_OP_T( op , unsigned long ) \
        DECL_ASN_OP_T( op , float) \
        DECL_ASN_OP_T( op , double ) \
        DECL_ASN_OP_T( op , const char* )\
        DECL_ASN_OP_T( op , const sc_fxval& )\
        DECL_ASN_OP_T( op , const sc_fxval_fast& )\
        DECL_ASN_OP_T( op , const sc_fxnum& )\
        DECL_ASN_OP_T( op , const sc_fxnum_fast& )\
        DECL_ASN_OP_OTHER( op )

    DECL_ASN_OP( = )
    DECL_ASN_OP( *= )
    DECL_ASN_OP( /= )
    DECL_ASN_OP( += )
    DECL_ASN_OP( -= )
    DECL_ASN_OP_T( <<= , int )
    DECL_ASN_OP_T( >>= , int )
    DECL_ASN_OP_T( &= , const sc_fix& )
    DECL_ASN_OP_T( &= , const sc_fix_fast& )
    DECL_ASN_OP_T( |= , const sc_fix& )
    DECL_ASN_OP_T( |= , const sc_fix_fast& )
    DECL_ASN_OP_T( ^= , const sc_fix& )
    DECL_ASN_OP_T( ^= , const sc_fix_fast& )

    sc_fxval_fast operator++ ( int );
    sc_fxval_fast operator-- ( int );
    sc_fix_fast& operator++ ();
    sc_fix_fast& operator-- ();
};

}    // namespace sc_dt
```

7.10.17.3 Constraints on usage

The word length shall be greater than zero. The number of saturated bits, if specified, shall not be less than zero.

sc_fix_fast shall use double-precision (floating-point) values. The mantissa of a double-precision value is limited to 53 bits, so bit-true behavior cannot be guaranteed with the limited-precision types.

7.10.17.4 Public constructors

The constructor arguments may specify the fixed-point type parameters, as described in 7.10.2. The default constructor shall set fixed-point type parameters according to the fixed-point context in scope at the point of construction. An initial value may additionally be specified as a C++ or SystemC numeric object or as a

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

string literal. A fixed-point cast switch may also be passed as a constructor argument to set the fixed-point casting, as described in 7.10.8.

7.10.17.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ numeric representation to **sc_fix_fast**, using truncation or sign-extension, as described in 7.10.5.

7.10.17.6 Bitwise operators

Bitwise operators for operands of type **sc_fix_fast** shall be defined, as described in 7.10.5.

7.10.18 sc_ufix_fast

7.10.18.1 Description

Class **sc_ufix_fast** shall represent an unsigned limited-precision fixed-point value. The fixed-point type parameters **wl**, **iwl**, **q_mode**, **o_mode**, and **n_bits** may be specified as constructor arguments.

7.10.18.2 Class definition

```
namespace sc_dt {

class sc_ufix_fast
: public sc_fxnum_fast
{
    public:
        // Constructors
        explicit sc_ufix_fast();
        sc_ufix_fast( int , int );
        sc_ufix_fast( sc_q_mode , sc_o_mode );
        sc_ufix_fast( sc_q_mode , sc_o_mode , int );
        sc_ufix_fast( int , int , sc_q_mode , sc_o_mode );
        sc_ufix_fast( int , int , sc_q_mode , sc_o_mode , int );
        explicit sc_ufix_fast( const sc_fxcast_switch& );
        sc_ufix_fast( int , int , const sc_fxcast_switch& );
        sc_ufix_fast( sc_q_mode , sc_o_mode , const sc_fxcast_switch& );
        sc_ufix_fast( sc_q_mode , sc_o_mode , int , const sc_fxcast_switch& );
        sc_ufix_fast( int , int , sc_q_mode , sc_o_mode , const sc_fxcast_switch& );
        sc_ufix_fast( int , int , sc_q_mode , sc_o_mode , int , const sc_fxcast_switch& );
        explicit sc_ufix_fast( const sc_fxtype_params& );
        sc_ufix_fast( const sc_fxtype_params& , const sc_fxcast_switch& );

        #define DECL_CTORS_T( tp ) \
            sc_ufix_fast( tp , int , int ); \
            sc_ufix_fast( tp , sc_q_mode , sc_o_mode ); \
            sc_ufix_fast( tp , sc_q_mode , sc_o_mode , int ); \
            sc_ufix_fast( tp , int , int , sc_q_mode , sc_o_mode ); \
            sc_ufix_fast( tp , int , int , sc_q_mode , sc_o_mode , int ); \
            sc_ufix_fast( tp , const sc_fxcast_switch& ); \
            sc_ufix_fast( tp , int , int , const sc_fxcast_switch& ); \
            sc_ufix_fast( tp , sc_q_mode , sc_o_mode , const sc_fxcast_switch& ); \
            sc_ufix_fast( tp , sc_q_mode , sc_o_mode , int , const sc_fxcast_switch& ); \
            sc_ufix_fast( tp , int , int , sc_q_mode , sc_o_mode , const sc_fxcast_switch& ); \
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    sc_ufix_fast( tp , int , int , sc_q_mode , sc_o_mode , int , const sc_fxcast_switch& ); \
    sc_ufix_fast( tp , const sc_fxtype_params& ); \
    sc_ufix_fast( tp , const sc_fxtype_params& , const sc_fxcast_switch& );

#define DECL_CTORS_T_A( tp )  \
    sc_ufix_fast( tp ); \
    DECL_CTORS_T( tp )
#define DECL_CTORS_T_B( tp )  \
    explicit sc_ufix_fast( tp ); \
    DECL_CTORS_T( tp )

DECL_CTORS_T_A( int )
DECL_CTORS_T_A( unsigned int )
DECL_CTORS_T_A( long )
DECL_CTORS_T_A( unsigned long )
DECL_CTORS_T_A( float )
DECL_CTORS_T_A( double )
DECL_CTORS_T_A( const char* )
DECL_CTORS_T_A( const sc_fxval& )
DECL_CTORS_T_A( const sc_fxval_fast& )
DECL_CTORS_T_A( const sc_fxnum& )
DECL_CTORS_T_A( const sc_fxnum_fast& )
DECL_CTORS_T_B( int64 )
DECL_CTORS_T_B( uint64 )
DECL_CTORS_T_B( const sc_int_base& )
DECL_CTORS_T_B( const sc_uint_base& )
DECL_CTORS_T_B( const sc_signed& )
DECL_CTORS_T_B( const sc_unsigned& )

#undef DECL_CTORS_T
#undef DECL_CTORS_T_A
#undef DECL_CTORS_T_B

// Copy constructor
sc_ufix_fast( const sc_ufix_fast& );

// Unary bitwise operators
sc_ufix_fast operator~ () const;

// Binary bitwise operators
friend sc_ufix_fast operator& ( const sc_ufix_fast& , const sc_ufix_fast& );
friend sc_ufix_fast operator^ ( const sc_ufix_fast& , const sc_ufix_fast& );
friend sc_ufix_fast operator| ( const sc_ufix_fast& , const sc_ufix_fast& );

// Assignment operators
sc_ufix_fast& operator= ( const sc_ufix_fast& );
#define DECL_ASN_OP_T( op , tp ) \
    sc_ufix_fast& operator op ( tp );
#define DECL_ASN_OP_OTHER( op ) \
    DECL_ASN_OP_T( op , int64 ) \
    DECL_ASN_OP_T( op , uint64 ) \
    DECL_ASN_OP_T( op , const sc_int_base& )\
    DECL_ASN_OP_T( op , const sc_uint_base& ) \
    DECL_ASN_OP_T( op , const sc_signed& ) \
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    DECL_ASN_OP_T( op , const sc_unsigned& )
#define DECL_ASN_OP( op ) \
    DECL_ASN_OP_T( op , int ) \
    DECL_ASN_OP_T( op , unsigned int )\
    DECL_ASN_OP_T( op , long )  \
    DECL_ASN_OP_T( op , unsigned long ) \
    DECL_ASN_OP_T( op , float ) \
    DECL_ASN_OP_T( op , double ) \
    DECL_ASN_OP_T( op , const char* )\
    DECL_ASN_OP_T( op , const sc_fxval& ) \
    DECL_ASN_OP_T( op , const sc_fxval_fast& ) \
    DECL_ASN_OP_T( op , const sc_fxnum& ) \
    DECL_ASN_OP_T( op , const sc_fxnum_fast& ) \
    DECL_ASN_OP_OTHER( op )

    DECL_ASN_OP( = )
    DECL_ASN_OP( *= )
    DECL_ASN_OP( /= )
    DECL_ASN_OP( += )
    DECL_ASN_OP( -= )
    DECL_ASN_OP_T( <<= , int )
    DECL_ASN_OP_T( >>= , int )
    DECL_ASN_OP_T( &= , const sc_ufix& )
    DECL_ASN_OP_T( &= , const sc_ufix_fast& )
    DECL_ASN_OP_T( |= , const sc_ufix& )
    DECL_ASN_OP_T( |= , const sc_ufix_fast& )
    DECL_ASN_OP_T( ^= , const sc_ufix& )
    DECL_ASN_OP_T( ^= , const sc_ufix_fast& )

#undef DECL_ASN_OP_T
#undef DECL_ASN_OP_OTHER
#undef DECL_ASN_OP

    // Auto-increment and auto-decrement
    sc_fxval_fast operator++ ( int );
    sc_fxval_fast operator-- ( int );
    sc_ufix_fast& operator++ ();
    sc_ufix_fast& operator-- ();
};

}    // namespace sc_dt
```

7.10.18.3 Constraints on usage

The word length shall be greater than zero. The number of saturated bits, if specified, shall not be less than zero.

sc_ufix_fast shall use double-precision (floating-point) values. The mantissa of a double-precision value is limited to 53 bits, so bit-true behavior cannot be guaranteed with the limited-precision types.

7.10.18.4 Public constructors

The constructor arguments may specify the fixed-point type parameters, as described in 7.10.2. The default constructor shall set fixed-point type parameters according to the fixed-point context in scope at the point of

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

construction. An initial value may additionally be specified as a C++ or SystemC numeric object or as a string literal. A fixed-point cast switch may also be passed as a constructor argument to set the fixed-point casting, as described in 7.10.8.

7.10.18.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ numeric representation to **sc_ufix_fast**, using truncation or sign-extension, as described in 7.10.5.

7.10.18.6 Bitwise operators

Bitwise operators for operands of type **sc_ufix_fast** shall be defined, as described in 7.10.5.

7.10.19 sc_fixed

7.10.19.1 Description

Class template **sc_fixed** shall represent a signed (two's complement) finite-precision fixed-point value. The fixed-point type parameters **wl**, **iwl**, **q_mode**, **o_mode**, and **n_bits** shall be specified by the template arguments.

Any public member functions of the base class **sc_fix** that are overridden in class **sc_fixed** shall have the same behavior in the two classes. Any public member functions of the base class not overridden in this way shall be publicly inherited by class **sc_fixed**.

7.10.19.2 Class definition

```
namespace sc_dt {

template <int W, int I,
    sc_q_mode Q = SC_DEFAULT_Q_MODE_,
    sc_o_mode O = SC_DEFAULT_O_MODE_, int N = SC_DEFAULT_N_BITS_>
class sc_fixed
: public sc_fix
{
    public:
        // Constructors
        sc_fixed();
        sc_fixed( const sc_fxcast_switch& );

    #define DECL_CTORS_T_A( tp ) \
        sc_fixed( tp ); \
        sc_fixed( tp , const sc_fxcast_switch& );
    #define DECL_CTORS_T_B( tp ) \
        sc_fixed( tp ); \
        sc_fixed( tp , const sc_fxcast_switch& );

        DECL_CTORS_T_A( int )
        DECL_CTORS_T_A( unsigned int )
        DECL_CTORS_T_A( long )
        DECL_CTORS_T_A( unsigned long )
        DECL_CTORS_T_A( float )
        DECL_CTORS_T_A( double )
        DECL_CTORS_T_A( const char* )
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
DECL_CTORS_T_A( const sc_fxval& )
DECL_CTORS_T_A( const sc_fxval_fast& )
DECL_CTORS_T_A( const sc_fxnum& )
DECL_CTORS_T_A( const sc_fxnum_fast& )
DECL_CTORS_T_B( int64 )
DECL_CTORS_T_B( uint64 )
DECL_CTORS_T_B( const sc_int_base& )
DECL_CTORS_T_B( const sc_uint_base& )
DECL_CTORS_T_B( const sc_signed& )
DECL_CTORS_T_B( const sc_unsigned& )
sc_fixed( const sc_fixed<W,I,Q,O,N>& );

 // Operators
sc_fixed& operator= ( const sc_fixed<W,I,Q,O,N>& );

#define DECL_ASN_OP_T( op , tp ) \
    sc_fixed& operator op ( tp );
#define DECL_ASN_OP_OTHER( op ) \
    DECL_ASN_OP_T( op , int64 ) \
    DECL_ASN_OP_T( op , uint64 ) \
    DECL_ASN_OP_T( op , const sc_int_base& ) \
    DECL_ASN_OP_T( op , const sc_uint_base& ) \
    DECL_ASN_OP_T( op , const sc_signed& ) \
    DECL_ASN_OP_T( op , const sc_unsigned& )
#define DECL_ASN_OP( op ) \
    DECL_ASN_OP_T( op , int ) \
    DECL_ASN_OP_T( op , unsigned int ) \
    DECL_ASN_OP_T( op , long ) \
    DECL_ASN_OP_T( op , unsigned long ) \
    DECL_ASN_OP_T( op, float ) \
    DECL_ASN_OP_T( op , double ) \
    DECL_ASN_OP_T( op , const char* ) \
    DECL_ASN_OP_T( op , const sc_fxval& ) \
    DECL_ASN_OP_T( op , const sc_fxval_fast& ) \
    DECL_ASN_OP_T( op , const sc_fxnum& ) \
    DECL_ASN_OP_T( op , const sc_fxnum_fast& ) \
    DECL_ASN_OP_OTHER( op )

DECL_ASN_OP( = )
DECL_ASN_OP( *= )
DECL_ASN_OP( /= )
DECL_ASN_OP( += )
DECL_ASN_OP( -= )
DECL_ASN_OP_T( <<= , int )
DECL_ASN_OP_T( >>= , int )
DECL_ASN_OP_T( &= , const sc_fix& )
DECL_ASN_OP_T( &= , const sc_fix_fast& )
DECL_ASN_OP_T( |= , const sc_fix& )
DECL_ASN_OP_T( |= , const sc_fix_fast& )
DECL_ASN_OP_T( ^= , const sc_fix& )
DECL_ASN_OP_T( ^= , const sc_fix_fast& )

sc_fxval operator++ ( int );
sc_fxval operator-- ( int );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_fixed& operator++ ();
        sc_fixed& operator-- ();
};

}       // namespace sc_dt
```

7.10.19.3 Constraints on usage

The word length shall be greater than zero. The number of saturated bits, if specified, shall not be less than zero.

7.10.19.4 Public constructors

The initial value of an **sc_fixed** object may be specified as a constructor argument, that is, a C++ or SystemC numeric object or a string literal. A fixed-point cast switch may also be passed as a constructor argument to set the fixed-point casting, as described in 7.10.8.

7.10.19.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ numeric representation to **sc_fixed**, using truncation or sign-extension, as described in 7.10.5.

7.10.20 sc_ufixed

7.10.20.1 Description

Class template **sc_ufixed** represents an unsigned finite-precision fixed-point value. The fixed-point type parameters **wl**, **iwl**, **q_mode**, **o_mode**, and **n_bits** shall be specified by the template arguments.

Any public member functions of the base class **sc_ufix** that are overridden in class **sc_ufixed** shall have the same behavior in the two classes. Any public member functions of the base class not overridden in this way shall be publicly inherited by class **sc_ufixed**.

7.10.20.2 Class definition

```
namespace sc_dt {

template <int W, int I,
      sc_q_mode Q = SC_DEFAULT_Q_MODE_,
      sc_o_mode O = SC_DEFAULT_O_MODE_, int N = SC_DEFAULT_N_BITS_>
class sc_ufixed
: public sc_ufix
{
    public:
        // Constructors
        explicit sc_ufixed();
        explicit sc_ufixed( const sc_fxcast_switch& );

        #define DECL_CTORS_T_A( tp )  \
            sc_ufixed( tp ); \
            sc_ufixed( tp , const sc_fxcast_switch& );
        #define DECL_CTORS_T_B( tp )  \
            explicit sc_ufixed( tp ); \
            sc_ufixed( tp , const sc_fxcast_switch& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
DECL_CTORS_T_A( int )
DECL_CTORS_T_A( unsigned int )
DECL_CTORS_T_A( long )
DECL_CTORS_T_A( unsigned long )
DECL_CTORS_T_A( float )
DECL_CTORS_T_A( double )
DECL_CTORS_T_A( const char* )
DECL_CTORS_T_A( const sc_fxval& )
DECL_CTORS_T_A( const sc_fxval_fast& )
DECL_CTORS_T_A( const sc_fxnum& )
DECL_CTORS_T_A( const sc_fxnum_fast& )
DECL_CTORS_T_B( int64 )
DECL_CTORS_T_B( uint64 )
DECL_CTORS_T_B( const sc_int_base& )
DECL_CTORS_T_B( const sc_uint_base& )
DECL_CTORS_T_B( const sc_signed& )
DECL_CTORS_T_B( const sc_unsigned& )

#undef DECL_CTORS_T_A
#undef DECL_CTORS_T_B

// Copy constructor
sc_ufixed( const sc_ufixed<W,I,Q,O,N>& );

// Assignment operators
sc_ufixed& operator= ( const sc_ufixed<W,I,Q,O,N>& );
#define DECL_ASN_OP_T( op , tp ) \
    sc_ufixed& operator op ( tp );
#define DECL_ASN_OP_OTHER( op ) \
    DECL_ASN_OP_T( op , int64 ) \
    DECL_ASN_OP_T( op , uint64 ) \
    DECL_ASN_OP_T( op , const sc_int_base& )\
    DECL_ASN_OP_T( op , const sc_uint_base& ) \
    DECL_ASN_OP_T( op , const sc_signed& ) \
    DECL_ASN_OP_T( op , const sc_unsigned& )
#define DECL_ASN_OP( op ) \
    DECL_ASN_OP_T( op , int ) \
    DECL_ASN_OP_T( op , unsigned int )\
    DECL_ASN_OP_T( op , long )  \
    DECL_ASN_OP_T( op , unsigned long )   \
    DECL_ASN_OP_T( op , float ) \
    DECL_ASN_OP_T( op , double ) \
    DECL_ASN_OP_T( op , const char* ) \
    DECL_ASN_OP_T( op , const sc_fxval& ) \
    DECL_ASN_OP_T( op , const sc_fxval_fast& ) \
    DECL_ASN_OP_T( op , const sc_fxnum& ) \
    DECL_ASN_OP_T( op , const sc_fxnum_fast& )  \
    DECL_ASN_OP_OTHER( op )

DECL_ASN_OP( = )
DECL_ASN_OP( *= )
DECL_ASN_OP( /= )
DECL_ASN_OP( += )
DECL_ASN_OP( -= )
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        DECL_ASN_OP_T( <<= , int )
        DECL_ASN_OP_T( >>= , int )
        DECL_ASN_OP_T( &= , const sc_ufix& )
        DECL_ASN_OP_T( &= , const sc_ufix_fast& )
        DECL_ASN_OP_T( |= , const sc_ufix& )
        DECL_ASN_OP_T( |= , const sc_ufix_fast& )
        DECL_ASN_OP_T( ^= , const sc_ufix& )
        DECL_ASN_OP_T( ^= , const sc_ufix_fast& )

        #undef DECL_ASN_OP_T
        #undef DECL_ASN_OP_OTHER
        #undef DECL_ASN_OP

        // Auto-increment and auto-decrement
        sc_fxval operator++ ( int );
        sc_fxval operator-- ( int );
        sc_ufixed& operator++ ();
        sc_ufixed& operator-- ();
};

}       // namespace sc_dt
```

7.10.20.3 Constraints on usage

The word length shall be greater than zero. The number of saturated bits, if specified, shall not be less than zero.

7.10.20.4 Public constructors

The initial value of an **sc_ufixed** object may be specified as a constructor argument that is a C++ or SystemC numeric object or a string literal. A fixed-point cast switch may also be passed as a constructor argument to set the fixed-point casting, as described in 7.10.8.

7.10.20.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ numeric representation to **sc_ufixed**, using truncation or sign-extension, as described in 7.10.5.

7.10.21 sc_fixed_fast

7.10.21.1 Description

Class template **sc_fixed_fast** shall represent a signed (two's complement) limited-precision fixed-point type. The fixed-point type parameters **wl**, **iwl**, **q_mode**, **o_mode**, and **n_bits** shall be specified by the template arguments.

Any public member functions of the base class **sc_fix_fast** that are overridden in class **sc_fixed_fast** shall have the same behavior in the two classes. Any public member functions of the base class not overridden in this way shall be publicly inherited by class **sc_fixed_fast**.

7.10.21.2 Class definition

```
namespace sc_dt {
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
template <int W, int I,
    sc_q_mode Q = SC_DEFAULT_Q_MODE_,
    sc_o_mode O = SC_DEFAULT_O_MODE_, int N = SC_DEFAULT_N_BITS_>
class sc_fixed_fast
: public sc_fix_fast
{
    public:
        // Constructors
        sc_fixed_fast();
        sc_fixed_fast( const sc_fxcast_switch& );

        #define DECL_CTORS_T_A( tp ) \
            sc_fixed_fast( tp ); \
            sc_fixed_fast( tp , const sc_fxcast_switch& );
        #define DECL_CTORS_T_B( tp ) \
            sc_fixed_fast( tp ); \
            sc_fixed_fast( tp , const sc_fxcast_switch& );

        DECL_CTORS_T_A( int )
        DECL_CTORS_T_A( unsigned int )
        DECL_CTORS_T_A( long )
        DECL_CTORS_T_A( unsigned long )
        DECL_CTORS_T_A( float )
        DECL_CTORS_T_A( double )
        DECL_CTORS_T_A( const char* )
        DECL_CTORS_T_A( const sc_fxval& )
        DECL_CTORS_T_A( const sc_fxval_fast& )
        DECL_CTORS_T_A( const sc_fxnum& )
        DECL_CTORS_T_A( const sc_fxnum_fast& )
        DECL_CTORS_T_B( int64 )
        DECL_CTORS_T_B( uint64 )
        DECL_CTORS_T_B( const sc_int_base& )
        DECL_CTORS_T_B( const sc_uint_base& )
        DECL_CTORS_T_B( const sc_signed& )
        DECL_CTORS_T_B( const sc_unsigned& )

         sc_fixed_fast( const sc_fixed_fast<W,I,Q,O,N>& );

         // Operators
        sc_fixed_fast& operator= ( const
            sc_fixed_fast<W,I,Q,O,N>& );
        #define DECL_ASN_OP_T( op , tp ) \
            sc_fixed_fast& operator op ( tp );
        #define DECL_ASN_OP_OTHER( op ) \
            DECL_ASN_OP_T( op , int64 ) \
            DECL_ASN_OP_T( op , uint64 ) \
            DECL_ASN_OP_T( op , const sc_int_base& ) \
            DECL_ASN_OP_T( op , const sc_uint_base& ) \
            DECL_ASN_OP_T( op , const sc_signed& ) \
            DECL_ASN_OP_T( op , const sc_unsigned& )
        #define DECL_ASN_OP( op ) \
            DECL_ASN_OP_T( op , int ) \
            DECL_ASN_OP_T( op , unsigned int ) \
            DECL_ASN_OP_T( op , long ) \
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    DECL_ASN_OP_T( op , unsigned long ) \
    DECL_ASN_OP_T( op , float ) \
    DECL_ASN_OP_T( op , double ) \
    DECL_ASN_OP_T( op , const char* ) \
    DECL_ASN_OP_T( op , const sc_fxval& ) \
    DECL_ASN_OP_T( op , const sc_fxval_fast& ) \
    DECL_ASN_OP_T( op , const sc_fxnum& ) \
    DECL_ASN_OP_T( op , const sc_fxnum_fast& ) \
    DECL_ASN_OP_OTHER( op )
    DECL_ASN_OP( = )
    DECL_ASN_OP( *= )
    DECL_ASN_OP( /= )
    DECL_ASN_OP( += )
    DECL_ASN_OP( -= )
    DECL_ASN_OP_T( <<= , int )
    DECL_ASN_OP_T( >>= , int )
    DECL_ASN_OP_T( &= , const sc_fix& )
    DECL_ASN_OP_T( &= , const sc_fix_fast& )
    DECL_ASN_OP_T( |= , const sc_fix& )
    DECL_ASN_OP_T( |= , const sc_fix_fast& )
    DECL_ASN_OP_T( ^= , const sc_fix& )
    DECL_ASN_OP_T( ^= , const sc_fix_fast& )
    sc_fxval_fast operator++ ( int );
    sc_fxval_fast operator-- ( int );
    sc_fixed_fast& operator++ ();
    sc_fixed_fast& operator-- ();
};

}       // namespace sc_dt
```

7.10.21.3 Constraints on usage

The word length shall be greater than zero. The number of saturated bits, if specified, shall not be less than zero.

sc_fixed_fast shall use double-precision (floating-point) values whose mantissa is limited to 53 bits.

7.10.21.4 Public constructors

The initial value of an **sc_fixed_fast** object may be specified as a constructor argument that is a C++ or SystemC numeric object or a string literal. A fixed-point cast switch may also be passed as a constructor argument to set the fixed-point casting, as described in 7.10.8.

7.10.21.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ numeric representation to **sc_fixed_fast**, using truncation or sign-extension, as described in 7.10.5.

7.10.22 sc_ufixed_fast

7.10.22.1 Description

Class template **sc_ufixed_fast** shall represent an unsigned limited-precision fixed-point type. The fixed-point type parameters **wl**, **iwl**, **q_mode**, **o_mode**, and **n_bits** shall be specified by the template arguments.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Any public member functions of the base class **sc_ufix_fast** that are overridden in class **sc_ufixed_fast** shall have the same behavior in the two classes. Any public member functions of the base class not overridden in this way shall be publicly inherited by class **sc_ufixed_fast**.

7.10.22.2 Class definition

```
namespace sc_dt {

template <int W, int I,
    sc_q_mode Q = SC_DEFAULT_Q_MODE_,
    sc_o_mode O = SC_DEFAULT_O_MODE_, int N = SC_DEFAULT_N_BITS_>
class sc_ufixed_fast
: public sc_ufix_fast
{
    public:
        // Constructors
        explicit sc_ufixed_fast();
        explicit sc_ufixed_fast( const sc_fxcast_switch& );

    #define DECL_CTORS_T_A( tp )  \
        sc_ufixed_fast( tp ); \
        sc_ufixed_fast( tp , const sc_fxcast_switch& );
    #define DECL_CTORS_T_B( tp )  \
        explicit sc_ufixed_fast ( tp );\
        sc_ufixed_fast( tp , const sc_fxcast_switch& );

        DECL_CTORS_T_A( int )
        DECL_CTORS_T_A( unsigned int )
        DECL_CTORS_T_A( long )
        DECL_CTORS_T_A( unsigned long )
        DECL_CTORS_T_A( float )
        DECL_CTORS_T_A( double )
        DECL_CTORS_T_A( const char* )
        DECL_CTORS_T_A( const sc_fxval& )
        DECL_CTORS_T_A( const sc_fxval_fast& )
        DECL_CTORS_T_A( const sc_fxnum& )
        DECL_CTORS_T_A( const sc_fxnum_fast& )
        DECL_CTORS_T_B( int64 )
        DECL_CTORS_T_B( uint64 )
        DECL_CTORS_T_B( const sc_int_base& )
        DECL_CTORS_T_B( const sc_uint_base& )
        DECL_CTORS_T_B( const sc_signed& )
        DECL_CTORS_T_B( const sc_unsigned& )

        #undef DECL_CTORS_T_A
        #undef DECL_CTORS_T_B

        // Copy constructor
        sc_ufixed_fast( const sc_ufixed_fast<W,I,Q,O,N>& );

        // Assignment operators
        sc_ufixed_fast& operator= ( const sc_ufixed_fast<W,I,Q,O,N>& );

        #define DECL_ASN_OP_T( op , tp ) \
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    sc_ufixed_fast& operator op ( tp );
#define DECL_ASN_OP_OTHER( op ) \
    DECL_ASN_OP_T( op , int64 ) \
    DECL_ASN_OP_T( op , uint64 ) \
    DECL_ASN_OP_T( op , const sc_int_base& ) \
    DECL_ASN_OP_T( op , const sc_uint_base& ) \
    DECL_ASN_OP_T( op , const sc_signed& ) \
    DECL_ASN_OP_T( op , const sc_unsigned& )

#define DECL_ASN_OP( op ) \
    DECL_ASN_OP_T( op , int ) \
    DECL_ASN_OP_T( op , unsigned int )\
    DECL_ASN_OP_T( op , long )  \
    DECL_ASN_OP_T( op , unsigned long )  \
    DECL_ASN_OP_T( op, float ) \
    DECL_ASN_OP_T( op , double ) \
    DECL_ASN_OP_T( op , const char* )\
    DECL_ASN_OP_T( op , const sc_fxval& ) \
    DECL_ASN_OP_T( op , const sc_fxval_fast& ) \
    DECL_ASN_OP_T( op , const sc_fxnum& ) \
    DECL_ASN_OP_T( op , const sc_fxnum_fast& \
    DECL_ASN_OP_OTHER( op )

DECL_ASN_OP( = )
DECL_ASN_OP( *= )
DECL_ASN_OP( /= )
DECL_ASN_OP( += )
DECL_ASN_OP( -= )
DECL_ASN_OP_T( <<= , int )
DECL_ASN_OP_T( >>= , int )
DECL_ASN_OP_T( &= , const sc_ufix& )
DECL_ASN_OP_T( &= , const sc_ufix_fast& )
DECL_ASN_OP_T( |= , const sc_ufix& )
DECL_ASN_OP_T( |= , const sc_ufix_fast& )
DECL_ASN_OP_T( ^= , const sc_ufix& )
DECL_ASN_OP_T( ^= , const sc_ufix_fast& )

#undef DECL_ASN_OP_T
#undef DECL_ASN_OP_OTHER
#undef DECL_ASN_OP

// Auto-increment and auto-decrement
sc_fxval_fast operator++ ( int );
sc_fxval_fast operator-- ( int );
sc_ufixed_fast& operator++ ();
sc_ufixed_fast& operator-- ();
};

}       // namespace sc_dt
```

7.10.22.3 Constraints on usage

The word length shall be greater than zero. The number of saturated bits, if specified, shall not be less than zero.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

sc_ufixed_fast shall use double-precision (floating-point) values whose mantissa is limited to 53 bits.

7.10.22.4 Public constructors

The initial value of an **sc_fixed_fast** object may be specified as a constructor argument, that is, a C++ or SystemC numeric object or a string literal. A fixed-point cast switch may also be passed as a constructor argument to set the fixed-point casting, as described in 7.10.8.

7.10.22.5 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ numeric representation to **sc_fixed_fast**, using truncation or sign-extension, as described in 7.10.5.

7.10.23 Bit-selects

7.10.23.1 Description

Class *sc_fxnum_bitref_r*† shall represent a read-only bit selected from an **sc_fxnum**.

Class *sc_fxnum_bitref*† shall represent a read-write bit selected from an **sc_fxnum**.

Class *sc_fxnum_fast_bitref_r*† shall represent a read-only bit selected from an **sc_fxnum_fast**.

Class *sc_fxnum_fast_bitref*† shall represent a read-write bit selected from an **sc_fxnum_fast**.

No distinction shall be made between a bit-select used as an lvalue or as an rvalue.

7.10.23.2 Class definition

```
class  sc_fxnum_fast_bitref_r {
    friend class sc_fxnum;
    friend class sc_fxnum_fast_bitref;

    protected:
        bool get() const;
        sc_fxnum_fast_bitref_r(sc_fxnum&, int );

    public:
        sc_fxnum_fast_bitref_r ( const sc_fxnum_fast_bitref_r& );

        operator bool() const;

        // print or dump content
        void print( std::ostream& = std::cout ) const;
        void dump( std::ostream& = std::cout ) const;

    protected:
        sc_fxnum& m_num;
        int       m_idx;
};
class  sc_fxnum_bitref : public sc_fxnum_bitref_r
{

    friend class sc_fxnum;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    void set( bool );

    // constructor
    sc_fxnum_bitref( sc_fxnum&, int );

    public:
        // copy constructor
        sc_fxnum_bitref( const sc_fxnum_bitref& );

        // assignment operators
        #define DECL_ASN_OP_T(op,tp)                                    \
            sc_fxnum_bitref& operator op ( tp );

        #define DECL_ASN_OP(op)                                         \
            DECL_ASN_OP_T(op,const sc_fxnum_bitref&)                    \
            DECL_ASN_OP_T(op,const sc_fxnum_fast_bitref&)               \
            DECL_ASN_OP_T(op,const sc_bit&)                             \
            DECL_ASN_OP_T(op,bool)

        DECL_ASN_OP(=)

        DECL_ASN_OP(&=)
        DECL_ASN_OP(|=)
        DECL_ASN_OP(^=)

        #undef DECL_ASN_OP_T
        #undef DECL_ASN_OP

        void scan( std::istream& = std::cin );
};

        class  sc_fxnum_fast_bitref : public sc_fxnum_fast_bitref_r
{
        friend class sc_fxnum_fast;

        void set( bool );

        sc_fxnum_fast_bitref( sc_fxnum_fast&, int );

    public:
        sc_fxnum_fast_bitref( const sc_fxnum_fast_bitref& );

        // assignment operators
        #define DECL_ASN_OP_T(op,tp)                                    \
            sc_fxnum_fast_bitref& operator op ( tp );

        #define DECL_ASN_OP(op)                                         \
            DECL_ASN_OP_T(op,const sc_fxnum_bitref&)                    \
            DECL_ASN_OP_T(op,const sc_fxnum_fast_bitref&)               \
            DECL_ASN_OP_T(op,const sc_bit&)                             \
            DECL_ASN_OP_T(op,bool)

        DECL_ASN_OP(=)
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        DECL_ASN_OP(&=)
        DECL_ASN_OP(|=)
        DECL_ASN_OP(^=)

        #undef DECL_ASN_OP_T
        #undef DECL_ASN_OP

        void scan( std::istream& = std::cin );
};

class  sc_fxnum_fast_bitref_r
{
    friend class sc_fxnum_fast;
    friend class sc_fxnum_bitref;

    protected:
        bool get() const;
        sc_fxnum_fast_bitref_r( sc_fxnum_fast&, int );

    public:
        // copy constructor
        sc_fxnum_fast_bitref_r( const sc_fxnum_fast_bitref_r& );

        operator bool() const;

        void print( std::ostream& = std::cout ) const;
        void dump( std::ostream& = std::cout ) const;

    protected:
        sc_fxnum_fast& m_num;
        int          m_idx;
};
```

7.10.23.3 Constraints on usage

Bit-select objects shall only be created using the bit-select operators of an instance of a class derived from **sc_fxnum** or **sc_fxnum_fast**.

An application shall not explicitly create an instance of any bit-select class.

An application should not declare a reference or pointer to any bit-select object.

7.10.23.4 Assignment operators

Overloaded assignment operators shall provide conversion from **bool** values.

7.10.23.5 Implicit type conversion

operator bool() const;

Operator **bool** can be used for implicit type conversion from a bit-select to the native C++ bool representation.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.24 Part-selects

7.10.24.1 Description

Class *sc_fxnum_subref_r*† shall represent a read-only bit selected from an **sc_fxnum**.

Class *sc_fxnum_subref*† shall represent a read-write bit selected from an **sc_fxnum**.

Class *sc_fxnum_fast_subref_r*† shall represent a read-only bit selected from an **sc_fxnum_fast**.

Class *sc_fxnum_fast_subref*† shall represent a read-write bit selected from an **sc_fxnum_fast**.

No distinction shall be made between a part-select used as an lvalue or as an rvalue.

7.10.24.2 Class definition

```
class sc_fxnum_subref_r
{
    friend class sc_fxnum;

    protected:
        bool get() const;

        sc_fxnum_subref_r( sc_fxnum&, int, int );

    public:
        // copy constructor
        sc_fxnum_subref_r( const sc_fxnum_subref_r& );

        // destructor
        ~sc_fxnum_subref_r();

        // relational operators

        #define DECL_REL_OP_T(op,tp)                                \
            friend bool operator op ( const sc_fxnum_subref_r&, tp );   \
            friend bool operator op ( tp, const sc_fxnum_subref_r& );

        #define DECL_REL_OP(op)                                     \
            friend bool operator op ( const sc_fxnum_subref_r&,     \
                const sc_fxnum_subref_r& );                         \
            friend bool operator op ( const sc_fxnum_subref_r&,     \
                const sc_fxnum_fast_subref_r& );                    \
        DECL_REL_OP_T(op,const sc_bv_base&)                         \
        DECL_REL_OP_T(op,const sc_lv_base&)                         \
        DECL_REL_OP_T(op,const char*)                               \
        DECL_REL_OP_T(op,const bool*)                               \
        DECL_REL_OP_T(op,const sc_signed&)                          \
        DECL_REL_OP_T(op,const sc_unsigned&)                        \
        DECL_REL_OP_T(op,int)                                       \
        DECL_REL_OP_T(op,unsigned int)                              \
        DECL_REL_OP_T(op,long)                                      \
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        DECL_REL_OP_T(op,unsigned long)

        DECL_REL_OP(==)
        DECL_REL_OP(!=)

        #undef DECL_REL_OP_T
        #undef DECL_REL_OP

        // reduce functions
        bool and_reduce() const;
        bool nand_reduce() const;
        bool or_reduce() const;
        bool nor_reduce() const;
        bool xor_reduce() const;
        bool xnor_reduce() const;

        // query parameter
        int length() const;

        // explicit conversions
        int             to_int() const;
        unsigned int    to_uint() const;
        long            to_long() const;
        unsigned long   to_ulong() const;
        int64           to_int64() const;
        uint64          to_uint64() const;

        #ifdef SC_DT_DEPRECATED
        int             to_signed() const;
        unsigned int    to_unsigned() const;
        #endif

        std::string to_string() const;
        std::string to_string( sc_numrep ) const;
        std::string to_string( sc_numrep, bool ) const;

        // implicit conversion
        operator sc_bv_base() const;

        // print or dump content
        void print( std::ostream& = std::cout ) const;
        void dump( std::ostream& = std::cout ) const;
};

class sc_fxnum_subref : public sc_fxnum_subref_r
{
    friend class sc_fxnum;
    friend class sc_fxnum_fast_subref;

    bool set();

        // constructor
        sc_fxnum_subref( sc_fxnum&, int, int );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    public:
        // copy constructor
        sc_fxnum_subref( const sc_fxnum_subref& );

        // assignment operators
        #define DECL_ASN_OP_T(tp)                                   \
        sc_fxnum_subref& operator = ( tp );

        DECL_ASN_OP_T(const sc_fxnum_subref&)
        DECL_ASN_OP_T(const sc_fxnum_fast_subref&)
        DECL_ASN_OP_T(const sc_bv_base&)
        DECL_ASN_OP_T(const sc_lv_base&)
        DECL_ASN_OP_T(const char*)
        DECL_ASN_OP_T(const bool*)
        DECL_ASN_OP_T(const sc_signed&)
        DECL_ASN_OP_T(const sc_unsigned&)
        DECL_ASN_OP_T(const sc_int_base&)
        DECL_ASN_OP_T(const sc_uint_base&)
        DECL_ASN_OP_T(int64)
        DECL_ASN_OP_T(uint64)
        DECL_ASN_OP_T(int)
        DECL_ASN_OP_T(unsigned int)
        DECL_ASN_OP_T(long)
        DECL_ASN_OP_T(unsigned long)
        DECL_ASN_OP_T(char)

        #undef DECL_ASN_OP_T

        #define DECL_ASN_OP_T_A(op,tp)                              \
        sc_fxnum_subref& operator op ## = ( tp );

        #define DECL_ASN_OP_A(op)                                   \
        DECL_ASN_OP_T_A(op,const sc_fxnum_subref&)                  \
        DECL_ASN_OP_T_A(op,const sc_fxnum_fast_subref&)             \
        DECL_ASN_OP_T_A(op,const sc_bv_base&)                       \
        DECL_ASN_OP_T_A(op,const sc_lv_base&)

        DECL_ASN_OP_A(&)
        DECL_ASN_OP_A(|)
        DECL_ASN_OP_A(^)

        #undef DECL_ASN_OP_T_A
        #undef DECL_ASN_OP_A

        void scan( std::istream& = std::cin );
};

class  sc_fxnum_fast_subref_r
{
    friend class sc_fxnum_fast;

    protected:
        bool get() const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    sc_fxnum_fast_subref_r( sc_fxnum_fast&, int, int );

public:
    // copy constructor
    sc_fxnum_fast_subref_r( const sc_fxnum_fast_subref_r& );

    // destructor
    ~sc_fxnum_fast_subref_r();

    // relational operators
    #define DECL_REL_OP_T(op,tp)                                    \
    friend bool operator op ( const sc_fxnum_fast_subref_r&, tp );  \
    friend bool operator op ( tp, const sc_fxnum_fast_subref_r& );

    #define DECL_REL_OP(op)                                         \
    friend bool operator op ( const sc_fxnum_fast_subref_r&,        \
        const sc_fxnum_fast_subref_r& );                            \
    friend bool operator op ( const sc_fxnum_fast_subref_r&,        \
        const sc_fxnum_subref_r& );                                 \
    DECL_REL_OP_T(op,const sc_bv_base&)                             \
    DECL_REL_OP_T(op,const sc_lv_base&)                             \
    DECL_REL_OP_T(op,const char*)                                   \
    DECL_REL_OP_T(op,const bool*)                                   \
    DECL_REL_OP_T(op,const sc_signed&)                              \
    DECL_REL_OP_T(op,const sc_unsigned&)                            \
    DECL_REL_OP_T(op,int)                                           \
    DECL_REL_OP_T(op,unsigned int)                                  \
    DECL_REL_OP_T(op,long)                                          \
    DECL_REL_OP_T(op,unsigned long)

    DECL_REL_OP(==)
    DECL_REL_OP(!=)

    #undef DECL_REL_OP_T
    #undef DECL_REL_OP

    // reduce functions
    bool and_reduce() const;
    bool nand_reduce() const;
    bool or_reduce() const;
    bool nor_reduce() const;
    bool xor_reduce() const;
    bool xnor_reduce() const;

    // query parameter
    int length() const;

    // explicit conversions
    int             to_int() const;
    unsigned int    to_uint() const;
    long            to_long() const;
    unsigned long   to_ulong() const;
    int64           to_int64() const;
    uint64          to_uint64() const;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        #ifdef SC_DT_DEPRECATED
        int             to_signed() const;
        unsigned int    to_unsigned() const;
        #endif

        std::string to_string() const;
        std::string to_string( sc_numrep ) const;
        std::string to_string( sc_numrep, bool ) const;

        // implicit conversion
        operator sc_bv_base() const;

        // print or dump content
        void print( std::ostream& = std::cout ) const;
        void dump( std::ostream& = std::cout ) const;
};

class  sc_fxnum_fast_subref : public sc_fxnum_fast_subref_r
{
    friend class sc_fxnum_fast;
    friend class sc_fxnum_subref;

    bool set();

        // constructor
        sc_fxnum_fast_subref( sc_fxnum_fast&, int, int );

    public:
        // copy constructor
        sc_fxnum_fast_subref( const sc_fxnum_fast_subref& );

        // assignment operators
        #define DECL_ASN_OP_T(tp)                                       \
        sc_fxnum_fast_subref& operator = ( tp );

        DECL_ASN_OP_T(const sc_fxnum_subref&)
        DECL_ASN_OP_T(const sc_fxnum_fast_subref&)
        DECL_ASN_OP_T(const sc_bv_base&)
        DECL_ASN_OP_T(const sc_lv_base&)
        DECL_ASN_OP_T(const char*)
        DECL_ASN_OP_T(const bool*)
        DECL_ASN_OP_T(const sc_signed&)
        DECL_ASN_OP_T(const sc_unsigned&)
        DECL_ASN_OP_T(const sc_int_base&)
        DECL_ASN_OP_T(const sc_uint_base&)
        DECL_ASN_OP_T(int64)
        DECL_ASN_OP_T(uint64)
        DECL_ASN_OP_T(int)
        DECL_ASN_OP_T(unsigned int)
        DECL_ASN_OP_T(long)
        DECL_ASN_OP_T(unsigned long)
        DECL_ASN_OP_T(char)
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    #undef DECL_ASN_OP_T

    #define DECL_ASN_OP_T_A(op,tp)                          \
    sc_fxnum_fast_subref& operator op ## = ( tp );

    #define DECL_ASN_OP_A(op)                               \
    DECL_ASN_OP_T_A(op,const sc_fxnum_subref&)              \
    DECL_ASN_OP_T_A(op,const sc_fxnum_fast_subref&)         \
    DECL_ASN_OP_T_A(op,const sc_bv_base&)                   \
    DECL_ASN_OP_T_A(op,const sc_lv_base&)

    DECL_ASN_OP_A(&)
    DECL_ASN_OP_A(|)
    DECL_ASN_OP_A(^)

    #undef DECL_ASN_OP_T_A
    #undef DECL_ASN_OP_A

    void scan( std::istream& = std::cin );
};
```

7.10.24.3 Constraints on usage

Fixed-point part-select objects shall only be created using the part-select operators of an instance of a class derived from **sc_fxnum** or **sc_fxnum_fast**.

An application shall not explicitly create an instance of any fixed-point part-select class.

An application should not declare a reference or pointer to any fixed-point part-select object.

No arithmetic operators are provided for fixed-point part-selects.

7.10.24.4 Assignment operators

Overloaded assignment operators shall provide conversion from SystemC data types and the native C++ integer representation to fixed-point part-selects. If the size of a data type or string literal operand differs from the fixed-point part-select word length, truncation, zero-extension, or sign-extension shall be used, as described in 7.2.2.

7.10.24.5 Bitwise operators

Overloaded bitwise operators shall be provided for fixed-point part-select, bit-vector, and logic-vector operands.

7.10.24.6 Implicit type conversion

sc_fxnum_subref_r† :: **operator sc_bv_base**() const;
sc_fxnum_fast_subref_r† :: **operator sc_bv_base**() const;

Operator **sc_bv_base** can be used for implicit type conversion from integer part-selects to the SystemC bit-vector representation.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.10.24.7 Explicit type conversion

int **to_int**() const;
unsigned int **to_uint**() const;
long **to_long**() const;
unsigned long **to_ulong**() const;
int64 **to_int64**() const;
uint64 **to_uint64**() const;

These member functions shall perform the conversion to C++ integer types.

std::string **to_string**() const;
std::string **to_string**(sc_numrep) const;
std::string **to_string**(sc_numrep , bool) const;

Member function **to_string** shall perform the conversion to a **string** representation, as described in 7.2.12, 7.10.9.1, and 7.10.9.2.

7.11 Contexts

7.11.1 Overview

Subclause 7.11 describes the classes that are provided to set the contexts for the data types.

7.11.2 sc_length_param

7.11.2.1 Description

Class **sc_length_param** shall represent a length parameter and shall be used to create a length context, as described in 7.2.4.

7.11.2.2 Class definition

```
namespace sc_dt {

class sc_length_param
{
    public:
        sc_length_param();
        sc_length_param( int );
        sc_length_param( const sc_length_param& );

        sc_length_param& operator= ( const sc_length_param& );
        friend bool operator== ( const sc_length_param& , const sc_length_param& );
        friend bool operator!= ( const sc_length_param& , const sc_length_param& );

        int len() const;
        void len( int );
        std::string to_string() const;
        void print( std::ostream& = std::cout ) const;
        void dump( std::ostream& = std::cout ) const;
};

}       // namespace sc_dt
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.11.2.3 Constraints on usage

The length (where specified) shall be greater than zero.

7.11.2.4 Public constructors

sc_length_param();

Default constructor **sc_length_param** shall create an **sc_length_param** object with the default word length of 32.

sc_length_param(int n) ;

Constructor **sc_length_param** shall create an **sc_length_param** with **n** as the word length with **n > 0**.

sc_length_param(const sc_length_param&);

Constructor **sc_length_param** shall create a copy of the object given as its argument.

7.11.2.5 Public member functions

int **len**() const;

Member function **len** shall return the word length stored in the **sc_length_param**.

void **len**(int n);

Member function len shall set the word length of the **sc_length_param** to **n**, with **n > 0**.

std::string **to_string**() const;

Member function **to_string** shall convert the **sc_length_param** into its string representation.

void **print**(std::ostream& = std::cout) const;

Member function **print** shall print the contents to a stream.

7.11.2.6 Public operators

sc_length_param& **operator=** (const sc_length_param& a);

operator= shall assign the word-length value of **a** to the left-hand side **sc_length_param** instance.

friend bool **operator==** (const sc_length_param& a , sc_length_param& b);

operator== shall return **true** if the stored lengths of **a** and **b** are equal.

friend bool **operator!=** (const sc_length_param& a , const sc_length_param& b);

operator!= shall return **true** if the stored lengths of **a** and **b** are not equal.

7.11.3 sc_length_context

7.11.3.1 Description

Class **sc_length_context** shall be used to create a length context for SystemC integer and vector objects.

7.11.3.2 Class definition

namespace sc_dt {

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
class sc_length_context
{
    public:
        explicit sc_length_context( const sc_length_param& , sc_context_begin† = SC_NOW );
        ~sc_length_context();

        void begin();
        void end();
        static const sc_length_param& default_value();
        const sc_length_param& value() const;
};

}       // namespace sc_dt
```

7.11.3.3 Public constructor

explicit **sc_length_context**(const sc_length_param& , *sc_context_begin*† = SC_NOW);

Constructor **sc_length_context** shall create an **sc_length_context** object. The first argument shall be the length parameter to use. The second argument, if supplied, shall have the value SC_NOW or SC_LATER.

7.11.3.4 Public member functions

void **begin**();

Member function **begin** shall set the current length context, as described in 7.2.4.

static const sc_length_param& **default_value**();

Member function **default_value** shall return the length parameter currently in context.

void **end**();

Member function **end** shall deactivate the length context and shall remove it from the top of the length context stack, as described in 7.2.4.

const sc_length_param& **value**() const;

Member function **value** shall return the length parameter.

7.11.4 sc_fxtype_params

7.11.4.1 Description

Class **sc_fxtype_params** shall represent a length parameter and shall be used to create a length context for fixed-point objects, as described in 7.2.4.

7.11.4.2 Class definition

```
namespace sc_dt {

class sc_fxtype_params
{
    public:
        // Constructors and destructor
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        sc_fxtype_params();
        sc_fxtype_params( int , int );
        sc_fxtype_params( sc_q_mode , sc_o_mode, int = 0 );
        sc_fxtype_params( int , int , sc_q_mode , sc_o_mode , int = 0 );
        sc_fxtype_params( const sc_fxtype_params& );
        sc_fxtype_params( const sc_fxtype_params& , int , int );
        sc_fxtype_params( const sc_fxtype_params& , sc_q_mode , sc_o_mode , int = 0 );

        // Operators
        sc_fxtype_params& operator= ( const sc_fxtype_params& );
        friend bool operator== ( const sc_fxtype_params& , const sc_fxtype_params& );
        friend bool operator!= ( const sc_fxtype_params& , const sc_fxtype_params& );

        // Member functions
        int wl() const;
        void wl( int );
        int iwl() const;
        void iwl( int );
        sc_q_mode q_mode() const;
        void q_mode( sc_q_mode );
        sc_o_mode o_mode() const;
        void o_mode( sc_o_mode );
        int n_bits() const;
        void n_bits( int );
        std::string to_string() const;
        void print( std::ostream& = std::cout ) const;
        void dump( std::ostream& = std::cout ) const;
};

}       // namespace sc_dt
```

7.11.4.3 Constraints on usage

The length (where specified) shall be greater than zero.

7.11.4.4 Public constructors

sc_fxtype_params (int wl , int iwl) ;
sc_fxtype_params (sc_q_mode q_mode , sc_o_mode o_mode) ;
sc_fxtype_params (sc_q_mode q_mode , sc_o_mode o_mode , int n_bits) ;
sc_fxtype_params (int wl , int iwl , sc_q_mode q_mode , sc_o_mode o_mode , int n_bits) ;
sc_fxtype_params (int wl , int iwl , sc_q_mode q_mode , sc_o_mode o_mode) ;
sc_fxtype_params () ;

Constructor **sc_fxtype_params** shall create an **sc_fxtype_params** object**.**

wl shall be the total number of bits in the fixed-point format. **wl** shall be greater than zero. The default value for **wl** shall be obtained from the fixed-point context currently in scope.

iwl shall be the number of integer bits in the fixed-point format. **iwl** may be positive or negative. The default value for **iwl** shall be obtained from the fixed-point context currently in scope.

q_mode shall be the quantization mode to use. Valid values for **o_mode** are given in 7.10.10.10. The default value for **q_mode** shall be obtained from the fixed-point context currently in scope.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

o_mode shall be the overflow mode to use. Valid values for **o_mode** are given in 7.10.10.2. The default value for **o_mode** shall be obtained from the fixed-point context currently in scope.

n_bits shall be the number of saturated bits parameter for the selected overflow mode. **n_bits** shall be greater than or equal to zero. If the overflow mode is specified, the default value shall be zero. If the overflow mode is not specified, the default value shall be obtained from the fixed-point context currently in scope.

If no fixed-point context is currently in scope, the default values for **wl**, **iwl**, **q_mode**, **o_mode**, and **n_bits** shall be those defined in Table 38 (see 7.10.8).

7.11.4.5 Public member functions

int **iwl**() const;

Member function **iwl** shall return the iwl value.

void **iwl**(int val);

Member function **iwl** shall set the iwl value to **val**.

int **n_bits**() const;

Member function **n_bits** shall return the **n_bits** value.

void **n_bits**(int);

Member function **n_bits** shall set the **n_bits** value to val.

sc_o_mode **o_mode**() const;

Member function **o_mode** shall return the **o_mode**.

void **o_mode**(sc_o_mode mode);

Member function **o_mode** shall set the **o_mode** to mode.

sc_q_mode **q_mode**() const;

Member function **q_mode** shall return the **q_mode**.

void **q_mode**(sc_q_mode mode);

Member function **q_mode** shall set the **q_mode** to mode.

int **wl**() const;

Member function **wl** shall return the **wl** value.

void **wl**(int val);

Member function **wl** shall set the **wl** value to **val**.

7.11.4.6 Operators

sc_fxtype_params& **operator=** (const sc_fxtype_params& param_);

operator= shall assign the **wl**, **iwl**, **q_mode**, **o_mode**, and **n_bits** of **param_** of the right-hand side to the left-hand side.

friend bool **operator==** (const sc_fxtype_params& param_a , const sc_fxtype_params& param_b);

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

operator== shall return **true** if **wl**, **iwl**, **q_mode**, **o_mode**, and **n_bits** of **param_a** are equal to the corresponding values of **param_b**; otherwise, it shall return **false**.

friend bool **operator!=** (const sc_fxtype_params& , const sc_fxtype_params&);

operator!= shall return **true** if **wl**, **iwl**, **q_mode**, **o_mode**, and **n_bits** of **param_a** are not equal to the corresponding values of **param_b**; otherwise, it shall return **false**.

7.11.5 sc_fxtype_context

7.11.5.1 Description

Class **sc_fxtype_context** shall be used to create a length context for fixed-point objects.

7.11.5.2 Class definition

```
namespace sc_dt {

class sc_fxtype_context
{
    public:
        explicit sc_fxtype_context( const sc_fxtype_params& , sc_context_begin† = SC_NOW );
        ~sc_fxtype_context();

        void begin();
        void end();
        static const sc_fxtype_params& default_value();
        const sc_fxtype_params& value() const;
};

}       // namespace sc_dt
```

7.11.5.3 Public constructor

explicit **sc_fxtype_context**(const sc_fxtype_params& , *sc_context_begin*† = SC_NOW);

Constructor **sc_fxtype_context** shall create an **sc_fxtype_context** object. The first argument shall be the fixed-point length parameter to use. The second argument (if supplied) shall have the value SC_NOW or SC_LATER.

7.11.5.4 Public member functions

void **begin**();

Member function **begin** shall set the current length context, as described in 7.2.4.

static const sc_fxtype_params& **default_value**();

Member function **default_value** shall return the length parameter currently in context.

void **end**();

Member function **end** shall deactivate the length context and remove it from the top of the length context stack, as described in 7.2.4.

const sc_fxtype_params& **value**() const;

Member function **value** shall return the length parameter.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

7.11.6 sc_fxcast_switch

7.11.6.1 Description

Class **sc_fxcast_switch** shall be used to set the floating-point cast context, as described in 7.10.8.

7.11.6.2 Class definition

```
namespace sc_dt {

class sc_fxcast_switch
{
    public:
        // Constructors
        sc_fxcast_switch();
        sc_fxcast_switch( sc_switch† );
        sc_fxcast_switch( const sc_fxcast_switch& );

        // Operators
        sc_fxcast_switch& operator= ( const sc_fxcast_switch& );
        friend bool operator== ( const sc_fxcast_switch& , const sc_fxcast_switch& );
        friend bool operator!= ( const sc_fxcast_switch& , const sc_fxcast_switch& );

        // Member functions
        std::string to_string() const;
        void print( std::ostream& = std::cout ) const;
        void dump( std::ostream& = std::cout ) const;
};

}       // namespace sc_dt
```

7.11.6.3 Public constructors

sc_fxcast_switch ();
sc_fxcast_switch (*sc_switch*†);

The argument (if supplied) shall have the value SC_OFF or SC_ON, as described in 7.10.8. The default constructor shall use the floating-point cast context currently in scope.

7.11.6.4 Public member functions

void **print**(std::ostream& = std::cout) const;

Member function **print** shall print the **sc_fxcast_switch** instance value to an output stream.

7.11.6.5 Explicit conversion

std::string **to_string**() const;

Member function **to_string** shall return the switch state as the character string "SC_OFF" or "SC_ON".

7.11.6.6 Operators

sc_fxcast_switch& **operator=** (const sc_fxtype_params& cast_switch);

operator= shall assign **cast_switch** to the **sc_fxcast_switch** on its left-hand side.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

friend bool **operator==** (const sc_fxcast_switch& switch_a , const sc_fxcast_switch& switch_b) ;

operator== shall return **true** if **switch_a** is equal to **switch_b**; otherwise, it shall return **false**.

friend bool **operator!=** (const sc_fxcast_switch& switch_a , const sc_fxcast_switch& switch_b);

operator!= shall return **true** if **switch_a** is not equal to **switch_b**; otherwise, it shall return **false**.

std::ostream& **operator<<** (std::ostream& os , const sc_fxcast_switch& a);

operator<< shall print the instance value of **a** to an output stream **os**.

7.11.7 sc_fxcast_context

7.11.7.1 Description

Class **sc_fxcast_context** shall be used to create a floating-point cast context for fixed-point objects.

7.11.7.2 Class definition

```
namespace sc_dt {

class sc_fxcast_context
{
    public:
        explicit sc_fxcast_context( const sc_fxcast_switch& , sc_context_begin† = SC_NOW );
        sc_fxcast_context();

        void begin();
        void end();
        static const sc_fxcast_switch& default_value();
        const sc_fxcast_switch& value() const;
};

}       // namespace sc_dt
```

7.11.7.3 Public constructor

explicit **sc_fxcast_context**(const sc_fxcast_switch&, *sc_context_begin*† = SC_NOW);

Constructor **sc_fxcast_context** shall create an **sc_fxcast_context** object. Its first argument shall be the floating-point cast switch to use. The second argument (if supplied) shall have the value SC_NOW or SC_LATER.

7.11.7.4 Public member functions

void **begin**();

Member function **begin** shall set the current floating-point cast context, as described in 7.10.8.

static const sc_fxcast_switch& **default_value**();

Member function **default_value** shall return the cast switch currently in context.

void **end**();

Member function **end** shall deactivate the floating-point cast context and remove it from the top of the floating-point cast context stack.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

const sc_fxcast_switch& **value**() const;

Member function **value** shall return the cast switch.

7.12 Control of string representation

7.12.1 Description

Type **sc_numrep** is used to control the formatting of number representations as character strings when passed as an argument to the **to_string** member function of a data type object.

7.12.2 Class definition

```
namespace sc_dt {

    enum sc_numrep
    {
        SC_NOBASE = 0,
        SC_BIN = 2,
        SC_OCT = 8,
        SC_DEC = 10,
        SC_HEX = 16,
        SC_BIN_US,
        SC_BIN_SM,
        SC_OCT_US,
        SC_OCT_SM,
        SC_HEX_US,
        SC_HEX_SM,
        SC_CSD
    };

    std::string to_string( sc_numrep );

};      // namespace sc_dt
```

7.12.3 Functions

std::string **to_string**(sc_numrep);

Function **to_string** shall return a string consisting of the same sequence of characters as the name of the corresponding constant value of the enumerated type **sc_numrep**.

Example:

```
to_string(SC_HEX) == "SC_HEX"  // is true
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

8. SystemC utilities

8.1 Trace files

8.1.1 Overview

A trace file records a time-ordered sequence of value changes during simulation. The VCD trace file format shall be supported.

A VCD trace file can only be created and opened by calling function **sc_create_vcd_trace_file**. A trace file may be opened during elaboration or at any time during simulation. Values can only be traced by calling function **sc_trace**. A trace file shall be opened before values can be traced to that file, and values shall not be traced to a given trace file if one or more delta cycles have elapsed since opening the file. A VCD trace file shall be closed by calling function **sc_close_vcd_trace_file**. A trace file shall not be closed before the final delta cycle of simulation.

An implementation may support other trace file formats by providing alternatives to the functions **sc_create_vcd_trace_file** and **sc_close_vcd_trace_file**.

The lifetime of a traced object need not extend throughout the entire time the trace file is open.

NOTE—A trace file can be opened at any time, but no mechanism is available to switch off tracing before the end of simulation.

8.1.2 Class definition and function declarations

```
namespace sc_core {

class sc_trace_file
{
public:
    virtual void set_time_unit( double , sc_time_unit ) = 0;
    implementation-defined
};

sc_trace_file* sc_create_vcd_trace_file( const char* name );
void sc_close_vcd_trace_file( sc_trace_file* tf );
void sc_write_comment( sc_trace_file* tf , const std::string& comment );
void sc_trace ...

}          // namespace sc_core
```

8.1.3 sc_trace_file

```
class sc_trace_file
{
    public:
        virtual void set_time_unit( double , sc_time_unit ) = 0;
        implementation-defined
};
```

Class **sc_trace_file** is the abstract base class from which the classes that provide file handles for VCD or other implementation-defined trace file formats are derived. An application shall not construct objects of class **sc_trace_file** but may define pointers and references to this type.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Member function **set_time_unit** shall be overridden in the derived class to set the time unit for the trace file. The value of the **double** argument shall be positive and shall be a power of 10. In the absence of any call function **set_time_unit**, the default trace file time unit shall be 1 picosecond.

8.1.4 sc_create_vcd_trace_file

sc_trace_file* **sc_create_vcd_trace_file**(const char* name);

Function **sc_create_vcd_trace_file** shall create a new file handle object of class **sc_trace_file**, open a new VCD file associated with the file handle, and return a pointer to the file handle. The file name shall be constructed by appending the character string "**.vcd**" to the character string passed as an argument to the function.

8.1.5 sc_close_vcd_trace_file

void **sc_close_vcd_trace_file**(sc_trace_file* tf);

Function **sc_close_vcd_trace_file** shall close the VCD file and delete the file handle pointed to by the argument.

8.1.6 sc_write_comment

void **sc_write_comment**(sc_trace_file* tf , const std::string& comment);

Function **sc_write_comment** shall write the string given as the second argument to the trace file given by the first argument, as a comment, at the simulation time at which the function is called.

8.1.7 sc_trace

void **sc_trace**(sc_trace_file* , const bool& , const std::string&);
void **sc_trace**(sc_trace_file* , const bool* , const std::string&);
void **sc_trace**(sc_trace_file* , const float& , const std::string&);
void **sc_trace**(sc_trace_file* , const float* , const std::string&);
void **sc_trace**(sc_trace_file* , const double& , const std::string&);
void **sc_trace**(sc_trace_file* , const double* , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_logic& , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_logic* , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_int_base& , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_int_base* , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_uint_base& , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_uint_base* , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_signed& , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_signed* , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_unsigned& , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_unsigned* , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_bv_base& , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_bv_base* , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_lv_base& , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_lv_base* , const std::string&);

void **sc_trace**(sc_trace_file* , const sc_dt::sc_fxval& , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_fxval* , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_fxval_fast& , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_fxval_fast* , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_fxnum& , const std::string&);
void **sc_trace**(sc_trace_file* , const sc_dt::sc_fxnum* , const std::string&);

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
void sc_trace( sc_trace_file* , const sc_dt::sc_fxnum_fast& , const std::string& );
void sc_trace( sc_trace_file* , const sc_dt::sc_fxnum_fast* , const std::string& );

void sc_trace( sc_trace_file* , const unsigned char& , const std::string& ,
               int width = 8 * sizeof( unsigned char ) );

void sc_trace( sc_trace_file* , const unsigned char* , const std::string& ,
               int width = 8 * sizeof( unsigned char ) );

void sc_trace( sc_trace_file* , const unsigned short& , const std::string& ,
               int width = 8 * sizeof( unsigned short ) );

void sc_trace( sc_trace_file* , const unsigned short* , const std::string& ,
                int width = 8 * sizeof( unsigned short ) );

void sc_trace( sc_trace_file* , const unsigned int& , const std::string& ,
               int width = 8 * sizeof( unsigned int ) );

void sc_trace( sc_trace_file* , const unsigned int* , const std::string& ,
                int width = 8 * sizeof( unsigned int ) );

void sc_trace( sc_trace_file* , const unsigned long& , const std::string& ,
               int width = 8 * sizeof( unsigned long ) );

void sc_trace( sc_trace_file* , const unsigned long* , const std::string& ,
               int width = 8 * sizeof( unsigned long ) );

void sc_trace( sc_trace_file* , const char& , const std::string& , int width = 8 * sizeof( char ) );

void sc_trace( sc_trace_file* , const char* , const std::string& , int width = 8 * sizeof( char ) );

void sc_trace( sc_trace_file* , const short& , const std::string& , int width = 8 * sizeof( short ) );

void sc_trace( sc_trace_file* , const short* , const std::string& , int width = 8 * sizeof( short ) );

void sc_trace( sc_trace_file* , const int& , const std::string& , int width = 8 * sizeof( int ) );

void sc_trace( sc_trace_file* , const int* , const std::string& , int width = 8 * sizeof( int ) );

void sc_trace( sc_trace_file* , const long& , const std::string& , int width = 8 * sizeof( long ) );

void sc_trace( sc_trace_file* , const long* , const std::string& , int width = 8 * sizeof( long ) );

void sc_trace( sc_trace_file* , const sc_dt::int64& , const std::string& ,
                                                      int width = 8 * sizeof( sc_dt::int64 ) );

void sc_trace( sc_trace_file* , const sc_dt::int64* , const std::string& ,
                                                      int width = 8 * sizeof( sc_dt::int64 ) );

void sc_trace( sc_trace_file* , const sc_dt::uint64& , const std::string& ,
                                                      int width = 8 * sizeof( sc_dt::uint64 ) );

void sc_trace( sc_trace_file* , const sc_dt::uint64* , const std::string& ,
                                                      int width = 8 * sizeof( sc_dt::uint64 ) );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
template <class T>
void sc_trace( sc_trace_file* , const sc_signal_in_if<T>& , const std::string& );

void sc_trace( sc_trace_file* , const sc_signal_in_if<char>& , const std::string& , int width );

void sc_trace( sc_trace_file* , const sc_signal_in_if<short>& , const std::string& , int width );

void sc_trace( sc_trace_file* , const sc_signal_in_if<int>& , const std::string& , int width );

void sc_trace( sc_trace_file* , const sc_signal_in_if<long>& , const std::string& , int width );

void sc_trace( sc_trace_file* , const event& , const std::string& );

void sc_trace( sc_trace_file* , const sc_time& , const std::string&);

void sc_trace( sc_trace_file* , const event* , const std::string& );

void sc_trace( sc_trace_file* , const sc_time* , const std::string&);
```

Function **sc_trace** shall trace the value passed as the second argument to the trace file passed as the first argument, using the string passed as the third argument to identify the value in the trace file. All changes to the value of the second argument that occur between the time the function is called and the time the trace file is closed shall be recorded in the trace file.

The **sc_time** value shall be expressed in units specified by **sc_trace_file**::**set_time_unit**.

NOTE 1—The function **sc_trace** is also overloaded elsewhere in this standard to support additional data types (see 6.8.4 and 6.10.5).

NOTE 2—It is possible that time values to be traced may incur a loss of precision based upon the value passed to the **sc_trace** function, the value passed to the **sc_trace_file**::**set_time_unit** function, and value passed to the **sc_core**::**sc_set_time_resolution** function. Implementations may issue a warning in such cases.

8.2 sc_report

8.2.1 Description

Class **sc_report** represents an instance of a report as generated by function **sc_report_handler::report**. **sc_report** objects are accessible to the application if the action SC_CACHE_REPORT is set for a given severity level and message type. Also, **sc_report** objects may be caught by the application when thrown by the report handler (see 8.3).

Type **sc_severity** represents the severity level of a report.

8.2.2 Class definition

```
namespace sc_core {

enum sc_severity {
    SC_INFO = 0,
    SC_WARNING,
    SC_ERROR,
    SC_FATAL,
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    SC_MAX_SEVERITY
};

enum sc_verbosity {
    SC_NONE = 0,
    SC_LOW = 100,
    SC_MEDIUM = 200,
    SC_HIGH = 300,
    SC_FULL = 400,
    SC_DEBUG = 500
};

class sc_report
: public std::exception
{
public:

    sc_report( const sc_report& );
    sc_report& operator= ( const sc_report& );
    virtual ~sc_report() throw();

    sc_severity get_severity() const;
    const char* get_msg_type() const;
    const char* get_msg() const;
    int get_verbosity() const;
    const char* get_file_name() const;
    int get_line_number() const;

    const sc_time& get_time() const;
    const char* get_process_name() const;

    virtual const char* what() const throw();
};

}       // namespace sc_core
```

8.2.3 Constraints on usage

Objects of class **sc_report** are generated by calling the function **sc_report_handler::report**. An application shall not directly create a new object of class **sc_report** other than by calling the copy constructor. The individual attributes of an **sc_report** object may only be set by function **sc_report_handler::report**.

An implementation shall throw an object of class **sc_report** from function **default_handler** of class **sc_report_handler** in response to the action SC_THROW. An application may throw an object of class **sc_report** from an application-specific report handler function. An application may catch an **sc_report** in a try-block.

8.2.4 sc_verbosity

The enumeration **sc_verbosity** provides the values of indicative verbosity levels that may be passed as arguments to member function **set_verbosity_level** of class **sc_report_handler** and to member function **report** of class **sc_report_handler**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

8.2.5 sc_severity

There shall be four severity levels. SC_MAX_SEVERITY shall not be a severity level. It shall be an error to pass the value SC_MAX_SEVERITY to a function requiring an argument of type **sc_severity**.

Table 50 describes the intended meanings of the four severity levels. The precise meanings can be overridden by the class **sc_report_handler**.

Table 50—Levels for sc_severity

Severity levels	Description
SC_INFO	An informative message
SC_WARNING	A potential problem
SC_ERROR	An actual problem from which an application may be able to recover
SC_FATAL	An actual problem from which an application cannot recover

8.2.6 Copy constructor and assignment

sc_report(const sc_report&);
sc_report& **operator=** (const sc_report&);

The copy constructor and the assignment operator shall each create a deep copy of the **sc_report** object passed as an argument.

8.2.7 Member functions

Several member functions specified in this subclause return a pointer to a null-terminated character string. The implementation is only obliged to keep the returned string valid during the lifetime of the **sc_report** object.

sc_severity **get_severity**() const;
const char* **get_msg_type**() const;
const char* **get_msg**() const;
int **get_verbosity**() const;
const char* **get_file_name**() const;
int **get_line_number**() const;

Each of these six member functions shall return the corresponding property of the **sc_report** object. The properties themselves can only be set by passing their values as arguments to the function **sc_report_handler::report**. If the value of the severity level is not SC_INFO, the value returned from **get_verbosity** shall be implementation-defined.

const sc_time& **get_time**() const;
const char* **get_process_name**() const;

Each of these two member functions shall return the corresponding property of the **sc_report** object. The properties themselves shall be set by function **sc_report_handler::report** according to the simulation time at which the report was generated and the process instance within which it was generated.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

virtual const char* **what**() const;

Member function **what** shall return a text string composed from the severity level, message type, message, file name, line number, process name, and time of the **sc_report** object. An implementation may vary the content of the text string, depending on the severity level.

Example:

```
try {
    ...
    SC_REPORT_ERROR("msg_type", "msg");
    ...
    } catch ( sc_core::sc_report e ) {
        std::cout << "Caught " << e.what() << std::endl;
    }
```

8.3 sc_report_handler

8.3.1 Description

Class **sc_report_handler** provides features for writing out textual reports on the occurrence of exceptional circumstances and for defining application-specific behavior to be executed when those reports are generated.

Member function **report** is the central feature of the reporting mechanism, and by itself, it is sufficient for the generation of reports using the default actions and default handler. Other member functions of class **sc_report_handler** provide for application-specific report handling. Member function **report** shall be called by an implementation whenever it needs to report an exceptional circumstance. Member function **report** may also be called from SystemC applications created by IP vendors, EDA tool vendors, or end users. The intention is that the behavior of reports embedded in an implementation or in precompiled SystemC code distributed as object code may be modified by end users to calling the member functions of class **sc_report_handler**.

In order to define application-specific actions to be taken when a report is generated, reports are categorized according to their severity level and message type. Care should be taken when choosing the message types passed to function **report** in order to give the end user adequate control over the definition of actions. It is recommended that each message type take the following general form:

"/originating_company_or_institution/product_identifier/subcategory/subcategory..."

It is the responsibility of any party who distributes precompiled SystemC code to ensure that any reports that the end user may need to distinguish for the purpose of setting actions are allocated unique message types.

8.3.2 Class definition

```
namespace sc_core {

typedef unsigned sc_actions;

enum {
    SC_UNSPECIFIED,
    SC_DO_NOTHING,
    SC_THROW,
    SC_LOG,
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    SC_DISPLAY,
    SC_CACHE_REPORT,
    SC_INTERRUPT,
    SC_STOP,
    SC_ABORT,

    // default action constants
    SC_DEFAULT_INFO_ACTIONS = SC_LOG | SC_DISPLAY,
    SC_DEFAULT_WARNING_ACTIONS = SC_LOG | SC_DISPLAY,
    SC_DEFAULT_ERROR_ACTIONS = SC_LOG | SC_CACHE_REPORT | SC_THROW,
    SC_DEFAULT_FATAL_ACTIONS = SC_LOG | SC_DISPLAY | SC_CACHE_REPORT |
SC_ABORT
};


typedef void ( * sc_report_handler_proc ) ( const sc_report& , const sc_actions& );

class sc_report_handler
{
public:
    static void report( sc_severity , const char* msg_type , const char* msg , const char* file , int line );
    static void report( sc_severity , const char* msg_type , const char* msg , int verbosity,
                                                                    const char* file , int line );

    static sc_actions set_actions( sc_severity , sc_actions = SC_UNSPECIFIED );
    static sc_actions set_actions( const char * msg_type , sc_actions = SC_UNSPECIFIED );
    static sc_actions set_actions( const char * msg_type , sc_severity , sc_actions = SC_UNSPECIFIED );

    static int stop_after( sc_severity , int limit = –1 );
    static int stop_after( const char* msg_type , int limit = –1 );
    static int stop_after( const char* msg_type , sc_severity , int limit = –1 );

    static int get_count( sc_severity );
    static int get_count( const char* msg_type );
    static int get_count( const char* msg_type , sc_severity );

    int set_verbosity_level( int );
    int get_verbosity_level();

    static sc_actions suppress( sc_actions );
    static sc_actions suppress();
    static sc_actions force( sc_actions );
    static sc_actions force();

    static void set_handler( sc_report_handler_proc );
    static sc_report_handler_proc get_handler();
    static void default_handler( const sc_report& , const sc_actions& );
    static sc_actions get_new_action_id();

    static sc_report* get_cached_report();
    static void clear_cached_report();

    static bool set_log_file_name( const char* );
    static const char* get_log_file_name();
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
};

#define SC_REPORT_INFO_VERB( msg_type , msg, verbosity ) \
                sc_report_handler::report( SC_INFO , msg_type , msg , verbosity, __FILE__ , __LINE__ )

#define SC_REPORT_INFO( msg_type , msg ) \
                sc_report_handler::report( SC_INFO , msg_type , msg , __FILE__ , __LINE__ )

#define SC_REPORT_WARNING( msg_type , msg ) \
                sc_report_handler::report( SC_WARNING , msg_type , msg , __FILE__ , __LINE__ )

#define SC_REPORT_ERROR( msg_type , msg ) \
                sc_report_handler::report( SC_ERROR , msg_type , msg , __FILE__ , __LINE__ )

#define SC_REPORT_FATAL( msg_type , msg ) \
                sc_report_handler::report( SC_FATAL , msg_type , msg , __FILE__ , __LINE__ )

#define sc_assert( expr ) \
                ( ( void ) ( ( expr ) ? 0 : ( ::sc_core::sc_assertion_failed(#expr,__FILE__,__LINE__ ) , 0 ) ) )
    [[noreturn]] void sc_assertion_failed(const char* msg, const char* file, int line);

void sc_interrupt_here( const char* msg_type , sc_severity );
void sc_stop_here( const char* msg_type , sc_severity );

}       // namespace sc_core
```

The actual integer values for **sc_actions enum values** are implementation-defined and shall be chosen to make possible bitwise combinations of the various values.

8.3.3 Constraints on usage

The member functions of class **sc_report_handler** can be called at any time during elaboration or simulation. Actions can be set for a severity level or a message type both before and after the first use of that severity level or message type as an argument to member function **report**.

8.3.4 sc_actions

The typedef **sc_actions** represents a word where each bit in the word represents a distinct action. More than one bit may be set, in which case all of the corresponding actions shall be executed. The enumeration defines the set of actions recognized and performed by the default handler. An application-specific report handler set by calling function **set_handler** may modify or extend this set of actions.

The value SC_UNSPECIFIED is not an action as such but serves as the default value for a variable or argument of type **sc_actions**, meaning that no action has been set. In contrast, the value SC_DO_NOTHING is a specific action and shall inhibit any actions set with a lower precedence according to the rules given in 8.3.6.

Each severity level is associated with a set of default actions chosen to be appropriate for the given name, but those defaults can be overridden by calling member function **set_actions**. The default actions shall be defined by the macros SC_DEFAULT_INFO_ACTIONS, SC_DEFAULT_WARNING_ACTIONS, SC_DEFAULT_ERROR_ACTIONS, and SC_DEFAULT_FATAL_ACTIONS.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

8.3.5 report

static void **report**(sc_severity , const char* msg_type , const char* msg , const char* file , int line);
static void **report**(sc_severity , const char* msg_type , const char* msg , int verbosity, const char* file , int line);

Member function **report** shall generate a report and cause the appropriate actions to be taken as defined below.

Member function **report** shall use the severity passed as the first argument and the message type passed as the second argument to determine the set of actions to be executed as a result of previous calls to functions **set_actions**, **stop_after**, **suppress**, and **force**. Member function **report** shall create an object of class **sc_report** initialized using all five argument values and shall pass this object to the handler set by the member function **set_handler**. The object of class **sc_report** shall not persist beyond the call to member function **report** unless the action SC_CACHE_REPORT is set, in which case the object can be retrieved by calling function **get_cached_reports**. An implementation shall maintain a separate cache of **sc_report** objects for each process instance and a single global report cache for calls to function **report** from outside any process. Each such cache shall store only the most recent report.

Member function **report** shall be responsible for determining the set of actions to be executed. The handler function set by function **set_handler** shall be responsible for executing those actions.

Member function **report** shall maintain counts of the number of reports generated as described in 8.3.7. These counts shall be incremented regardless of whether actions are executed or suppressed, except where reports are ignored due to their verbosity level, in which case the counts shall not be incremented.

If the verbosity argument is present, and the value of the severity argument is SC_INFO, and the value of the verbosity argument is greater than the maximum verbosity level, member function **report** shall return without executing any actions and without incrementing any counts. If the verbosity argument is absent and the value of the severity argument is SC_INFO, member function **report** shall behave as if the verbosity argument were present and had a value of SC_MEDIUM. If the severity argument has a value other than SC_INFO, the implementation shall ignore the verbosity argument.

The macros SC_REPORT_INFO_VERB, SC_REPORT_INFO, SC_REPORT_WARNING, SC_REPORT_ERROR, SC_REPORT_FATAL, and **sc_assert** are provided for convenience when calling member function **report**, but there is no obligation on an application to use these macros. The function **sc_assertion_failed** called by **sc_assert** is implementation-defined but is required to have a [[noreturn]] C++ attribute and to cause a FATAL error to be reported.

NOTE—Class **sc_report** may provide a constructor for the exclusive use of class **sc_report_handler** in initializing these properties.

8.3.6 set_actions

static sc_actions **set_actions**(sc_severity , sc_actions = SC_UNSPECIFIED);
static sc_actions **set_actions**(const char * msg_type , sc_actions = SC_UNSPECIFIED);
static sc_actions **set_actions**(const char * msg_type , sc_severity , sc_actions = SC_UNSPECIFIED);

Member function **set_actions** shall set the actions to be taken by member function **report** when **report** is called with the given severity level, message type, or both. In determining which set of actions to take, the message type shall take precedence over the severity level, and the message type and severity level combined shall take precedence over the message type and severity level

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

considered individually. In other words, the three member functions **set_actions** just listed appear in order of increasing precedence. The actions of any lower precedence match shall be inhibited.

Each call to **set_actions** shall replace the actions set by the previous call for the given severity, message type, or severity-message type pair. The value returned from the member function **set_actions** shall be the actions set by the previous call to that very same overloading of the function **set_actions** for the given severity level, message type, or severity-message type pair. The first call to function **set_actions(sc_severity , sc_actions)** shall return the default actions associated with the given severity level. The first call to one of the remaining two functions for a given message type shall return the value SC_UNSPECIFIED. Each of the three overloaded functions operates independently in this respect. Precedence is only relevant when **report** is called.

Example:

```
sc_core::sc_report_handler::set_actions(sc_core::SC_WARNING, sc_core::SC_DO_NOTHING);
sc_core::sc_report_handler::set_actions("/Acme_IP", sc_core::SC_DISPLAY);
sc_core::sc_report_handler::set_actions("/Acme_IP", sc_core::SC_INFO, sc_core::SC_DISPLAY |
sc_core::SC_CACHE_REPORT);
...
SC_REPORT_WARNING("", "1");                 // Silence
SC_REPORT_WARNING("/Acme_IP", "2");         // Written to standard output
SC_REPORT_INFO("/Acme_IP", "3");            // Written to standard output and cached
```

8.3.7 stop_after

```
static int stop_after( sc_severity , int limit = –1 );
static int stop_after( const char* msg_type , int limit = –1 );
static int stop_after( const char* msg_type , sc_severity , int limit = –1 );
```

Member function **report** shall maintain independent counts of the number of reports generated for each severity level, each message type, and each severity-message type pair. Member function **stop_after** shall set a limit on the number of reports that will be generated in each case. Member function **report** shall call the function **sc_stop** when exactly the number of reports given by argument **limit** to function **stop_after** have been generated for the given severity level, message type, or severity-message type pair.

In determining when to call function **sc_stop**, the message type shall take precedence over the severity level, and the message type and severity level combined shall take precedence over the message type and severity level considered individually. In other words, the three member functions **stop_after** just listed appear in order of increasing precedence. If function **report** is called with a combination of severity level and message type that matches more than one limit set by calling **stop_after**, only the higher precedence limit shall have any effect.

The appropriate counts shall be initialized to the value 1 the first time function **report** is called with a particular severity level, message type, or severity-message type pair and shall not be modified or reset when function **stop_after** is called. All three counts shall be incremented for each call to function **report** whether or not any actions are executed. When a count for a particular severity-message type pair is incremented, the counts for the given severity level and the given message type shall be incremented also. If the limit being set has already been reached or exceeded by the count at the time **stop_after** is called, **sc_stop** shall not be called immediately but shall be called the next time the given count is incremented.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The default limit is –1, which means that no stop limit is set. Calling function **stop_after** with a limit of –1 for a particular severity level, message type, or severity-message type pair shall remove the stop limit for that particular case.

A limit of 0 shall mean that there is no stop limit for the given severity level, message type, or severity-message type pair, and, moreover an explicit limit of 0 shall override the behavior of any lower precedence case. However, even with an explicit limit of 0, the actions set for the given case (by calling function **sc_action** or the default actions) may nonetheless result in function **sc_stop** or **abort** being called or an exception thrown.

If function **report** is called with a severity level of SC_FATAL, the default behavior in the absence of any calls to either function **set_actions** or function **stop_after** is to execute a set of actions, including SC_ABORT.

The value returned from the member function **stop_after** shall be the limit set by the previous call to that very same overloading of the function **stop_after** for the given severity level, message type, or severity-message type pair. Otherwise, the value returned is the default limit of –1.

Example 1:

```
sc_core::sc_report_handler::stop_after(sc_core::SC_WARNING, 1);
sc_core::sc_report_handler::stop_after("/Acme_IP", 2);
sc_core::sc_report_handler::stop_after("/Acme_IP", sc_core::SC_WARNING, 3);
...
SC_REPORT_WARNING("/Acme_IP", "Overflow");
SC_REPORT_WARNING("/Acme_IP", "Conflict");
SC_REPORT_WARNING("/Acme_IP", "Misuse");                    // sc_core::sc_stop() called
```

Example 2:

```
sc_core::sc_report_handler::stop_after(sc_core::SC_WARNING, 5);
sc_core::sc_report_handler::stop_after("/Acme_IP", sc_core::SC_WARNING, 1);
...
SC_REPORT_WARNING("/Star_IP", "Unexpected");
SC_REPORT_INFO("/Acme_IP", "Invoked");
SC_REPORT_WARNING("/Acme_IP", "Mistimed");                  // sc_core::sc_stop() called
```

8.3.8 get_count

static int **get_count**(sc_severity);
static int **get_count**(const char* msg_type);
static int **get_count**(const char* msg_type , sc_severity);

Member function **get_count** shall return the count of the number of reports generated for each severity level, each message type, and each severity-message type pair maintained by member function **report**. If member function **report** has not been called for the given severity level, message type, or severity-message type pair, member function **get_count** shall return the value zero.

8.3.9 Verbosity level

The maximum verbosity level is a single global quantity that shall only apply to reports with a severity level of SC_INFO. Any individual reports having a severity level of SC_INFO and a verbosity level greater than the maximum verbosity level shall be ignored; that is, the implementation shall in effect suppress all the actions associated with that report.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

int **set_verbosity_level**(int);

Member function **set_verbosity_level** shall set the maximum verbosity level to the value passed as an argument and shall return the previous value of the maximum verbosity level as the value of the function.

int **get_verbosity_level**();

Member function **get_verbosity_level** shall return the value of the maximum verbosity level.

8.3.10 suppress and force

static sc_actions **suppress**(sc_actions);
static sc_actions **suppress**();

Member function **suppress** shall suppress the execution of a given set of actions for subsequent calls to function **report**. The actions to be suppressed are passed as an argument to function **suppress**. The return value from function **suppress** shall be the set of actions that were suppressed immediately prior to the call to function **suppress**. The actions passed as an argument shall replace entirely the previously suppressed actions, there being only a single, global set of suppressed actions. By default, there are no suppressed actions. If the argument list is empty, the set of suppressed actions shall be cleared, restoring the default behavior.

The suppression of certain actions shall not hinder the execution of any other actions that are not suppressed.

static sc_actions **force**(sc_actions);
static sc_actions **force**();

Member function **force** shall force the execution of a given set of actions for subsequent calls to function **report**. The actions to be forced are passed as an argument to function **force**. The return value from function **force** shall be the set of actions that were forced immediately prior to the call to function **force**. The actions passed as an argument shall replace entirely the previously forced actions, there being only a single, global set of forced actions. By default, there are no forced actions. If the argument list is empty, the set of forced actions shall be cleared, restoring the default behavior.

Forced actions shall be executed in addition to the default actions for the given severity level and in addition to any actions set by calling function **set_actions**.

If the same action is both suppressed and forced, the force shall take precedence.

8.3.11 set_handler

typedef void (* **sc_report_handler_proc**) (const sc_report& , const sc_actions&);
static void **set_handler**(sc_report_handler_proc);

Member function **set_handler** shall set the handler function to be called from function **report**. This allows an application-specific report handler to be provided.

static sc_report_handler_proc **get_handler**();

Member function **get_handler** shall return the handler function.

static void **default_handler**(const sc_report& , const sc_actions&);

Member function **default_handler** shall be the default handler; that is, member function **default_handler** shall be called from function **report** in the absence of any call to function **set_handler**. Member function **default_handler** shall perform zero, one, or more than one of the

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

actions set out in Table 51, as determined by the value of its second argument. In this table, the *composite message* shall be a text string composed from the severity level, message type, message, file name, line number, process name, and time of the **sc_report** object. An implementation may vary the content of the *composite message,* depending on the severity level.

NOTE—To restore the default handler, call set_handler(&sc_report_handler::default_handler).

Table 51—Actions by default_handler

Severity levels	Description
SC_UNSPECIFIED	No action (but function **report** will execute any lower precedence actions).
SC_DO_NOTHING	No action (but causes function **report** to inhibit lower precedence actions).
SC_THROW	Throw the **sc_report** object.
SC_LOG	Write the composite message to the log file as set by function **set_log_file_name**.
SC_DISPLAY	Write the composite message to standard output.
SC_CACHE_REPORT	No action (but causes function **report** to cache the report).
SC_INTERRUPT	Call function **sc_interrupt_here**, passing the message type and severity level of the **sc_report** object as arguments.
SC_STOP	Call function **sc_stop_here**, passing the message type and severity level of the **sc_report** object as arguments, then call function **sc_stop**.
SC_ABORT	Call **abort()**.

8.3.12 get_new_action_id

static sc_actions **get_new_action_id**();

Member function **get_new_action_id** shall return a value of type **sc_actions** that represents an unused action. The returned value shall be a word with exactly one bit set. The intention is that such a value can be used to extend the set of actions when writing an application-specific report handler. If there are no more unique values available, the function shall return the value SC_UNSPECIFIED. An application shall not call function **get_new_action_id** before the start of elaboration.

8.3.13 sc_interrupt_here and sc_stop_here

void **sc_interrupt_here**(const char* msg_type , sc_severity);
void **sc_stop_here**(const char* msg_type , sc_severity);

Functions **sc_interrupt_here** and **sc_stop_here** shall be called from member function **default_handler** in response to action types SC_INTERRUPT and SC_STOP, respectively. These two functions may also be called from application-specific report handlers. The intention is that these two functions serve as a debugging aid by allowing a user to set a breakpoint on or within either function. To this end, an implementation may choose to implement each of these functions with a switch statement dependent on the severity parameter such that a user can set a breakpoint dependent on the severity level of the report.

8.3.14 get_cached_report and clear_cached_report

static sc_report* **get_cached_report**();

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Member function **get_cached_report** shall return a pointer to the most recently cached report for the current process instance if called from a process or the global cache otherwise. Previous reports shall not be accessible.

static void **clear_cached_report**();

Member function **clear_cached_report** shall empty the report cache for the current process instance if called from a process or the global cache otherwise. A subsequent call to **get_cached_report** would return a null pointer until such a time as a further report was cached in the given cache.

8.3.15 set_log_file_name and get_log_file_name

static bool **set_log_file_name**(const char*);
static const char* **get_log_file_name**();

Member function **set_log_file_name** shall set the log file name. Member function **get_log_file_name** shall return the log file name as set by **set_log_file_name**. The default value for the log file name is a pointer to an empty string. If function **set_log_file_name** is called with a pointer to a non-empty string and there is no existing log file name, the log file name shall be set by duplicating the string passed as an argument and the function shall return **true**. If called with a pointer to a non-empty string and there is already a log file name, function **set_log_file_name** shall not modify the existing name and shall return **false**. If called with a pointer to an empty string, any existing log file name shall be deleted and the function shall return **false**.

Opening, writing, and closing the log file shall be the responsibility of the report handler. Member function **default_handler** shall call function **get_log_file_name** in response to the action SC_LOG. Function **get_log_file_name** may also be called from an application-specific report handler.

Example:

```
sc_core::sc_report_handler::set_log_file_name("foo");      // Returns true
sc_core::sc_report_handler::get_log_file_name();           // Returns "foo"
sc_core::sc_report_handler::set_log_file_name("bar");      // Returns false
sc_core::sc_report_handler::get_log_file_name();           // Returns "foo"
sc_core::sc_report_handler::set_log_file_name(0);          // Returns false
sc_core::sc_report_handler::get_log_file_name();           // Returns 0
```

8.4 sc_exception

8.4.1 Description

Class **sc_report**, which represents a report generated by the SystemC report handler, is derived from class **std::exception**. The typedef **sc_exception** exists to provide a degree of backward compatibility with earlier versions of the SystemC class library (see 8.2).

8.4.2 Class definition

```
namespace sc_core {

    typedef std::exception sc_exception;
}
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

8.5 sc_vector

8.5.1 Description

Class **sc_vector** is used to construct vectors of modules, channels, ports, exports, or objects of any other type that is derived from **sc_object**. As such it provides a convenient way of describing repetitive or parameterized structures. Class **sc_vector** provides member functions for picking out elements from a vector and for port binding, including member functions to bind a vector-of-ports to a vector-of-objects.

Function **sc_assemble_vector** (together with its associated proxy class **sc_vector_assembly**) provides a mechanism for assembling a vector of objects from a set of individual objects distributed across a vector of modules. Such a vector assembly can be used in place of a vector in many contexts, such as when binding ports.

Class **sc_vector** provides a mechanism for passing through user-defined module constructor arguments when creating a vector of modules.

Throughout the current clause only, the term *vector* refers to an object of type **sc_vector<T>** for some appropriate choice of T.

8.5.2 Class definition

```
namespace sc_core {

enum sc_vector_init_policy
{
    SC_VECTOR_LOCK_AFTER_INIT,
    SC_VECTOR_LOCK_AFTER_ELABORATION
};

class sc_vector_base : public sc_object
{
    public:
        typedef implementation-defined size_type;

        virtual const char* kind() const;
        size_type size() const;
        const std::vector<sc_object*>& get_elements() const;
};

template< typename T >
class sc_vector_iter† : public std::iterator< std::random_access_iterator_tag, T >
{
    // Conforms to Random Access Iterator category.
    // See ISO/IEC 14882:2017 [lib.iterator.requirements]
    implementation-defined
};

template< typename T >
class sc_vector : public sc_vector_base
{
    public:
        using sc_vector_base::size_type;
        typedef sc_vector_iter†<T> iterator;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
typedef sc_vector_iter† <const T> const_iterator;

sc_vector();
explicit sc_vector( const char* prefix );
sc_vector( const char* prefix, size_type n,
           sc_vector_init_policy init_pol = SC_VECTOR_LOCK_AFTER_INIT );
template< typename Creator >
sc_vector( const char* prefix, size_type n,
           Creator creator, sc_vector_init_policy init_pol = SC_VECTOR_LOCK_AFTER_INIT );
virtual ~sc_vector();

void init( size_type n,
          sc_vector_init_policy init_pol = SC_VECTOR_LOCK_AFTER_INIT );
static T* create_element( const char* prefix, size_type index );

template< typename Creator >
void init( size_type n,
          Creator creator, sc_vector_init_policy init_pol = SC_VECTOR_LOCK_AFTER_INIT );

template< typename... Args >
void emplace_back( Args&&... args );

template< typename... Args >
void emplace_back_with_name( Args &&... args );

T& operator[] ( size_type );
const T& operator[] ( size_type ) const;

T& at( size_type );
const T& at( size_type ) const;

iterator begin();
iterator end();

const_iterator begin() const;
const_iterator end() const;

const_iterator cbegin() const;
const_iterator cend() const;

template< typename ContainerType, typename ArgumentType >
iterator bind( sc_vector_assembly<ContainerType,ArgumentType> );

template< typename BindableContainer >
iterator bind( BindableContainer& );

template< typename BindableIterator >
iterator bind( BindableIterator , BindableIterator );

template< typename BindableIterator >
iterator bind( BindableIterator , BindableIterator , iterator );

template< typename ContainerType, typename ArgumentType >
iterator operator() ( sc_vector_assembly<ContainerType,ArgumentType> c );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        template< typename ArgumentContainer >
        iterator operator() ( ArgumentContainer& );

        template< typename ArgumentIterator >
        iterator operator() ( ArgumentIterator , ArgumentIterator );

        template< typename ArgumentIterator >
        iterator operator() ( ArgumentIterator , ArgumentIterator , iterator );

    private:
        // Disabled
        sc_vector( const sc_vector& );
        sc_vector& operator= ( const sc_vector& );
};

template< typename T, typename MT >
class sc_vector_assembly
{
    public:
        typedef implementation-defined size_type;
        typedef implementation-defined iterator;
        typedef implementation-defined const_iterator;
        typedef MT (T::*member_type);

         sc_vector_assembly( const sc_vector_assembly& );

        iterator begin();
        iterator end();

        const_iterator begin() const;
        const_iterator end() const;

        const_iterator cbegin() const;
        const_iterator cend() const;

        size_type size() const;
        std::vector< sc_object* > get_elements() const;

        iterator::reference operator[] ( size_type );
        const_iterator::reference operator[] ( size_type ) const;

        iterator::reference at( size_type );
        const_iterator::reference at( size_type ) const;

        template< typename ContainerType, typename ArgumentType >
        iterator bind( sc_vector_assembly<ContainerType, ArgumentType> );

        template< typename BindableContainer >
        iterator bind( BindableContainer& );

        template< typename BindableIterator >
        iterator bind( BindableIterator , BindableIterator );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        template< typename BindableIterator >
        iterator bind( BindableIterator , BindableIterator , iterator );

        template< typename BindableIterator >
        iterator bind( BindableIterator , BindableIterator , sc_vector<T>::iterator );

        template< typename ContainerType, typename ArgumentType >
        iterator operator() ( sc_vector_assembly<ContainerType, ArgumentType> );

        template< typename ArgumentContainer >
        iterator operator() ( ArgumentContainer& );

        template< typename ArgumentIterator >
        iterator operator() ( ArgumentIterator , ArgumentIterator );

        template< typename ArgumentIterator >
        iterator operator() ( ArgumentIterator , ArgumentIterator , iterator );

        template< typename ArgumentIterator >
        iterator operator()( ArgumentIterator , ArgumentIterator , sc_vector<T>::iterator );

    private:
        // Disabled
        sc_vector_assembly& operator=( const sc_vector_assembly& );
};

template< typename T, typename MT >
sc_vector_assembly<T,MT> sc_assemble_vector( sc_vector<T> & , MT (T::*member_ptr ) );

} // namespace sc_core
```

8.5.3 Constraints on usage

An application shall only instantiate the template **sc_vector<T>** with a template argument T that is a type derived from **sc_object**. Except where used with a custom *creator*, type T shall provide a constructor with a single argument of a type that is convertible from **const char***. If an application needs to pass additional arguments to the element constructor, it can use a *creator* function or function object (see 8.5.5).

The constraints on when an object of type **sc_vector<T>** may be constructed are dependent on the choice of the template argument T. Subject only to any such constraints that apply to type T itself, vectors may be constructed dynamically during simulation. For example, a vector-of-modules can only be constructed during elaboration.

The size of a vector can only be set once, either when the vector is constructed or using a call to member function **init**. Vectors cannot be resized dynamically.

8.5.4 Constructors and destructors

```
sc_vector();
explicit sc_vector( const char* );
sc_vector( const char* , size_type n,
           sc_vector_init_policy init_pol = SC_VECTOR_LOCK_AFTER_INIT );
template< typename Creator >
sc_vector( const char* , size_type n,
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Creator creator, sc_vector_init_policy init_pol = SC_VECTOR_LOCK_AFTER_INIT);

The above constructors shall pass the character string argument (if there is one) through to the constructor belonging to the base class **sc_object** in order to set the string name of the vector instance in the module hierarchy.

The default constructor shall call function **sc_gen_unique_name("vector")** in order to generate a unique string name that it shall then pass through to the constructor for the base class **sc_object**.

If the second argument is present, the constructor shall call the member function **init**, passing through the value of this second argument as the first argument to **init**. Otherwise the constructor shall construct an empty vector that may subsequently be initialized by calling member function **init** explicitly.

If the second and third arguments are present, the constructor shall call the member function **init**, passing through the values of the second and third arguments as the first and second arguments to **init**.

Example:

```
SC_MODULE(my_module) {
    sc_core::sc_vector<sc_core::sc_port<i_f> > ports;
    sc_core::sc_vector<sc_core::sc_signal<bool> > signals;
    ...

    SC_CTOR(my_module)
    : ports ("ports", 4)              // Vector-of-ports with 4 elements
    , signals("signals") {            // Uninitialized vector-of-signals
        signals.init(8);              // Initialize the vector with 8 elements, each one a signal
        ...
    }
};
```

virtual ~**sc_vector**();

The destructor shall delete every element of the vector.

8.5.5 init and create_element

void **init**(size_type n, sc_vector_init_policy init_pol = SC_VECTOR_LOCK_AFTER_INIT);
static T* **create_element**(const char* prefix, size_type index);

Static member function **create_element** shall allocate and return a pointer to a new object of type T with the string name given by its first argument. The newly created object shall have the same parent as the vector; that is, the elements of a vector shall be siblings of the vector object itself in the SystemC object hierarchy. The reason for the existence of this function is that an application may provide an alternative function in order to pass user-defined constructor arguments to each element of the vector (see below).

Member function **init** shall allocate the number of objects of type T given by the value of its first argument and shall populate the vector with those objects. Each object shall be allocated by calling the function **create_element**, where the first argument shall be set to the string name of the element and the second argument shall be set to the number of the element in the vector, counting up from 0. The string name of each element shall be determined by calling the function **sc_gen_unique_name(this->basename())**. The value of the second argument of the **init** function determines when the vector is locked. SC_VECTOR_LOCK_AFTER_INIT locks the vector at

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

construction time (default value), while SC_VECTOR_LOCK_AFTER_ELABORATION locks the vector *after elaboration*.

Calling member function **init** with an argument value of 0 shall be permitted and shall have no effect.

It shall be an error to call member function **init** more than once with an argument value greater than 0 for any given vector. Note that since a constructor with a second argument calls **init**, it shall be an error to both construct a vector with a size and to call **init**, assuming the size is nonzero in both cases.

template< typename Creator >
void **init**(size_type n, Creator creator, sc_vector_init_policy init_pol =
SC_VECTOR_LOCK_AFTER_INIT);

Instead of calling **create_element**, member function template **init** shall use the value of the second argument, which may be a function or a function object, to allocate each element of the vector. The value passed as the second argument c must be such that

T* placeholder1 = c((const char*)placeholder2, (size_type)placeholder3);

is a well-formed statement. In other words, the actual argument must be callable in place of **create_element**. This allows the creator to be used to pass additional constructor arguments to each element of the vector, rather than passing the string name only.

The expressions **V.init(N, sc_vector<T>::create_element)** and **V.init(N)** shall be equivalent for all vectors V. The value of the third argument determines when the vector is locked. SC_VECTOR_LOCK_AFTER_INIT locks the vector at construction time (default value), while SC_VECTOR_LOCK_AFTER_ELABORATION locks the vector *after elaboration*.

Example:

```
struct my_module : sc_core::sc_module {
    my_module(sc_core::sc_module_name n, std::string extra_arg);
    ...
};

struct Top : sc_core::sc_module {
    sc_core::sc_vector<my_module> vector1; // Vector-of-modules
    sc_core::sc_vector<my_module> vector2;

    // Case 1: creator is a function object
    struct my_module_creator {
        my_module_creator(std::string arg) : extra_arg(arg) {}

        my_module *operator()(const char *name, size_t) {
            return new my_module(name, extra_arg);
            }
            std::string extra_arg;
        };

        // Case 2: creator is a member function
        my_module *my_module_creator_func(const char *name, size_t i) {
            return new my_module(name, "value_of_extra_arg");
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    }

    Top(sc_core::sc_module_name _name, int N) {
      // Initialize vector passing through constructor arguments to my_module
      // Case 1: construct and pass in a function object
      vector1.init(N, my_module_creator("value_of_extra_arg"));

      // Case 2: pass in a member function using Boost bind
      vector2.init(N,
                   sc_bind(&Top::my_module_creator_func, this, sc_unnamed::_1, sc_unnamed::_2));
  }
};
```

8.5.6 Incremental additions to sc_vector during elaboration phase

void **emplace_back**(Args&&... args);

The member function **emplace_back** appends an element to the vector if not locked, calling the element constructor with an autogenerated name as first argument. The arguments args... are forwarded to the constructor.

void **emplace_back_with_name**(Args &&... args);

The member function **emplace_back_with_name** appends an element to the vector if not locked, calling the element constructor with a user-defined name as first argument. The arguments args... are forwarded to the constructor.

8.5.7 kind, size, get_elements

virtual const char* **kind**() const;

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

size_type **size**() const;

Member function **size** shall return the number of elements in the vector. In the case of an uninitialized vector, the size shall be 0. Once set to a nonzero value, the size cannot be modified.

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

Member function **get_elements** of class **sc_vector** shall return a const reference to a **std::vector** that contains pointers to the elements of the **sc_vector**, one pointer per element, in the same order. The **std::vector** shall have the same size as the **sc_vector**. The reference shall be valid for the lifetime of the **sc_vector** object.

8.5.8 operator[] and at

T& **operator[]** (size_type);
const T& **operator[]** (size_type) const;

T& **at**(size_type);
const T& **at**(size_type) const;

operator[] and member functions **at** shall each return a reference or const-qualified reference to the object stored at the index position within the vector given by the value of their one-and-only argument. The reference shall be valid for the lifetime of the vector. If the value of the argument is greater than the size of the vector, the behavior of **operator[]** is undefined, whereas member function **at** shall detect and report the error.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The value of the relation **&V[i] + j == &V[i + j]** is undefined for all vectors v and for all indices i and j.

8.5.9 Iterators

iterator **begin**();
iterator **end**();

const_iterator **begin**() const;
const_iterator **end**() const;

const_iterator **cbegin**() const;
const_iterator **cend**() const;

Class **sc_vector** shall provide an iterator interface that fulfils the Random Access Iterator requirements as defined in ISO/IEC 14882:2017 [lib.iterator.requirements].

begin shall return an iterator that refers to the first element of the vector. **end** shall return an iterator that refers to an imaginary element following the last element of the vector. If the vector is empty, then **begin() == end()**.

Type **iterator** shall be implicitly convertible to type **const_iterator**, where the conversion shall preserve the identity of the element being referred to by the iterator.

Type **sc_vector_assembly<T,MT>::iterator** shall be implicitly convertible both to type **sc_vector<T>::iterator** and to type **sc_vector<T>::const_iterator**, where the conversion shall preserve the identity of the element being referred to by the iterator.

Type **sc_vector_assembly<T,MT>::const_iterator** shall be implicitly convertible to type **sc_vector<T>::const_iterator**, where the conversion shall preserve the identity of the element being referred to by the iterator.

8.5.10 bind

template< typename ContainerType, typename ArgumentType >
iterator **bind**(sc_vector_assembly<ContainerType,ArgumentType>);

template< typename BindableContainer >
iterator **bind**(BindableContainer&);

template< typename BindableIterator >
iterator **bind**(BindableIterator , BindableIterator);

template< typename BindableIterator >
iterator **bind**(BindableIterator , BindableIterator , iterator);

template< typename ContainerType, typename ArgumentType >
iterator **operator()** (sc_vector_assembly<ContainerType,ArgumentType> c);

template< typename ArgumentContainer >
iterator **operator()** (ArgumentContainer&);

template< typename ArgumentIterator >
iterator **operator()** (ArgumentIterator , ArgumentIterator);

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
template< typename ArgumentIterator >
iterator operator() ( ArgumentIterator , ArgumentIterator , iterator );
```

Each member function **bind** and **operator()** shall perform element-by-element binding of the elements of the current vector ***this** to the elements of the vector determined by the arguments to the function call. In each case, the implementation shall bind the elements by calling function **bind** or **operator()**, respectively, of each individual element of the current vector.

These functions exist in multiple forms that support partial binding of either vector as follows:

The one- and two-argument forms shall start binding from the first element of the current object. The three-argument forms shall start binding from the element referred to by the iterator passed as the third argument. If the third argument does not refer to an element of the current object, the behavior shall be undefined.

The one-argument forms shall start binding to the first element of the container passed as an argument, which may be a vector or a vector assembly, and may bind to every element of that container. The two- and three-argument forms shall start binding to the element referred to by the iterator passed as the first argument and shall not bind any element including or following that referred to by the iterator passed as the second argument.

In each case, member function **bind** and **operator()** shall return an iterator that refers to the first unbound element within the current vector ***this** after the binding has been performed.

A given vector may be bound multiple times subject only to the binding policy of the individual elements. In other words, it is possible to make multiple calls to **bind**, each call binding different elements of a vector.

Example:

```
typedef sc_core::sc_vector<sc_core::sc_inout<int> > port_type;
typedef sc_core::sc_vector<sc_core::sc_signal<int> > signal_type;

struct M : sc_core::sc_module {
    port_type ports;                                    // Vector-of-ports
    M(sc_core::sc_module_name _name, int N)
    : ports("ports", N) {
    ...
    }
};

struct Top : sc_core::sc_module {
    signal_type sigs;                                   // Vector-of-signals
    signal_type hi_sigs;                                // Vector-of-signals
    signal_type lo_sigs;                                // Vector-of-signals
    M *m1, *m2;

    Top(sc_core::sc_module_name _name)
    : sigs("sigs", 4), hi_sigs("hi_sigs", 2), lo_sigs("lo_sigs", 2) {
        m1 = new M("m1", 4);
        m2 = new M("m2", 4);

        port_type::iterator it;

        // Bind all 4 elements of ports vector to all 4 elements of sigs vector
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        it = m1->ports.bind(sigs);
        sc_assert((it - m1->ports.begin()) == 4);

        // Bind first 2 elements of ports vector to 2 elements of hi_sigs vector
        it = m2->ports.bind(hi_sigs.begin(), hi_sigs.end());
        sc_assert((it - m2->ports.begin()) == 2);

        // Explicit loop equivalent to the above vector bind
        // port_type::iterator from;
        // signal_type::iterator to;
        //
        // for ( from = m2->ports.begin(), to = hi_sigs.begin();
        // (from != m2->ports.end()) && (to != hi_sigs.end());
        // from++, to++ )
        // (*from).bind( *to );

        // Bind last 2 elements of ports vector to 2 elements of lo_sigs vector
        it = m2->ports.bind(lo_sigs.begin(), lo_sigs.end(), it);
        sc_assert((it - m2->ports.begin()) == 4);
    }
    ...
};
```

8.5.11 sc_assemble_vector

```
template< typename T, typename MT >
sc_vector_assembly<T,MT> sc_assemble_vector( sc_vector<T> & , MT (T::*member_ptr ) );
```

Function **sc_assemble_vector** shall return an object of class **sc_vector_assembly**, which shall serve as a proxy for **sc_vector** by providing the member functions **begin**, **end**, **cbegin**, **cend**, **size**, **get_elements**, **operator[]**, **at**, **bind**, and **operator()**. Each of these member functions shall have the same behavior as the corresponding member functions of class **sc_vector** for the vector represented by this proxy.

The first argument passed to function **sc_assemble_vector** by the application shall be an object of type **sc_vector<T>**, where type T is a type derived from class **sc_module**. In other words, the first argument shall be a vector-of-modules.

The second argument passed to function **sc_assemble_vector** by the application shall be the address of a member sub-object of the user-defined module class that forms the type of the elements of the vector passed as the first argument. In other words, the second argument shall be a member of the module from the first argument.

The vector represented by the object of the proxy class **sc_vector_assembly** shall contain elements consisting of references to the member sub-objects (as specified by the second argument) of every element of the vector-of-modules (as specified by the first argument).

sc_assemble_vector may be used to create a proxy for a vector of any object type derived from the class **sc_object**.

A call to any member function of the object of class **sc_vector_assembly** shall act on the elements of the vector represented by the proxy; that is, the member sub-objects identified by the second argument that are actually distributed across the members of the vector identified by the first

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

argument. These sub-objects shall appear to be members of a single vector with respect to the behavior of each of these member functions. As a consequence, the following relations shall hold:

```
sc_vector_assembly<T, MT> assembly = sc_assemble_vector(vector, &module_type::member);

sc_assert( &*(assembly.begin()) == &(*vector.begin()).member );
sc_assert( &*(assembly.end())   == &(*vector.end()).member );
sc_assert( assembly.size()      ==    vector.size() );

for (unsigned int i = 0; i < assembly.size(); i++)
{
    sc_assert( &assembly[i]     == &vector[i].member );
    sc_assert( &assembly.at(i)  == &vector[i].member );
}
```

Member function **get_elements** of class **sc_vector_assembly** shall return an object of type **std::vector<sc_object*>** by value. Each element of this std::vector shall be set by statically casting the member pointers in the vector assembly to type **sc_object***.

An application shall not construct an object of class **sc_vector_assembly** except by calling **sc_assemble_vector**.

Class **sc_vector_assembly** shall be copyable.

Example:

```
struct i_f: virtual sc_core::sc_interface { ... };

struct Init : sc_core::sc_module {
    sc_core::sc_port<i_f> port;
    ...
    Init(sc_core::sc_module_name _name)
    : port("port") { ... }
};

struct Targ : public sc_core::sc_module, private i_f {
    sc_core::sc_export<i_f> xp;
    ...
    Targ(sc_core::sc_module_name _name)
    : xp("xp") { ... }
};

struct Top : sc_core::sc_module {
    sc_core::sc_vector<Init> init_vec;
    sc_core::sc_vector<Targ> targ_vec;
    ...
    Top(sc_core::sc_module_name _name, int N)
    : init_vec("init_vec", N), targ_vec("targ_vec", N) {

        // Vector-to-vector bind from vector-of-ports to vector-of-exports
        sc_core::sc_assemble_vector(init_vec, &Init::port).bind(sc_core::sc_assemble_vector(targ_vec,
&Targ::xp));
        ...
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    }
    ...
};
```

8.6 Utility functions

8.6.1 Function declarations

```
namespace sc_dt {

    template <class T>
    T sc_abs( const T& );

    template <class T>
    T sc_max( const T& a , const T& b ) { return (( a >= b ) ? a : b ); }

    template <class T>
    T sc_min( const T& a , const T& b ) { return (( a <= b ) ? a : b ); }
}

namespace sc_core {

    #define IEEE_1666_SYSTEMC 201101L

    #define SC_VERSION_MAJOR                implementation-defined_number
    #define SC_VERSION_MINOR                implementation-defined_number
    #define SC_VERSION_PATCH                implementation-defined_number
    #define SC_VERSION_ORIGINATOR           implementation-defined_string
    #define SC_VERSION_RELEASE_DATE         implementation-defined_date
    #define SC_VERSION_PRERELEASE           implementation-defined_string
    #define SC_IS_PRERELEASE                implementation-defined_bool
    #define SC_VERSION                      implementation-defined_string
    #define SC_COPYRIGHT                    implementation-defined_string

    extern const unsigned int       sc_version_major;
    extern const unsigned int       sc_version_minor;
    extern const unsigned int       sc_version_patch;
    extern const std::string        sc_version_originator;
    extern const std::string        sc_version_release_date;
    extern const std::string        sc_version_prerelease;
    extern const bool               sc_is_prerelease;
    extern const std::string        sc_version_string;
    extern const std::string        sc_copyright_string;

    const char* sc_copyright();
    const char* sc_version();
    const char* sc_release();
}
```

8.6.2 sc_abs

```
template <class T>
T sc_abs( const T& );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Function **sc_abs** shall return the absolute value of the argument. This function shall be implemented by calling the operators **bool T::operator>=(const T&)** and **T T::operator-()**, and hence, the template argument can be any SystemC numeric type or any fundamental C++ type.

8.6.3 sc_max

```
template <class T>
T sc_max( const T& a , const T& b ) { return (( a >= b ) ? a : b ); }
```

Function **sc_max** shall return the greater of the two values passed as arguments as defined above.

NOTE—The template argument should be a type for which **operator>=** is defined or for which a user-defined conversion to such a type is defined, such as any SystemC numeric type or any fundamental C++ type.

8.6.4 sc_min

```
template <class T>
T sc_min( const T& a , const T& b ) { return (( a <= b ) ? a : b ); }
```

Function **sc_min** shall return the lesser of the two values passed as arguments as defined above.

NOTE—The template argument should be a type for which **operator<=** is defined or for which a user-defined conversion to such a type is defined, such as any SystemC numeric type or any fundamental C++ type.

8.6.5 Version and copyright

```
#define IEEE_1666_SYSTEMC 201101L
```

The implementation shall define the macro IEEE_1666_SYSTEMC with precisely the value given above. It is the intent that future versions of this standard will replace the value of this macro with a numerically greater value.

```
#define SC_VERSION_MAJOR          implementation-defined_number   // For example, 2
#define SC_VERSION_MINOR          implementation-defined_number   // 3
#define SC_VERSION_PATCH          implementation-defined_number   // 4
#define SC_VERSION_ORIGINATOR     implementation-defined_string   // "OSCI"
#define SC_VERSION_RELEASE_DATE   implementation-defined_date     // "20110411"
#define SC_VERSION_PRERELEASE     implementation-defined_string   // "beta"
#define SC_IS_PRERELEASE          implementation-defined_bool     // 1
#define SC_VERSION                implementation-defined_string   // "2.3.4_beta-OSCI"
#define SC_COPYRIGHT              implementation-defined_string
```

```
extern const unsigned int     sc_version_major;
extern const unsigned int     sc_version_minor;
extern const unsigned int     sc_version_patch;
extern const std::string      sc_version_originator;
extern const std::string      sc_version_release_date;
extern const std::string      sc_version_prerelease;
extern const bool             sc_is_prerelease;
extern const std::string      sc_version_string;
extern const std::string      sc_copyright_string;
```

Each implementation shall define the macros and constants given above. It is recommended that each implementation should in addition define one or more implementation-specific macros in order to allow an application to determine which implementation is being run.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The values of the macros and constants defined in this clause may be independent of the values of the corresponding set of definitions for TLM-2.0 given in 10.8.3.

Each *implementation-defined_number* shall consist of a sequence of decimal digits from the character set [0–9] not enclosed in quotation marks.

The originator and pre-release strings shall each consist of a sequence of characters from the character set [A–Z][a–z][0–9]_ enclosed in quotation marks.

The version release date shall consist of an ISO 8601 basic format calendar date of the form YYYYMMDD, where each of the eight characters is a decimal digit, enclosed in quotation marks.

The value of the SC_IS_PRERELEASE flag shall be either 0 or 1, not enclosed in quotation marks.

The version string shall be set to the value "major.minor.patch_prerelease-originator" or "major.minor.patch-originator", where major, minor, patch, prerelease, and originator are the values of the corresponding strings (without enclosing quotation marks), and the presence or absence of the prerelease string shall depend on the value of the SC_IS_PRERELEASE flag.

The copyright string should be set to a copyright notice. The intent is that this string should contain a legal copyright notice, which an application may print to the console window or to a log file.

Each constant shall be initialized with the value defined by the macro of the corresponding name converted to the appropriate data type.

const char* **sc_release**();

Function **sc_release** shall return the value of the **sc_version_string** converted to a C string.

const char* **sc_version**();

Function **sc_version** shall return an implementation-defined string. The intent is that this string should contain information concerning the version of the SystemC class library implementation, which an application may print to the console window or to a log file.

const char* **sc_copyright**();

Function **sc_copyright** shall return the value of the copyright string converted to a C string.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

9. Overview of TLM-2.0 and compliance with TLM-2.0 standard

9.1 Overview

Clause 10 through Clause 17 define the SystemC Transaction-Level Modeling Standard, version 2.0, also known as TLM-2.0. This version of the standard includes the core interfaces from TLM-1.

TLM-2.0 consists of a set of core interfaces, the global quantum, initiator and target sockets, the generic payload and base protocol, and the utilities. The TLM-1 core interfaces, analysis interface, and analysis ports are also included, although they are separate from the main body of the TLM-2.0 standard. The TLM-2.0 core interfaces consist of the blocking and non-blocking transport interfaces, the direct memory interface (DMI), and the debug transport interface. The generic payload supports the abstract modeling of memory-mapped buses, together with an extension mechanism to support the modeling of specific bus protocols while maximizing interoperability.

The TLM-2.0 classes are layered on top of the SystemC class library as shown in Figure 16. For maximum interoperability, and particularly for memory-mapped bus modeling, it is recommended that the TLM-2.0 core interfaces, sockets, generic payload, and base protocol be used together in concert. These classes are known collectively as the *interoperability layer*. In cases where the generic payload is inappropriate, it is possible for the core interfaces and the initiator and target sockets, or the core interfaces alone, to be used with an alternative transaction type. It is even technically possible for the generic payload to be used directly with the core interfaces without the initiator and target sockets, although this approach is not recommended.

It is not strictly necessary to use the utilities to achieve interoperability between bus models. Nonetheless, these classes should be used where possible for consistency of style and are documented and maintained as part of the TLM-2.0 standard.

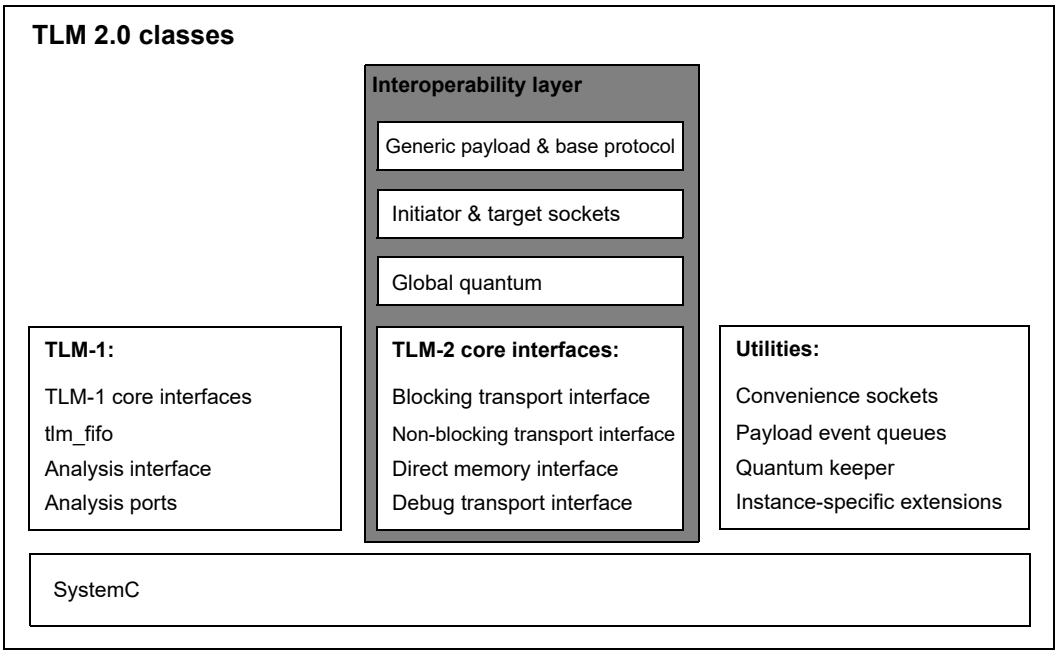

Figure 16—TLM 2.0 classes

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The generic payload is primarily intended for memory-mapped bus modeling, but it may also be used to model other non-bus protocols with similar attributes. The attributes and phases of the generic payload can be extended to model specific protocols, but such extensions may lead to a reduction in interoperability depending on the degree of deviation from the standard non-extended generic payload.

A fast, loosely-timed model is typically expected to use the blocking transport interface, the direct memory interface, and temporal decoupling. A more accurate, approximately-timed model is typically expected to use the non-blocking transport interface and the payload event queues. These statements are just coding style suggestions, however, and are not a normative part of the TLM-2.0 standard.

9.2 Compliance with the TLM-2.0 standard

This standard defines three notions of compliance related to the TLM-2.0 classes, the first concerning compliance of the implementation and the latter two concerning compliance of the application.

a) A **TLM-2.0-compliant implementation** is an implementation that provides all of the TLM-2.0 classes described in this standard with the semantics described in this standard, including both the TLM-2.0 interoperability layer and the TLM-2.0 utilities. TLM-2.0-compliance alone does not infer full compliance with the entire standard, although it does implicitly infer compliance with the subset of SystemC used by the TLM-2.0 classes (see also 1.4).

b) A **TLM-2.0 base-protocol-compliant** model is a part of an application that obeys all of the rules of the TLM-2.0 base protocol as described in this standard. Such a model will necessarily consist of one or more SystemC modules with *standard sockets* (as defined below) specialized using the protocol traits class **tlm_base_protocol**, and precisely obeying each and every rule defined in 15.2. See 15.2.1 for an introduction to the base protocol rules and 14.2.2 regarding the use of extensions with the base protocol.

c) A **TLM-2.0 custom-protocol-compliant** model is a part of an application that has *standard sockets* specialized with a user-defined protocol traits class (specifically not **tlm_base_protocol**) and that uses the generic payload, including the generic payload extension and memory management mechanisms where appropriate. A custom-protocol-compliant model is not obliged to obey any of the rules of the base protocol, although such a model is recommended to follow the rules of the base protocol as closely as possible in order to minimize the amount of engineering effort required to interface any two such models. Although this recommendation is necessarily informal, there being no *a priori* limits on the kinds of protocols modeled using TLM-2.0, it is key to gaining benefit from the TLM-2.0 standard. See 14.2.3 regarding the relationship between custom protocol rules and the base protocol.

In case b) and case c), the term *standard socket* means an object of type **tlm_initiator_socket**, **tlm_target_socket**, or any class derived from one of these two classes.

A model that uses only isolated features of the TLM-2.0 class library may be compliant with this standard but is neither TLM-2.0 base-protocol-compliant nor TLM-2.0 custom-protocol-compliant.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

10. Introduction to TLM-2.0

10.1 Background

The TLM-1 standard defined a set of core interfaces for transporting transactions by value or const reference. This set of interfaces is being used successfully in some applications, but it has three shortcomings with respect to the modeling of memory-mapped buses and other on-chip communication networks:

a) TLM-1 has no standard transaction class, so each application has to create its own non-standard classes, resulting in very poor interoperability between models from different sources. TLM-2.0 addresses this shortcoming with the generic payload.
b) TLM-1 has no explicit support for timing annotation, so no standardized way of communicating timing information between models. TLM-2.0 addresses this shortcoming with the addition of timing annotation function arguments to the blocking and non-blocking transport interface.
c) The TLM-1 interfaces require all transaction objects and data to be passed by value or const reference, which may slow down simulation in certain use cases (although not all). TLM-2.0 passes transaction objects by non-const reference, which is a fast solution for modeling memory-mapped buses.

10.2 Transaction-level modeling, use cases, and abstraction

There has been a longstanding discussion in the ESL community concerning what is the most appropriate taxonomy of abstraction levels for transaction-level modeling. Models have been categorized according to a range of criteria, including granularity of time, frequency of model evaluation, functional abstraction, communication abstraction, and use cases. The TLM-2.0 activity explicitly recognizes the existence of a variety of use cases for transaction-level modeling, but rather than defining an abstraction level around each use case, TLM-2.0 takes the approach of distinguishing between interfaces (APIs), on the one hand, and coding styles, on the other. The TLM-2.0 standard defines a set of interfaces that should be thought of as low-level programming mechanisms for implementing transaction-level models, and then describes a number of coding styles that are appropriate for, but not locked to, the various use cases (Figure 17).

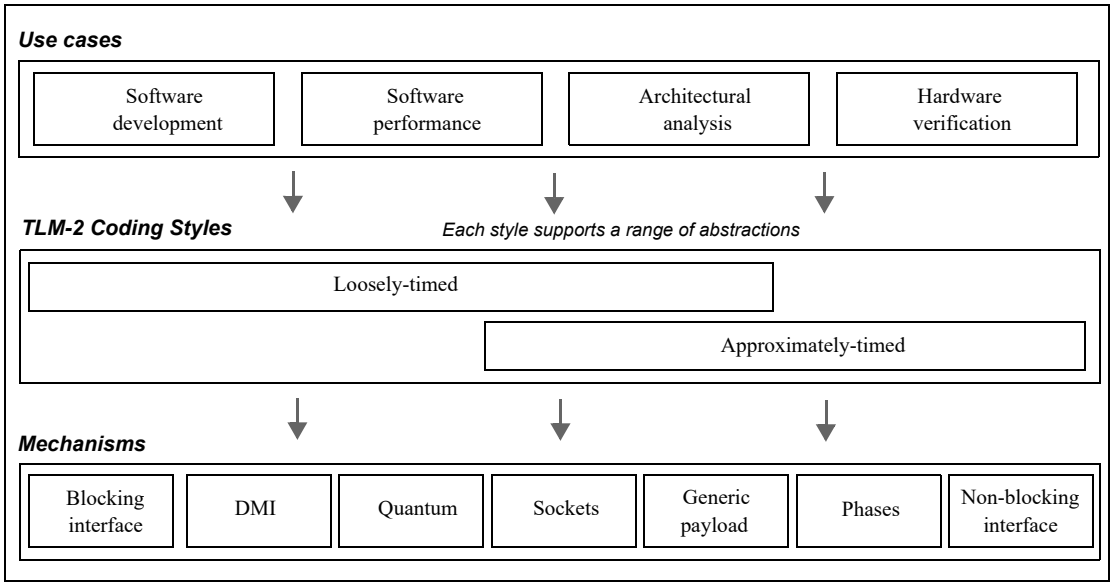

Figure 17—Use cases, coding styles, and mechanisms

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The definitions of the standard TLM-2.0 interfaces stand apart from the descriptions of the coding styles. It is the TLM-2.0 interfaces that form the normative part of the standard and help ensure interoperability. Each coding style can support a range of abstraction across functionality, timing, and communication. In principle users can create their own coding styles.

An untimed functional model consisting of a single software thread can be written as a C function or as a single SystemC process, and it is sometimes termed an *algorithmic* model. Such a model is not *transaction-level* per se, because by definition a transaction is an abstraction of communication, and a single-threaded model has no inter-process communication. A transaction-level model requires multiple SystemC processes to simulate concurrent execution and communication.

An abstract transaction-level model containing multiple processes (multiple software threads) requires some mechanism by which those threads can yield control to one another. This is because SystemC uses a cooperative multitasking model where an executing process cannot be preempted by any other process. SystemC processes yield control by calling *wait* in the case of a thread process, or returning to the kernel in the case of a method process. Calls to *wait* are usually hidden behind a programming interface (API), which may model a particular abstract or concrete protocol that may or may not rely on timing information.

Synchronization may be *strong* in the sense that the sequence of communication events is precisely determined in advance, or *weak* in the sense that the sequence of communication events is partially determined by the detailed timing of the individual processes. Strong sychronization is easily implemented in SystemC using FIFOs or semaphores, allowing a completely untimed modeling style where in principle simulation can run without advancing simulation time. Untimed modeling in this sense is outside the scope of TLM-2.0. On the other hand, a fast virtual platform model allowing multiple embedded software threads to run in parallel may use either strong or weak synchronization. In this standard, the appropriate coding style for such a model is termed *loosely-timed*.

A more detailed transaction-level model may need to associate multiple protocol-specific timing points with each transaction, such as timing points to mark the start and the end of each phase of the protocol. By choosing an appropriate number of timing points, it is possible to model communication to a high degree of timing accuracy without the need to execute the component models on every single clock cycle. In this standard, such a coding style is termed *approximately-timed*.

10.3 Coding styles

10.3.1 Overview

A coding style is a set of programming language idioms that work well together, not a specific abstraction level or software programming interface. For simplicity and clarity, this document restricts itself to elaborating two specific named coding styles: *loosely-timed* and *approximately-timed*. By their nature the coding styles are not precisely defined, and the rules governing the TLM-2.0 core interfaces are defined independently from these coding styles. In principle, it would be possible to define other coding styles based on the TLM-1 and TLM-2.0 mechanisms.

10.3.2 Untimed coding style

TLM-2.0 does not make explicit provision for an untimed coding style because all contemporary bus-based systems require some notion of time in order to model software running on one or more embedded processors. However, untimed modeling is supported by the TLM-1 core interfaces. (The term *untimed* is sometimes used to refer to models that contain a limited amount of timing information of unspecified accuracy. In TLM-2.0, such models would be termed *loosely-timed*.)

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

10.3.3 Loosely-timed coding style and temporal decoupling

The loosely-timed coding style makes use of the blocking transport interface. This interface allows only two timing points to be associated with each transaction, corresponding to the call to and return from the blocking transport function. In the case of the base protocol, the first timing point marks the beginning of the request, and the second marks the beginning of the response. These two timing points could occur at the same simulation time or at different times.

The loosely-timed coding style is appropriate for the use case of software development using a virtual platform model of an MPSoC, where the software content may include one or more operating systems. The loosely-timed coding style supports the modeling of timers and interrupts, sufficient to boot an operating system and run arbitrary code on the target machine.

The loosely-timed coding style also supports *temporal decoupling*, where individual SystemC processes are permitted to run ahead in a local "time warp" without actually advancing simulation time until they reach the point when they need to synchronize with the rest of the system. Temporal decoupling can result in very fast simulation for certain systems because it increases the data and code locality and reduces the scheduling overhead of the simulator. Each process is allowed to run for a certain time slice or *quantum* before switching to the next, or instead, it may yield control when it reaches an explicit synchronization point.

Just considering SystemC itself, the SystemC scheduler keeps a tight hold on simulation time. The scheduler advances simulation time to the time of the next event; then it runs any processes due to run at that time or sensitive to that event. SystemC processes only run at the current simulation time (as obtained by calling the member function **sc_time_stamp**), and whenever a SystemC process reads or writes a variable, it accesses the state of the variable as it would be at the current simulation time. When a process finishes running, it must pass control back to the simulation kernel. If the simulation model is written at a fine-grained level, then the overhead of event scheduling and process context switching becomes the dominant factor in simulation speed. One way to speed up simulation is to allow processes to run ahead of the current simulation time, or temporal decoupling.

When implementing temporal decoupling in SystemC, a process can be allowed to run ahead of simulation time until it needs to interact with another process, for example, to read or update a variable belonging to another process. At that point, the process may either access the current value and continue (with some possible loss of timing accuracy) or may return control to the simulation kernel, only resuming the process when simulation time has caught up with the local "time warp." Each process is responsible for determining whether it can run ahead of simulation time without breaking the functionality of the model. When a process encounters an external dependency, it has two choices: Either force synchronization, which means yielding to allow all other processes to run as normal until simulation time catches up, or sample or update the current value and continue. The synchronization option is functionally congruent with the standard SystemC simulation semantics. Continuing with the current value relies on making assumptions concerning communication and timing in the modeled system. It assumes that no damage will be done by sampling or updating the value too early or too late. This assumption is usually valid in the context of a virtual platform simulation, where the software stack should not be dependent on the low-level details of the hardware timing anyway.

Temporal decoupling is characteristic of the loosely-timed coding style.

If a process were permitted to run ahead of simulation time with no limit, the SystemC scheduler would be unable to operate and other processes would never get the chance to execute. This may be avoided by reference to the *global quantum*, which imposes an upper limit on the time a process is allowed to run ahead of simulation time. The quantum is set by the application, and the quantum value represents a trade-off between simulation speed and accuracy. Too small a quantum forces processes to yield and synchronize very frequently, slowing down simulation. Too large a quantum might introduce timing inconsistencies across the system, possibly to the point where the system ceases to function.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

For example, consider the simulation of a system consisting of a processor, a memory, a timer, and some slow external peripherals. The software running on the processor spends most of its time fetching and executing instructions from system memory, and only interacts with the rest of the system when it is interrupted by the timer, say every 1 ms. The ISS that models the processor could be permitted to run ahead of SystemC simulation time with a quantum of up to 1 ms, making direct accesses to the memory model, but only synchronizing with the peripheral models at the rate of timer interrupts. The point here is that the ISS does not have to be locked to the simulation time clock of the hardware part of the system, as would be the case with more traditional hardware-software co-simulation. Depending on the detail of the models, temporal decoupling alone could give a simulation speed improvement of approximately 10X, or 100X, when combined with DMI.

It is quite possible for some processes to be temporally decoupled and others not, and for different processes to use different values for the time quantum. However, any process that is not temporally decoupled is likely to become a simulation speed bottleneck.

In TLM-2.0, temporal decoupling is supported by the **tlm_global_quantum** class and the timing annotation of the blocking and non-blocking transport interface. The utility class **tlm_quantumkeeper** provides a convenient way to access the global quantum.

10.3.4 Synchronization in loosely-timed models

An untimed model relies on the presence of explicit synchronization points (that is calls to **wait** or blocking method calls) in order to pass control between initiators at predetermined points during execution. A loosely-timed model can also benefit from explicit synchronization in order to help guarantee deterministic execution, but a loosely-timed model is able to make progress even in the absence of explicit synchronization points (calls to **wait**), because each initiator will only run ahead as far as the end of the time quantum before yielding control. A loosely-timed model can increase its timing accuracy by using synchronization-on-demand, that is, yielding control to the scheduler before reaching the end of the time quantum.

The time quantum mechanism is not intended to ensure correct system synchronization, but it is a simulation mechanism that allows multiple system initiators to make progress in a scheduler-based simulation environment. The time quantum mechanism is not an alternative to designing an explicit synchronization scheme at the system level.

10.3.5 Approximately-timed coding style

The approximately-timed coding style is supported by the non-blocking transport interface, which is appropriate for the use cases of architectural exploration and performance analysis. The non-blocking transport interface provides for timing annotation and for multiple phases and timing points during the lifetime of a transaction.

For approximately-timed modeling, a transaction is broken down into multiple phases, with an explicit timing point marking the transition between phases. In the case of the base protocol there are exactly four timing points marking the beginning and the end of the request and the beginning and the end of the response. Specific protocols may need to add further timing points, which may possibly cause the loss of direct compatibility with the generic payload.

Although it is possible to use the non-blocking transport interface with just two phases to indicate the start and end of a transaction, the blocking transport interface is generally preferred for loosely-timed modeling.

The approximately-timed coding style cannot generally exploit temporal decoupling because of the need for timing accuracy. Instead, each process typically executes in lock step with the SystemC scheduler. Process interactions are annotated with specific delays. To create an approximately-timed model, it is generally

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

sufficient to annotate delays representing the data transfer times for write and read commands and the latency of the target. For the base protocol, the data transfer times are effectively the same as the minimum initiation interval or accept delay between two successive requests or two successive responses. The annotated delays are implemented by making calls to the SystemC scheduler, that is, **wait**(delay) or **notify**(delay).

10.3.6 Characterization of loosely-timed and approximately-timed coding styles

The coding styles can be characterized in terms of timing points and temporal decoupling.

Loosely-timed. Each transaction has just two timing points, marking the start and the end of the transaction. Simulation time is used, but processes may be temporally decoupled from simulation time. Each process keeps a tally of how far it has run ahead of simulation time, and it may yield because it reaches an explicit synchronization point or because it has consumed its time quantum.

Approximately-timed. Each transaction has multiple timing points. Processes typically need to run in lock-step with SystemC simulation time. Delays annotated onto process interactions are implemented using time-outs (wait) or timed event notifications.

Untimed. The notion of simulation time is unnecessary. Processes yield at explicit pre-determined synchronization points.

10.3.7 Switching between loosely-timed and approximately-timed modeling

A model may switch between the loosely-timed and approximately-timed coding style during simulation. The idea is to run rapidly through the reset and boot sequence at the loosely-timed level, then switch to approximately-timed modeling for more detailed analysis once the simulation has reached an interesting stage.

10.3.8 Cycle-accurate modeling

Cycle-accurate modeling is beyond the scope of TLM-2.0 at present. It is possible to create cycle-accurate models using SystemC and TLM-1 as it stands, but the requirement for the standardization of a cycle-accurate coding style remains an open issue, possibly to be addressed by a future Accellera Systems Initiative standard.

In principle only, the approximately-timed coding style might be extended to encompass cycle-accurate modeling by defining an appropriate set of phases and rules. The TLM-2.0 release includes sufficient machinery for this, but the details have not been worked out.

10.3.9 Blocking versus non-blocking transport interfaces

The blocking and non-blocking transport interfaces are distinct interfaces that exist in TLM-2.0 to support different levels of timing detail. The blocking transport interface is only able to model the start and end of a transaction, with the transaction being completed within a single function call. The non-blocking transport interface allows a transaction to be broken down into multiple timing points, and typically requires multiple function calls for a single transaction.

For interoperability, the blocking and non-blocking transport interfaces are combined into a single interface. A model that initiates transactions may use the blocking or non-blocking transport interfaces (or both) according to coding style. Any model that provides a TLM-2.0 transport interface is obliged to provide both the blocking and non-blocking forms for maximal interoperability, although such a model is not obliged to implement both interfaces internally.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

TLM-2.0 provides a mechanism (the so-called *convenience socket*) to automatically convert incoming blocking or non-blocking transport calls to non-blocking or blocking transport calls, respectively. Converting transport call types does incur some cost, particularly converting an incoming non-blocking call to a blocking implementation. However, the cost overhead is mitigated by the fact that the presence of an approximately-timed model is likely to have a negative impact on simulation speed anyway.

The C++ static typing rules enforce the implementation of both interfaces, but an initiator can choose dynamically whether to call the blocking or the non-blocking transport method. It is possible for different initiators to call different methods, or for a given initiator to switch between blocking and non-blocking calls on-the-fly. This standard includes rules governing the mixing and ordering of blocking and non-blocking transport calls to the same target.

The strength of the blocking transport interface is that it allows a simplified coding style for models that are able to complete a transaction in a single function call. The strength of the non-blocking transport interface is that it supports the association of multiple timing points with a single transaction. In principle, either interface supports temporal decoupling, but the speed benefits of temporal decoupling are likely to be nullified by the presence of multiple timing points for approximately-timed models.

10.3.10 Use cases and coding styles

Table 52 summarizes the mapping between use cases for transaction-level modeling and coding styles.

Table 52—Mapping between use cases for transaction-level modeling and coding styles

Use case	Coding style
Software application development	Loosely-timed
Software performance analysis	Loosely-timed
Hardware architectural analysis	Loosely-timed or approximately-timed
Hardware performance verification	Approximately-timed or cycle-accurate
Hardware functional verification	Untimed (verification environment), loosely-timed or approximately-timed

10.4 Initiators, targets, sockets, and transaction bridges

The TLM-2.0 core interfaces pass transactions between initiators and targets (Figure 18). An initiator is a module that can initiate transactions, that is, create new transaction objects and pass them on by calling a method of one of the core interfaces. A target is a module that acts as the final destination for a transaction. In the case of a write transaction, an initiator (such as a processor) writes data to a target (such as a memory). In the case of a read transaction, an initiator reads data from a target. An interconnect component is a module that accesses a transaction but does not act as an initiator or a target for that transaction, typical examples being arbiters and routers. The roles of initiator, interconnect, and target can change dynamically. For example, a given component may act as an interconnect for some transactions but as a target for other transactions.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

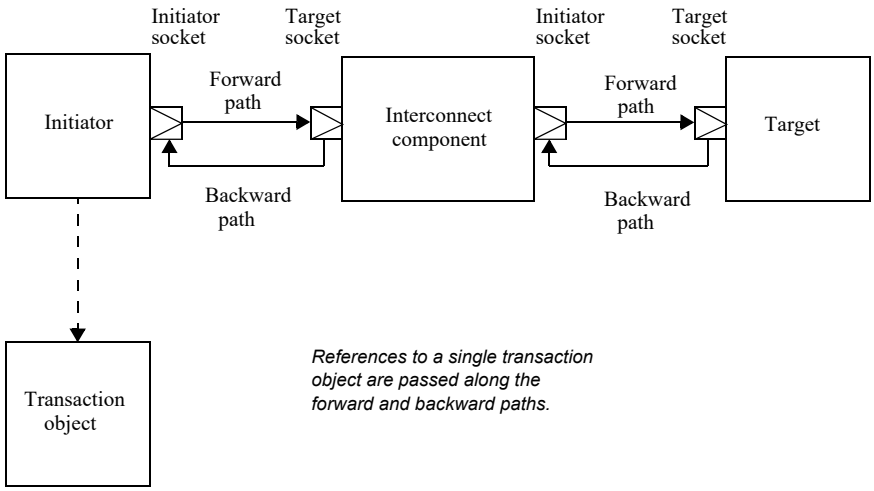

Figure 18—Initiators and targets

To illustrate the idea, this paragraph will describe the lifetime of a typical transaction object (Figure 19). The transaction object is created by an initiator and passed as an argument of a method of the transport interface (blocking or non-blocking). That method is implemented by an interconnect component such as an arbiter, which may read attributes of the transaction object before passing it on to a further transport call. That second transport method is implemented by a second interconnect component, such as a router, which in turn passes on the transaction through a third transport call to a target such as a memory, the final destination for the transaction object. (The actual number of interconnect components will vary from transaction to transaction. There may be none.) This sequence of method calls is known as the *forward path*. The transaction is executed in the target, and the transaction object may be returned to the initiator in one of two ways, either carried with the return from the transport method calls as they unwind, known as the *return path*, or passed by making explicit transport method calls on the opposite path from target back to initiator, known as the *backward path*. This choice is determined by the return value from the non-blocking transport method. (Strictly speaking there are two return paths corresponding to the forward and backward paths, but the meaning is usually clear from the context.)

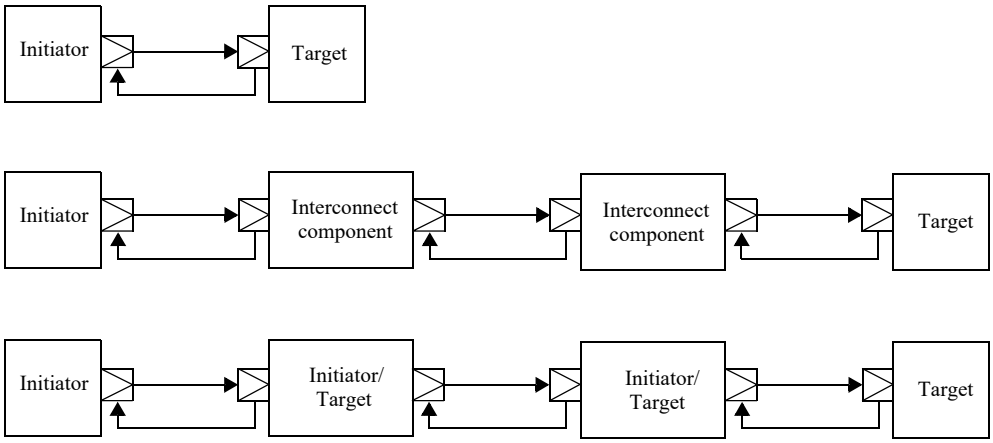

Figure 19—TLM-2.0 connectivity

The forward path is the calling path by which an initiator or interconnect component makes interface method calls forward in the direction of another interconnect component or the target. The backward path is the calling path by which a target or interconnect component makes interface method calls back in the direction

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

of another interconnect component or the initiator. The entire path between an initiator and a target consists of a number of *hops*, each hop connecting two adjacent components. A hop consists of one initiator socket bound to one target socket. The number of hops from initiator to target is one greater than the number of interconnect components on that path. When using the generic payload, the forward and backward paths should each pass through the same set of components and sockets in opposing directions.

In order to support both forward and backward paths, each connection between components requires a port and an export, both of which have to be bound. This is facilitated by the *initiator socket* and the *target socket*. An initiator socket provides a port for interface method calls on the forward path, and an export for interface method calls on the backward path. A target socket provides the opposite. (More specifically, an initiator socket is derived from class **sc_port** and *has* an **sc_export**, and vice versa for a target socket.) The initiator and target socket classes overload the SystemC port binding operator to bind implicitly both forward and backward paths.

As well as the transport interfaces, the sockets also encapsulate the DMI and debug transport interfaces (see below).

When using sockets, an initiator component will have at least one initiator socket, a target component at least one target socket, and an interconnect component at least one of each. A component may have several sockets transporting different transaction types, in which case a single component may act as an initiator or as a target for multiple independent transactions.

In order to model a bus bridge, there are two alternatives. Either model the bus bridge as an interconnect component or model the bus bridge as a transaction bridge between two separate TLM-2.0 transactions. An interconnect component would pass on a pointer to a single transaction object, which is the best approach for simulation speed. A transaction bridge would require the transaction object to be copied, which gives much more flexibility because the two transactions could have different attributes.

The use of TLM-2.0 sockets is recommended for maximal interoperability, convenience, and a consistent coding style. Although it is possible for components to use SystemC ports and exports directly with the TLM-2.0 core interfaces, this is not recommended.

10.5 DMI and debug transport interfaces

The direct memory interface (DMI) and debug transport interface are specialized interfaces distinct from the transport interface, providing direct access and debug access to an area of memory owned by a target. Once a DMI request has been granted, the DMI interface enables an initiator to bypass the usual path through the interconnect components used by the transport interface. DMI is intended to accelerate regular memory transactions in a loosely-timed simulation, whereas the debug transport interface is for debug access free of the delays or side-effects associated with regular transactions.

The DMI has both forward (initiator-to-target) and backward (target-to-initiator) interfaces, whereas debug only has a forward interface (see 11.3 and 11.4).

10.6 Combined interfaces and sockets

The blocking and non-blocking transport interfaces are combined with the DMI and debug transport interfaces in the standard initiator and target sockets. All four interfaces (the two transport interfaces, DMI, and debug) can be used in parallel to access a given target (subject to the rules described in this standard). These combined interfaces are one of the keys to interoperability between components using the TLM-2.0 standard, the other key being the generic payload (see 13.1).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The standard target sockets provide all four interfaces, so each target component must effectively implement the methods of all four interfaces. However, the design of the blocking and non-blocking transport interfaces together with the provision of convenience sockets to convert between the two means that a given target need only implement either the blocking or the non-blocking transport method, not both, according to the speed and accuracy requirements of the model.

A given initiator may choose to call methods through any or all of the core interfaces, again according to the speed and accuracy requirements. The coding styles mentioned above help guide the choice of an appropriate set of interface features. Typically, a loosely-timed initiator will call blocking transport, DMI and debug, whereas an approximately-timed initiator will call non-blocking transport and debug.

10.7 Namespaces

The TLM-2.0 classes shall be declared in two top-level C++ namespaces, **tlm** and **tlm_utils**. Particular implementations of the TLM-2.0 classes may choose to nest further namespaces within these two namespaces, but such nested namespaces shall not be used in applications.

Namespace **tlm** contains the classes that comprise the interoperability interface for memory-mapped bus modeling.

Namespace **tlm_utils** contains utility classes that are not strictly necessary for interoperability at the interface between memory-mapped bus models but that are nevertheless a proper part of the TLM-2.0 standard.

10.8 Header files and version numbers

10.8.1 Overview

Applications should #include the header file **tlm**, which shall contain all of the public declarations for the TLM-2.0 interoperability layer. The header files **tlm** and **tlm.h** shall be equivalent in the sense that they shall include precisely the same set of declarations and macros, and they may be used interchangeably. The header file **tlm.h** is provided for backward compatibility with earlier versions of TLM-2.0 and may be deprecated in future versions of this standard. The TLM-2.0 utilities shall not be present in the header files **tlm** or **tlm.h**. Applications should also explicitly #include the header files for any TLM-2.0 utilities they may wish to use, which shall be placed in a directory named **tlm_utils**. The precise file names for each of these header files are defined in their respective clauses in this standard.

10.8.2 Software version information

The header files **tlm** and **tlm.h** shall include a set of macros, constants, and functions that provide information concerning the version number of the TLM-2.0 software distribution. Applications may use these macros and constants.

The value of the macros and constants defined in this clause may be independent of the values of the corresponding set of definitions for SystemC given in 8.6.5.

10.8.3 Definitions

```
namespace tlm
{

#define  TLM_VERSION_MAJOR          implementation-defined_number     // For example, 2
#define  TLM_VERSION_MINOR          implementation-defined_number     // 0
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
#define  TLM_VERSION_PATCH              implementation-defined_number    // 1
#define  TLM_VERSION_ORIGINATOR         implementation-defined_string    // "OSCI"
#define  TLM_VERSION_RELEASE_DATE       implementation-defined_date      // "20090329"
#define  TLM_VERSION_PRERELEASE         implementation-defined_string    // "beta"
#define  TLM_IS_PRERELEASE              implementation-defined_bool      // 1
#define  TLM_VERSION                    implementation-defined_string
                                                                  // "2.0.1_beta-OSCI"
#define  TLM_COPYRIGHT                  implementation-defined_string

const  unsigned  int       tlm_version_major;
const  unsigned  int       tlm_version_minor;
const  unsigned  int       tlm_version_patch;
const  std::string         tlm_version_originator;
const  std::string         tlm_version_release_date;
const  std::string         tlm_version_prerelease;
const  bool                tlm_is_prerelease;
const  std::string         tlm_version_string;
const  std::string         tlm_copyright_string;

inline  const  char*       tlm_release();
inline  const  char*       tlm_version();
inline  const  char*       tlm_copyright();

} // namespace tlm
```

10.8.4 Rules

a) Each *implementation-defined_number* shall consist of a sequence of decimal digits from the character set [0–9] not enclosed in quotation marks.

b) The originator and pre-release strings shall each consist of a sequence of characters from the character set [A–Z][a–z][0–9]_ enclosed in quotation marks.

c) The version release date shall consist of an ISO 8601 basic format calendar date of the form YYYYMMDD, where each of the eight characters is a decimal digit, enclosed in quotation marks.

d) The TLM_IS_PRERELEASE flag shall be either 0 or 1, not enclosed in quotation marks.

e) The version string shall be set to the value "major.minor.patch_prerelease-originator" or "major.minor.patch-originator", where major, minor, patch, prerelease, and originator are the values of the corresponding strings (without enclosing quotation marks), and the presence or absence of the prerelease string shall depend on the value of the TLM_IS_PRERELEASE flag.

f) The copyright string should be set to a copyright notice.

g) Each constant shall be initialized with the value defined by the macro of the corresponding name converted to the appropriate data type.

h) Functions **tlm_release** and **tlm_version** shall each return the value of the version string converted to a C string.

i) Function **tlm_copyright** shall return the value of the copyright string converted to a C string.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

11. TLM-2.0 core interfaces

11.1 Overview

In addition to the core interfaces from TLM-1, TLM-2.0 adds blocking and non-blocking transport interfaces, a direct memory interface (DMI), and a debug transport interface.

11.2 Transport interfaces

11.2.1 Overview

The transport interfaces are the primary interfaces used to transport transactions between initiators, targets, and interconnect components. Both the blocking and non-blocking transport interfaces support timing annotation and temporal decoupling, but only non-blocking transport supports multiple phases within the lifetime of a transaction. Blocking transport does not have an explicit phase argument, and any association between blocking transport and the phases of the non-blocking transport interface is purely notional. Only the non-blocking transport method returns a value indicating whether or not the return path was used.

The transport interfaces and the generic payload were designed to be used together for the fast, abstract modeling of memory-mapped buses. The transport interface templates are specialized with the transaction type allowing them to be used separately from the generic payload, although many of the interoperability benefits would be lost.

The rules governing memory management of the transaction object, transaction ordering, and the permitted function calling sequence depend on the specific transaction type passed as a template argument to the transport interface, which in turn depends on the protocol traits class passed as a template argument to the socket (if a socket is used).

11.2.2 Blocking transport interface

11.2.2.1 Introduction

The TLM-2.0 blocking transport interface is intended to support the loosely-timed coding style. The blocking transport interface is appropriate where an initiator wishes to complete a transaction with a target during the course of a single function call, the only timing points of interest being those that mark the start and the end of the transaction.

The blocking transport interface only uses the forward path from initiator to target.

The TLM-2.0 blocking transport interface has deliberate similarities with the transport interface from TLM-1, which is still part of the TLM-2.0 standard, but the TLM-1 transport interface and the TLM-2.0 blocking transport interface are not identical. In particular, the new member function **b_transport** has a single transaction argument passed by non-const reference and a second argument to annotate timing, whereas the TLM-1 member function **transport** takes a request as a single const reference request argument, has no timing annotation, and returns a response by value. TLM-1 assumes separate request and response objects passed by value (or const reference), whereas TLM-2.0 assumes a single transaction object passed by reference, whether using the blocking or the non-blocking TLM-2.0 interfaces.

The member function **b_transport** has a timing annotation argument. This single argument is used on both the call to and the return from **b_transport** to indicate the time of the start and end of the transaction, respectively, relative to the current simulation time.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

11.2.2.2 Class definition

```
namespace tlm {

template <typename TRANS = tlm_generic_payload>
class tlm_blocking_transport_if : public virtual sc_core::sc_interface {
public:
  virtual void b_transport(TRANS& trans, sc_core::sc_time& t) = 0;
};

} // namespace tlm
```

11.2.2.3 The TRANS template argument

The intent is that this core interface may be used to transport transactions of any type. A specific transaction type, **tlm_generic_payload**, is provided to ease interoperability between models where the precise details of the transaction attributes are less important.

For maximum interoperability, applications should use the default transaction type **tlm_generic_payload** with the base protocol (see 15.2). In order to model specific protocols, applications may substitute their own transaction type. Sockets that use interfaces specialized with different transaction types cannot be bound together, providing compile-time checking but restricting interoperability.

11.2.2.4 Rules

a) The member function **b_transport** may call **wait**, directly or indirectly.

b) The member function **b_transport** shall not be called from a method process.

c) The initiator may reuse a transaction object from one call to the next and across calls to the transport interfaces, DMI, and the debug transport interface.

d) The call to **b_transport** marks the first timing point of the transaction. The return from **b_transport** marks the final timing point of the transaction.

e) The timing annotation argument allows the timing points to be offset from the simulation times (value returned by **sc_time_stamp()**) at which the function call and return are executed.

f) The callee may modify or update the transaction object, subject to any constraints imposed by the transaction class **TRANS**.

g) It is recommended that the transaction object should not contain timing information. Timing should be annotated using the **sc_time** argument to **b_transport**.

h) Typically, an interconnect component should pass the **b_transport** call along the forward path from initiator to target. In other words, the implementation of **b_transport** for the target socket of the interconnect component may call the member function **b_transport** of an initiator socket.

i) Whether or not the implementation of **b_transport** is permitted to call **nb_transport_fw** depends on the rules associated with the protocol. For the base protocol, the convenience socket **simple_target_socket** is able to make this conversion automatically (see 16.2.2).

j) The implementation of **b_transport** shall not call **nb_transport_bw**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

11.2.2.5 Message sequence chart—blocking transport

The blocking transport method may return immediately (that is, in the current SystemC evaluation phase) or may yield control to the scheduler and only return to the initiator at a later point in simulation time (Figure 20). Although the initiator thread may be blocked, another thread in the initiator may be permitted to call **b_transport** before the first call has returned, depending on the protocol.

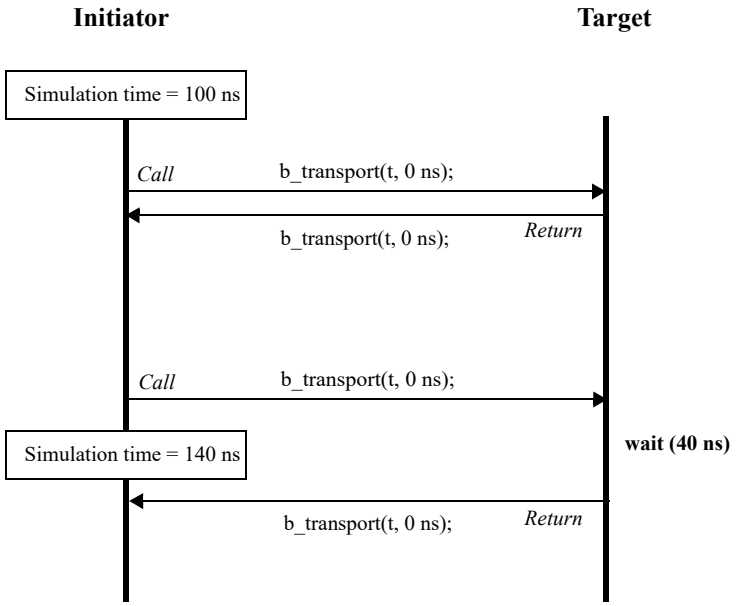

Figure 20—Blocking transport

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

11.2.2.6 Message sequence chart—temporal decoupling

A temporally decoupled initiator may run at a notional local time in advance of the current simulation time, in which case it should pass a non-zero value for the time argument to **b_transport**, as shown below. The initiator and target may each further advance the local time offset by increasing the value of the time argument. Adding the time argument returned from the call to the current simulation time gives the notional time at which each the transaction completes, but simulation time itself cannot advance until the initiator thread yields.

The body of **b_transport** may itself call **wait**, in which case the local time offset should be reset to zero. In Figure 21, the final return from the initiator happens at simulation time 140 ns, but with an annotated delay of 5 ns, giving an effective local time of 145 ns.

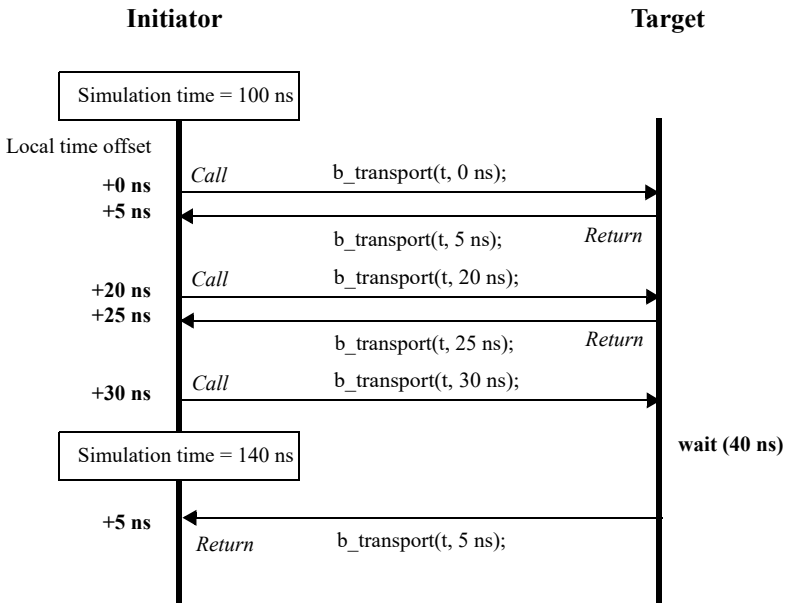

Figure 21—Temporal decoupling

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

11.2.2.7 Message sequence chart—the time quantum

A temporally decoupled initiator will continue to advance local time until the time quantum is exceeded (Figure 22). At that point, the initiator is obliged to synchronize by suspending execution, directly or indirectly calling the member function **wait** with the local time as an argument. This allows other initiators in the model to run and to catch up, which effectively means that the initiators execute in turn, each being responsible for determining when to hand back control by keeping track of its own local time. The original initiator should only run again after simulation time has advanced to the next quantum.

The primary purpose of delays in the loosely-timed coding style is to allow each initiator to determine when to hand back control. It is best if the model does not rely on the details of the timing in order to function correctly.

Within each quantum, the transactions generated by a given initiator happen in strict sequential order but without advancing simulation time. The local time is not tracked by the SystemC scheduler.

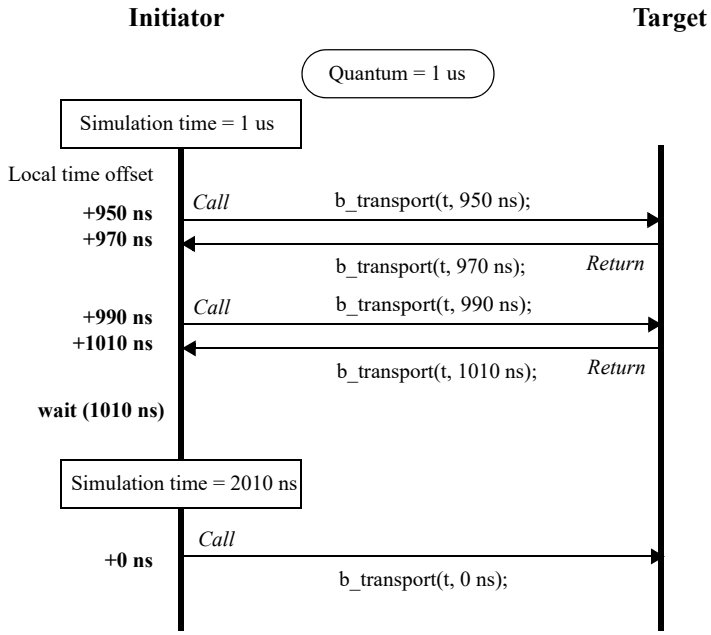

Figure 22—The time quantum

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

11.2.3 Non-blocking transport interface

11.2.3.1 Introduction

The non-blocking transport interface is intended to support the approximately-timed coding style. The non-blocking transport interface is appropriate where it is desired to model the detailed sequence of interactions between initiator and target during the course of each transaction. In other words, to break down a transaction into multiple phases, where each phase transition is associated with a timing point. Each call to and return from the non-blocking transport method may correspond to a phase transition.

By restricting the number of timing points to two, it is possible to use the non-blocking transport interface with the loosely-timed coding style, but this is not generally recommended. For loosely-timed modeling, the blocking transport interface is generally preferred for its simplicity. The non-blocking transport interface is particularly suited for modeling pipelined transactions, which would be awkward to model using blocking transport.

The non-blocking transport interface uses both the forward path from initiator to target and the backward path from target to initiator. There are two distinct interfaces, **tlm_fw_nonblocking_transport_if** and **tlm_bw_nonblocking_transport_if**, for use on opposite paths.

The non-blocking transport interface uses a similar argument-passing mechanism to the blocking transport interface in that the non-blocking transport methods pass a non-const reference to the transaction object and a timing annotation, but there the similarity ends. The non-blocking transport method also passes a phase to indicate the state of the transaction, and it returns an enumeration value to indicate whether the return from the function also represents a phase transition.

Both blocking and non-blocking transport support timing annotation, but only non-blocking transport supports multiple phases within the lifetime of a transaction. The blocking and non-blocking transport interface and the generic payload were designed to be used together for the fast, abstract modeling of memory-mapped buses. However, the transport interfaces can be used separately from the generic payload to model specific protocols. Both the transaction type and the phase type are template parameters of the non-blocking transport interface.

11.2.3.2 Class definition

```
namespace tlm {

enum tlm_sync_enum { TLM_ACCEPTED,  TLM_UPDATED,  TLM_COMPLETED };

template <typename TRANS = tlm_generic_payload, typename PHASE = tlm_phase>
class tlm_fw_nonblocking_transport_if : public virtual sc_core::sc_interface {
public:
  virtual tlm_sync_enum nb_transport_fw(TRANS& trans, PHASE& phase, sc_core::sc_time& t) = 0;
};

template <typename TRANS = tlm_generic_payload, typename PHASE = tlm_phase>
class tlm_bw_nonblocking_transport_if : public virtual sc_core::sc_interface {
public:
  virtual tlm_sync_enum nb_transport_bw(TRANS& trans, PHASE& phase, sc_core::sc_time& t) = 0;
};

} // namespace tlm
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

11.2.3.3 The TRANS and PHASE template arguments

The intent is that the non-blocking transport interface may be used to transport transactions of any type and with any number of phases and timing points. A specific transaction type, **tlm_generic_payload**, is provided to ease interoperability between models where the precise details of the transaction attributes are less important, and a specific type **tlm_phase** is provided for use with the base protocol (see 15.2).

For maximum interoperability, applications should use the default transaction type **tlm_generic_payload** and the default phase type **tlm_phase** with the base protocol. In order to model specific protocols, applications may substitute their own transaction type and phase type. Sockets that use interfaces specialized with different transaction types cannot be bound together, providing compile-time checking but restricting interoperability.

11.2.3.4 The nb_transport_fw and nb_transport_bw calls

a) There are two non-blocking transport member functions, **nb_transport_fw** for use on the forward path, and **nb_transport_bw** for use on the backward path. Aside from their names and calling direction these two member functions have similar semantics. In this document, the italicized term *nb_transport* is used to describe both member functions in situations where there is no need to distinguish between them.

   In the case of the base protocol, the forward and backward paths should pass through exactly the same sequence of components and sockets in opposing order. It is the responsibility of each component to route any transaction returning toward the initiator using the target socket through which that transaction was first received.

b) **nb_transport_fw** shall only be called on the forward path, and **nb_transport_bw** shall only be called on the backward path.

c) An **nb_transport_fw** call on the forward path shall under no circumstances directly or indirectly make a call to **nb_transport_bw** on the backward path, and vice versa.

d) The *nb_transport* methods shall not call **wait**, directly or indirectly.

e) The *nb_transport* methods may be called from a thread process or from a method process.

f) *nb_transport* is not permitted to call **b_transport**. One solution would be to call **b_transport** from a separate thread process, spawned or notified by the original member function **nb_transport_fw**. For the base protocol, a convenience socket **simple_target_socket** is provided, which is able to make this conversion automatically (see 16.2.2).

g) The non-blocking transport interface is explicitly intended to support pipelined transactions. In other words, several successive calls to **nb_transport_fw** from the same process could each initiate separate transactions without having to wait for the first transaction to complete.

h) In principle, the final timing point of a transaction may be marked by a call to or a return from *nb_transport* on either the forward path or the backward path.

11.2.3.5 The trans argument

a) The lifetime of a given transaction object may extend beyond the return from *nb_transport* such that a series of calls to *nb_transport* may pass a single transaction object forward and backward between initiators, interconnect components, and targets.

b) If there are multiple calls to *nb_transport* associated with a given transaction instance, one and the same transaction object shall be passed as an argument to every such call. In other words, a given transaction instance shall be represented by a single transaction object.

c) An initiator may re-use a given transaction object to represent more than one transaction instance, or across calls to the transport interfaces, DMI, and the debug transport interface.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

d) Since the lifetime of the transaction object may extend over several calls to *nb_transport*, either the caller or the callee may modify or update the transaction object, subject to any constraints imposed by the transaction class **TRANS**. For example, for the generic payload, the target may update the data array of the transaction object in the case of a read command but shall not update the command field (see 14.7).

11.2.3.6 The phase argument

a) Each call to *nb_transport* passes a reference to a phase object. In the case of the base protocol, successive calls to *nb_transport* with the same phase are not permitted. Each phase transition has an associated timing point. A timing annotation using the **sc_time** argument shall delay the timing point relative to the phase transition.

b) The phase argument is passed by reference. Either caller or callee may modify the phase.

c) The intent is that the phase argument should be used to inform components as to whether and when they are permitted to read or modify the attributes of a transaction. If the rules of a protocol allow a given component to modify the value of a transaction attribute during a particular phase, then that component may modify the value at any time during that phase and any number of times during that phase. The protocol should forbid other components from reading the value of that attribute during that phase, only permitting the value to be read after the next phase transition.

d) The value of the phase argument represents the current state of the protocol state machine for the given hop. Where a single transaction object is passed between more than two components (initiator, interconnect, target), each hop requires (notionally, at least) a separate protocol state machine.

e) Whereas the transaction object has a lifetime and a scope that may extend beyond any single call to *nb_transport*, the phase object is normally local to the caller. Each *nb_transport* call for a given transaction may have a separate phase object. Corresponding phase transitions on different hops may occur at different points in simulation time.

f) The default phase type **tlm_phase** is specific to the base protocol. Other protocols may use or extend type **tlm_phase** or may substitute their own phase type (with a corresponding loss of interoperability) (see 15.1).

11.2.3.7 The tlm_sync_enum return value

a) The concept of sychronization is referred to in several places. To *synchronize* is to yield control to the SystemC scheduler in order that other processes may run, but it has additional connotations for temporal decoupling. This is discussed more fully elsewhere (see 16.3.4).

b) In principle, synchronization can be accomplished by yielding (calling **wait** in the case of a thread process or returning to the kernel in the case of a method process), but a temporally decoupled initiator should synchronize by calling the member function **sync** of class **tlm_quantumkeeper**. In general, it is necessary for an initiator to synchronize from time-to-time in order to allow other SystemC processes to run.

c) The following rules apply to both the forward and backward paths.

d) The meaning of the return value from *nb_transport* is fixed and does not vary according to the transaction type or phase type. Hence the following rules are not restricted to the base protocol but apply to every transaction and phase type used to parameterize the non-blocking transport interface.

e) **TLM_ACCEPTED**. The callee shall not have modified the state of the transaction object, the phase, or the time argument during the call. In other words, TLM_ACCEPTED indicates that the return path is not being used. The caller may ignore the values of the *nb_transport* arguments following the call, since the callee is obliged to leave them unchanged. In general, the caller will have to yield before the component containing the callee can respond to the transaction. For the base protocol, a callee that is ignoring an ignorable phase should return TLM_ACCEPTED.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

f) **TLM_UPDATED**. The callee has updated the transaction object. The callee may have modified the state of the phase argument, may have modified the state of the transaction object, and may have increased the value of the time argument during the call. In other words, TLM_UPDATED indicates that the return path is being used, and the callee has advanced the state of the protocol state machine associated with the transaction. Whether or not the callee is actually obliged to modify each of the arguments depends on the protocol. Following the call to *nb_transport*, the caller should inspect the phase, transaction, and time arguments and take the appropriate action.

g) **TLM_COMPLETED**. The callee has updated the transaction object, and the transaction is complete. The callee may have modified the state of the transaction object and may have increased the value of the time argument during the call. The value of the phase argument is undefined. In other words, TLM_COMPLETED indicates that the return path is being used and the transaction is complete with respect to a particular socket. Following the call to *nb_transport*, the caller should inspect the transaction object and take the appropriate action but should ignore the phase argument. There shall be no further transport calls associated with this particular transaction through the current socket along either the forward or backward paths. Completion in this sense does not necessarily imply successful completion, so depending on the transaction type, the caller may need to inspect a response status embedded in the transaction object.

h) In general there is no obligation to complete a transaction by having *nb_transport* return TLM_COMPLETED. A transaction is in any case complete with respect to a particular socket when the final phase of the protocol is passed as an argument to *nb_transport*. (For the base protocol, the final phase is END_RESP.) In other words, TLM_COMPLETED is not mandatory.

i) For any of the three return values, and depending on the protocol, following the call to *nb_transport* the caller may need to yield in order to allow the component containing the callee to generate a response or to release the transaction object.

11.2.3.8 tlm_sync_enum summary

Table 53 provides a summary of tlm_sync_enum.

Table 53—tlm_sync_enum summary

tlm_sync_enum	Unmodified	Unchanged	Unchanged
TLM_ACCEPTED	Unmodified	Unchanged	Unchanged
TLM_UPDATED	Updated	Changed	May be increased
TLM_COMPLETED	Updated	Ignored	May be increased

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

11.2.3.9 Message sequence chart—using the backward path

The following message sequence charts illustrate various calling sequences to *nb_transport*. The arguments and return value passed to and from *nb_transport* are shown using the notation *return*, *phase*, *delay*, where return is the value returned from the function call, phase is the value of the phase argument, and delay is the value of the **sc_time** argument. The notation '-' indicates that the value is unused.

The following message sequence charts use the phases of the base protocol as an example, that is, BEGIN_REQ, END_REQ and so on. With the approximately-timed coding style and the base protocol, a transaction is passed back-and-forth twice between initiator and target. For other protocols, the number of phases and their names may be different.

If the recipient of an *nb_transport* call is unable immediately to calculate the next state of the transaction or the delay to the next timing point, it should return a value of TLM_ACCEPTED. The caller should yield control to the scheduler and expect to receive a call to *nb_transport* on the opposite path when the callee is ready to respond. Notice that in this case, unlike the loosely-timed case, the caller could be the initiator or the target (Figure 23).

Transactions may be pipelined. The initiator could call *nb_transport* to send another transaction to the target before having seen the final phase transition of the previous transaction.

Because processes are regularly yielding control to the scheduler in order to allow simulation time to advance, the approximately-timed coding style is expected to simulate a lot more slowly than the loosely-timed coding style.

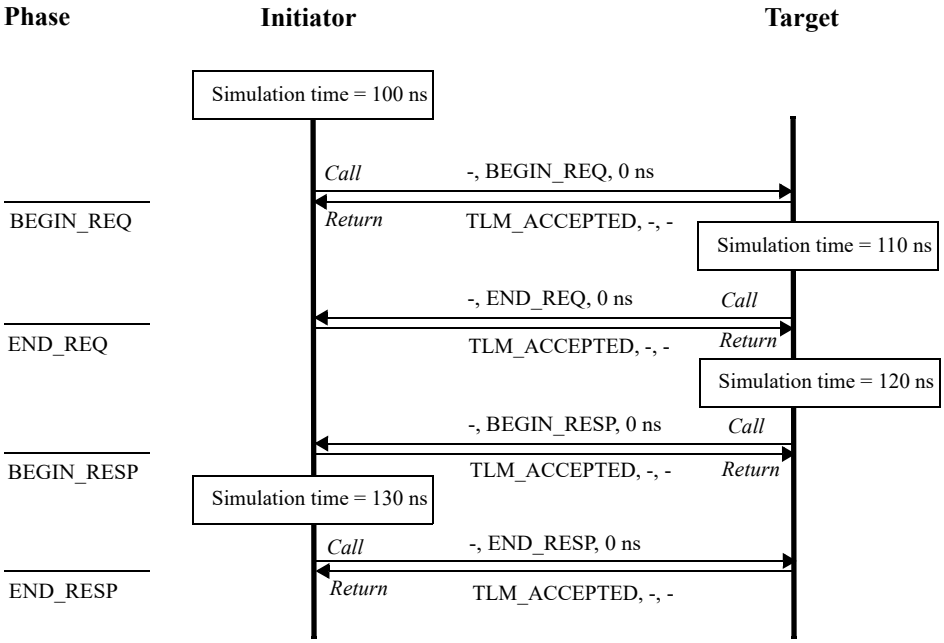

Figure 23—Using the backward path

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

11.2.3.10 Message sequence chart—using the return path

If the recipient of an *nb_transport* call can immediately calculate the next state of the transaction and the delay to the next timing point, it may return the new state on return from *nb_transport* rather than using the opposite path. If the next timing point marks the end of the transaction, the recipient can return either TLM_UPDATED or TLM_COMPLETED. A callee can return TLM_COMPLETED at any stage (subject to the rules of the protocol) to indicate to the caller that it has preempted the other phases and jumped to the final phase, completing the transaction. This applies to initiator and target alike.

With TLM_UPDATED, the callee should update the transaction, the phase, and the timing annotation.

In Figure 24, the non-zero timing annotation argument passed on return from the function calls indicates to the caller the delay between the phase transition on the hop and the corresponding timing point.

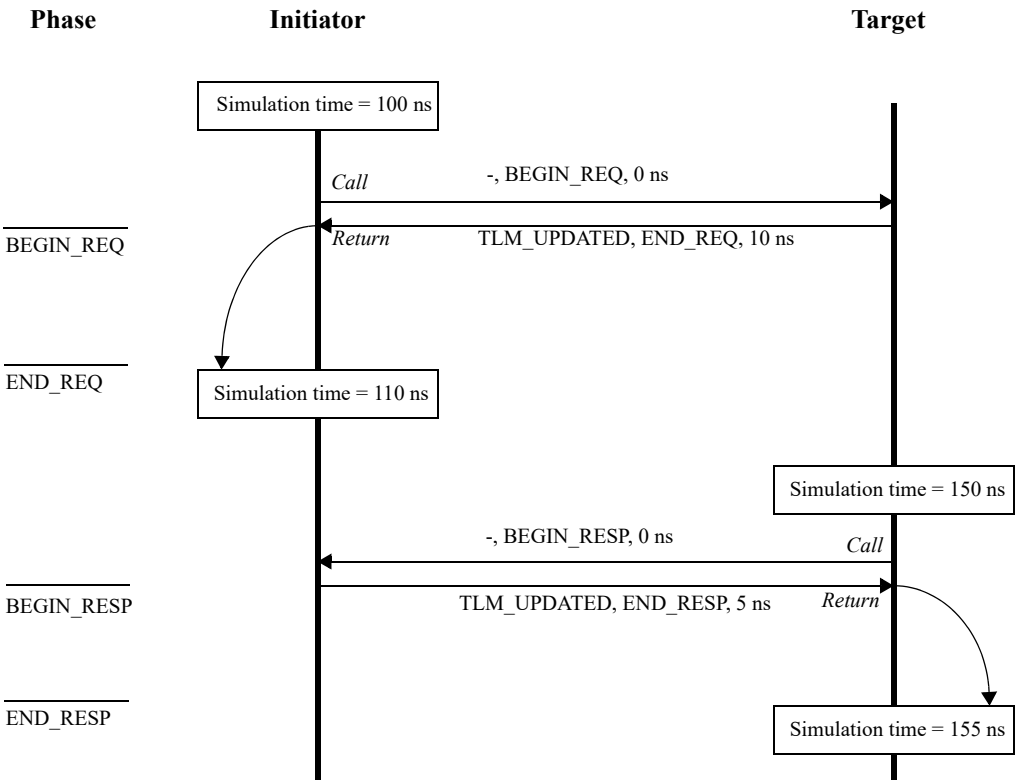

Figure 24—Using the return path

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

11.2.3.11 Message sequence chart—early completion

Depending on the protocol, an initiator or a target may return TLM_COMPLETED from *nb_transport* at any point in order to complete the transaction early. Neither initiator nor target may make any further *nb_transport* calls for this particular transaction instance. Whether or not an initiator or target is actually permitted to shortcut a transaction in this way depends on the rules of the specific protocol.

In Figure 25, the timing annotation on the return path indicates to the initiator that the final timing point is to occur after the given delay. The phase transitions from BEGIN_REQ through END_REQ and BEGIN_RESP to END_RESP are implicit, rather than being passed explicitly as arguments to *nb_transport*.

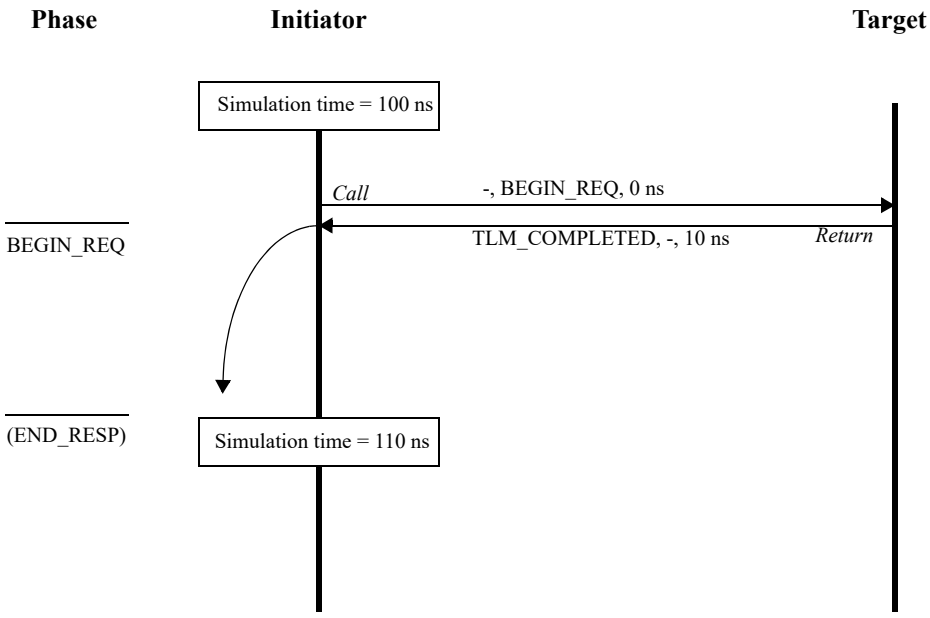

Figure 25—Early completion

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

11.2.3.12 Message sequence chart—timing annotation

A caller may annotate a delay onto an *nb_transport* call (Figure 26). This is an indication to the callee that the transaction should be processed *as if* it had been received after the given delay. An approximately-timed callee would typically handle this situation by putting the transaction into a payload event queue for processing when simulation time has *caught up* with the annotated delay. Depending on the implementation of the payload event queue, this processing may occur either in a SystemC process sensitive to an event notification from the payload event queue or in a callback registered with the payload event queue.

Delays can be annotated onto calls on the forward and backward paths and the corresponding return paths. An approximately-timed initiator would be expected to handle incoming transactions on both the forward return path and the backward path in the same way. Similarly, an approximately-timed target would be expected to handle incoming transactions on both the backward return path and the forward path in the same way.

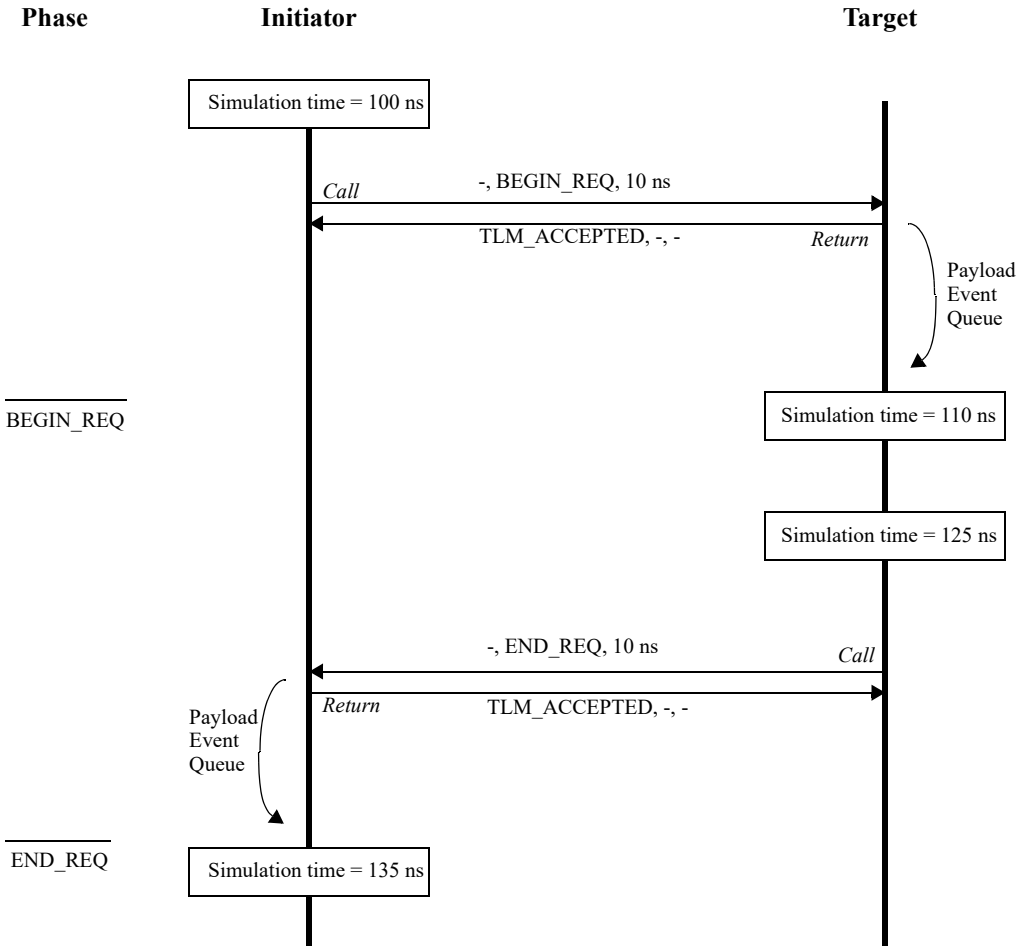

Figure 26—Timing annotation

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

11.2.4 Timing annotation with the transport interfaces

11.2.4.1 Overview

Timing annotation is a shared feature of the blocking and non-blocking transport interfaces, expressed using the **sc_time** argument to the member functions **b_transport**, **nb_transport_fw**, and **nb_transport_bw**. In this document, the italicized term *transport* is used to denote the three member functions **b_transport**, **nb_transport_fw**, and **nb_transport_bw**.

Transaction ordering is governed by a combination of the core interface rules and the protocol rules. The rules in the following clause apply to the core interfaces regardless of the choice of protocol. For the base protocol, the rules given here should be read in conjunction with those in 15.2.7.

11.2.4.2 The sc_time argument

a) It is recommended that the transaction object should not contain timing information. Any timing annotation should be expressed using the **sc_time** argument to *transport.*

b) The time argument shall be non-negative and shall be expressed relative to the current simulation time **sc_time_stamp()**.

c) The time argument shall apply on both the call to and the return from *transport* (subject to the rules of the **tlm_sync_enum** return value of *nb_transport*).

d) The *nb_transport* method may itself increase the value of the time argument but shall not decrease the value. The member function **b_transport** may increase the value of the time argument or may decrease the value in the case that it has called **wait** and thus synchronized with simulation time, but only by an amount that is less than or equal to the time for which the process was suspended. This rule is consistent with time not running backward in a SystemC simulation.

e) In the following description, the *recipient* of the transaction on the call to *transport* is the callee, and the *recipient* of the transaction on return from *transport* is the caller.

f) The *recipient* shall behave as if it had received the transaction at an effective local time of **sc_time_stamp() + t**, where t is the value of the time argument. In other words, the recipient shall behave as if the timing point associated with the interface method call is to occur at the effective local time.

g) Given a sequence of calls to *transport*, the effective local times at which the transactions are to be processed may or may not be in increasing time order in general. For transactions created by different initiators, it is fundamental to temporal decoupling that interface method call order may be different from effective local time order. The responsibility to handle transactions with out-of-order timing annotations lies with the *recipient*.

h) On receipt of a transaction with a non-zero timing annotation, any recipient always has choices that reflect the modeling trade-off between speed and accuracy. The recipient can execute any state changes caused by the transaction immediately and pass on the timing annotation, possibly increased, or it can schedule some internal process to resume after part or all of the annotated time has elapsed and execute the state changes only then. The choice is not enforced by the transport interface but may be documented as part of a protocol traits class or coding style.

i) If the recipient is not concerned with timing accuracy or with processing a sequence of incoming transactions in the order given by their timing annotations, it may process each transaction immediately, without delay. Having done so, the recipient may also choose to increase the value of the timing annotation to model the time needed to process the transaction. This scenario assumes that the system design can tolerate out-of-order execution because of the existence of some explicit mechanism (over and above the TLM-2.0 interfaces) to enforce the correct causal chain of events.

j) If the recipient is to implement an accurate model of timing and execution order, it should process the transaction at the correct time relative to any other SystemC processes with which it may

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

interact. In SystemC, the appropriate mechanism to schedule an event at a future time is the timed event notification. For convenience, TLM-2.0 provides a family of utility classes, known as payload event queues, which can be used to queue transactions for processing at the proper simulation time according to the natural semantics of SystemC (see 16.4). In other words, an approximately-timed recipient should typically put the transaction into a payload event queue.

k) Rather than processing the transaction directly, the recipient may pass the transaction on with a further call to or return from a *transport* method without modification to the transaction and using the same phase and timing annotation (or with an increased timing annotation).

l) With the loosely-timed coding style, transactions are typically executed immediately such that execution order matches interface method call order, and the member function **b_transport** is recommended.

m) With the approximately-timed coding style, transactions are typically delayed such that their execution order matches the effective local time order, and the *nb_transport* method is recommended.

n) Each component can make the above choice dynamically on a call-by-call basis. For example, a loosely-timed component may execute a series of transactions immediately in call order, passing on the timing annotations but may then choose to schedule the very next transaction for execution only after the delay given by the timing annotation has elapsed (known as *synchronization on demand*). This is a matter of coding style.

o) The above choice exists for both blocking and non-blocking transport. For example, **b_transport** may increase the timing annotation and return immediately or may **wait** for the timing annotation to elapse before returning. *nb_transport* may increase the timing annotation and return TLM_COMPLETED or may return TLM_ACCEPTED and schedule the transaction for execution later.

p) As a consequence of the above rules, if a component is the recipient of a series of transactions where the order of the incoming interface method calls is different from the effective local time order, the component is free to choose the mutual execution order of those particular transactions. The recommendation is to execute all transactions in call order or to execute all transactions in effective local time order, but this is not an obligation.

q) Note that the order in which incoming transactions are executed by a component should *in effect* always be the same as interface method call order because a component will either execute an incoming transaction before returning from the interface method call regardless of the timing annotation (loosely-timed), or will schedule the transaction for execution at the proper future time and return TLM_ACCEPTED (approximately-timed), thus indicating to the caller that it should wait before issuing the next transaction. (TLM_ACCEPTED alone does not forbid the caller from issuing the next transaction, but in the case of the base protocol, the request and response exclusion rules may do so.)

r) Timing annotation can also be described in terms of temporal decoupling. A non-zero timing annotation can be considered as an invitation to the recipient to “warp time.” The recipient can choose to enter a time warp, or it can put the transaction in a queue for later processing and yield. In a loosely-timed model, time warping is generally acceptable. On the other hand, if the target has dependencies on other asynchronous events, the target may have to wait for simulation time to advance before it can predict the future state of the transaction with certainty.

s) For a general description of temporal decoupling, see 10.3.3.

t) For a description of the quantum, see 16.3.

Example:

The following pseudo-code fragments illustrate just some of the many possible coding styles:

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
// ----------------------------------------------------------------------------
// Various interface method definitions
// ----------------------------------------------------------------------------

void b_transport(TRANS &trans, sc_core::sc_time &t) {
    // Loosely-timed coding style
    execute_transaction(trans);
    t = t + latency;
}

void b_transport(TRANS &trans, sc_core::sc_time &t) {
    // Loosely-timed with synchronization at the target or synchronization-on-demand
    wait(t);
    execute_transaction(trans);
    t = sc_core::SC_ZERO_TIME;
}

tlm::tlm_sync_enum nb_transport_fw(TRANS &trans, PHASE &phase, sc_core::sc_time &t) {
    // Pseudo-loosely-timed coding style using non-blocking transport (not recommended)
    execute_transaction(trans);
    t = t + latency;
    return tlm::TLM_COMPLETED;
}

tlm::tlm_sync_enum nb_transport_fw(TRANS &trans, PHASE &phase, sc_core::sc_time &t) {
    // Approximately-timed coding style
    // Post the transaction into a payload event queue for execution at time sc_time_stamp() + t
    payload_event_queue->notify(trans, phase, t);
    // Increment the transaction reference count
    trans.acquire();
    return tlm::TLM_ACCEPTED;
}

tlm::tlm_sync_enum nb_transport_fw(TRANS &trans, PHASE &phase, sc_core::sc_time &t) {
    // Approximately-timed coding style making use of the backward path
    payload_event_queue->notify(trans, phase, t);
    trans.acquire();
    // Modify the phase and time arguments
    phase = tlm::END_REQ;
    t = t + accept_delay;
    return tlm::TLM_UPDATED;
}

// ----------------------------------------------------------------------------
// b_transport interface method calls, loosely-timed coding style
// ----------------------------------------------------------------------------

initialize_transaction(T1);
socket->b_transport(T1, t);                 // t may increase
process_response(T1);

initialize_transaction(T2);
socket->b_transport(T2, t);                 // t may increase
process_response(T2);
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
// Initiator may sync after each transaction or after a series of transactions
quantum_keeper->set(t);
if (quantum_keeper->need_sync())
    quantum_keeper->sync();

// ---------------------------------------------------------------------------
// nb_transport interface method call, approximately-timed coding style
// ---------------------------------------------------------------------------

initialize_transaction(T3);
status = socket->nb_transport_fw(T3, phase, t);

if (status == tlm::TLM_ACCEPTED) {
    // No action, but expect an incoming nb_transport_bw method call
} else if (status == tlm::TLM_UPDATED) {          // Backward path is being used
    payload_event_queue->notify(T3, phase, t);
} else if (status == tlm::TLM_COMPLETED) {      // Early completion
    // Timing annotation may be taken into account in one of several ways
    // Either (1) by waiting, as shown here
    wait(t);
    process_response(T3);
    // or (2) by creating an event notification
    // response_event.notify( t );
    // or (3) by being passed on to the next transport method call (code not shown here)
}
```

11.2.5 Migration path from TLM-1

The old TLM-1 and the new TLM-2.0 interfaces are both part of the TLM-2.0 standard. The TLM-1 blocking and non-blocking interfaces are still useful in their own right. For example, a number of vendors have used these interfaces in building functional verification environments for HDL designs.

The intent is that the similarity between the old and new blocking transport interfaces should ease the task of building adapters between legacy models using the TLM-1 interfaces and the new TLM-2.0 interfaces.

11.3 Direct memory interface

11.3.1 Introduction

The Direct Memory Interface, or DMI, provides a means by which an initiator can get direct access to an area of memory owned by a target, thereafter accessing that memory using a direct pointer rather than through the transport interface. The DMI offers a large potential increase in simulation speed for memory access between initiator and target because once established it is able to bypass the normal path of multiple **b_transport** or *nb_transport* calls from initiator through interconnect components to target.

There are two direct memory interfaces, one for calls on the forward path from initiator to target, and a second for calls on the backward path from target to initiator. The forward path is used to request a particular mode of DMI access (e.g., read or write) to a given address, and returns a reference to a DMI descriptor of type **tlm_dmi**, which contains the bounds of the DMI region. The backward path is used by the target to invalidate DMI pointers previously established using the forward path. The forward and backward paths may pass through zero, one, or many interconnect components, but it should be identical to the forward and backward paths for the corresponding transport calls through the same sockets.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

A DMI pointer is requested by passing a transaction along the forward path. The default DMI transaction type is **tlm_generic_payload**, where only the command and address attributes of the transaction object are used. DMI follows the same approach to extension as the transport interface; that is, a DMI request may contain ignorable extensions, but any non-ignorable or mandatory extension requires the definition of a new protocol traits class (see 14.2.3).

The DMI descriptor returns latency values for use by the initiator and so provides sufficient timing accuracy for loosely-timed modeling.

DMI pointers may be used for debug, but the debug transport interface itself is usually sufficient because debug traffic is usually light, and usually dominated by I/O rather than memory access. Debug transactions are not usually on the critical path for simulation speed. If DMI pointers were used for debug, the latency values should be ignored.

11.3.2 Class definition

```
namespace tlm {

class tlm_dmi
{
public:
    tlm_dmi() { init(); }

    void init();

    enum dmi_access_e {
        DMI_ACCESS_NONE= 0x00,
        DMI_ACCESS_READ = 0x01,
        DMI_ACCESS_WRITE= 0x02,
        DMI_ACCESS_READ_WRITE  = DMI_ACCESS_READ | DMI_ACCESS_WRITE
    };

    unsigned char* get_dmi_ptr() const;
    sc_dt::uint64 get_start_address() const;
    sc_dt::uint64 get_end_address() const;
    sc_core::sc_time get_read_latency() const;
    sc_core::sc_time get_write_latency() const;
    dmi_access_e get_granted_access() const;
    bool is_none_allowed() const;
    bool is_read_allowed() const;
    bool is_write_allowed() const;
    bool is_read_write_allowed() const;

    void set_dmi_ptr(unsigned char* p);
    void set_start_address(sc_dt::uint64 addr);
    void set_end_address(sc_dt::uint64 addr);
    void set_read_latency(sc_core::sc_time t);
    void set_write_latency(sc_core::sc_time t);
    void set_granted_access(dmi_access_e t);
    void allow_none();
    void allow_read();
    void allow_write();
    void allow_read_write();
};
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
template <typename TRANS = tlm_generic_payload>
class tlm_fw_direct_mem_if : public virtual sc_core::sc_interface
{
public:
    virtual bool get_direct_mem_ptr(TRANS& trans, tlm_dmi& dmi_data) = 0;
};

class tlm_bw_direct_mem_if : public virtual sc_core::sc_interface
{
public:
    virtual void invalidate_direct_mem_ptr(sc_dt::uint64 start_range, sc_dt::uint64 end_range) = 0;
};

} // namespace tlm
```

11.3.3 get_direct_mem_ptr

a) The member function **get_direct_mem_ptr** shall only be called by an initiator or by an interconnect component, not by a target.

b) The **trans** argument shall pass a reference to a DMI transaction object constructed by the initiator.

c) The **dmi_data** argument shall be a reference to a DMI descriptor constructed by the initiator.

d) Any interconnect component should pass the **get_direct_mem_ptr** call along the forward path from initiator to target. In other words, the implementation of **get_direct_mem_ptr** for the target socket of the interconnect component may call the member function **get_direct_mem_ptr** of an initiator socket.

e) Each **get_direct_mem_ptr** call shall follow exactly the same path from initiator to target as a corresponding set of transport calls. In other words, each DMI request shall involve an interaction between one initiator and one target, where those two components also serve the role of initiator and target for a single transaction object passed through the transport interface. DMI cannot be used on a path through a component that initiates a second transaction object, such as a non-trivial width converter. (If DMI is an absolute requirement for simulation speed, the simulation model may need to be restructured to permit it.)

f) Any interconnect components shall pass on the **trans** and **dmi_data** arguments in the forward direction, the only permitted modification being to the value of the address attribute of the DMI transaction object as described below. The address attribute of the transaction and the DMI descriptor may both be modified on return from the member function **get_direct_mem_ptr**, that is, when unwinding the function calls from target back to initiator.

g) If the target is able to support DMI access to the given address, it shall set the members of the DMI descriptor as described below and set the return value of the function to **true**. When a target grants DMI access, the DMI descriptor is used to indicate the details of the access being granted.

h) If the target is not able to support DMI access to the given address, it need set only the address range and type members of the DMI descriptor as described below and set the return value of the function to **false**. When a target denies DMI access, the DMI descriptor is used to indicate the details of the access being denied.

i) A target may grant or deny DMI access to any part or parts of its memory region, including non-contiguous regions, subject to the rules given in this clause.

j) In the case that a target has granted DMI access and has set the return value of the function to **true**, an interconnect component may deny DMI access by setting the return value of the function to **false** on return from the member function **get_direct_mem_ptr**. The reverse is not permitted; in the case that a target has denied DMI access, an interconnect component shall not grant DMI access.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

k) Given multiple calls to **get_direct_mem_ptr**, a target may grant DMI access to multiple initiators for the same memory region at the same time. The application is responsible for synchronization and coherency.

l) Since each call to **get_direct_mem_ptr** can only return a single DMI pointer to a contiguous memory region, each DMI request can only be fulfilled by a single target in practice. In other words, if a memory region is scattered across multiple targets, then even though the address range is contiguous, each target will likely require a separate DMI request.

m) If read or write access to a certain region of memory causes side effects in a target (that is, causes some other change to the state of the target over and above the state of the memory), the target should not grant DMI access of the given type to that memory region. But if, for example, only write access causes side effects in a target, the target may still grant DMI read access to a given region.

n) The implementation of **get_direct_mem_ptr** may call **invalidate_direct_mem_ptr**.

o) The implementation of **get_direct_mem_ptr** shall not call **wait**, directly or indirectly.

11.3.4 template argument and tlm_generic_payload class

a) The **tlm_fw_direct_mem_if** template shall be parameterized with the type of a DMI transaction class.

b) The transaction object shall contain attributes to indicate the address for which direct memory access is requested and the type of access requested, namely read access or write access to the given address. In the case of the base protocol, these shall be the command and address attributes of the generic payload.

c) The default value of the TRANS template argument shall be the class **tlm_generic_payload**.

d) For maximal interoperability, the DMI transaction class should be the **tlm_generic_payload** class. The use of non-ignorable extensions or other transaction types will restrict interoperability.

e) The initiator shall be responsible for constructing and managing the DMI transaction object, and for setting the appropriate attributes of the object before passing it as an argument to **get_direct_mem_ptr**.

f) The command attribute of the transaction object shall be set by the initiator to indicate the kind of DMI access being requested, and shall not be modified by any interconnect component or target. For the base protocol, the permitted values are TLM_READ_COMMAND for read access, and TLM_WRITE_COMMAND for write access.

g) For the base protocol, the command attribute is forbidden from having the value TLM_IGNORE_COMMAND. However, this value may be used by other protocols.

h) The address attribute of the transaction object shall be set by the initiator to indicate the address for which direct memory access is being requested.

i) An interconnect component passing the DMI transaction object along the forward path should decode and where necessary modify the address attribute of the transaction exactly as it would for the corresponding transport interface of the same socket. For example, an interconnect component may need to mask the address (reducing the number of significant bits) according to the address width of the target and its location in the system memory map.

j) An interconnect component need not pass on the **get_direct_mem_ptr** call in the event that it detects an addressing error.

k) In the case of the base protocol, if the generic payload option attribute is TLM_MIN_PAYLOAD, the initiator is not obliged to set any other attributes of the generic payload aside from command and address, and the target and any interconnect components may ignore all other attributes. In particular, the response status attribute and the DMI allowed attribute may be ignored. If the target sets the value of the generic payload option attribute to TLM_FULL_PAYLOAD_ACCEPTED, the target shall set the response status attribute and the initiator should check the response status as

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

described in 14.8. The DMI allowed attribute is only intended for use with the transport and debug transport interfaces.

l) The initiator may re-use a transaction object from one DMI call to the next and across calls to DMI, the transport interfaces, and the debug transport interface.

m) If an application needs to add further attributes to a DMI transaction object for use by the target when determining the kind of DMI access being requested, the recommended approach is to add extensions to the generic payload rather than substituting an unrelated transaction class. For example, the DMI transaction might include a CPU ID to allow the target to service DMI requests differently depending on the kind of CPU making the request. In the case that such extensions are non-ignorable, this will require the definition of a new protocol traits class.

11.3.5 tlm_dmi class

a) A DMI descriptor is an object of class **tlm_dmi**. DMI descriptors shall be constructed by initiators, but their members may be set by interconnect components or targets.

b) A DMI descriptor shall have the following attributes: the DMI pointer attribute, the granted access type attribute, the start address attribute, the end address attribute, the read latency attribute, and the write latency attribute. The default values of these attributes shall be as follows: DMI pointer attribute = 0, access type = DMI_ACCESS_NONE, start address = 0, end address = the maximum value of type **sc_dt::uint64**, read latency = SC_ZERO_TIME, and write latency = SC_ZERO_TIME.

c) Member function **init** shall initialize the members of the DMI descriptor to their default values.

d) A DMI descriptor shall be in its default state whenever it is passed as an argument to **get_direct_mem_ptr** by the initiator. If DMI descriptor objects are pooled, the initiator shall reset the DMI descriptor to its default state before passing it as an argument to **get_direct_mem_ptr**. Member function **init** may be called for this purpose.

e) Since an interconnect component is not permitted to modify the DMI descriptor as it is passed on toward the target, the DMI descriptor shall be in its initial state when it is received by the target.

f) Member function **set_dmi_ptr** shall set the DMI pointer attribute to the value passed as an argument. Member function **get_dmi_ptr** shall return the current value of the DMI pointer attribute.

g) The DMI pointer attribute shall be set by the target to point to the storage location corresponding to the value of the start address attribute. This shall be less than or equal to the address requested in the call to **get_direct_mem_ptr**. The initial value shall be 0.

h) The storage in the DMI region is represented with type **unsigned char***. The storage shall have the same organization as the data array of the generic payload. If a target is unable to return a pointer to a memory region with that organization, the target is unable to support DMI and **get_direct_mem_ptr** should return the value false. For a full description of how memory organization and endianness are handled in TLM-2.0, see 14.18.

i) An interconnect component is permitted to modify the DMI pointer attribute on the return path from the **get_direct_mem_ptr** function call in order to restrict the region to which DMI access is being granted.

j) Member function **set_granted_access** shall set the granted access type attribute to the value passed as an argument. Member function **get_granted_access** shall return the current value of the granted access type attribute. The initial value shall be DMI_ACCESS_NONE.

k) Member functions **allow_none**, **allow_read**, **allow_write**, and **allow_read_write** shall set the granted access type attribute to the value DMI_ACCESS_NONE, DMI_ACCESS_READ, DMI_ACCESS_WRITE, or DMI_ACCESS_READ_WRITE, respectively.

l) Member function **is_none_allowed** shall return true if and only if the granted access type attribute has the value DMI_ACCESS_NONE. Member function **is_read_allowed** shall return true if and only if the granted access type attribute has the value DMI_ACCESS_READ or

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

DMI_ACCESS_READ_WRITE. Member function **is_write_allowed** shall return true if and only if the granted access type attribute has the value DMI_ACCESS_WRITE or DMI_ACCESS_READ_WRITE. Member function **is_read_write_allowed** shall return true if and only if the granted access type attribute has the value DMI_ACCESS_READ_WRITE.

m) The target shall set the granted access type attribute to the type of access being granted or being denied. A target is permitted to respond to a request for read access by granting (or denying) read or read/write access, and to a request for write access by granting (or denying) write or read/write access. An interconnect component is permitted to restrict the granted access type by overwriting a value of DMI_ACCESS_READ_WRITE with DMI_ACCESS_READ or DMI_ACCESS_WRITE on the return path from the **get_direct_mem_ptr** function call.

n) A target wishing to deny read and write access to the DMI region should set the granted access type to DMI_ACCESS_READ_WRITE, not to DMI_ACCESS_NONE.

Example:

```
bool get_direct_mem_ptr( TRANS& trans, tlm::tlm_dmi& dmi_data ) {
  // Deny DMI access to entire memory region
  dmi_data.allow_read_write();
  dmi_data.set_start_address( 0x0 );
  dmi_data.set_end_address( (sc_dt::uint64)-1 );
  return false;
}
```

o) The target should set the granted access type to DMI_ACCESS_NONE to indicate that it is not granting (or denying) read, write, or read/write access to the initiator, but it is granting (or denying) some other kind of access as requested by an extension to the DMI transaction object. This value should only be used in cases where an extension to the DMI transaction object makes the pre-defined access types read, write, and read/write unnecessary or meaningless. This value should not be used in the case of the base protocol.

p) The initiator is responsible for using only those modes of DMI access that have been granted by the target (and possibly modified by the interconnect) using the granted access type attribute (or in cases other than the base protocol, granted using extensions to the generic payload or using other DMI transaction types).

q) Member functions **set_start_address** and **set_end_address** shall set the start and end address attributes, respectively, to the values passed as arguments. Member functions **get_start_address** and **get_end_address** shall return the current values of the start and end address attributes, respectively.

r) The start and end address attributes shall be set by the target (or modified by the interconnect) to point to the addresses of the first and the last bytes in the DMI region. The DMI region is either being granted or being denied, as determined by the value returned from the member function **get_direct_mem_ptr** (**true** or **false**). A target wishing to deny access to its entire memory region may set the start address to **0** and the end address to the maximum value of type **sc_dt::uint64**.

s) A target can only grant or deny a single contiguous memory region for each **get_direct_mem_ptr** call. A target can set the DMI region to a single address by having the start and end address attributes be equal or can set the DMI region to be arbitrarily large.

t) Having been granted DMI access of a given type to a given region, an initiator may perform access of the given type anywhere in that region until it is invalidated. In other words, access is not restricted to the address given in the DMI request.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

u) Any interconnect components that pass on the **get_direct_mem_ptr** call are obliged to transform the start and end address attributes as they do the address argument. Any transformations on the addresses in the DMI descriptor shall occur as the descriptor is passed along the return path from the **get_direct_mem_ptr** function call. For example, the target may set the start address attribute to a relative address within the memory map known to that target, in which case the interconnect component is obliged to transform the relative address back to an absolute address in the system memory map. The initial values shall be 0 and the maximum value of type **sc_dt::uint64**, respectively.

v) An interconnect component is permitted to modify the start and end address attributes in order to restrict the region to which DMI access is being granted, or expand the range to which DMI access is being denied.

w) If **get_direct_mem_ptr** returns the value **true**, the DMI region indicated by the start and end address attributes is a region for which DMI access is allowed. On the other hand, if **get_direct_mem_ptr** return the value **false**, it is a region for which DMI access is disallowed.

x) A target or interconnect component receiving two or more calls to **get_direct_mem_ptr** may return two or more overlapping allowed DMI regions or two or more overlapping disallowed DMI regions.

y) A target or interconnect component shall not return overlapping DMI regions where one region is allowed, and the other is disallowed for the same access type, for example, both read or read/write or both write or read/write, without making an intervening call to **invalidate_direct_mem_ptr** to invalidate the first region.

z) In other words, the definition of the DMI regions shall not be dependent on the order in which they were created unless the first region is invalidated by an intervening call to **invalidate_direct_mem_ptr**. Specifically, the creation of a disallowed DMI region shall not be permitted to punch a hole in an existing allowed DMI region for the same access type, or vice versa.

aa) A target may disallow DMI access to the entire address space (start address attribute = 0, end address attribute = maximum value), perhaps because the target does not support DMI access at all, in which case an interconnect component should clip this disallowed region down to the part of the memory map occupied by the target. Otherwise, if an interconnect component fails to clip the address range, then an initiator would be misled into thinking that DMI was disallowed across the entire system address space.

ab) Member functions **set_read_latency** and **set_write_latency** shall set the read and write latency attributes, respectively, to the values passed as arguments. Member functions **get_read_latency** and **get_write_latency** shall return the current values of the read and write latency attributes, respectively.

ac) The read and write latency attributes shall be set to the average latency per byte for read and write memory transactions, respectively. In other words, the initiator performing the direct memory operation shall calculate the actual latency by multiplying the read or write latency from the DMI descriptor by the number of bytes that would have been transferred by the equivalent *transport* transaction. The initial values shall be SC_ZERO_TIME. Both interconnect components and the target may increase the value of either latency such that the latency accumulates as the DMI descriptor is passed back from target to initiator on return from the member function **get_direct_mem_ptr**. One or both latencies will be valid, depending on the value of the granted access type attribute.

ad) The initiator is responsible for respecting the latencies whenever it accesses memory using the direct memory pointer. If the initiator chooses to ignore the latencies, this may result in timing inaccuracies.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

11.3.6 invalidate_direct_mem_ptr

a) The member function **invalidate_direct_mem_ptr** shall only be called by a target or an interconnect component.

b) A target is obliged to call **invalidate_direct_mem_ptr** before any change that would modify the validity or the access type of any existing DMI region. For example, before restricting the address range of an existing DMI region, before changing the access type from read/write to read, or before re-mapping the address space.

c) The **start_range** and **end_range** arguments shall be the first and last addresses of the address range for which DMI access is to be invalidated.

d) An initiator receiving an incoming call to **invalidate_direct_mem_ptr** shall immediately invalidate and discard any DMI region (previously received from a call to **get_direct_mem_ptr**) that overlaps with the given address range.

e) In the case of a partial overlap, that is, only part of an existing DMI region is invalidated, an initiator may adjust the boundaries of the existing region or may invalidate the entire region.

f) Each DMI region shall remain valid until it is explicitly invalidated by a target using a call to **invalidate_direct_mem_ptr**. Each initiator may maintain a table of valid DMI regions and may continue to use each region until it is invalidated.

g) Any interconnect components are obliged to pass on the **invalidate_direct_mem_ptr** call along the backward path from target to initiator, decoding and where necessary modifying the address arguments as they would for the corresponding transport interface. Because the transport interface transforms the address on the forward path and DMI on the backward path, the transport and DMI transformations should be the inverse of one another.

h) Given a single **invalidate_direct_mem_ptr** call from a target, an interconnect component may make multiple **invalidate_direct_mem_ptr** calls to initiators. Since there may be multiple initiators each getting direct memory pointers to the same target, a safe implementation is for an interconnect component to call **invalidate_direct_mem_ptr** for every initiator.

i) An interconnect component can invalidate all direct memory pointers in an initiator by setting **start_range** to 0 and **end_range** to the maximum value of the type **sc_dt::uint64**.

j) The implementation of any TLM-2.0 core interface method may call **invalidate_direct_mem_ptr**.

k) The implementation of **invalidate_direct_mem_ptr** shall not call **get_direct_mem_ptr**, directly or indirectly.

l) The implementation of **invalidate_direct_mem_ptr** shall not call **wait**, directly or indirectly.

11.3.7 DMI versus transport

a) By definition, the direct memory interface provides a direct interface between initiator and target that bypasses any interconnect components. The transport interfaces, on the other hand, cannot bypass interconnect components.

b) Care must be taken with interconnect components retaining state or having side effects, such as buffered interconnects or interconnects modeling cache memory. The transport interfaces may access and update the state of the interconnect component, whereas the direct memory interface will bypass the interconnect component. The safest alternative is for such interconnect components always to deny DMI access.

c) It is possible for an initiator to switch back and forth between calling the transport interfaces and using a direct memory pointer. It is also possible that one initiator may use DMI, while another initiator is using the transport interfaces. In this situation, care must be taken, particularly considering that transport calls may carry a timing annotation. This is the responsibility of the application. For example, a given target could support DMI and transport simultaneously or could invalidate every DMI pointer whenever transport is called.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

11.3.8 DMI and temporal decoupling

a) A DMI region can only be invalidated by means of a target or interconnect component making a call to **invalidate_direct_mem_ptr**.

b) An initiator is responsible for checking that a DMI region is still valid before using the associated DMI pointer, subject to the following considerations.

c) Due to the co-routine semantics of SystemC, once an initiator has started running, no other SystemC process will be able to run until the initiator yields. In particular, no other SystemC process would be able to invalidate a DMI pointer (although the current process might). As a consequence, a temporally decoupled initiator does not necessarily need to check repeatedly that a given DMI region is still valid each time it uses the associated DMI pointer.

d) It is possible that an interface method call made from an initiator may cause another component to call **invalidate_direct_mem_ptr**, thus invalidating a DMI region being used by that initiator. This could be true of a temporally decoupled initiator that runs without yielding.

e) While an initiator is running without interacting with any other components and without yielding, any valid DMI region will remain valid.

f) It is possible that after a temporally decoupled initiator using DMI has yielded, another temporally decoupled initiator may cause that same DMI region to be invalidated within the current time quantum. This reflects the fundamental inaccuracy intrinsic to temporal decoupling in general but does not represent a violation of the rules given in this clause.

11.3.9 Optimization using a DMI hint

a) The DMI hint, otherwise known as the DMI allowed attribute, is a mechanism to optimize simulation speed by avoiding the need to poll repeatedly for DMI access. Instead of calling **get_direct_mem_ptr** to check for the availability of a DMI pointer, an initiator can check the DMI allowed attribute of a normal transaction passed through the transport interface.

b) The generic payload provides a DMI allowed attribute. User-defined transactions could implement a similar mechanism, in which case the target should set the value of the DMI allowed attribute appropriately.

c) Use of the DMI allowed attribute is optional. An initiator is free to ignore the DMI allowed attribute of the generic payload.

d) For an initiator wishing to take advantage of the DMI allowed attribute, the recommended sequence of actions is as follows:

   1) The initiator should check the address against its cache of valid DMI regions.
   2) If there is no existing DMI pointer, the initiator should perform a normal transaction through the transport interface.
   3) Following that, the initiator should check the DMI allowed attribute of the transaction.
   4) If the attribute indicates DMI is allowed, the initiator should call **get_direct_mem_ptr**.
   5) The initiator should modify its cache of valid DMI regions according to the values returned from the call.

11.4 Debug transport interface

11.4.1 Introduction

The debug transport interface provides a means to read and write to storage in a target, over the same forward path from initiator to target as is used by the transport interface, but without any of the delays, waits, event notifications, or side effects associated with a regular transaction. In other words, the debug transport interface is non-intrusive. Because the debug transport interface follows the same path as the transport

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

interface, the implementation of the debug transport interface can perform the same address translation as for regular transactions.

For example, the debug transport interface could permit a software debugger attached to an ISS to peek or poke an address in the memory of the simulated system from the point of view of the simulated CPU. The debug transport interface could also allow an initiator to take a snapshot of system memory contents during simulation for diagnostic purposes, or to initialize some area of system memory at the end of elaboration.

The default debug transaction type is **tlm_generic_payload**, where only the command, address, data length, and data pointer attributes of the transaction object are used. Debug transactions follow the same approach to extension as the transport interface; that is, a debug transaction may contain ignorable extensions, but any non-ignorable or mandatory extension requires the definition of a new protocol traits class (see 7.2.3).

11.4.2 Class definition

```
namespace tlm {

template <typename TRANS = tlm_generic_payload>
class tlm_transport_dbg_if : public virtual sc_core::sc_interface
{
public:
  virtual unsigned int transport_dbg(TRANS& trans) = 0;
};

} // namespace tlm
```

11.4.3 TRANS template argument and tlm_generic_payload class

a) The **tlm_transport_dbg_if** template shall be parameterized with the type of a debug transaction class.

b) The debug transaction class shall contain attributes to indicate to the target the command, address, data length, and date pointer for the debug access. In the case of the base protocol, these shall be the corresponding attributes of the generic payload.

c) The default value of the TRANS template argument shall be the class **tlm_generic_payload**.

d) For maximal interoperability, the debug transaction class should be **tlm_generic_payload**. The use of non-ignorable extensions or other transaction types will restrict interoperability.

e) If an application needs to add further attributes to a debug transaction, the recommended approach is to add extensions to the generic payload rather than substituting an unrelated transaction class. In the case that such extensions are non-ignorable or mandatory, this will require the definition of a new protocol traits class.

11.4.4 Rules

a) Calls to **transport_dbg** shall follow the same forward path as the transport interface used for normal transactions.

b) The **trans** argument shall pass a reference to a debug transaction object.

c) The initiator shall be responsible for constructing and managing the debug transaction object, and for setting the appropriate attributes of the object before passing it as an argument to **transport_dbg**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

d) The command attribute of the transaction object shall be set by the initiator to indicate the kind of debug access being requested, and shall not be modified by any interconnect component or target. For the base protocol, the permitted values are TLM_READ_COMMAND for read access to the target, TLM_WRITE_COMMAND for write access to the target, and TLM_IGNORE_COMMAND.

e) On receipt of a transaction with the command attribute equal to TLM_IGNORE_COMMAND, the target should not execute a read or a write, but may use the value of any attribute in the generic payload, including any extensions, in executing an extended debug transaction.

f) As is the case for the transport interface, the use of any non-ignorable or mandatory generic payload extension with the debug transport interface requires the definition of a new protocol traits class.

g) The address attribute shall be set by the initiator to the first address in the region to be read or written.

h) An interconnect component passing the debug transaction object along the forward path should decode and where necessary modify the address attribute of the transaction object exactly as it would for the corresponding transport interface of the same socket. For example, an interconnect component may need to mask the address (reducing the number of significant bits) according to the address width of the target and its location in the system memory map.

i) An interconnect component need not pass on the **transport_dbg** call in the event that it detects an addressing error.

j) The address attribute may be modified several times if a debug payload is forwarded through several interconnect components. When the debug payload is returned to the initiator, the original value of the address attribute may have been overwritten.

k) The data length attribute shall be set by the initiator to the number of bytes to be read or written and shall not be modified by any interconnect component or target. The data length attribute may be 0, in which case the target shall not read or write any bytes, and the data pointer attribute may be null.

l) The data pointer attribute shall be set by the initiator to the address of an array from which values are to be copied to the target (for a write), or to which values are to be copied from the target (for a read), and shall not be modified by any interconnect component or target. This array shall be allocated by the initiator and shall not be deleted before the return from **transport_dbg**. The size of the array shall be at least equal to the value of the data length attribute. If the data length attribute is 0, the data pointer attribute may be the null pointer and the array need not be allocated.

m) The implementation of **transport_dbg** in the target shall read or write the given number of bytes using the given address (after address translation through the interconnect), if it is able. In the case of a write command, the target shall not modify the contents of the data array.

n) The data array shall have the same organization as the data array of the generic payload when used with the transport interface. The implementation of **transport_dbg** shall be responsible for converting between the organization of the local data storage within the target and the generic payload organization.

o) In the case of the base protocol, if the generic payload option attribute is TLM_MIN_PAYLOAD, the initiator is not obliged to set any other attributes of the generic payload aside from command, address, data length and data pointer, and the target and any interconnect components may ignore all other attributes. In particular, the response status attribute may be ignored.

p) In the case of the base protocol, if the initiator sets the value of the generic payload option attribute to TLM_FULL_PAYLOAD, the initiator shall set the values of the byte enable pointer, byte enable length, and streaming width attributes, and shall set the *DMI allowed* and response status attributes to their default values as described in 14.8.

q) In the case of the base protocol, if the target sets the value of the generic payload option attribute to TLM_FULL_PAYLOAD_ACCEPTED, the target shall act on the values of the byte enable pointer, byte enable length, and streaming width attributes, and shall set the *DMI allowed* and response status attributes as described in 14.8.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

r) The initiator may re-use a transaction object from one call to the next and across calls to the debug transport interface, the transport interfaces, and DMI.

s) **transport_dbg** shall return a count of the number of bytes actually read or written, which may be less than the value of the data length attribute. In the case of the base protocol, if the initiator sets the value of the generic payload option attribute to TLM_FULL_PAYLOAD, the count shall include both enabled and disabled bytes. If the target is not able to perform the operation, it shall return a value of 0. In the case of TLM_IGNORE_COMMAND, the target is free to choose the value returned from **transport_dbg**.

t) Directly or indirectly, **transport_dbg** shall not call **wait**, and should not cause any state changes in any interconnect component or in the target aside from the immediate effect of executing a debug write command.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

12. TLM-2.0 global quantum

12.1 Introduction

Temporal decoupling permits SystemC processes to run ahead of simulation time for an amount of time known as the time quantum and is associated with the loosely-timed coding style. Temporal decoupling permits a significant simulation speed improvement by reducing the number of context switches and events. The use of a time quantum is not strictly necessary in the presence of explicit synchronization between temporally decoupled processes, in which case processes may run arbitrarily far ahead to the point when the next synchronization point is reached. However, any processes that do require a time quantum should use the global quantum.

When using temporal decoupling, the delays annotated to the member function **b_transport** and *nb_transport* methods are to be interpreted as local time offsets defined relative to the current simulation time as returned by **sc_time_stamp()**, also known as the quantum boundary. The global quantum is the default time interval between successive quantum boundaries. The value of the global time quantum is maintained by the singleton class **tlm_global_quantum**. It is recommended that each process should use the global time quantum, but a process is permitted to calculate its own local time quantum.

For a general description of temporal decoupling, see 10.3.3.

For a description of timing annotation, see 11.2.4.

The utility class **tlm_quantumkeeper** provides a set of member functions for managing and interacting with the time quantum. For a description of how to use a quantum keeper, see 16.3.

12.2 Header file

The class definition for the global quantum shall be in the header file **tlm**.

12.3 Class definition

```
namespace tlm {

class tlm_global_quantum
{
public:
    static tlm_global_quantum& instance();
    virtual ~tlm_global_quantum();
    void set( const sc_core::sc_time& );
    const sc_core::sc_time& get() const;
    sc_core::sc_time compute_local_quantum();

protected:
    tlm_global_quantum();
};

} // namespace tlm
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

12.4 tlm_global_quantum

a) There is a unique global quantum maintained by the class **tlm_global_quantum**. This should be considered the default time quantum. The intent is that all temporally decoupled initiators should synchronize on integer multiples of the global quantum, or more frequently where required.

b) It is possible for each initiator to use a different time quantum but more typical for all initiators to use the global quantum. An initiator that only requires infrequent synchronization could conceivably have a longer time quantum than the rest, but it is usually the shortest time quantum that has the biggest negative impact on simulation speed.

c) Member function **instance** shall return a reference to the singleton global quantum object.

d) Member function **set** shall set the value of the global quantum to the value passed as an argument.

e) Member function **get** shall return the value of the global quantum.

f) Member function **compute_local_quantum** shall calculate and return the value of the local quantum based on the unique global quantum. The local quantum shall be calculated by subtracting the value of **sc_time_stamp** from the next larger integer multiple of the global quantum. The local quantum shall equal the global quantum in the case where **compute_local_quantum** is called at a simulation time that is an integer multiple of the global quantum. Otherwise, the local quantum shall be less than the global quantum.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

13. Combined TLM-2.0 interfaces and sockets

13.1 Combined interfaces

13.1.1 Introduction

The combined forward and backward transport interfaces group the core TLM-2.0 interfaces for use by the initiator and target sockets. Note that the combined interfaces include the transport, DMI, and debug transport interfaces, but they do not include any TLM-1 core interfaces. The forward interface provides method calls on the forward path from initiator socket to target socket, and the backward interface on the backward path from target socket to initiator socket. Neither the blocking transport interface nor the debug transport interface require a backward calling path.

It would be technically possible to define new socket class templates unrelated to the standard initiator and target sockets and then to instantiate those class templates using the combined interfaces as template arguments, but for the sake of interoperability, this is not recommended. On the other hand, deriving new socket classes from the standard sockets is recommended for convenience.

The combined interface templates are parameterized with a *protocol traits* class that defines the types used by the forward and backward interfaces, namely, the payload type and the phase type. A protocol traits class is associated with a specific protocol. The default protocol type is the class **tlm_base_protocol_types** (see 15.2).

13.1.2 Class definition

```
namespace tlm {

// The default protocol traits class:
struct tlm_base_protocol_types
{
    typedef  tlm_generic_payload     tlm_payload_type;
    typedef  tlm_phase               tlm_phase_type;
};

// The combined forward interface:
template< typename TYPES = tlm_base_protocol_types >
class tlm_fw_transport_if
    : public virtual tlm_fw_nonblocking_transport_if<typename TYPES::tlm_payload_type,
                                                     typename TYPES::tlm_phase_type>
    , public virtual tlm_blocking_transport_if<      typename TYPES::tlm_payload_type>
    , public virtual tlm_fw_direct_mem_if    <       typename TYPES::tlm_payload_type>
    , public virtual tlm_transport_dbg_if<           typename TYPES::tlm_payload_type>
{};

// The combined backward interface:
template < typename TYPES = tlm_base_protocol_types >
class tlm_bw_transport_if
    : public virtual tlm_bw_nonblocking_transport_if<      typename TYPES::tlm_payload_type,
                                                           typename TYPES::tlm_phase_type >
    , public virtual tlm_bw_direct_mem_if
{};

} // namespace tlm
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

13.2 Initiator and target sockets

13.2.1 Introduction

A socket combines a port with an export. An initiator socket has a port for the forward path and an export for the backward path, while a target socket has an export for the forward path and a port for the backward path. The sockets also overload the SystemC port binding operators to bind both the port and export to the export and port in the opposing socket. When binding sockets hierarchically, parent to child or child to parent, it is important to carefully consider the binding order.

Both the initiator and target sockets are coded using a C++ inheritance hierarchy. Only the most derived classes **tlm_initiator_socket** and **tlm_target_socket** are typically used directly by applications. These two sockets are parameterized with a *protocol traits* class that defines the types used by the forward and backward interfaces. Sockets can only be bound together if they have the identical protocol type. The default protocol type is the class **tlm_base_protocol_types**. If an application defines a new protocol, it should instantiate combined interface templates with a new protocol traits class, whether or not the new protocol is based on the generic payload.

The initiator and target sockets provide the following benefits:

a) They group the transport, direct memory, and debug transport interfaces for both the forward and backward paths together into a single object.
b) They provide methods to bind port and export of both the forward and backward paths in a single call.
c) They offer strong type checking when binding sockets parameterized with incompatible protocol types.
d) They include a bus width parameter that may be used to interpret the transaction.

The socket classes **tlm_initiator_socket** and **tlm_target_socket** belong to the interoperability layer of the TLM-2.0 standard. In addition, there is a family of derived socket classes provided in the utilities namespace, collectively known as *convenience sockets*.

13.2.2 Class definition

```
namespace tlm {

enum tlm_socket_category
{
    TLM_UNKNOWN_SOCKET,
    TLM_INITIATOR_SOCKET,
    TLM_TARGET_SOCKET
};

class tlm_base_socket_if
{
public:
    virtual sc_core::sc_port_base &              get_base_port() = 0;
    virtual sc_core::sc_port_base const &        get_base_port() const = 0;
    virtual sc_core::sc_export_base &            get_base_export() = 0;
    virtual sc_core::sc_export_base const &      get_base_export() const = 0;
    virtual unsigned int                         get_bus_width() const = 0;
    virtual std::type_index                      get_protocol_types() const = 0;
    virtual tlm_socket_category                  get_socket_category() const = 0;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    virtual bool                                    is_multi_socket() const = 0;

protected:
    virtual ~tlm_base_socket_if() {}
};

// Abstract base class for initiator sockets
template <
    unsigned int BUSWIDTH = 32,
    typename FW_IF = tlm_fw_transport_if<>,
    typename BW_IF = tlm_bw_transport_if<>
>
class tlm_base_initiator_socket_b : public tlm_base_socket_if
{
public:
    virtual ~tlm_base_initiator_socket_b() {}

    virtual sc_core::sc_port_b<FW_IF> &          get_base_port() = 0;
    virtual const sc_core::sc_port_b<FW_IF> &    get_base_port() const = 0;

    virtual  BW_IF  &                            get_base_interface() = 0;
    virtual const BW_IF &                        get_base_interface() const = 0;

    virtual sc_core::sc_export<BW_IF> &          get_base_export() = 0;
    virtual const sc_core::sc_export<BW_IF> &    get_base_export() const = 0;
};

// Abstract base class for target sockets
template <
    unsigned int BUSWIDTH = 32,
    typename FW_IF = tlm_fw_transport_if<>,
    typename BW_IF = tlm_bw_transport_if<>
>
class tlm_base_target_socket_b : public tlm_base_socket_if
{
public:
    virtual ~tlm_base_target_socket_b();

    virtual sc_core::sc_port_b<BW_IF> &          get_base_port() = 0;
    virtual const sc_core::sc_port_b<BW_IF> &    get_base_port() const = 0;

    virtual sc_core::sc_export<FW_IF> &          get_base_export() = 0;
    virtual const sc_core::sc_export <FW_IF> &   get_base_export() const = 0;

    virtual  FW_IF  &                            get_base_interface() = 0;
    virtual const FW_IF &                        get_base_interface() const = 0;
};

// Base class for initiator sockets, providing binding methods
template <
    unsigned int BUSWIDTH = 32,
    typename FW_IF = tlm_fw_transport_if<>,
    typename BW_IF = tlm_bw_transport_if<>,
    int N = 1,
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
   sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
>
class tlm_base_initiator_socket :    public tlm_base_initiator_socket_b<BUSWIDTH, FW_IF, BW_IF>,
                                     public sc_core::sc_port<FW_IF, N, POL>
{
public:
    typedef FW_IF                                                    fw_interface_type;
    typedef BW_IF                                                    bw_interface_type;
    typedef sc_core::sc_port<fw_interface_type, N, POL>              port_type;
    typedef sc_core::sc_export<bw_interface_type>                    export_type;
    typedef tlm_base_target_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
                                                                     base_initiator_socket_type;
    typedef tlm_base_initiator_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
                                                                     base_type;

    tlm_base_initiator_socket();
    explicit tlm_base_initiator_socket(const char* name);
    virtual const char* kind() const;


    virtual void bind(base_target_socket_type& s);
    void operator() (base_target_socket_type& s);
    virtual void bind(base_type& s);
    void operator() (base_type& s);
    virtual void bind(bw_interface_type& ifs);
    void operator() (bw_interface_type& s);

    virtual unsigned int             get_bus_width() const { return BUSWIDTH; }
    virtual tlm_socket_category      get_socket_category() const final { return
TLM_INITIATOR_SOCKET; }
    virtual bool                     is_multi_socket() const final { return (N != 1); }


    // Implementation of pure virtual functions of base class
    virtual sc_core::sc_port_b<FW_IF> &          get_base_port()          { return *this;   }
    virtual const sc_core::sc_port_b<FW_IF> &    get_base_port() const    { return *this;   }

    virtual  BW_IF &                             get_base_interface()     { return m_export; }
    virtual const BW_IF &                        get_base_interface() const { return m_export; }

    virtual sc_core::sc_export<BW_IF> &          get_base_export()        { return m_export; }
    virtual const sc_core::sc_export<BW_IF> &    get_base_export() const  { return m_export; }

protected:
    export_type m_export;
};

// Base class for target sockets, providing binding methods
template <
    unsigned int BUSWIDTH = 32,
    typename FW_IF = tlm_fw_transport_if<>,
    typename BW_IF = tlm_bw_transport_if<>,
    int N = 1,
    sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
>
class tlm_base_target_socket :      public tlm_base_target_socket_b<BUSWIDTH, FW_IF, BW_IF>,
                                    public sc_core::sc_export<FW_IF>
{
public:
    typedef  FW_IF                                                  fw_interface_type;
    typedef  BW_IF                                                  bw_interface_type;
    typedef sc_core::sc_port<bw_interface_type, N, POL>             port_type;
    typedef sc_core::sc_export<fw_interface_type>                   export_type;
    typedef tlm_base_initiator_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
                                                                    base_initiator_socket_type;
    typedef tlm_base_target_socket_b<BUSWIDTH, fw_interface_type, bw_interface_type>
                                                                    base_type;

    tlm_base_target_socket();
    explicit tlm_base_target_socket(const char* name);
    virtual const char* kind() const;


    virtual void bind(base_initiator_socket_type& s);
    void operator() (base_initiator_socket_type& s);
    virtual void bind(base_type& s);
    void operator() (base_type& s);
    virtual void bind(fw_interface_type& ifs);
    void operator() (fw_interface_type& s);

    int size() const;
    bw_interface_type* operator-> ();
    bw_interface_type* operator[] (int i);

    virtual unsigned int                        get_bus_width() const { return BUSWIDTH; }
    virtual tlm_socket_category                 get_socket_category() const final {return
TLM_TARGET_SOCKET;}
    virtual bool                                is_multi_socket() const final { return (N != 1; }


    // Implementation of pure virtual functions of base class
    virtual sc_core::sc_port_b<BW_IF> &         get_base_port()             { return m_port;   }
    virtual const sc_core::sc_port_b<BW_IF> &   get_base_port() const       { return *this;   }

    virtual  FW_IF  &                           get_base_interface()        { return *this;   }
    virtual const FW_IF &                       get_base_interface() const  { return *this;   }

    virtual sc_core::sc_export<FW_IF> &         get_base_export()           { return *this;   }
    virtual const sc_core::sc_export<FW_IF> &   get_base_export() const     { return *this;   }

protected:
    port_type m_port;
};

// Principal initiator socket, parameterized with protocol traits class
template <
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm_base_protocol_types,
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    int N = 1,
    sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
>
class tlm_initiator_socket : public tlm_base_initiator_socket <
    BUSWIDTH, tlm_fw_transport_if<TYPES>, tlm_bw_transport_if<TYPES>, N, POL>
{
public:
    tlm_initiator_socket();
    explicit tlm_initiator_socket(const char* name);
    virtual const char* kind() const;

    virtual std::type_index get_protocol_types() const final { return std::type_index(typeid(TYPES)); }
};

// Principal target socket, parameterized with protocol traits class
template <
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm_base_protocol_types,
    int N = 1,
    sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
>
class tlm_target_socket : public tlm_base_target_socket <
    BUSWIDTH, tlm_fw_transport_if<TYPES>, tlm_bw_transport_if<TYPES>, N, POL>
{
public:
    tlm_target_socket();
    explicit tlm_target_socket(const char* name);
    virtual const char* kind() const;

    virtual std::type_index get_protocol_types() const final { return std::type_index(typeid(TYPES)); }
};

} // namespace tlm
```

13.2.3 tlm_base_socket_if

a) The abstract base class **tlm_base_socket_if** declares pure virtual functions that should be overridden in any derived classes.

b) Member functions **get_base_port** and **get_base_export** shall return the port and export, respectively, associated with the derived socket class, as non-templated base types.

c) Member functions **get_bus_width**, **get_protocol_types**, **get_socket_category**, and **is_multi_socket** are intended to return the value or information about the value of template parameters used in derived socket class.

13.2.4 tlm_base_initiator_socket_b and tlm_base_target_socket_b

a) The abstract base classes **tlm_base_initiator_socket_b** and **tlm_base_target_socket_b** shall inherit from the **tlm_base_socket_if** abstract base class.

b) These classes shall provide overriding member functions for **get_base_port** and **get_base_export**, returning a covariant of the return values defined in **tlm_base_socket_if**, templated on FW_IF, BW_IF.

c) Member function **get_interface** is intended to return the interface object associated with the derived socket class.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

13.2.5 tlm_base_initiator_socket and tlm_base_target_socket

a) The class **tlm_base_initiator_socket** shall derive from **tlm_base_initiator_socket_b**.

b) The class **tlm_base_target_socket** shall derive from **tlm_base_target_socket_b**.

c) For class **tlm_base_initiator_socket**, the constructor with a name argument shall pass the character string argument to the constructor belonging to the base class **sc_port** to set the string name of the instance in the module hierarchy, and shall also pass the same character string to set the string name of the corresponding **sc_export** on the backward path, adding the suffix **"_export"** and calling **sc_gen_unique_name** to avoid name clashes. For example, the call **tlm_initiator_socket("foo")** would set the port name to **"foo"** and the export name to **"foo_export"**. In the case of the default constructor, the names shall be created by calling **sc_gen_unique_name("tlm_base_initiator_socket")** for the port, and **sc_gen_unique_name("tlm_base_initiator_socket_export")** for the export.

d) For class **tlm_base_target_socket**, the constructor with a name argument shall pass the character string argument to the constructor belonging to the base class **sc_export** to set the string name of the instance in the module hierarchy, and shall also pass the same character string to set the string name of the corresponding **sc_port** on the backward path, adding the suffix **"_port"** and calling **sc_gen_unique_name** to avoid name clashes. For example, the call **tlm_target_socket("foo")** would set the export name to **"foo"** and the port name to **"foo_port"**. In the case of the default constructor, the names shall be created by calling **sc_gen_unique_name("tlm_base_target_socket")** for the export, and **sc_gen_unique_name("tlm_base_target_socket_port")** for the port.

e) Member function **kind** shall return the class name as a C string, that is, "tlm_base_initiator_socket" or "tlm_base_target_socket", respectively.

f) Member function **get_bus_width** shall return the value of the BUSWIDTH template argument.

g) Template argument BUSWIDTH shall determine the word length for each individual data word transferred through the socket, expressed as the number of bits in each word. For a burst transfer, BUSWIDTH shall determine the number of bits in each beat of the burst. The precise interpretation of this attribute shall depend on the transaction type. For the meaning of BUSWIDTH with the generic payload, see 14.12.

h) When binding socket-to-socket, the two sockets shall have identical values for the BUSWIDTH template argument. Executable code in the initiator or target may get and act on the BUSWIDTH.

i) Each of the member function **bind** and **operator()** that take a socket as an argument shall bind the socket instance to which the member function belongs to the socket instance passed as an argument to the member function.

j) Each of the member function **bind** and **operator()** that take an interface as an argument shall bind the export of the socket instance to which the member function belongs to the channel instance passed as an argument to the member function. (A channel is the SystemC term for a class that implements an interface.)

k) In each case, the implementation of **operator()** shall achieve its effect by calling the corresponding virtual member function **bind**.

l) When binding initiator socket to target socket, the member function **bind** and **operator()** shall each bind the port of the initiator socket to the export of the target socket, and the port of the target socket to the export of the initiator socket. This is for use when binding socket-to-socket at the same level in the hierarchy.

m) An initiator socket can be bound to a target socket by calling the member function **bind** or **operator()** of either socket, with precisely the same effect. In either case, the forward path lies in the direction from the initiator socket to the target socket.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

n) When binding initiator socket to initiator socket or target socket to target socket, the member function **bind** and **operator()** shall each bind the port of one socket to the port of the other socket, and the export of one socket to the export of the other socket. This is for use in hierarchical binding, that is, when binding a socket on a child module to a socket on a parent module, or a socket on a parent module to a socket on a child module, passing transactions up or down the module hierarchy.

o) For hierarchical binding, it is necessary to bind sockets in the correct order. When binding initiator socket to initiator socket, the socket of the child must be bound to the socket of the parent. When binding target socket to target socket, the socket of the parent must be bound to the socket of the child. This rule is consistent with the fact the **tlm_base_initiator_socket** is derived from **sc_port**, and **tlm_base_target_socket** from **sc_export**. Port must be bound to port going up the hierarchy, port-to-export across the top, and export-to-export going down the hierarchy.

p) In order for two sockets of classes **tlm_base_initiator_socket** and **tlm_base_target_socket** to be bound together, they must share the same forward and backward interface types and bus widths.

q) Member function **size** of the target socket shall call member function **size** of the port in the target socket (on the backward path), and shall return the value returned by **size** of the port.

r) The **operator->** of the target socket shall call **operator->** of the port in the target socket (on the backward path), and shall return the value returned by **operator->** of the port.

s) The **operator[]** of the target socket shall call **operator[]** of the port in the target socket (on the backward path) with the same argument, and shall return the value returned by **operator[]** of the port.

t) Class **tlm_base_initiator_socket** and class **tlm_base_target_socket** may act as multi-sockets; that is, a single initiator socket may be bound to multiple target sockets, and a single target socket may be bound to multiple initiator sockets. The two class templates have template parameters specifying the number of bindings and the port binding policy, which are used within the class implementation to parameterize the associated **sc_port** template instantiation.

u) If an object of class **tlm_base_initiator_socket** or **tlm_base_target_socket** is bound multiple times, then the **operator[]** can be used to address the corresponding object to which the socket is bound. The index value is determined by the order in which the member function **bind** or **operator()** were called to bind the sockets. However, any incoming interface method calls received by such a socket will be anonymous in the sense that there is no mechanism provided to identify the caller. On the other hand, such a mechanism is provided by the convenience sockets (see 16.2.4).

v) For example, consider a socket bound to two separate targets. The calls **socket[0]->nb_transport_fw(...)** and **socket[1]->nb_transport_fw()** would address the two targets, but there is no way to identify the caller of in incoming member function **nb_transport_bw()** from one of those two targets.

w) The implementations of the virtual member functions **get_base_port** and **get_base_export** shall return the port and export objects of the socket, respectively. The implementation of the virtual member function **get_base_interface** shall return the export object in the case of the initiator port, or the socket object itself in the case of the target socket.

x) Member function **get_socket_category** shall return TLM_INITIATOR_SOCKET for **tlm_base_initiator_socket** and TLM_TARGET_SOCKET for **tlm_base_target_socket**.

y) Member function **is_multi_socket** shall return true if the template parameter N is different from 1, i.e., if the socket is a multi-socket.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

13.2.6 tlm_initiator_socket and tlm_target_socket

a) The socket **tlm_initiator_socket** and **tlm_target_socket** take a protocol traits class as a template parameter. These sockets (or convenience sockets derived from them) should typically be used by an application rather than the base sockets.

b) The class **tlm_initiator_socket** shall derive from **tlm_base_initiator_socket**; the class **tlm_target_socket** shall derive from **tlm_base_target_socket**.

c) The constructors of the classes **tlm_initiator_socket** and **tlm_target_socket** shall call the corresponding constructors of their respective base classes, passing the **char*** argument where it exists.

d) In order for two sockets of classes **tlm_initiator_socket** and **tlm_target_socket** to be bound together, they must share the same protocol traits class (default **tlm_base_protocol_types**) and bus width. Strong type checking between sockets can be achieved by defining a new protocol traits class for each distinct protocol, whether or not that protocol is based on the generic payload.

e) Member function **kind** shall return the class name as a C string, that is, "tlm_initiator_socket" or "tlm_target_socket", respectively.

f) Member function **get_protocol_types** shall return a **std::type_index** object corresponding to the TYPES template parameter.

Example:

```
#include <systemc>
#include <tlm>
using namespace sc_core;
using namespace std;

struct Initiator : sc_module, tlm::tlm_bw_transport_if<> {  // Initiator implements the bw interface

    tlm::tlm_initiator_socket<32> init_socket;              // Protocol types default to base protocol
    SC_CTOR(Initiator)
            : init_socket("init_socket") {
        SC_THREAD(thread);
        init_socket.bind(*this);                            // Initiator socket bound to the initiator itself
    }

    void thread() {                                         // Process generates one placeholder transaction
        tlm::tlm_generic_payload trans;
        sc_time delay = SC_ZERO_TIME;
        init_socket->b_transport(trans, delay);
    }

    virtual tlm::tlm_sync_enum nb_transport_bw(
            tlm::tlm_generic_payload &trans,
            tlm::tlm_phase &phase,
            sc_core::sc_time &t) {
        return tlm::TLM_COMPLETED;                          // Placeholder implementation
    }

    virtual void invalidate_direct_mem_ptr(sc_dt::uint64 start_range, sc_dt::uint64 end_range)
    {}                                                      // Placeholder implementation
};

struct Target : sc_module, tlm::tlm_fw_transport_if<> {     // Target implements the fw interface
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    tlm::tlm_target_socket<32> targ_socket;                 // Protocol types default to base protocol

    SC_CTOR(Target) : targ_socket("targ_socket") {
        targ_socket.bind(*this);                            // Target socket bound to the target itself
    }

    virtual tlm::tlm_sync_enum nb_transport_fw(
            tlm::tlm_generic_payload &trans, tlm::tlm_phase &phase, sc_core::sc_time &t) {
        return tlm::TLM_COMPLETED;                          // Placeholder implementation
    }

    virtual void b_transport(tlm::tlm_generic_payload &trans, sc_time &delay) {
    }                                                       // Placeholder implementation

    virtual bool get_direct_mem_ptr(tlm::tlm_generic_payload &trans, tlm::tlm_dmi &dmi_data) {
        return false;                                       // Placeholder implementation
    }

    virtual unsigned int transport_dbg(tlm::tlm_generic_payload &trans) {
        return 0;                                           // Placeholder implementation
    }
};

SC_MODULE(Top1) {                   // Showing a simple non-hierarchical binding of initiator to target
    Initiator *init;
    Target *targ;

    SC_CTOR(Top1) {
        init = new Initiator("init");
        targ = new Target("targ");
        init->init_socket.bind(targ->targ_socket);          // Bind initiator socket to target socket
    }
};

struct Parent_of_initiator : sc_module {                    // Showing hierarchical socket binding
    tlm::tlm_initiator_socket<32> init_socket;
    Initiator *initiator;

    SC_CTOR(Parent_of_initiator) : init_socket("init_socket") {
        initiator = new Initiator("initiator");
        initiator->init_socket.bind(init_socket);       // Bind initiator socket to parent initiator socket
    }
};

struct Parent_of_target : sc_module {
    tlm::tlm_target_socket<32> targ_socket;
    Target *target;

    SC_CTOR(Parent_of_target) : targ_socket("targ_socket") {
        target = new Target("target");
        targ_socket.bind(target->targ_socket);              // Bind parent target socket to target socket
    }
};
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
SC_MODULE(Top2) {
    Parent_of_initiator *init;
    Parent_of_target *targ;

    SC_CTOR(Top2) {
        init = new Parent_of_initiator("init");
        targ = new Parent_of_target("targ");
        init->init_socket.bind(targ->targ_socket);    // Bind initiator socket to target socket at top level
    }
};
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

14. TLM-2.0 generic payload

14.1 Introduction

The generic payload is the class type offered by the TLM-2.0 standard for transaction objects passed through the core interfaces. The generic payload is closely related to the base protocol, which itself defines further rules to help ensure interoperability when using the generic payload (see 15.2).

The generic payload is intended to improve the interoperability of memory-mapped bus models, which it does at two levels. First, the generic payload provides an off-the-shelf general-purpose payload that allows for immediate interoperability when creating abstract models of memory-mapped buses where the precise details of the bus protocol are unimportant, while at the same time providing an extension mechanism for *ignorable* attributes. Second, the generic payload can be used as the basis for creating detailed models of specific bus protocols, with the advantage of reducing the implementation cost and increasing simulation speed when there is a need to bridge or adapt between different protocols, sometimes to the point where the bridge becomes trivial to write.

The generic payload is specifically aimed at modeling memory-mapped buses. It includes some attributes found in typical memory-mapped bus protocols such as command, address, data, byte enables, single word transfers, burst transfers, streaming, and response status. The generic payload may also be used as the basis for modeling protocols other than memory-mapped buses.

The generic payload does not include every attribute found in typical memory-mapped bus protocols, but it does include an extension mechanism so that applications can add their own specialized attributes.

For specific protocols, whether bus-based or not, modeling and interoperability are the responsibility of the protocol owners and are outside the scope of the Accellera Systems Initiative. It is up to the protocol owners or subject matter experts to proliferate models or coding guidelines for their own particular protocol. However, the generic payload is still applicable here, because it provides a common starting point for model creation, and in many cases will reduce the cost of bridging between different protocols in a transaction-level model.

It is recommended that the generic payload be used with the initiator and target sockets, which provide a bus width parameter used when interpreting the data array of the generic payload as well as forward and backward paths and a mechanism to enforce strong type checking between different protocols whether or not they are based on the generic payload.

The generic payload can be used with both the blocking and non-blocking transport interfaces. It can also be used with the direct memory and debug transport interfaces, in which case only a restricted set of attributes is used.

14.2 Extensions and interoperability

14.2.1 Overview

The goal of the generic payload is to enable interoperability between memory-mapped bus models, but all buses are not created equal. Given two transaction-level models that use different protocols and that model those protocols at a detailed level, then just as in a physical system, an adapter or bridge must be inserted between those models to perform protocol conversion and allow them to communicate. On the other hand, many transaction level models produced early in the design flow do not care about the specific details of any particular protocol. For such models it is sufficient to copy a block of data starting at a given address, and for those models, the generic payload can be used directly to give excellent interoperability.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The generic payload extension mechanism permits any number of extensions of any type to be defined and added to a transaction object. Each extension represents a new set of attributes, transported along with the transaction object. Extensions can be created, added, written and read by initiators, interconnect components, and targets alike. The extension mechanism itself does not impose any restrictions. Of course, undisciplined use of this extension mechanism would compromise interoperability, so disciplined use is strongly encouraged. But the flexibility is there where you need it.

The use of the extension mechanism represents a trade-off between increased coding convenience when binding sockets, and decreased compile-time type checking. If the undisciplined use of generic payload extensions were allowed, each application would be obliged to detect any incompatibility between extensions by including explicit run-time checks in each interconnect component and target, and there would be no mechanism to enforce the existence of a given extension. The TLM-2.0 standard prescribes specific coding guidelines to avoid these pitfalls.

There are three, and only three, recommended alternatives for the transaction template argument TRANS of the blocking and non-blocking transport interfaces and the template argument TYPES of the combined interfaces:

a) Use the generic payload directly, with ignorable extensions, and obey the rules of the base protocol. Such a model is said to be TLM-2.0 base-protocol-compliant (see 9.2).
b) Define a new protocol traits class containing a **typedef** for **tlm_generic_payload**. Such a model is said to be TLM-2.0 custom-protocol-compliant (see 9.2).
c) Define a new protocol traits class and a new transaction type. Such a model may use isolated features of the TLM-2.0 class library but is neither TLM-2.0 base-protocol-compliant nor TLM-2.0 custom-protocol-compliant (see 9.2).

These three alternatives are defined below in order of decreasing interoperability.

It should be emphasized that although deriving a new class from the generic payload is possible, it is not the recommended approach for interoperability

It should also be emphasized that these three options may be mixed in a single system model. In particular, there is value in mixing the first two options since the extension mechanism has been designed to permit efficient interoperability.

14.2.2 Use the generic payload directly, with ignorable extensions

a) In this case, the transaction type is **tlm_generic_payload**, the phase type is **tlm_phase**, and the protocol traits class for the combined interfaces is **tlm_base_protocol_types**. These are the default values for the TRANS argument of the transport interfaces and TYPES argument of the combined interfaces, respectively. Any model that uses the standard initiator and target sockets with the base protocol will be interoperable with any other such model, provided that those models respect the semantics of the generic payload and the base protocol (see 15.2).
b) In this case, any generic payload extension or extended phase shall be ignorable. *Ignorable* means that any component other than the component that added the extension is permitted to behave as if the extension were absent (see 14.21.1.2).
c) If an extension is ignorable, then by definition compile-time checking to enforce support for that extension in a target is not wanted, and indeed, the ignorable extension mechanism does not support compile-time checking.
d) The generic payload intrinsically supports minor variations in protocol. As a general principle, a target is recommended to support every feature of the generic payload. But, for example, a particular component may or may not support byte enables. A target that is unable to support a particular

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

feature of the generic payload is obliged to generate the standard error response. This should be thought of as being part of the specification of the generic payload.

14.2.3 Define a new protocol traits class containing a typedef for tlm_generic_payload

a) In this case, the transaction type is **tlm_generic_payload** and the phase type **tlm_phase**, but the protocol traits class used to specialize the socket is a new application-defined class, not the default **tlm_base_protocol_types**. In this case, the extended generic payload is treated as a distinct type and provides compile-time type checking when the initiator and target sockets are bound.

b) The new protocol type may set its own rules, and these rules may extend or contradict any of the rules of the base protocol, including the generic payload memory management rules (see 14.5) and the rules for the modifiability of attributes (see 14.7). However, for the sake of consistency and interoperability, it is recommended to follow the rules and coding style of the base protocol as far as possible (see 15.2).

c) The generic payload extension mechanism may be used for ignorable, non-ignorable, or mandatory extensions with no restrictions. The semantics of any extensions should be thoroughly documented with the new protocol traits class.

d) Because the transaction type is **tlm_generic_payload**, the transaction can be transported through interconnect components and targets that use the generic payload type, and can be cloned in its entirety, including all extensions. This provides a good starting point for building interoperable components and for creating adapters or bridges between different protocols, but the user should consider the semantics of the extended generic payload very carefully.

e) It is usual to use one and the same protocol traits class along the entire length of the path followed by a transaction from an initiator through zero or more interconnect components to a target. However, it may be possible to model an adapter or bus bridge as an interconnect component that takes incoming transactions of one protocol type and converts them to outgoing transactions of another protocol type. It is also possible to create a transaction bridge, which acts as a target for incoming transactions and as an initiator for outgoing transactions.

f) When passing a generic payload transaction between sockets specialized using different protocol traits classes, the user is obliged to consider the semantics of each extension very carefully so that the transaction can be transported through components that are aware of the generic payload but not the extensions. There is no general rule. Some extensions can be transported through components ignorant of the extension without mishap, for example, an attribute specifying the security level of the data. Other extensions will require explicit adaption or might not be supportable at all, for example, an attribute specifying that the interconnect is to be locked.

14.2.4 Define a new protocol traits class and a new transaction type

a) In this case, the transaction type may be unrelated to the generic payload.

b) A new protocol traits class will need to be defined to parameterize the combined interfaces and the sockets.

c) This choice may be justified when the new transaction type is significantly different from the generic payload or represents a very specific protocol.

d) If the intention is to use the generic payload for maximal interoperability, the recommended approach is to use the generic payload as described in one of the previous two clauses rather than to use it in the definition of a new class.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

14.3 Generic payload attributes and member functions

The generic payload class contains a set of private attributes, and a set of public access functions to get and set the values of those attributes. The exact implementation of those access functions is implementation-defined.

The majority of the attributes are set by the initiator and shall not be modified by any interconnect component or target. Only the address, DMI allowed, response status and extension attributes may be modified by an interconnect component or by the target. In the case of a read command, the target may also modify the data array.

14.4 Class definition

```
namespace tlm {

class tlm_generic_payload;

class tlm_mm_interface {
public:
    virtual void free(tlm_generic_payload*) = 0;
    virtual ~tlm_mm_interface() {}
};

unsigned int max_num_extensions();

class tlm_extension_base
{
public:
    virtual tlm_extension_base* clone() const = 0;
    virtual void free() { delete this; }
    virtual void copy_from(tlm_extension_base const &) = 0;
protected:
    virtual ~tlm_extension_base() {}
};

template <typename T>
class tlm_extension : public tlm_extension_base
{
public:
    virtual tlm_extension_base* clone() const = 0;
    virtual void copy_from(tlm_extension_base const &) = 0;
    virtual ~tlm_extension() {}
    const static unsigned int ID;
};

enum tlm_gp_option {
    TLM_MIN_PAYLOAD,
    TLM_FULL_PAYLOAD,
    TLM_FULL_PAYLOAD_ACCEPTED
};

enum tlm_command {
    TLM_READ_COMMAND,
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    TLM_WRITE_COMMAND,
    TLM_IGNORE_COMMAND
};

enum tlm_response_status {
    TLM_OK_RESPONSE = 1,
    TLM_INCOMPLETE_RESPONSE = 0,
    TLM_GENERIC_ERROR_RESPONSE = –1,
    TLM_ADDRESS_ERROR_RESPONSE = –2,
    TLM_COMMAND_ERROR_RESPONSE = –3,
    TLM_BURST_ERROR_RESPONSE = –4,
    TLM_BYTE_ENABLE_ERROR_RESPONSE = –5
};

#define TLM_BYTE_DISABLED 0x0
#define TLM_BYTE_ENABLED 0xff

class tlm_generic_payload {
public:
    // Constructors and destructor
    tlm_generic_payload();
    explicit tlm_generic_payload( tlm_mm_interface* );
    virtual ~tlm_generic_payload();

private:
    // Disable copy constructor and assignment operator
    tlm_generic_payload( const tlm_generic_payload& );
    tlm_generic_payload& operator= ( const tlm_generic_payload& );

public:
    // Memory management
    void set_mm( tlm_mm_interface* );
    bool has_mm() const;
    void acquire();
    void release();
    int get_ref_count() const;
    void reset();
    void deep_copy_from( const tlm_generic_payload & );
    void update_original_from( const tlm_generic_payload & , bool use_byte_enable_on_read = true );
    void update_extensions_from( const tlm_generic_payload & );
    void free_all_extensions();

    // Access member functions
    tlm_gp_option get_gp_option() const;
    void set_gp_option( const tlm_gp_option );

    tlm_command get_command() const;
    void set_command( const tlm_command );
    bool is_read();
    void set_read();
    bool is_write();
    void set_write();

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


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    void set_address( const sc_dt::uint64 );

    unsigned char* get_data_ptr() const;
    void set_data_ptr( unsigned char* );

    unsigned int get_data_length() const;
    void set_data_length( const unsigned int );

    unsigned int get_streaming_width() const;
    void set_streaming_width( const unsigned int );

    unsigned char* get_byte_enable_ptr() const;
    void set_byte_enable_ptr( unsigned char* );
    unsigned int get_byte_enable_length() const;
    void set_byte_enable_length( const unsigned int );

    // DMI hint
    void set_dmi_allowed( bool );
    bool is_dmi_allowed() const;

    tlm_response_status get_response_status() const;
    void set_response_status( const tlm_response_status );
    std::string get_response_string();
    bool is_response_ok();
    bool is_response_error();

    // Extension mechanism
    template <typename T> T* set_extension( T* );
    tlm_extension_base* set_extension( unsigned int , tlm_extension_base* );

    template <typename T> T* set_auto_extension( T* );
    tlm_extension_base* set_auto_extension( unsigned int , tlm_extension_base* );

    template <typename T> void get_extension( T*& ) const;
    template <typename T> T* get_extension() const;
    tlm_extension_base* get_extension( unsigned int ) const;

    template <typename T> void clear_extension( const T* );
    template <typename T> void clear_extension();

    template <typename T> void release_extension( T* );
    template <typename T> void release_extension();

    void resize_extensions();
};

} // namespace tlm
```

14.5 Generic payload memory management

a) The initiator shall be responsible for setting the data pointer and byte enable pointer attributes to existing storage, which could be static, automatic (stack), or dynamically allocated (new) storage.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The initiator shall not delete this storage before the lifetime of the transaction is complete. The generic payload destructor does not delete these two arrays.

b) This clause should be read in conjunction with the rules on generic payload extensions (see 14.21).

c) The generic payload supports two distinct approaches to memory management; reference counting with an explicit memory manager and ad hoc memory management by the initiator. The two approaches can be combined. Any memory management approach should manage both the transaction object itself and any extensions to the transaction object.

d) The construction and destruction of objects of type **tlm_generic_payload** is expected to be expensive in terms of CPU time due to the implementation of the extension array. As a consequence, repeated construction and destruction of generic payload objects should be avoided. There are two recommended strategies; either use a memory manager that implements a pool of transaction objects, or if using ad hoc memory management, reuse the very same generic payload object across successive calls to **b_transport** (effectively a transaction pool with a size of one). In particular, having a generic payload object constructed and destructed once per call to *transport* would be prohibitively slow and should be avoided.

e) A memory manager is a user-defined class that implements at least the member function **free** of the abstract base class **tlm_mm_interface**. The intent is that a memory manager would provide a method to allocate a generic payload transaction object from a pool of transactions, would implement the member function **free** to return a transaction object to that same pool, and would implement a destructor to delete the entire pool. The member function **free** is called by the member function **release** of class **tlm_generic_payload** when the reference count of a transaction object reaches 0. The member function **free** of class **tlm_mm_interface** would typically call the member function **reset** of class **tlm_generic_payload** in order to delete any extensions marked for automatic deletion.

f) Member functions **set_mm**, **acquire**, **release**, **get_ref_count**, and **reset** of the generic payload shall only used in the presence of a memory manager. By default, a generic payload object does not have a memory manager set.

g) Ad hoc memory management by the initiator without a memory manager requires the initiator to allocate memory for the transaction object before the TLM-2.0 core interface call, and delete or pool the transaction object and any extension objects after the call.

h) When the generic payload is used with the blocking transport interface, the direct memory interface or the debug transport interface, either approach may be used. Ad hoc memory management by the initiator is sufficient. In the absence of a memory manager, the member function **b_transport**, **get_direct_mem_ptr**, or **transport_dbg** should assume that the transaction object and any extensions will be invalidated or deleted on return.

i) When the generic payload is used with the non-blocking transport interface, a memory manager shall be used. Any transaction object passed as an argument to *nb_transport* shall have a memory manager already set. This applies whether the caller is the initiator, an interconnect component, or a target.

j) A blocking-to-non-blocking transport adapter shall set a memory manager for a given transaction if none existed already, in which case it shall remove that same memory manager from the transaction before returning control to the caller. A memory manager cannot be removed until the reference count has returned to 0, so the implementation will necessarily require that the member function **free** of the memory manager does not delete the transaction object. The **simple_target_socket** provides an example of such an adapter.

k) When using a memory manager, the transaction object and any extension objects shall be allocated from the heap (ultimately by calling **new** or **malloc**).

l) When using ad hoc memory management, the transaction object and any extensions may be allocated from the heap or from the stack. When using stack allocation, particular care needs to be taken with the memory management of extension objects in order to avoid memory leaks and segmentation faults.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

m) Member function **set_mm** shall set the memory manager of the generic payload object to the object whose address is passed as an argument. The argument may be null, in which case any existing memory manager would be removed from the transaction object, but not itself deleted. **set_mm** shall not be called for a transaction object that already has a memory manager and a reference count greater than 0.

n) Member function **has_mm** shall return true if and only if a memory manager has been set. When called from the body of an *nb_transport* method, **has_mm** should return true.

o) When called from the body of the member function **b_transport**, **get_direct_mem_ptr**, or **transport_dbg**, **has_mm** may return true or false. An interconnect component may call **has_mm** and take the appropriate action depending on whether or not a transaction has a memory manager. Otherwise, it shall assume all the obligations of a transaction with a memory manager (for example, heap allocation), but shall not call any of the member functions that require the presence of a memory manager (for example, **acquire**).

p) Each generic payload object has a reference count. The default value of the reference count is 0.

q) Member function **acquire** shall increment the value of the reference count. If **acquire** is called in the absence of a memory manager, a run-time error will occur.

r) Member function **release** shall decrement the value of the reference count, and if this leaves the value equal to 0, shall call the member function **free** of the memory manager object, passing the address of the transaction object as an argument. If **release** is called in the absence of a memory manager, a run-time error will occur.

s) Member function **get_ref_count** shall return the value of the reference count. In the absence of a memory manager, the value returned would be 0.

t) In the presence of a memory manager, each initiator should call the member function **acquire** of each transaction object before first passing that object as an argument to an interface method call, and should call the member function **release** of that transaction object when the object is no longer required.

u) In the presence of a memory manager, each interconnect component and target should call the member function **acquire** whenever they need to extend the lifetime of a transaction object beyond the current interface method call, and call the member function **release** when the object is no longer required.

v) In the presence of a memory manager, a component may call the member function **release** from any interface method call or process. Thus, a component cannot assume a transaction object is still valid after making an interface method call or after yielding control unless it has previously called the member function **acquire**. For example, an initiator may call **release** from its implementation of **nb_transport_bw**, or a target from its implementation of **nb_transport_fw**.

w) If an interconnect component or a target wishes to extend the lifetime of a transaction object indefinitely for analysis purposes, it should make a clone of the transaction object rather than using the reference counting mechanism. In other words, the reference count should not be used to extend the lifetime of a transaction object beyond the normal phases of the protocol.

x) In the presence of a memory manager, a transaction object shall not be reused to represent a new transaction or reused with a different interface until the reference count indicates that no component other than the initiator itself still has a reference to the transaction object. That is, assuming the initiator has called **acquire** for the transaction object, until the reference count equals 1. This rule applies when reusing transactions with the same interface or across the transport, direct memory, and debug transport interfaces. When reusing transaction objects to represent different transaction instances, it is best practice not to reuse the object until the reference count equals 0, that is, until the object has been freed.

y) The member function **reset** shall delete any extensions marked for automatic deletion, and shall set the corresponding extension pointers to null. Each extension shall be deleted by calling the member function **free** of the extension object, which could conceivably be overloaded if a user wished to provide explicit memory management for extension objects. The member function **reset** shall set the

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

value of the option attribute to TLM_MIN_PAYLOAD. The member function **reset** should typically be called from the member function **free** of class **tlm_mm_interface** in order to delete extensions at the end of the lifetime of a transaction.

z) An extension object added by calling **set_extension** may be deleted by calling **release_extension**. Calling **clear_extension** would only clear the extension pointer, not delete the extension object itself. This latter behavior would be required in the case that transaction objects are stack-allocated without a memory manager, and extension objects pooled.

aa) In the absence of a memory manager, whichever component allocates or sets a given extension should also delete or clear that same extension before returning control from **b_transport**, **get_direct_mem_ptr**, or **transport_dbg**. For example, an interconnect component that implements **b_transport** and calls **set_mm** to add a memory manager to a transaction object shall not return from **b_transport** until it has removed from the transaction object all extensions added by itself (and assuming that any downstream components will already have removed any extensions added by themselves, by virtue of this very same rule).

ab) In the presence of a memory manager, extensions can be added by calling **set_auto_extension**, and thus deleted or pooled automatically by the memory manager. Alternatively, extensions added by calling **set_extension** and not explicitly cleared are so-called *sticky* extensions, meaning that they will not be automatically deleted when the transaction reference count reaches 0 but may remain associated with the transaction object even when it is pooled. Sticky extensions are a particularly efficient way to manage extension objects because the extension object need not be deleted and reconstructed between transport calls. Sticky extensions rely on transaction objects being pooled (or reused singly).

ac) If it is unknown whether or not a memory manager is present, extensions should be added by calling **set_extension** and deleted by calling **release_extension**. This calling sequence is safe in the presence or absence of a memory manager. This circumstance can only occur within an interconnect component or target that chooses not to call **has_mm**. (Within an initiator, it is always known whether or not a memory manager is present, and a call to **has_mm** will always reveal whether or not a memory manager is present.)

ad) Member function **free_all_extensions** shall delete all extensions, including but not limited to those marked for automatic deletion, and shall set the corresponding extension pointers to null. Each extension shall be deleted by calling the member function **free** of the extension object. The member function **free** could conceivably be overloaded if a user wished to provide explicit memory management for extension objects.

ae) **free_all_extensions** would be useful when removing the extensions from a pooled transaction object that does not use a memory manager. With a memory manager, extensions marked for automatic deletion would indeed have been deleted automatically, while sticky extensions would not need to be deleted.

af) Member function **deep_copy_from** shall modify the attributes and extensions of the current transaction object by copying those of another transaction object, which is passed as an argument to the member function. The option, command, address, data length, byte enable length, streaming width, response status, and DMI allowed attributes shall be copied. The data and byte enable arrays shall be deep copied if and only if the corresponding pointers in both transactions are non-null. The application is responsible for ensuring that the arrays in the current transaction are sufficiently large. If an extension on the other transaction already exists on the current transaction, it shall be copied by calling the member function **copy_from** of the extension class. Otherwise, a new extension object shall be created by calling the member function **clone** of the extension class, and set on the current transaction. In the case of cloning, the new extension shall be marked for automatic deletion if and only if a memory manager is present for the current transaction.

ag) In other words, in the presence of a memory manager **deep_copy_from** will mark for automatic deletion any new extensions that were not already on the current object. Without a memory manager, extensions cannot be marked for auto-deletion.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

ah) Member function **update_original_from** shall modify certain attributes and extensions of the current transaction object by copying those of another transaction object, which is passed as an argument to the member function. The intent is that **update_original_from** should be called to pass back the response for a transaction created using **deep_copy_from**. The response status and DMI allowed attributes of the current transaction object shall be modified. The data array shall be deep copied if and only if the command attribute of the current transaction is TLM_READ_COMMAND and the data pointers in the two transactions are both non-null and are unequal. The byte enable array shall be used to mask the copy operation, as per the read command, if and only if the byte enable pointer is non-null and the **use_byte_enable_on_read** argument is **true**. Otherwise, the entire data array shall be deep copied. The extensions of the current transaction object shall be updated as per the member function **update_extensions_from**.

ai) Member function **update_extensions_from** shall modify the extensions of the current transaction object by copying from another transaction object only those extensions that were already present on the current object. The extensions shall be copied by calling the member function **copy_from** of the extension class.

aj) The typical use case for **deep_copy_from**, **update_original_from**, and **update_extensions_from** is within a transaction bridge where they are used to deep copy an incoming request, send the copy out through an initiator socket, then on receiving back the response copy the appropriate attributes and extensions back to the original transaction object. The transaction bridge may choose to deep copy the arrays or merely to copy the pointers.

ak) These obligations apply to the generic payload. In principle, similar obligations might apply to transaction types unrelated to the generic payload.

14.6 Constructors, assignment, and destructor

a) The default constructor shall set the generic payload attributes to their default values, as defined in the following clauses.

b) The constructor **tlm_generic_payload(tlm_mm_interface*)** shall set the generic payload attributes to their default values, and shall set the memory manager of the generic payload object to the object whose address is passed as an argument. This is equivalent to calling the default constructor and then immediately calling **set_mm**.

c) The copy constructor and assignment operators are disabled.

d) The virtual destructor **~tlm_generic_payload** shall delete all extensions, including but not limited to those marked for automatic deletion. Each extension shall be deleted by calling the member function **free** of the extension object. The destructor shall not delete the data array or the byte enable array.

14.7 Default values and modifiability of attributes

The default values and modifiability of the generic payload attributes and arrays for the base protocol are summarized in Table 54 and Table 55.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 54—Default values and modifiability of attributes

Attribute	Default value	Modifiable by interconnect?	Modifiable by target?
Option	TLM_MIN_PAYLOAD	No	**Yes**
Command	TLM_IGNORE_COMMAND	No	No
Address	0	**Yes**	No
Data pointer	0	No	No
Data length	0	No	No
Byte enable pointer	0	No	No
Byte enable length	0	No	No
Streaming width	0	No	No
DMI allowed	false	**Yes**	**Yes**
Response status	TLM_INCOMPLETE_RESPONSE	No	**Yes**
Extension pointers	0	**Yes**	**Yes**

Table 55—Modifiability of generic payload arrays

Arrays	Default value	Modifiable by interconnect?	Modifiable by target?
Data array	—	No	**Read command only**
Byte enable array	—	No	No

a) It is the responsibility of the initiator to set the values of the generic payload attributes prior to passing the transaction object through an interface method call according to the rules of the core interface being used. In the case of the transport interfaces, every generic payload attribute shall be set with the exception of the extension pointers. The DMI and debug transport interfaces have their own rules, described in 11.3.4 and 11.4.4, respectively. The option attribute can be used to determine whether the DMI and debug transport interfaces use a minimal or a full set of generic playload attributes. Care should be taken so that the attributes are set correctly in the case where transaction objects are pooled and reused.

b) In the case that a transaction object is returned to a pool or otherwise reused, these modifiability rules cease to apply at the end of the lifetime of that transaction instance. In the presence of a memory manager, this is the point at which the reference count reaches 0 or, otherwise, on return from the interface method call. The modifiability rules would apply afresh if the transaction object was reused for a new transaction.

c) After passing the transaction object as an argument to an interface method call (**b_transport**, **nb_transport_fw**, **get_direct_mem_ptr**, or **transport_dbg**), the only generic payload attributes that the initiator is permitted to modify during the lifetime of the transaction are the extension pointers.

d) An interconnect component is permitted to modify the address attribute, but only before passing the transaction concerned as an argument to any TLM-2.0 core interface method on the forward path. Once an interconnect component has passed a reference to the transaction to a downstream

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

component, it is not permitted to modify the address attribute of that transaction object again throughout the entire lifetime of the transaction.

e) As a consequence of the previous rule, the address attribute is valid immediately on entering any of the forward path interface method calls **b_transport**, **get_direct_mem_ptr**, or **transport_dbg**. In the case of **nb_transport_fw**, the address attribute is valid immediately on entering the function but only when the phase is BEGIN_REQ. Following the return from any forward path TLM-2.0 interface method call, the address attribute will have the value set by the interconnect component lying furthest downstream, and so should be regarded as being undefined for the purposes of transaction routing.

f) The interconnect and target are not permitted to modify the data array in the case of a write command, but the target alone is permitted to modify the data array in the case of a read command.

g) For a given transaction object, the target is permitted to modify the DMI allowed attribute, the response status attribute, and (for a read command) the data array at any time between having first received the transaction object and the time at which it passes a response in the upstream direction. A target is not permitted to modify these attributes after having sent a response in the upstream direction. A target sends a response in this sense whenever it returns control from the member function **b_transport**, **get_direct_mem_ptr**, or **transport_dbg**, whenever it passes the BEGIN_RESP phase as an argument to *nb_transport*, or whenever it returns the value TLM_COMPLETED from *nb_transport*.

h) If the DMI allowed attribute is **false**, an interconnect component is not permitted to modify the DMI allowed attribute. But if the target sets the DMI allowed attribute to **true**, an interconnect component is permitted to reset the DMI allowed attribute to **false** as it passes the response in an upstream direction. In other words, an interconnect component is permitted to clear the DMI allowed attribute, despite the DMI allowed attribute having been set by the target.

i) The initiator is permitted to assume it is seeing the values of the DMI allowed attribute, the response status attribute, and (for a read command) the data array as modified by the target only after it has received the response.

j) If the above rules permit a component to modify the value of a transaction attribute within a particular window of time, that attribute may be modified at any time during that window and any number of times during that window. Any other component shall only read the value of the attribute as it is left at the end of the time window (with the exception of extensions).

k) The roles of initiator, interconnect, and target may change dynamically. For example, although an interconnect component is not permitted to modify the response status attribute, that same component could modify the response status attribute by taking on the role of target for a given transaction. In its role as a target, the component would be forbidden from passing that particular transaction any further downstream.

l) In the case where the generic payload is used as the transaction type for the direct memory and debug transport interfaces, the modifiability rules given in this section shall apply to the appropriate attributes, that is, the command and address attributes in the case of direct memory, and the command, address, data pointer and data length attributes in the case of debug transport.

14.8 Option attribute

a) The option attribute is used to determine whether the DMI and debug transport interfaces use a minimal or a full set of generic payload attributes. The minimal set is supported for backward compatibility with previous versions of the TLM-2.0 standard. New components may choose to use the minimal or the full set of attributes.

b) Member function **set_gp_option** shall set the option attribute to the value passed as an argument. Member function **get_gp_option** shall return the current value of the option attribute.

c) The default value of the option attribute shall be TLM_MIN_PAYLOAD.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

d) With one exception, initiator, interconnect, and target components may ignore the option attribute. This applies both to legacy components developed before the publication of this standard and to new components developed after the publication of this standard. The one exception is when an initiator receives a transaction with the option attribute having the value TLM_FULL_PAYLOAD_ACCEPTED.

e) When sending a generic payload transaction through the direct memory interface, an initiator that requires the target to set the response status attribute shall set the option attribute to TLM_FULL_PAYLOAD, in which case the initiator shall set the values of the byte enable pointer, byte enable length, streaming width, *DMI allowed*, and response status attributes to their default values (as given in Table 54).

f) When sending a generic payload transaction through the debug transport interface, an initiator that requires the target to use the byte enable pointer, byte enable length, or streaming width attributes or that requires the target to set the *DMI allowed* or response status attributes shall set the option attribute to TLM_FULL_PAYLOAD, in which case the initiator shall set the *DMI allowed* and response status attributes to their default values (as given in Table 54).

g) When a generic payload transaction is sent through the blocking or the non-blocking transport interface, the option attribute shall be set to TLM_MIN_PAYLOAD and shall not be modified by any component.

h) If the option attribute has the value TLM_MIN_PAYLOAD, the value of the option attribute shall not be modified by any interconnect or target component.

i) In the case of the direct memory interface, if the option attribute is TLM_MIN_PAYLOAD, the target may ignore the values of all attributes except the command and address attributes.

j) In the case of the direct memory interface, if the option attribute is TLM_FULL_PAYLOAD, the target may set the value to TLM_FULL_PAYLOAD_ACCEPTED, in which case the initiator and the target shall set and act on the value of the response status attribute according to the rules given in 14.17.

k) In the case of the debug transport interface, if the option attribute is TLM_MIN_PAYLOAD, the target may ignore the values of the byte enable pointer, byte enable length, streaming width, *DMI allowed*, and response status attributes.

l) In the case of the debug transport interface, if the option attribute is TLM_FULL_PAYLOAD, the target may set the value to TLM_FULL_PAYLOAD_ACCEPTED, in which case the initiator and the target shall set and act on the values of the byte enable pointer, byte enable length, streaming width, *DMI allowed*, and response status attributes according to the rules given in 14.13, 14.14, 14.15, 14.16, and 14.17, respectively.

m) If the target chooses not to set the option attribute to TLM_FULL_PAYLOAD_ACCEPTED, the target shall ignore the values of the byte enable pointer, byte enable length, and streaming width attributes and shall not set the *DMI allowed* or response status attributes.

n) If the initiator sets the option attribute to TLM_FULL_PAYLOAD and the target does not set the option attribute to TLM_FULL_PAYLOAD_ACCEPTED, the initiator shall assume that the target has acted as if the option attribute were set to TLM_MIN_PAYLOAD. In this situation, it is possible that the target may have wrongly interpreted the generic payload attributes; for example, the target may have ignored the byte enable attributes for a debug transport transaction.

o) An interconnect component shall not modify the value of the option attribute. (A component that generally acts as an interconnect component may act as a target component in order to return an error response, in which case it may set the value of the option attribute to TLM_FULL_PAYLOAD_ACCEPTED.)

p) An initiator that sets the option attribute to TLM_FULL_PAYLOAD shall set the option attribute back to TLM_MIN_PAYLOAD at the end of the lifetime of the transaction object. In the presence of a memory manager, the member function **reset** of class **tlm_generic_payload** shall set the option attribute to TLM_MIN_PAYLOAD. In the absence of a memory manager, the initiator is obliged to reset the option attribute explicitly.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

q) The value of the option attribute shall apply only to the current transaction instance, and implies nothing about the behavior of an initiator, interconnect, or target with respect to other transactions or to other interfaces. For example, a given initiator may set TLM_MIN_PAYLOAD and TLM_FULL_PAYLOAD in two consecutive debug transport transactions, or may set TLM_MIN_PAYLOAD in a debug transport transaction and TLM_FULL_PAYLOAD in a DMI transaction. A target may set TLM_FULL_PAYLOAD_ACCEPTED for one transaction but not for the next. Each component should inspect each transaction individually rather than building a map of the initiators and targets and their capabilities.

14.9 Command attribute

a) Member function **set_command** shall set the command attribute to the value passed as an argument. Member function **get_command** shall return the current value of the command attribute.

b) Member functions **set_read** and **set_write** shall set the command attribute to TLM_READ_COMMAND and TLM_WRITE_COMMAND, respectively. Member functions **is_read** and **is_write** shall return **true** if and only if the current value of the command attribute is TLM_READ_COMMAND and TLM_WRITE_COMMAND, respectively.

c) A read command is a generic payload transaction with the command attribute equal to TLM_READ_COMMAND. A write command is a generic payload transaction with the command attribute equal to TLM_WRITE_COMMAND. An ignore command is a generic payload transaction with the command attribute equal to TLM_IGNORE_COMMAND.

d) On receipt of a read command, the target shall copy the contents of a local array in the target to the array pointed to be the data pointer attribute, honoring all the semantics of the generic payload as defined by this standard.

e) On receipt of a write command, the target shall copy the contents of the array pointed to by the data pointer attribute to a local array in the target, honoring all the semantics of the generic payload as defined by this standard.

f) If the target is unable to execute a read or write command, it shall generate a standard error response. The recommended response status is TLM_COMMAND_ERROR_RESPONSE.

g) An ignore command is a null command. The intent is that an ignore command may be used as a vehicle for transporting generic payload extensions without the need to execute a read or a write command, although the rules concerning extensions are the same for all three commands.

h) On receipt of an ignore command, the target shall not execute a write command or a read command. In particular, it shall not modify the value of the local array that would be modified by a write command, or modify the value of the array pointed to by the data pointer attribute. The target may, however, use the value of any attribute in the generic payload, including any extensions.

i) On receipt of an ignore command, a component that usually acts as an interconnect component may either forward the transaction onward toward the target (that is, act as an interconnect), or may return an error response (that is, act as a target). A component that routes read and write commands differently would be expected to return an error response.

j) A target is deemed to have executed an ignore command successfully if it has received the transaction and has checked the values of the generic payload attributes to its own satisfaction (see 14.17).

k) The command attribute shall be set by the initiator, and shall not be overwritten by any interconnect component or target.

l) The default value of the command attribute shall be TLM_IGNORE_COMMAND.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

14.10 Address attribute

a) Member function **set_address** shall set the address attribute to the value passed as an argument. Member function **get_address** shall return the current value of the address attribute.

b) For a read command or a write command, the target shall interpret the current value of the address attribute as the start address in the system memory map of the contiguous block of data being read or written. This address may or may not correspond to the first byte in the array pointed to by the data pointer attribute, depending on the endianness of the host computer.

c) The address associated with any given byte in the data array is dependent on the address attribute, the array index, the streaming width attribute, the endianness of the host computer, and the width of the socket (see 14.18).

d) The value of the address attribute need not be word-aligned (although address calculations can be considerably simplified if the address attribute is a multiple of the local socket width expressed in bytes).

e) If the target is unable to execute the transaction with the given address attribute (because the address is out-of-range, for example), it shall generate a standard error response. The recommended response status is TLM_ADDRESS_ERROR_RESPONSE.

f) The address attribute shall be set by the initiator but may be overwritten by one or more interconnect components. This may be necessary if an interconnect component performs address translation, for example, to translate an absolute address in the system memory map to a relative address in the memory map known to the target. Once the address attribute has been overwritten in this way, the old value is lost (unless it was explicitly saved somewhere).

g) The default value of the address attribute shall be 0.

14.11 Data pointer attribute

a) Member function **set_data_ptr** shall set the data pointer attribute to the value passed as an argument. Member function **get_data_ptr** shall return the current value of the data pointer attribute. Note that the data pointer attribute is a pointer to the data array, and these member functions set or get the value of the pointer, not the contents of the array.

b) For a read command or a write command, the target shall copy data to or from the data array, respectively, honoring the semantics of the remaining attributes of the generic payload.

c) The initiator is responsible for allocating storage for the data and byte enable arrays. The storage may represent the final source or destination of the data in the initiator, such as a register file or cache memory, or may represent a temporary buffer used to transfer data to and from the transaction level interface.

d) In general, the organization of the generic payload data array is independent of the organization of local storage within the initiator and the target. However, the generic payload has been designed so that data can be copied to and from the target with a single call to **memcpy** in most circumstances. This assumes that the target uses the same storage organization as the generic payload. This assumption is made for simulation efficiency but does not restrict the expressive power of the generic payload: the target is free to transform the data in any way it wishes as it copies the data to and from the data array.

e) For a read command or a write command, it is an error to call the transport interface with a transaction object having a null data pointer attribute.

f) In the case of TLM_IGNORE_COMMAND, the data pointer may point to a data array or may be null.

g) If the data pointer is not null, the length of the data array shall be greater than or equal to the value of the data length attribute, in bytes.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

h) The data pointer attribute shall be set by the initiator, and shall not be overwritten by any interconnect component or target.

i) For a write command or TLM_IGNORE_COMMAND, the contents of the data array shall be set by the initiator, and shall not be overwritten by any interconnect component or target

j) For a read command, the contents of the data array may be overwritten by the target (honoring the semantics of the byte enable) but by no other component and only before the target sends a response. A target sends a response in this sense whenever it returns control from the member functions **b_transport**, **get_direct_mem_ptr**, or **transport_dbg**, whenever it passes the BEGIN_RESP phase as an argument to *nb_transport*, or whenever it returns the value TLM_COMPLETED from *nb_transport*.

k) The default value of the data pointer attribute shall be 0, the null pointer.

14.12 Data length attribute

a) Member function **set_data_length** shall set the data length attribute to the value passed as an argument. Member function **get_data_length** shall return the current value of the data length attribute.

b) For a read command or a write command, the target shall interpret the data length attribute as the number of bytes to be copied to or from the data array, inclusive of any bytes disabled by the byte enable attribute.

c) The data length attribute shall be set by the initiator, and shall not be overwritten by any interconnect component or target.

d) For a read command or a write command, the data length attribute shall not be set to 0. In order to transfer zero bytes, the command attribute should be set to TLM_IGNORE_COMMAND.

e) In the case of TLM_IGNORE_COMMAND, if the data pointer is null, the value of the data length attribute is undefined.

f) When using the standard socket classes of the interoperability layer (or classes derived from these) for burst transfers, the word length for each transfer shall be determined by the BUSWIDTH template parameter of the socket. BUSWIDTH is independent of the data length attribute. BUSWIDTH shall be expressed in bits. If the data length is less than or equal to the BUSWIDTH / 8, the transaction is effectively modeling a single-word transfer, and if greater, the transaction is effectively modeling a burst. A single transaction can be passed through sockets of different bus widths. The BUSWIDTH may be used to calculate the latency of the transfer.

g) The target may or may not support transactions with data length greater than the word length of the target, whether the word length is given by the BUSWIDTH template parameter or by some other value.

h) If the target is unable to execute the transaction with the given data length, it shall generate a standard error response, and it shall not modify the contents of the data array. The recommended response status is TLM_BURST_ERROR_RESPONSE.

i) The default value of the data length attribute shall be 0, which is an invalid value unless the data pointer is null. Hence, unless the data pointer is null, the data length attribute shall be set explicitly before the transaction object is passed through an interface method call.

14.13 Byte enable pointer attribute

a) Member function **set_byte_enable_ptr** shall set the pointer to the byte enable array to the value passed as an argument. Member function **get_byte_enable_ptr** shall return the current value of the byte enable pointer attribute.

b) The elements in the byte enable array shall be interpreted as follows. A value of 0 shall indicate that that corresponding byte is disabled, and a value of 0xff shall indicate that the corresponding byte is enabled. The meaning of all other values shall be undefined. The value 0xff has been chosen so that

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

the byte enable array can be used directly as a mask. The two macros TLM_BYTE_DISABLED and TLM_BYTE_ENABLED are provided for convenience.

c) Byte enables may be used to create burst transfers where the address increment between each beat is greater than the number of significant bytes transferred on each beat, or to place words in selected byte lanes of a bus. At a more abstract level, byte enables may be used to create "lacy bursts" where the data array of the generic payload has an arbitrary pattern of holes punched in it.

d) The byte enable mask may be defined by a small pattern applied repeatedly or by a large pattern covering the whole data array (see 14.18).

e) The number of elements in the byte enable array shall be given by the byte enable length attribute.

f) The byte enable pointer may be set to 0, the null pointer, in which case byte enables shall not be used for the current transaction, and the byte enable length shall be ignored.

g) If byte enables are used, the byte enable pointer attribute shall be set by the initiator, the storage for the byte enable array shall be allocated by the initiator, the contents of the byte enable array shall be set by the initiator, and neither the byte enable pointer nor the contents of the byte enable array shall be overwritten by any interconnect component or target.

h) If the byte enable pointer is non-null, the target shall either implement the semantics of the byte enable as defined below or shall generate a standard error response. The recommended response status is TLM_BYTE_ENABLE_ERROR_RESPONSE.

i) In the case of a write command, any interconnect component or target should ignore the values of any disabled bytes in the data array. It is recommended that disabled bytes have no effect on the behavior of any interconnect component or target. The initiator may set those bytes to any values since they are going to be ignored.

j) In the case of a write command, when a target is doing a byte-by-byte copy from the transaction data array to a local array, the target should not modify the values of bytes in the local array corresponding to disabled bytes in the generic payload.

k) In the case of a read command, any interconnect component or target should not modify the values of disabled bytes in the data array. The initiator can assume that disabled bytes will not be modified by any interconnect component or target.

l) In the case of a read command, when a target is doing a byte-by-byte copy from a local array to the transaction data array, the target should ignore the values of bytes in the local array corresponding to disabled bytes in the generic payload.

m) If the application needs to violate these semantics for byte enables, or to violate any other semantics of the generic payload as defined in this document, the recommended approach would be to create a new protocol traits class (see 14.2.3).

n) The default value of the byte enable pointer attribute shall be 0, the null pointer.

14.14 Byte enable length attribute

a) Member function **set_byte_enable_length** shall set the byte enable length attribute to the value passed as an argument. Member function **get_byte_enable_length** shall return the current value of the byte enable length attribute.

b) For a read command or a write command, the target shall interpret the byte enable length attribute as the number of elements in the byte enable array.

c) The byte enable length attribute shall be set by the initiator, and shall not be overwritten by any interconnect component or target.

d) The byte enable to be applied to a given element of the data array shall be calculated using the formula **byte_enable_array_index = data_array_index % byte_enable_length**. In other words, the byte enable array is applied repeatedly to the data array.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

e) The byte enable length attribute may be greater than the data length attribute, in which case any superfluous byte enables should not affect the behavior of a read or write command, but could be used by extensions.

f) If the byte enable pointer is 0, the null pointer, then the value of the byte enable length attribute shall be ignored by any interconnect component or target. If the byte enable pointer is non-0, the byte enable length shall be non-0.

g) If the target is unable to execute the transaction with the given byte enable length, it shall generate a standard error response. The recommended response status is TLM_BYTE_ENABLE_ERROR_RESPONSE.

h) The default value of the byte enable length attribute shall be 0.

14.15 Streaming width attribute

a) Member function **set_streaming_width** shall set the streaming width attribute to the value passed as an argument. Member function **get_streaming_width** shall return the current value of the streaming width attribute.

b) For a read command or a write command, the target shall interpret and act on the current value of the streaming width attribute.

c) Streaming affects the way a component should interpret the data array. A stream consists of a sequence of data transfers occurring on successive notional beats, each beat having the same start address as given by the generic payload address attribute. The streaming width attribute shall determine the width of the stream, that is, the number of bytes transferred on each beat. In other words, streaming affects the local address associated with each byte in the data array. In all other respects, the organization of the data array is unaffected by streaming.

d) The bytes within the data array have a corresponding sequence of local addresses within the component accessing the generic payload transaction. The lowest address is given by the value of the address attribute. The highest address is given by the formula **address_attribute + streaming_width – 1**. The address to or from which each byte is being copied in the target shall be set to the value of the address attribute at the start of each beat.

e) With respect to the interpretation of the data array, a single transaction with a streaming width shall be functionally equivalent to a sequence of transactions each having the same address as the original transaction, each having a data length attribute equal to the streaming width of the original, and each with a data array that is a different subset of the original data array on each beat. This subset effectively steps down the original data array maintaining the sequence of bytes.

f) A streaming width of 0 shall be invalid. If a streaming transfer is not required, the streaming width attribute should be set to a value greater than or equal to the value of the data length attribute.

g) The value of the streaming width attribute shall have no affect on the length of the data array or the number of bytes stored in the data array.

h) Width conversion issues may arise when the streaming width is different from the width of the socket (when measured as a number of bytes) (see 14.18).

i) If the target is unable to execute the transaction with the given streaming width, it shall generate a standard error response. The recommended response status is TLM_BURST_ERROR_RESPONSE.

j) Streaming may be used in conjunction with byte enables, in which case the streaming width would typically be equal to the byte enable length. It would also make sense to have the streaming width a multiple of the byte enable length. Having the byte enable length a multiple of the streaming width would imply that different bytes were enabled on each beat.

k) The streaming width attribute shall be set by the initiator, and shall not be overwritten by any interconnect component or target.

l) The default value of the streaming width attribute shall be 0.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

14.16 DMI allowed attribute

a) Member function **set_dmi_allowed** shall set the DMI allowed attribute to the value passed as an argument. Member function **is_dmi_allowed** shall return the current value of the DMI allowed attribute.

b) The DMI allowed attribute provides a hint to an initiator that it may try to obtain a direct memory pointer. The target should set this attribute to true if the transaction at hand could have been done through DMI (see 11.3.9).

c) The default value of the DMI allowed attribute shall be **false**.

14.17 Response status attribute

14.17.1 Overview

a) Member function **set_response_status** shall set the response status attribute to the value passed as an argument. Member function **get_response_status** shall return the current value of the response status attribute.

b) Member function **is_response_ok** shall return **true** if and only if the current value of the response status attribute is TLM_OK_RESPONSE. Member function **is_response_error** shall return **true** if and only if the current value of the response status attribute is not equal to TLM_OK_RESPONSE.

c) Member function **get_response_string** shall return the current value of the response status attribute as a text string.

d) As a general principle, a target is recommended to support every feature of the generic payload, but in the case that it does not, it shall generate the standard error response (see 14.17.2).

e) The response status attribute shall be set to TLM_INCOMPLETE_RESPONSE by the initiator, and may or may not be overwritten by the target. The response status attribute shall not be overwritten by an interconnect component. The value TLM_INCOMPLETE_RESPONSE should be used to indicate that the component acting as the target did not attempt to execute the command, as might be the case if the response was returned from a component that usually acts as an interconnect component. But note that such a component would be allowed to set the response status attribute to any error response, because it is acting as a target.

f) If the target is able to execute the command successfully, it shall set the response status attribute to TLM_OK_RESPONSE. Otherwise, the target may set the response status to any of the six error responses listed in Table 56. The target should choose the appropriate error response depending on the cause of the error.

Table 56—Error responses

Error response	Interpretation
TLM_INCOMPLETE_RESPONSE	Target did not attempt to execute the command
TLM_ADDRESS_ERROR_RESPONSE	Target was unable to act on the address attribute, or address out-of-range
TLM_COMMAND_ERROR_RESPONSE	Target was unable to execute the command
TLM_BURST_ERROR_RESPONSE	Target was unable to act on the data length or streaming width
TLM_BYTE_ENABLE_ERROR_RESPONSE	Target was unable to act on the byte enable
TLM_GENERIC_ERROR_RESPONSE	Any other error

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

g) If a target detects an error but is unable to select a specific error response, it may set the response status to TLM_GENERIC_ERROR_RESPONSE.

h) The default value of the response status attribute shall be TLM_INCOMPLETE_RESPONSE.

i) In the case of TLM_IGNORE_COMMAND, a target that has received the transaction and would have been in a position to execute a read or write command should return TLM_OK_RESPONSE. Otherwise, the target may choose, at its discretion, to set an error response using the same criteria it would have applied for a read or write command. For example, a target that does not support byte enables would be permitted (but not obliged) to return TLM_BYTE_ENABLE_ERROR_RESPONSE.

j) The presence of a generic payload extension or extended phase may cause a target to return a different response status, provided that the rules concerning ignorable extensions are honored. In other words, within the base protocol it is allowable that an extension may cause a command to fail, but it is also allowable that the target may ignore the extension and thus have the command succeed.

k) The target shall be responsible for setting the response status attribute at the appropriate point in the lifetime of the transaction. In the case of the blocking transport interface, this means before returning control from **b_transport**. In the case of the non-blocking transport interface and the base protocol, this means before sending the BEGIN_RESP phase or returning a value of TLM_COMPLETED.

l) It is recommended that the initiator should always check the response status attribute on receiving a transition to the BEGIN_RESP phase or after the completion of the transaction. An initiator *may* choose to ignore the response status if it is known in advance that the value will be TLM_OK_RESPONSE, perhaps because it is known in advance that the initiator is only connected to targets that always return TLM_OK_RESPONSE, but in general this will not be the case. In other words, the initiator ignores the response status at its own risk.

m) A target has some latitude when selecting an error response. For example, if the command and address attributes are in error, a target may be justified in setting any of TLM_ADDRESS_ERROR_RESPONSE, TLM_COMMAND_ERROR_RESPONSE, or TLM_GENERIC_ERROR_RESPONSE. When using the response status to determine its behavior an initiator should not rely on the distinction between the six categories of error response alone, although an initiator may use the response status to determine the content of diagnostic messages printed for the benefit of the user.

14.17.2 The standard error response

When a target receives a generic payload transaction, the target should perform one and only one of the following actions:

— Execute the command represented by the transaction, honoring the semantics of the generic payload attributes, and honoring the publicly documented semantics of the component being modeled, and set the response status to TLM_OK_RESPONSE.

— Set the response status attribute of the generic payload to one of the five error responses as described above.

— Generate a report using the standard SystemC report handler with any of the four standard SystemC severity levels indicating that the command has failed or been ignored, and set the response status to TLM_OK_RESPONSE.

It is recommended that the target should perform exactly one of these actions, but an implementation is not obliged or permitted to enforce this recommendation.

It is recommended that a target for a transaction type other than the generic payload should follow this same principle; that is, execute the command as expected, or generate an error response using an attribute of the

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

transaction, or generate a SystemC report. However, the details of the semantics and the error response mechanism for such a transaction are outside the scope of this standard.

The conditions for satisfying point a) are determined by the expected behavior of the target component as would be visible to a user of that component. The attributes of the generic payload have defined semantics that correspond to conventional usage in the context of memory-mapped buses but that do not necessarily assume that the target behaves as a random-access memory. There are many subtle corner cases. For example:

a) A target may have a memory-mapped register that supports both read and write commands, but the write command is non-sticky; that is, write modifies the state of the target, but a write followed by read will not return the data just written but some other value determined by the state of the target. If this is the normal expected behavior of the component, it is covered by point a).
b) A target may implement the write command to set a bit while totally ignoring the value of the data attribute. If this is the normal expected behavior of the target, it is covered by point a).
c) A read-only memory may ignore the write command without signalling an error to the initiator using the response status attribute. Since the write command is not changing the state of the target but is being ignored altogether, the target should at least generate a SystemC report with severity SC_INFO or SC_WARNING.
d) A target should not under any circumstances implement the write command by performing a read, or vice versa. That would be a fundamental violation of the semantics of the generic payload.
e) A target may implement the read command according to the intent of the generic payload but with additional side-effects. This is covered by point a).
f) A target with a set of memory-mapped registers forming an addressable register file receives a write command with an out-of-range address. The target should either set the response status attribute of the transaction to TLM_ADDRESS_ERROR_RESPONSE or generate a SystemC report.
g) A passive simulation bus monitor target receives a transaction with an address that is outside the physical range of the bus being modeled. The target may log the erroneous transaction for post-processing under point a) and not generate an error response under points b) or c). Alternatively, the target may generate a report under point c).

In other words, the distinction between points a), b), and c) is ultimately a pragmatic judgment to be made on a case-by-case basis, but the definitive rule for the generic payload is that a target should always perform exactly one of these actions.

Example:

```
// Showing generic payload with command, address, data, and response status

// The initiator
void thread() {
    tlm::tlm_generic_payload trans;                 // Construct default generic payloadsc_time delay;
    trans.set_command(tlm::TLM_WRITE_COMMAND);      // A write command
    trans.set_data_length(4);                       // Write 4 bytes
    trans.set_byte_enable_ptr(0);                   // Byte enables unused
    trans.set_streaming_width(4);                   // Streaming unused
    for (int i = 0; i < RUN_LENGTH; i += 4) {       // Generate a series of transactions
        int word = i;
        trans.set_address(i);                       // Set the address
        trans.set_data_ptr((unsigned char *)(&word));   // Write data from local variable 'word'
        trans.set_dmi_allowed(false);               // Clear the DMI hint
        trans.set_response_status(tlm::TLM_INCOMPLETE_RESPONSE); // Clear the response status
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        init_socket->b_transport(trans, delay);
        if (trans.is_response_error())                  // Check return value of b_transport
          SC_REPORT_ERROR("TLM-2.0", trans.get_response_string().c_str());
        ...
    }
}
...

// The target
virtual void b_transport(tlm::tlm_generic_payload & trans, sc_core::sc_time & t) {
    tlm::tlm_command cmd = trans.get_command();
    sc_dt::uint64 adr = trans.get_address();
    unsigned char *ptr = trans.get_data_ptr();
    unsigned int len = trans.get_data_length();
    unsigned char *byt = trans.get_byte_enable_ptr();
    unsigned int wid = trans.get_streaming_width();

    if (adr + len > m_length) {                         // Check for storage address overflow
        trans.set_response_status(tlm::TLM_ADDRESS_ERROR_RESPONSE);
        return;
    }

    if (byt) {                                          // Target unable to support byte enable attribute
        trans.set_response_status(tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE);
        return;
    }
    if (wid < len) {                                    // Target unable to support streaming width attribute
        trans.set_response_status(tlm::TLM_BURST_ERROR_RESPONSE);
        return;
    }

    if (cmd == tlm::TLM_WRITE_COMMAND)                  // Execute command
        memcpy(&m_storage[adr], ptr, len);
    else if (cmd == tlm::TLM_READ_COMMAND)
        memcpy(ptr, &m_storage[adr], len);

    trans.set_response_status(tlm::TLM_OK_RESPONSE);    // Successful completion
}
...

// Showing generic payload with byte enables

// The initiator
void thread() {
    tlm::tlm_generic_payload trans;
    sc_time delay;

    static word_t byte_enable_mask = 0x0000ffffful;     // MSB..LSB regardless of host-endianness

    trans.set_command(tlm::TLM_WRITE_COMMAND);
    trans.set_data_length(4);
    trans.set_byte_enable_ptr(reinterpret_cast<unsigned char *>(&byte_enable_mask));
    trans.set_byte_enable_length(4);
    trans.set_streaming_width(4);
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    ...
}
...

// The target
virtual void b_transport(tlm::tlm_generic_payload &trans, sc_core::sc_time &t) {
    tlm::tlm_command cmd = trans.get_command();
    sc_dt::uint64    adr = trans.get_address();
    unsigned char   *ptr = trans.get_data_ptr();
    unsigned int     len = trans.get_data_length();
    unsigned char   *byt = trans.get_byte_enable_ptr();
    unsigned int     bel = trans.get_byte_enable_length();
    unsigned int     wid = trans.get_streaming_width();

    if (cmd == tlm::TLM_WRITE_COMMAND) {
        if (byt) {
            for (unsigned int i = 0; i < len; i++)        // Byte enable applied repeatedly up data array
                if (byt[i % bel] == tlm::TLM_BYTE_ENABLED)
                    m_storage[adr + i] = ptr[i];          // Byte enable [i] corresponds to data ptr [i]
        } else
            memcpy(&m_storage[adr], ptr, len);        // No byte enables
    } else if (cmd == tlm::TLM_READ_COMMAND) {
        if (byt) {                                        // Target does not support read with byte enables
            trans.set_response_status(tlm::TLM_BYTE_ENABLE_ERROR_RESPONSE);
            return;
        } else
            memcpy(ptr, &m_storage[adr], len);
    }
    trans.set_response_status(tlm::TLM_OK_RESPONSE);
}
...
```

14.18 Endianness

14.18.1 Introduction

When using the generic payload to transfer data between initiator and target, both the endianness of the host machine (host endianness) and the endianness of the initiator and target being modeled (modeled endianness) are relevant. This clause defines rules to help ensure interoperability between initiators and targets using the generic payload, so is specifically concerned with the organization of the generic payload data array and byte enable array. However, the rules given here may have an impact on some of the choices made in modeling endianness beyond the immediate scope of the generic payload.

A general principle in the TLM-2.0 approach to endianness is that the organization of the generic payload data array depends only on information known locally within each initiator, interconnect component, or target. In particular, it depends on the width of the local socket through which the transaction is sent or received, the endianness of the host computer, and the endianness of the component being modeled.

The organization of the generic payload and the approach to endianness has been chosen to maximize simulation efficiency in certain common system scenarios, particularly mixed-endian systems. The rules given below dictate the organization of the generic payload, and this is independent of the organization of the system being modeled. For example, a “word” within the generic payload need not necessarily correspond in internal representation with any “word” within the modeled architecture.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

At a macroscopic level, the main principle is that the generic payload assumes components in a mixed-endian system to be wired up MSB to MSB (most-significant byte) and LSB to LSB (least-significant byte). In other words, if a word is transferred between components of differing endianness, the MSB ... LSB relationship is preserved, but the *local address* of each byte as seen within each component will necessarily change using the transformation generally called *address swizzling*. This is true within both the modeled system and the TLM-2.0 model. On the other hand, if a mixed-endian system is wired such that the local addresses are invariant within each component (that is, each byte has the same local address when seen from any component), then an explicit byte swap would need to be inserted in the TLM-2.0 model.

In order to achieve interoperability with respect to the endianness of the generic payload arrays, it is only necessary to obey the rules given in this clause. A set of helper functions is provided to assist with the organization of the data array (see 14.20).

14.18.2 Rules

a) In the following rules, the generic payload data array is denoted as **data** and the generic payload byte enable array as **be**.

b) When using the standard socket classes of the interoperability layer (or classes derived from these), the contents of the data and byte enable arrays shall be interpreted using the BUSWIDTH template parameter of the socket through which the transaction is sent or received locally. The effective word length shall be calculated as (BUSWIDTH + 7)/8 bytes and in the following rules is denoted as **W**.

c) This quantity **W** defines the length of a word within the data array, each word being the amount of data that could be transferred through the local socket on a single beat. The data array may contain a single word, a part-word, or several contiguous words or part-words. Only the first and last words in the data array may be part-words. This description refers to the internal organization of the generic payload, not to the organization of the architecture being modeled.

d) If a given generic payload transaction object is passed through sockets of different widths, the data array word length would appear different when calculated from the point of view of different sockets (see the description of width conversion below).

e) The order of the bytes within each word of the data array shall be host-endian. That is, on a little-endian host processor, within any given word **data[n]** shall be less significant than **data[n+1]**, and on a big-endian host processor, **data[n]** shall be the more significant than **data[n+1]**.

f) The word boundaries in the data array shall be address-aligned; that is, they shall fall on addresses that are integer multiples of the word length **W**. However, neither the address attribute nor the data length attribute are required to be multiples of the word length. Hence the possibility that the first and last words in the data array could be part-words.

g) The order of the words within the data array shall be determined by their addresses in the memory map of the modeled system. For array index values less than the value of the streaming width attribute, the local addresses of successive words shall be in increasing order, and (excluding any leading part-word) shall equal **address_attribute** – **(address_attribute % W) + NW**, where **N** is a non-negative integer, and **%** indicates remainder on division.

h) In other words, using the notation {a,b,c,d} to list the elements of the data array in increasing order of array index, and using LSBN to denote the least significant byte of the Nth word, on a little-endian host bytes are stored in the order {..., MSB_0, LSB_1, ..., MSB_1, LSB_2, ...}, and on a big-endian host {... LSB_0, MSB_1, ... LSB_1, MSB_2, ...}, where the number of bytes in each full word is given by **W**, and the total number of bytes is given by the **data_length** attribute.

i) The previous rules effectively mean that initiators and targets are connected LSB-to-LSB, MSB-to-MSB. The rules have been chosen to give optimal simulation speed in the case where the majority of initiators and targets are modeled using host endianness whatever their native endianness, also known as “arithmetic mode.”

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

j) It is strongly recommended that applications should be independent of host endianness, that is, should model the same behavior when run on a host of either endianness. This may require the use of helper functions or conditional compilation.

k) If an initiator or target is modeled using its native endianness and that is different from host endianness, it will be necessary to swap the order of bytes within a word when transferring data to or from the generic payload data array. Helper functions are provided for this purpose.

l) For example, consider the following SystemC code fragment, which uses the literal value 0xAABBCCDD to initialize the generic payload data array:

```
int data = 0xAABBCCDD;
trans.set_data_ptr( reinterpret_cast<unsigned char*>( &data ) );
trans.set_data_length(4);
trans.set_address(0);
socket->b_transport(trans, delay);
```

m) The C++ compiler will interpret the literal 0xAABBCCDD in host-endian form. In either case, the MSB has value 0xAA and the LSB has value 0xDD. Assuming this is the intent, the code fragment is valid and is independent of host endianness. However, the array index of the four bytes will differ depending on host endianness. On a little-endian host, data[0] = 0xDD, and on a big-endian host, data[0] = 0xAA. The correspondence between local addresses in the modeled system and array indexes will differ depending whether modeled endianness and host endianness are equal:

Little-endian model	and little-endian host:	data[0] is 0xDD and local address 0
Big-endian model	and little-endian host:	data[0] is 0xDD and local address 3
Little-endian model	and big-endian host:	data[0] is 0xAA and local address 3
Big-endian model	and big-endian host:	data[0] is 0xAA and local address 0

n) Code such as the fragment shown above would not be portable to a host computer that uses neither little nor big endianness. In such a case, the code would have to be re-written to access the generic payload data array using byte addressing only.

o) When a little-endian and a big-endian model interpret a given generic payload transaction, then by definition they will agree on which is the MSB and LSB of a word, but they will each use different local addresses to access the bytes of the word.

p) Neither the data length attribute nor the address attribute are required to be integer multiples of **W**. However, having address and data length aligned with word boundaries and having **W** be a power of 2 considerably simplifies access to the data array. Just to emphasize the point, it would be perfectly in order for a generic payload transaction to have an address and data length that indicated three bytes in the middle of a 48-bit socket. If a particular target is unable to support a given address attribute or data length, it should generate a standard error response (see 14.17).

q) For example, on a little-endian host and with **W = 4**, **address = 1**, and **data_length = 4**, the first word would contain three bytes at addresses 1...3, and the second word 1 byte at address 4.

r) Single byte and part-word transfers may be expressed using non-aligned addressing. For example, given **W** = 8, address = 5, and **data** = {1,2}, the two bytes with local addresses 5 and 6 are accessed in an order dependent on endianness.

s) Part-word and non-aligned transfers can always be expressed using integer multiples of **W** together with byte enables. This implies that a given transaction may have several equally valid generic payload representations. For example, given a little-endian host and a little-endian initiator:

address = 2, **W** = 4, **data** = {1} is equivalent to
address = 0, **W** = 4, **data** = {x, x, 1, x}, and **be** = {0, 0, 0xff, 0}
address = 2, **W** = 4, **data** = {1,2,3,4} is equivalent to
address = 0, **W** = 4, **data** = {x, x, 1, 2, 3, 4, x, x}, and **be** = {0, 0, 0xff, 0xff, 0xff, 0xff, 0, 0}.

t) For part-word access, the necessity to use byte enables is dependent on endianness. For example, given the intent to access the whole of the first word and the LSB of the second word, given a little-endian host this might be expressed as

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

address = 0, **W** = 4, **data** = {1,2,3,4,5}

Given a big-endian host, the equivalent would be

address = 0, **W** = 4, **data** = {4,3,2,1,x,x,x,5}, **be** = {0xff, 0xff, 0xff, 0xff, 0, 0, 0, 0xff }.

u) When two sockets are bound together, they necessarily have the same BUSWIDTH. However, a transaction may be forwarded from a target socket to an initiator socket of a different bus width. In this case, width conversion of the generic payload transaction must be considered (Figure 27). Any width conversion has its own intrinsic endianness, depending on whether the least- or most-significant byte of the wider socket is picked out first.

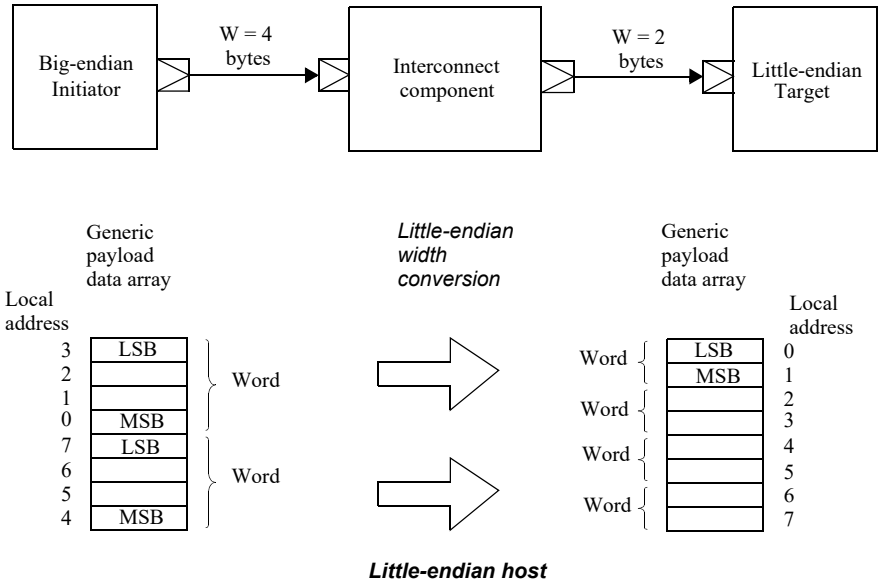

Figure 27—Width conversion

v) When the endianness chosen for a width conversion matches the host endianness, the width conversion is effectively free, meaning that a single transaction object can be forwarded from socket-to-socket without modification. Otherwise, two separate generic payload transaction objects would be required. In Figure 27, the width conversion between the 4-byte socket and the 2-byte socket uses host-endianness, moving the less-significant bytes to lower addresses while retaining the host-endian byte order within each word. The initiator and target both access the same sequence of bytes in the data array, but their local addressing schemes are quite different.

w) If a width conversion is performed from a narrower socket to a wider socket, the choice has to be made as to whether or not to perform address alignment on the outgoing transaction. Performing address alignment will always necessitate the construction of a new generic payload transaction object.

x) Similar width conversion issues arise when the streaming width attribute is non-zero but different from **W**. A choice has to be made as to the order in which to read off the bytes down the data array depending on host endianness and the desired endianness of the width conversion.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

14.19 Helper functions to determine host endianness

14.19.1 Introduction

A set of helper functions is provided to determine the endianness of the host computer. These are intended for use when creating or interpreting the generic payload data array.

14.19.2 Definition

```
namespace tlm {

enum tlm_endianness {
    TLM_UNKNOWN_ENDIAN,
    TLM_LITTLE_ENDIAN,
    TLM_BIG_ENDIAN };

inline tlm_endianness get_host_endianness(void);
inline bool host_has_little_endianness(void);
inline bool has_host_endianness(tlm_endianness endianness);

} // namespace tlm
```

14.19.3 Rules

a) The function **get_host_endianness** shall return the endianness of the host.

b) The function **host_has_little_endianness** shall return the value true if and only if the host is little-endian.

c) The function **has_host_endianness** shall return the value true if and only if the endianness of the host is the same as that indicated by the argument.

d) If the host is neither little- nor big-endian, the value returned from the above three functions shall be undefined.

14.20 Helper functions for endianness conversion

14.20.1 Introduction

The rules governing the organization of the generic payload data array are well-defined, and in many simple cases, writing host-independent C++ code to create and interpret the data array is a straightforward task. However, the rules do depend on the relationship between the endianness of the modeled component and host endianness, so creating host-independent code can become quite complex in cases involving non-aligned addressing and data word widths that differ from the socket width. A set of helper functions is provided to assist with this task.

With respect to endianness, interoperability depends only on the endianness rules being followed. Use of the helper functions is not necessary for interoperability.

The motivation behind the endianness conversion functions is to permit the C++ code that creates a generic payload transaction for an initiator to be written once with little regard for host endianness, and then to have the transaction converted to match host endianness with a single function call. Each conversion function takes an existing generic payload transaction and modifies that transaction in-place. The conversion functions are organized in pairs, a *to_hostendian* function and a *from_hostendian* function, which should always be used together. The *to_hostendian* function should be called by an initiator before sending a transaction through a transport interface, and *from_hostendian* on receiving back the response.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Four pairs of functions are provided, the *_generic* pair being the most general and powerful, and the *_word*, *_aligned* and *_single* functions being variants that can only handle restricted cases. The transformation performed by the *_generic* functions is relatively computationally expensive, so the other functions should be preferred for efficiency wherever possible.

The conversion functions provide sufficient flexibility to handle many common cases, including both *arithmetic mode* and *byte order mode*. *Arithmetic mode* is where a component stores data words in host-endian format for efficiency when performing arithmetic operations, regardless of the endianness of the component being modeled. *Byte order mode* is where a component stores bytes in an array in ascending address order, disregarding host endianness. The use of arithmetic mode is recommended for simulation speed. Byte order mode may necessitate byte swapping when copying data to and from the generic payload data array.

The conversion functions use the concept of a *data word*. The data word is independent of both the TLM-2.0 socket width and the word width of the generic payload data array. The data word is intended to represent a register that stores bytes in host-endian order within the component model (regardless of the endianness of the component being modeled). If the data word width is different to the socket width, the *hostendian* functions may have to perform an endianness conversion. If the data word is just one byte wide, the *hostendian* functions will effectively perform a conversion from and to *byte order mode*.

In summary, the approach to be taken with the *hostendian* conversion functions is to write the initiator code *as if* the endianness of the host computer matched the endianness of the component being modeled, while keeping the bytes within each data word in actual host-endian order. For data words wider than the host machine word length, use an array in host-endian order. Then if host endianness differs from modeled endianness, simply call the *hostendian* conversion functions.

14.20.2 Definition

```
namespace tlm {

template<class DATAWORD>
inline void tlm_to_hostendian_generic(tlm_generic_payload *, unsigned int );
template<class DATAWORD>
inline void tlm_from_hostendian_generic(tlm_generic_payload *, unsigned int );

template<class DATAWORD>
inline void tlm_to_hostendian_word(tlm_generic_payload *, unsigned int);
template<class DATAWORD>
inline void tlm_from_hostendian_word(tlm_generic_payload *, unsigned int);

template<class DATAWORD>
inline void tlm_to_hostendian_aligned(tlm_generic_payload *, unsigned int);
template<class DATAWORD>
inline void tlm_from_hostendian_aligned(tlm_generic_payload *, unsigned int);

template<class DATAWORD>
inline void tlm_to_hostendian_single(tlm_generic_payload *, unsigned int);
template<class DATAWORD>
inline void tlm_from_hostendian_single(tlm_generic_payload *, unsigned int);

inline void tlm_from_hostendian(tlm_generic_payload *);

} // namespace tlm
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

14.20.3 Rules

a) The first argument to a function of the form *to_hostendian* should be a pointer to a generic payload transaction object that would be valid if it were sent through a transport interface. The function should only be called after constructing and initializing the transaction object and before passing it to an interface method call.

b) The first argument to a function of the form *from_hostendian* shall be a pointer to a generic payload transaction object previously passed to *to_hostendian*. The function should only be called when the initiator receives a response for the given transaction or the transaction is complete. Since the function may modify the transaction and its arrays, it should only be called at the end of the lifetime of the transaction object.

c) If a *to_hostendian* function is called for a given transaction, the corresponding *from_hostendian* function should also be called with the same template and function arguments. Alternatively, the function **tlm_from_hostendian**(tlm_generic_payload *****) can be called for the given transaction. This function uses additional context information stored with the transaction object (as an ignorable extension) to recover the template and function argument values, but is marginally slower in execution.

d) The second argument to a *hostendian* function should be the width of the local socket through which the transaction is passed, expressed in bytes. This is equivalent to the word length of the generic payload data array with respect to the local socket. This shall be a power of 2.

e) The template argument to a *hostendian* function should be a type representing the internal initiator data word for the endianness conversion. The expression sizeof(DATAWORD) is used to determine the width of the data word in bytes, and the assignment operator of type DATAWORD is used during copying. sizeof(DATAWORD) shall be a power of 2.

f) The implementation of *to_hostendian* adds an extension to the generic payload transaction object to store context information. This means that *to_hostendian* can only be called once before calling *from_hostendian*.

g) The following constraints are common to every pair of *hostendian* functions. The term *integer multiple* means 1 x , 2 x , 3 x , ... and so forth:

   1) Socket width shall be a power of 2
   2) Data word width shall be a power of 2
   3) The streaming width attribute shall be an integer multiple of the data word width
   4) The data length attribute shall be an integer multiple of the streaming width attribute

h) The *hostendian_generic* functions are not subject to any further specific constraints. In particular, they support byte enables, streaming, and non-aligned addresses and word widths.

i) The remaining pairs of functions, namely *hostendian_word*, *hostendian_aligned*, and *hostendian_single*, all share the following additional constraints:

   1) Data word width shall be no greater than socket width, and as a consequence, socket width shall be a power-of-2 multiple of data word width.
   2) The streaming width attribute shall equal the data length attribute. That is, streaming is not supported.
   3) Byte enable granularity shall be no finer than data word width. That is, the bytes in a given data word shall be either all enabled or all disabled.
   4) If byte enables are present, the byte enable length attribute shall equal the data length attribute.

j) The *hostendian_aligned* functions alone are subject to the following additional constraints:

   1) The address attribute shall be an integer multiple of the socket width.
   2) The data length attribute shall be an integer multiple of the socket width.

k) The *hostendian_single* functions alone are subject to the following additional constraints:

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

1) The data length attribute shall equal the data word width.
2) The data array shall not cross a data word boundary and, as a consequence, shall not cross a socket boundary.

14.21 Generic payload extensions

14.21.1 Introduction

14.21.1.1 Overview

The extension mechanism is an integral part of the generic payload, and is not intended to be used separately from the generic payload. Its purpose is to permit attributes to be added to the generic payload. Extensions can be ignorable or non-ignorable, mandatory or non-mandatory.

14.21.1.2 Ignorable extensions

Being *ignorable* means that any component other than the component that added the extension is permitted to behave as if the extension were absent. As a consequence, the component that added the ignorable extension cannot rely on any other component reacting in any way to the presence of the extension, and a component receiving an ignorable extension cannot rely on other components having recognized that extension. This definition applies to generic payload extensions and to extended phases alike.

A component shall not fail and shall not generate an error response because of the absence of an ignorable extension. In this sense, ignorable extensions are also non-mandatory extensions. A component may fail or generate an error response because of the presence of an ignorable extension but also has the choice of ignoring the extension.

In general, an ignorable extension can be thought of as one for which there exists an obvious and safe default value such that any interconnect component or target can behave normally in the absence of the given extension by assuming the default value. An example might be the privilege level associated with a transaction, where the default is the lowest level.

Ignorable extensions may be used to transport auxiliary, side-band, or simulation-related information or meta-data. For example, a unique transaction identifier, the wall-time when the transaction was created, or a diagnostic filename.

Ignorable extensions are permitted by the base protocol.

14.21.1.3 Non-ignorable and mandatory extensions

A non-ignorable extension is an extension that every component receiving the transaction is obliged to act on if present. A mandatory extension is an extension that is required to be present. Non-ignorable and mandatory extensions may be used when specializing the generic payload to model the details of a specific protocol. Non-ignorable and mandatory extensions require the definition of a new protocol traits class.

14.21.2 Rationale

The rationale behind the extension mechanism is twofold. First, to permit TLM-2.0 sockets that carry variations on the core attribute set of the generic payload to be specialized with the same protocol traits class, thus allowing them to be bound together directly with no need for adaption or bridging. Second, to permit easy adaption between different protocols where both are based on the same generic payload and extension mechanism. Without the extension mechanism, the addition of any new attribute to the generic payload would require the definition of a new transaction class, leading to the need for multiple adapters.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The extension mechanism allows variations to be introduced into the generic payload, thus reducing the amount of coding work that needs to be done to traverse sockets that carry different protocols.

14.21.3 Extension pointers, objects and transaction bridges

An extension is an object of a type derived from the class **tlm_extension**. The generic payload contains an array of pointers to extension objects. Every generic payload object is capable of carrying a single instance of every type of extension.

The array-of-pointers to extensions has a slot for every registered extension. The member function **set_extension** simply overwrites a pointer and, in principle, can be called from an initiator, interconnect component, or target. This provides a very a flexible low-level mechanism, but it is open to misuse. The ownership and deletion of extension objects has to be well understood and carefully considered by the user.

When creating a transaction bridge between two separate generic payload transactions, it is the responsibility of the bridge to copy any extensions, if required, from the incoming transaction object to the outgoing transaction object, and to own and manage the outgoing transaction and its extensions. The same holds for the data array and byte enable array. The member functions **deep_copy_from** and **update_original_from** are provided so that a transaction bridge can perform a deep copy of a transaction object, including the data and byte enable arrays and the extension objects. If the bridge adds further extensions to the outgoing transaction, those extensions would be owned by the bridge.

The management of extensions is described more fully in 14.5.

14.21.4 Rules

a) An extension can be added by an initiator, interconnect, or target component. In particular, the creation of extensions is not restricted to initiators.

b) Any number of extensions may be added to each instance of the generic payload.

c) In the case of an ignorable extension, any component (excepting the component that added the extension) is allowed to ignore the extension, and ignorable extensions are not mandatory extensions. Having a component fail because of either the absence of an ignorable extension or the absence of a response to an ignorable extension would destroy interoperability.

d) There is no built-in mechanism to enforce the presence of a given extension, nor is there a mechanism to ensure that an extension is ignorable.

e) The semantics of each extension shall be application-defined. There are no pre-defined extensions.

f) An extension shall be created by deriving a user-defined class from the class **tlm_extension**, passing the name of the user-defined class itself as a template argument to **tlm_extension**, then creating an object of that class. The user-defined extension class may include members that represent extended attributes of the generic payload.

g) The virtual member function **free** of the class **tlm_extension_base** shall delete the extension object. This member function may be overridden to implement user-defined memory management of extension, but this is not necessary.

h) The pure virtual function **clone** of class **tlm_extension** shall be defined in the user-defined extension class to clone the extension object, including any extended attributes. This member function **clone** is intended for use in conjunction with generic payload memory management. It shall create a copy of any extension object such that the copy can survive the destruction of the original object with no visible side-effects.

i) The pure virtual function **copy_from** of class **tlm_extension** shall be defined in the user-defined extension class to modify the current extension object by copying the attributes of another extension object.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

j) The act of instantiating the class template **tlm_extension** shall cause the public data member **ID** to be initialized, and this shall have the effect of registering the given extension with the generic payload object and assigning a unique ID to the extension. The ID shall be unique across the whole executing program. That is, each instantiation of the class template **tlm_extension** shall have a distinct **ID**, whereas all extension objects of a given type shall share the same **ID**.

k) The generic payload shall behave as if it stored pointers to the extensions in a re-sizable array, where the **ID** of the extension gives the index of the extension pointer in the array. Registering the extension with the generic payload shall reserve an array index for that extension. Each generic payload object shall contain an array capable of storing pointers to every extension registered in the currently executing program.

l) The pointers in the extension array shall be null when the transaction is constructed.

m) Each generic payload object can store a pointer to at most one object of any given extension type (but to many objects of different extensions types). (There exists a utility class **instance_specific_extension**, which enables a generic payload object to reference several extension objects of the same type (see 16.5).)

n) The function **max_num_extensions** shall return the number of extension types, that is, the size of the extension array. As a consequence, the extension types shall be numbered from **0** to **max_num_extensions()-1**.

o) Member functions **set_extension**, **set_auto_extension**, **get_extension**, **clear_extension**, and **release_extension** are provided in several forms, each of which identify the extension to be accessed in different ways: using a function template, using an extension pointer argument, or using an ID argument. The functions with an ID argument are intended for specialist programming tasks such as when cloning a generic payload object, and not for general use in applications.

p) Member function **set_extension(T*)** shall set the pointer to the extension object of type **T** in the array of pointers with the value of the argument. The argument shall be a pointer to a registered extension. The return value of the function may be a null pointer. In previous versions of this standard, the return value was intended to be the previous value of the pointer to an existing extension of the same type; therefore, the return value may be other than a null pointer when needed for backward compatibility. The member function **set_auto_extension(T*)** shall behave similarly, except that the extension shall be marked for automatic deletion. These member functions shall not be used to attach multiple extensions of the same type to the same generic payload; such implementation shall generate a standard error response.

q) Member function **set_extension(unsigned int, tlm_extension_base*)** shall set the pointer to the extension object in the array of pointers at the array index given by the first argument with the value of the second argument. The given index shall have been registered as an extension ID; otherwise, the behavior of the function is undefined. The return value of the function shall be the previous value of the pointer at the given array index, which may be a null pointer. In previous versions of this standard, the return value was intended to be the previous value of the pointer to an existing extension of the same type; therefore, the return value may be other than a null pointer when needed for backward compatibility, The member function **set_auto_extension(unsigned int, tlm_extension_base*)** shall behave similarly, except that the extension shall be marked for automatic deletion. These member functions shall not be used to attach multiple extensions of the same type to the same generic payload; such implementation shall generate a standard error response.

r) In the presence of a memory manager, a call to **set_auto_extension** for a given extension is equivalent to a call to **set_extension** immediately followed by a call to **release_extension** for that same extension. In the absence of a memory manager, a call to **set_auto_extension** will cause a run-time error.

s) If an extension is marked for automatic deletion, the given extension object should be deleted or pooled by the implementation of the member function **free** of a user-defined memory manager. Member function **free** is called by member function **release** of class **tlm_generic_payload** when the

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

reference count of the transaction object reaches 0. The extension object may be deleted by calling member function **reset** of class **tlm_generic_payload** or by calling member function **free** of the extension object itself.

t) If the generic payload object already contained a non-null pointer to an extension of the type being set, then the old pointer is overwritten.

u) Member functions **get_extension(T*&)** and **T* get_extension()** shall each return a pointer to the extension object of the given type, if it exists, or a null pointer if it does not exist. The type **T** shall be a pointer to an object of a type derived from **tlm_extension**. It is not an error to attempt to retrieve a non-existent extension using this function template.

v) Member function **get_extension(unsigned int)** shall return a pointer to the extension object with the ID given by the argument. The given index shall have been registered as an extension ID; otherwise, the behavior of the function is undefined. If the pointer at the given index does not point to an extension object, the function shall return a null pointer.

w) Member functions **clear_extension(const T*)** and **clear_extension()** shall remove the given extension from the generic payload object, that is, shall set the corresponding pointer in the extension array to null. The extension may be specified either by passing a pointer to an extension object as an argument or by using the function template parameter type, for example, **clear_extension<ext_type>()**. If present, the argument shall be a pointer to an object of a type derived from **tlm_extension**. Member function **clear_extension** shall not delete the extension object.

x) Member functions **release_extension(T*)** and **release_extension()** shall mark the extension for automatic deletion if the transaction object has a memory manager or otherwise shall delete the given extension by calling the member function **free** of the extension object and setting the corresponding pointer in the extension array to null. The extension may be specified either by passing a pointer to an extension object as an argument, or by using the function template parameter type, for example, **release_extension<ext_type>()**. If present, the argument shall be a pointer to an object of a type derived from **tlm_extension**.

y) Note that the behavior of member function **release_extension** depends on whether or not the transaction object has a memory manager. With a memory manager, the extension is merely marked for automatic deletion, and continues to be accessible. In the absence of a memory manager, not only is the extension pointer cleared but also the extension object itself is deleted. Care should be taken not to release a non-existent extension object because doing so will result in a run-time error.

z) Member functions **clear_extension** and **release_extension** shall not be called for extensions marked for automatic deletion, for example, an extension set using **set_auto_extension** or already released using **release_extension**. Doing so may result in a run-time error.

aa) Each generic payload transaction should allocate sufficient space to store pointers to every registered extension. This can be achieved in one of two ways, either by constructing the transaction object *after* C++ static initialization or by calling the member function **resize_extensions** *after* static initialization but *before* using the transaction object for the first time. In the former case, it is the responsibility of the generic payload constructor to set the size of the extension array. In the latter case, it is the responsibility of the application to call **resize_extensions** before accessing the extensions for the first time.

ab) Member function **resize_extensions** shall increase the size of the extensions array in the generic payload to accommodate every registered extension.

Example:

```
// Showing an ignorable extension

// User-defined extension class
struct ID_extension : tlm::tlm_extension<ID_extension> {
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    ID_extension() : transaction_id(0) {}

    virtual tlm_extension_base *clone() const { // Must override pure virtual clone member function
        ID_extension *t = new ID_extension;
        t->transaction_id = this->transaction_id;
        return t;
    }

    // Must override pure virtual copy_from member function
    virtual void copy_from(tlm_extension_base const &ext) {
        transaction_id = static_cast<ID_extension const &>(ext).transaction_id;
    }

    unsigned int transaction_id;
};

// The initiator
struct Initiator : sc_core::sc_module {
    ...

    void thread() {
        tlm::tlm_generic_payload trans;
        ...
        ID_extension *id_extension = new ID_extension;
        trans.set_extension(id_extension);          // Add the extension to the transaction

        for (int i = 0; i < RUN_LENGTH; i += 4) {
        ...
            ++id_extension->transaction_id;         // Increment the id for each new transaction
            ...
            socket->b_transport(trans, delay);
            ...
        }
    }
}

// The target
virtual void b_transport(tlm::tlm_generic_payload &trans, sc_core::sc_time &t) {
    ...
    ID_extension *id_extension;
    trans.get_extension(id_extension);              // Retrieve the extension
    if (id_extension) {                             // Extension is not mandatory
        char txt[80];
        sprintf(txt, "Received transaction id %d", id_extension->transaction_id);
        SC_REPORT_INFO("TLM-2.0", txt);
    }
    ...
}

// Showing a new protocol traits class with a mandatory extension

struct cmd_extension : tlm::tlm_extension<cmd_extension> { // User-defined mandatory extension class
    cmd_extension() : increment(false) {}
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    virtual tlm::tlm_extension_base *clone() const {
        cmd_extension *t = new cmd_extension;
        t->increment = this->increment;
        return t;
    }

    virtual void copy_from(tlm::tlm_extension_base const &ext) {
        increment = static_cast<cmd_extension const &>(ext).increment;
    }

    bool increment;
};

struct my_protocol_types {                                  // User-defined protocol traits class
    typedef tlm::tlm_generic_payload tlm_payload_type;
    typedef tlm::tlm_phase tlm_phase_type;
};

struct Initiator : sc_core::sc_module {
    tlm_utils::simple_initiator_socket<Initiator, 32, my_protocol_types> socket;
    ...

    void thread() {
        tlm::tlm_generic_payload trans;
        cmd_extension *extension = new cmd_extension;
        trans.set_extension(extension);                     // Add the extension to the transaction
        ...
        trans.set_command(tlm::TLM_WRITE_COMMAND);          // Execute a write command
        socket->b_transport(trans, delay);
        ...
        trans.set_command(tlm::TLM_IGNORE_COMMAND);
        extension->increment = true;                        // Execute an increment command
        socket->b_transport(trans, delay);
        ...
    }
    ...
}

// The target
tlm_utils::simple_target_socket<Memory, 32, my_protocol_types> socket;

virtual void b_transport(tlm::tlm_generic_payload &trans, sc_core::sc_time &t) {
    tlm::tlm_command cmd = trans.get_command();
    ...
    cmd_extension *extension;
    trans.get_extension(extension);                         // Retrieve the command extension
    ...
    if (!extension) {                                       // Check the extension exists
        trans.set_response_status(tlm::TLM_GENERIC_ERROR_RESPONSE);
        return;
    }

    if (extension->increment) {
        if (cmd != tlm::TLM_IGNORE_COMMAND) {               // Detect clash with read or write
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
            trans.set_response_status(tlm::TLM_GENERIC_ERROR_RESPONSE);
            return;
        }
        ++m_storage[adr];                                   // Execute an increment command
        memcpy(ptr, &m_storage[adr], len);
    }
    ...
}
...
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

15. TLM-2.0 base protocol and phases

15.1 Phases

15.1.1 Introduction

Class **tlm_phase** is the default phase type used by the non-blocking transport interface class templates and the base protocol. Class **tlm_phase** is assignment compatible with an enumeration having values corresponding to the four phases of the base protocol, namely BEGIN_REQ, END_REQ, BEGIN_RESP, and END_RESP. Because type **tlm_phase** is a class rather than an enumeration, it is able to support an overloaded stream operator to display the value of the phase as ASCII text.

The set of four phases provided by **tlm_phase_enum** can be extended using the macro TLM_DECLARE_EXTENDED_PHASE. This macro creates a singleton class derived from **tlm_phase** with a member function **get_phase** that returns the corresponding object. That object can be used as a new phase.

For maximal interoperability, an application should only use the four phases of **tlm_phase_enum**. If further phases are required in order to model the details of a specific protocol, the intent is that TLM_DECLARE_EXTENDED_PHASE should be used since this retains assignment compatibility with type **tlm_phase**.

The principle of ignorable versus non-ignorable or mandatory extensions applies to phases in the same way as to generic payload extensions. In other words, ignorable phases are permitted by the base protocol. An ignorable phase has to be ignorable by the recipient in the sense that the recipient can simply act as if it had not received the phase transition, and consequently, the sender has to be able to continue in the absence of any response from the recipient. If a phase is not ignorable in this sense, a new protocol traits class should be defined (see 14.2.3).

15.1.2 Class definition

```
namespace tlm {

enum tlm_phase_enum {
    UNINITIALIZED_PHASE=0, BEGIN_REQ=1, END_REQ, BEGIN_RESP, END_RESP };

class tlm_phase{
public:
    tlm_phase();
    tlm_phase( const tlm_phase_enum& );
    tlm_phase& operator= ( const tlm_phase_enum& );
    operator unsigned int() const;
};

inline std::ostream& operator<< ( std::ostream& , const tlm_phase& );

#define TLM_DECLARE_EXTENDED_PHASE(name_arg) \
class tlm_phase_##name_arg : public tlm::tlm_phase{ \
public:\
    static const tlm_phase_##name_arg& get_phase();\
    implementation-defined \
}; \
static const tlm_phase_##name_arg& name_arg=tlm_phase_##name_arg::get_phase()
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
} // namespace tlm
```

15.1.3 Rules

a) The default constructor **tlm_phase** shall set the value of the phase to 0, corresponding to the enumeration literal UNINITIALIZED_PHASE.

b) Member functions **tlm_phase(unsigned int)**, **operator=**, and **operator unsigned int** shall get or set the value of the phase using the corresponding unsigned int or enum.

c) The function **operator<<** shall write a character string corresponding to the name of the phase to the given output stream. For example, "BEGIN_REQ".

d) The macro TLM_DECLARE_EXTENDED_PHASE(name_arg) shall create a new singleton class named **tlm_phase_name_arg**, derived from **tlm_phase**, and having a public member function **get_phase** that returns a reference to the static object so created. The macro argument shall be used as the character string written by **operator<<** to denote the corresponding phase.

e) The invocation of the macro TLM_DECLARE_EXTENDED_PHASE shall be terminated with a semicolon.

f) The intent is that the object denoted by the static const **name_arg** represents the extended phase that may be passed as a phase argument to *nb_transport*.

Example:

```
TLM_DECLARE_EXTENDED_PHASE(ignore_me);      // Declare two extended phases
TLM_DECLARE_EXTENDED_PHASE(internal_ph);    // Only used within target

struct Initiator : sc_core::sc_module {
    ...
    void thread() {
        ...
        phase = tlm::BEGIN_REQ;
        delay = sc_core::sc_time(10.0, sc_core::SC_NS);
        socket->nb_transport_fw(trans, phase, delay);      // Send phase BEGIN_REQ to target
        phase = ignore_me;                                 // Set phase variable to the extended phase
        delay = sc_core::sc_time(12.0, sc_core::SC_NS);
        socket->nb_transport_fw(trans, phase, delay);      // Send the extended phase 2ns later
        ...
    }
};

struct Target : sc_core::sc_module {
    ...

    SC_CTOR(Target)
    : m_peq("m_peq", this, &Target::peq_cb) {}             // Register callback with PEQ

    virtual tlm::tlm_sync_enum nb_transport_fw(tlm::tlm_generic_payload &trans, tlm::tlm_phase &phase,
sc_core::sc_time &delay) {
        std::cout << "Phase = " << phase << std::endl;     // use overloaded operator<< to print phase
        m_peq.notify(trans, phase, delay);                 // Move transaction to internal queue
        return tlm::TLM_ACCEPTED;
    }
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    void peq_cb(tlm::tlm_generic_payload &trans, const tlm::tlm_phase &phase) {          // PEQ callback
        sc_core::sc_time delay;
        tlm::tlm_phase phase_out;

        if (phase == tlm::BEGIN_REQ) {                      // Received BEGIN_REQ from initiator
            phase_out = tlm::END_REQ;
            delay = sc_core::sc_time(10.0, sc_core::SC_NS);
            socket->nb_transport_bw(trans, phase_out, delay);       // Send END_REQ back to initiator
            phase_out = internal_ph;                        // Use extended phase to signal internal event
            delay = sc_core::sc_time(15.0, sc_core::SC_NS);
            m_peq.notify(trans, phase_out, delay);                  // Put internal event into PEQ
        }
        else if (phase == internal_ph) {                            // Received internal event
            phase_out = tlm::BEGIN_RESP;
            delay = sc_core::sc_time(10.0, sc_core::SC_NS);
            socket->nb_transport_bw(trans, phase_out, delay);       // Send BEGIN_RESP back to initiator
        }
    }                                                               // Ignore phase ignore_me from initiator

    tlm_utils::peq_with_cb_and_phase<Target, tlm::tlm_base_protocol_types> m_peq;
};
```

15.2 Base protocol

15.2.1 Introduction

The base protocol consists of a set of rules to help ensure maximal interoperability between transaction level models of components that interface to memory-mapped buses. The base protocol requires the use of the classes of the TLM-2.0 interoperability layer listed here, together with the rules defined in this clause:

a) The TLM-2.0 core transport, direct memory, and debug transport interfaces (see Clause 11).
b) The socket classes **tlm_initiator_socket** and **tlm_target_socket** (or classes derived from these) (see 13.2).
c) The generic payload class **tlm_generic_payload** (see Clause 14).
d) The phase class **tlm_phase**.

The base protocol rules permit extensions to the generic payload and to the phases only if those extensions are ignorable. Non-ignorable extensions require the definition of a new protocol traits class (see 14.2.2).

The base protocol is represented by the pre-defined class **tlm_base_protocol_types**. However, this class contains nothing but two type definitions. All components that use this class (as template argument to a socket) are obliged by convention to respect the rules of the base protocol.

In cases where it is necessary to define a new protocol traits class (e.g., because the features of the base protocol are insufficient to model a particular protocol), the rules associated with the new protocol traits class override those of the base protocol. However, for consistency and interoperability, it is recommended that the rules and coding style associated with any new protocol traits class should follow those of the base protocol as far as possible (see 14.2.3).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Subclause 15.2 specifically concerns the base protocol, but nonetheless it may be used as a guide when modeling other protocols. Specific protocols represented by other protocol traits classes may include additional phases and may adopt their own rules for timing annotation, transaction ordering, and so forth. In doing so, they may cease to be compatible with the base protocol.

15.2.2 Class definition

```
namespace tlm {

struct tlm_base_protocol_types
{
    typedef    tlm_generic_payload        tlm_payload_type;
    typedef    tlm_phase                  tlm_phase_type;
};

} // namespace tlm
```

15.2.3 Base protocol phase sequences

a) The base protocol permits the use of the blocking transport interface, the non-blocking transport interface, or both together. The blocking transport interface does not carry phase information. When used with the base protocol, the constraints governing the order of calls to *nb_transport* are stronger than those governing the order of calls to **b_transport**. Hence *nb_transport* is more for the approximately-timed coding style, and **b_transport** is more for the loosely-timed coding style.

b) The full sequence of phase transitions for a given transaction through a given socket is:

BEGIN_REQ -> END_REQ -> BEGIN_RESP -> END_RESP

c) BEGIN_REQ and END_RESP shall be sent through initiator sockets only, and END_REQ and BEGIN_RESP through target sockets only.

d) In the case of the blocking transport interface, a transaction instance is associated with a single call to and return from **b_transport**. The correspondence between the call to **b_transport** and BEGIN_REQ, and the return from **b_transport** and BEGIN_RESP, is purely notional; **b_transport** has no associated phases.

e) For the base protocol, each call to *nb_transport* and each return from *nb_transport* with a value of TLM_UPDATED shall cause a phase transition. In other words, two consecutive calls to *nb_transport* for the same transaction shall have different values for the phase argument. Ignorable phase extensions are permitted, in which case the insertion of an extended phase shall count as a phase transition for the purposes of this rule, even if the phase is ignored.

f) The phase sequence can be cut short by having *nb_transport* return a value of TLM_COMPLETED but only in one of the following ways (Figure 28). An interconnect component or target may return TLM_COMPLETED when it receives BEGIN_REQ or END_RESP on the forward path. An interconnect component or initiator may return TLM_COMPLETED when it receives BEGIN_RESP on the backward path. A return value of TLM_COMPLETED indicates the end of the transaction with respect to a particular hop, in which case the phase argument should be ignored by the caller (see 11.2.3.7). TLM_COMPLETED does not imply successful completion, so the initiator should check the response status of the transaction for success or failure.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

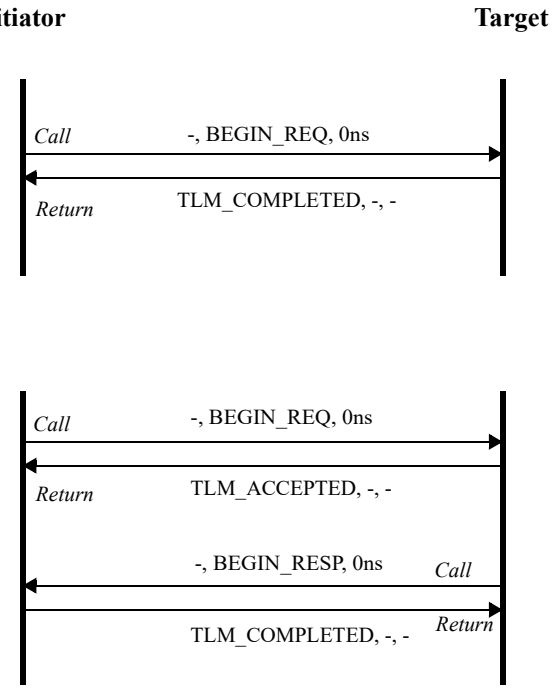

Figure 28—Examples of early completion

g) A transition to the phase END_RESP shall also indicate the end of the transaction with respect to a particular hop, in which case the callee is not obliged to return a value of TLM_COMPLETED.

h) When TLM_COMPLETED is returned in an upstream direction after having received BEGIN_REQ, this carries with it an implicit END_REQ and an implicit BEGIN_RESP. Hence, the initiator should check the response status of the generic payload, and may send BEGIN_REQ for the next transaction immediately.

i) Since TLM_COMPLETED returned after having received BEGIN_REQ carries with it an implicit BEGIN_RESP, this situation is forbidden by the response exclusion rule if there is already a response in progress through a given socket. In this situation, the callee should have returned TLM_ACCEPTED instead of TLM_COMPLETED and should wait for END_RESP before sending the next response upstream.

j) Since TLM_COMPLETED returned after having received BEGIN_REQ indicates the end of the transaction, an interconnect component or initiator is forbidden from then sending END_RESP for that same transaction through that same socket.

k) When TLM_COMPLETED is returned in a downstream direction by a component after having received BEGIN_RESP, this carries with it an implicit END_RESP.

l) If a component receives a BEGIN_RESP from a downstream component without having first received an END_REQ for that same transaction, the initiator shall assume an implicit END_REQ immediately preceding the BEGIN_RESP. This is only the case for the same transaction; a BEGIN_RESP does not imply an END_REQ for any other transaction, and a target that receives a BEGIN_REQ cannot infer an END_RESP for the previous transaction.

m) The above points hold regardless of the value of the timing annotation argument to *nb_transport*.

n) A base protocol transaction is complete (with respect to a particular hop) when TLM_COMPLETED is returned on either path, or when END_RESP is sent on the forward path or the return path.

o) In the case where END_RESP is sent on the forward path, the callee may return TLM_ACCEPTED or TLM_COMPLETED. The transaction is complete in either case.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

p) A given transaction may complete at different times on different hops. A transaction object passed to *nb_transport* is obliged to have a memory manager, and the lifetime of the transaction object ends when the reference count of the generic payload reaches zero. Any component that calls the member function **acquire** of a generic payload transaction object should also call the member function **release** at or before the completion of the transaction (see 14.5).

q) If a component receives an illegal or out-of-order phase transition, this is an error on the part of the sender. The behavior of the recipient is undefined, meaning that a run-time error may be caused.

15.2.4 Permitted phase transitions

Taking all of the rules in the previous clause into account, the set of permitted phase transitions over a given hop for the base protocol is shown in Table 57.

Table 57—Permitted phase transitions

Previous state		Calling path	Phase argument on call	Phase argument on return	Status on return	Response valid	End-of-life		Next state
//rsp		Forward	BEGIN_REQ	—	Accepted				req
//rsp		Forward	BEGIN_REQ	END_REQ	Updated				//req
//rsp		Forward	BEGIN_REQ	BEGIN_RESP	Updated	X			rsp
//rsp		Forward	BEGIN_REQ	—	Completed	X	X		//rsp
req		Backward	END_REQ	—	Accepted				//req
req		Backward	BEGIN_RESP	—	Accepted	X			rsp
req		Backward	BEGIN_RESP	END_RESP	Updated	X	X		//rsp
req		Backward	BEGIN_RESP	—	Completed	X	X		//rsp
//req		Backward	BEGIN_RESP	—	Accepted	X			rsp
//req		Backward	BEGIN_RESP	END_RESP	Updated	X	X		//rsp
//req		Backward	BEGIN_RESP	—	Completed	X	X		//rsp
rsp		Forward	END_RESP	—	Accepted	X	X		//rsp
rsp		Forward	END_RESP	—	Completed	X	X		//rsp

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

a) req, //req, rsp, //rsp stand for BEGIN_REQ, END_REQ, BEGIN_RESP, and END_RESP, respectively.

b) These phase state transitions are independent of the value of the **sc_time** argument to *nb_transport* (the timing annotation). In other words, a call to *nb_transport* will cause the state transition shown in the table regardless of the value of the timing annotation. (The timing annotation may have the effect of delaying the subsequent execution of the transaction.)

c) The **previous state** column shows the state of a given hop before a call to *nb_transport*.

d) The **calling path** column indicates whether the corresponding member function is called on the forward path (**nb_transport_fw**) or backward path (**nb_transport_bw**).

e) The **phase argument on call** column gives the value of the phase argument on the call to *nb_transport*. This will be the phase presented to the callee.

f) The **phase argument on return** column gives the value of the phase argument on return from *nb_transport*. The phase argument is only valid if the member function returns TLM_UPDATED.

g) The **status on return** column gives the return value of the *nb_transport* method, Accepted (TLM_ACCEPTED), Updated (TLM_UPDATED), or Completed (TLM_COMPLETED).

h) The **response valid** column is checked if the response status attribute of the transaction is valid on return from the *nb_transport* method.

i) The **end-of-life** column is checked if the transaction has reached the end of its lifetime with respect to this hop, that is, if no further *nb_transport* calls are permitted for the given transaction over the given hop.

j) The **next state** column shows the state of a given hop after return from *nb_transport*.

k) A phase transition can be caused either by the caller (indicated by a '-' in the phase argument on return column) or by the callee.

l) Ignorable phase extensions may be inserted at any point between BEGIN_REQ and END_RESP.

m) A valid response does not indicate successful completion. The transaction may or may not have been successful.

n) Figure 29 presents, in a graphical format, the *nb_transport* call sequences permitted by the base protocol over a given hop. A traversal of the graph from Start to End gives a permitted call sequence for a single transaction instance. The rectangles show calls to *nb_transport* together with the value of the phase argument, the rounded rectangles show the status and where appropriate the value of the phase argument on return. The larger shaded rectangles show phase transitions.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

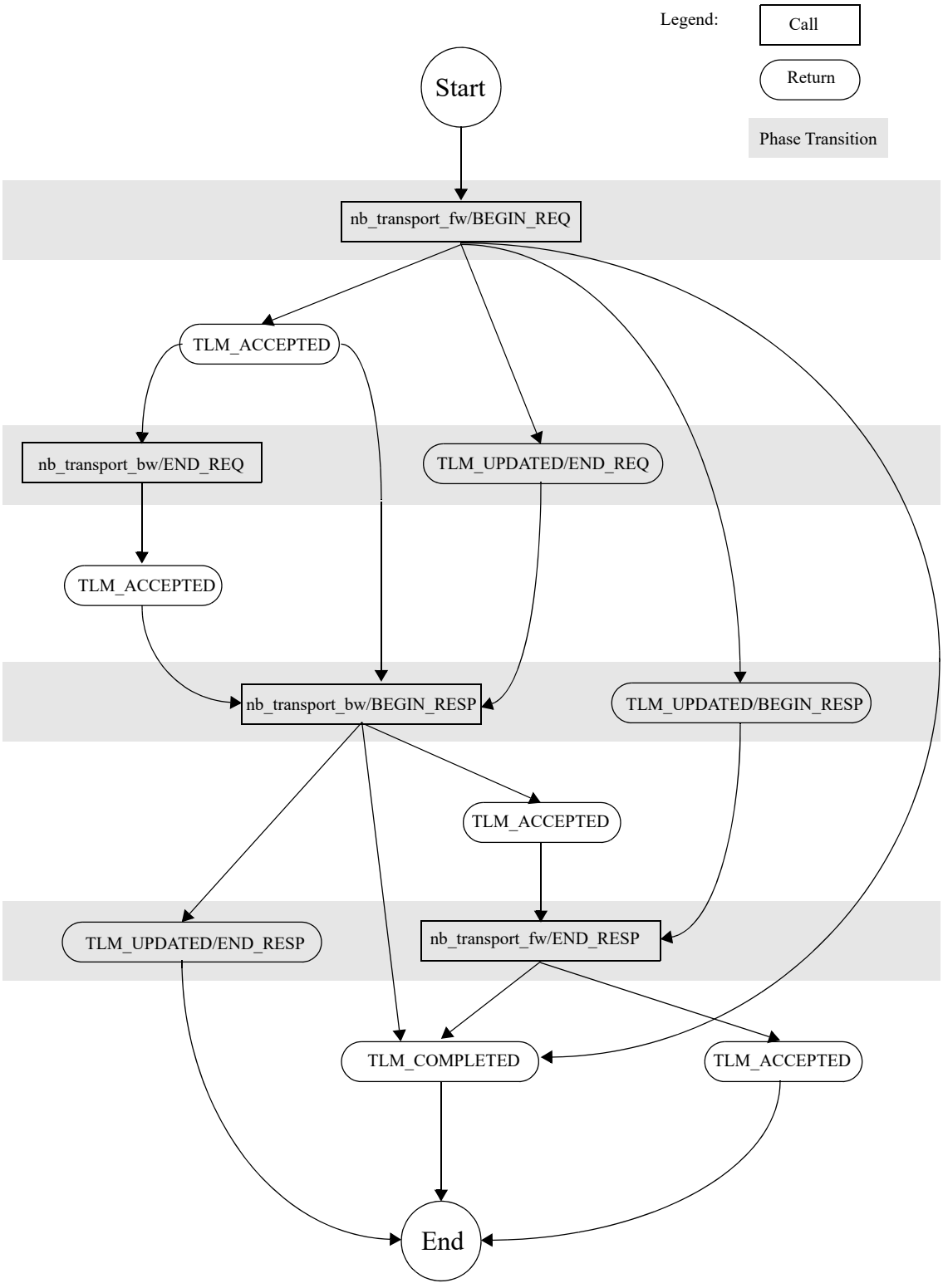

Figure 29—*nb_transport* call sequence for each base protocol transaction

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

15.2.5 Ignorable phases

Extended phases may be used with the base protocol provided that they are *ignorable phases*. An ignorable phase may be ignored by its recipient.

a) In general, the recommended way to add extended phases to the four phases of the base protocol is to define a new protocol traits class (see 14.2.3). Ignorable phases are a special and restricted case of extended phases. The main purpose of ignorable phases is to permit extra timing points to be added to the base protocol in order to increase the timing accuracy of the model. For example, an ignorable phase could mark the time of the start of the data transfer from initiator to target.

b) In the case of a call to *nb_transport*, if it is the callee that is ignoring the phase, it shall return a value of TLM_ACCEPTED. In the case that the callee returns TLM_UPDATED, the caller may ignore the phase being passed on the return path but is not obliged to take any specific action to indicate that the phase is being ignored.

c) The *nb_transport* interface does not provide any way for the caller of *nb_transport* to distinguish between the case where the callee is ignoring the phase, and the case where the callee will respond later on the opposite path. The callee shall return TLM_ACCEPTED in either case.

d) The presence of an ignorable phase shall not change the order or the semantics of the four phases BEGIN_REQ, END_REQ, BEGIN_RESP, and END_RESP of the base protocol, and is not permitted to result in any of the rules of the base protocol being broken.

e) An ignorable phase shall not occur before BEGIN_REQ or after END_RESP for a given transaction through a given socket. The presence of an ignorable phase before BEGIN_REQ or after END_RESP would violate the base protocol, and is an error.

f) The presence of an ignorable phase shall not change the rules concerning the validity of the generic payload attributes or the rules for modifying those attributes. For example, on receipt of an ignorable phase, an interconnect component is only permitted to modify the address attribute, DMI allowed attribute, and extensions (see 14.7).

g) With the exception of transparent components as defined below, if the recipient of an ignorable phase does not recognize that phase (that is, the phase is being ignored), the recipient shall not propagate that phase on the forward, the backward, or the return path. In other words, a component is only permitted to pass a phase as an argument to an *nb_transport* call if it fully understands the semantics of that phase.

h) If the recipient of an ignorable phase does recognize that phase, provided that the base protocol is not violated, the behavior of that component is otherwise outside the scope of the base protocol and is undefined by the base protocol. The recipient should obey the semantics of the extended protocol to which the phase belongs.

i) By definition, a component that sends an ignorable phase cannot require or demand any kind of response from the components to which that phase is sent other than the minimal response of *nb_transport* returning a value of TLM_ACCEPTED. A phase that demands a response is not ignorable, by definition, in which case the recommended approach is to define a new protocol traits class rather than using extensions to the base protocol. This prevents the binding of sockets that represent incompatible protocols.

j) On the other hand, a base-protocol-compliant component that does recognize an incoming extended phase may respond by sending another extended phase on the opposite path according to the rules of some extended protocol agreed in advance. This possibility is permitted by the TLM-2.0 standard, provided that the rules of the base protocol are not broken. For example, such an extended protocol could make use of generic payload extensions.

k) It is possible to create so-called *transparent* interconnect components, which immediately and directly pass through any TLM-2.0 interface method calls between a target socket and an initiator socket contained within the same component. The sole intent of recognizing transparent components

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

in this standard is to allow for checkers and monitors, which typically have one target socket, one initiator socket, and pass through all transactions in both directions without modification.

l) Within a transparent component, the implementation of any TLM-2.0 core interface method shall not consume any simulation time, insert any delay, or call **wait**, but shall immediately make the identical interface method call through the opposing socket (initiator socket to target socket or target socket to initiator socket), passing through all its arguments. Such an interface method shall not modify the value of any argument, including the transaction object, the phase, and the delay, with the one exception of generic payload extensions. The routing through such transparent components shall be fixed, and not depend on transaction attributes or phases.

m) As a consequence of the above rules, a transparent component would pass through any extended phase or ignorable phase in either direction.

Example:

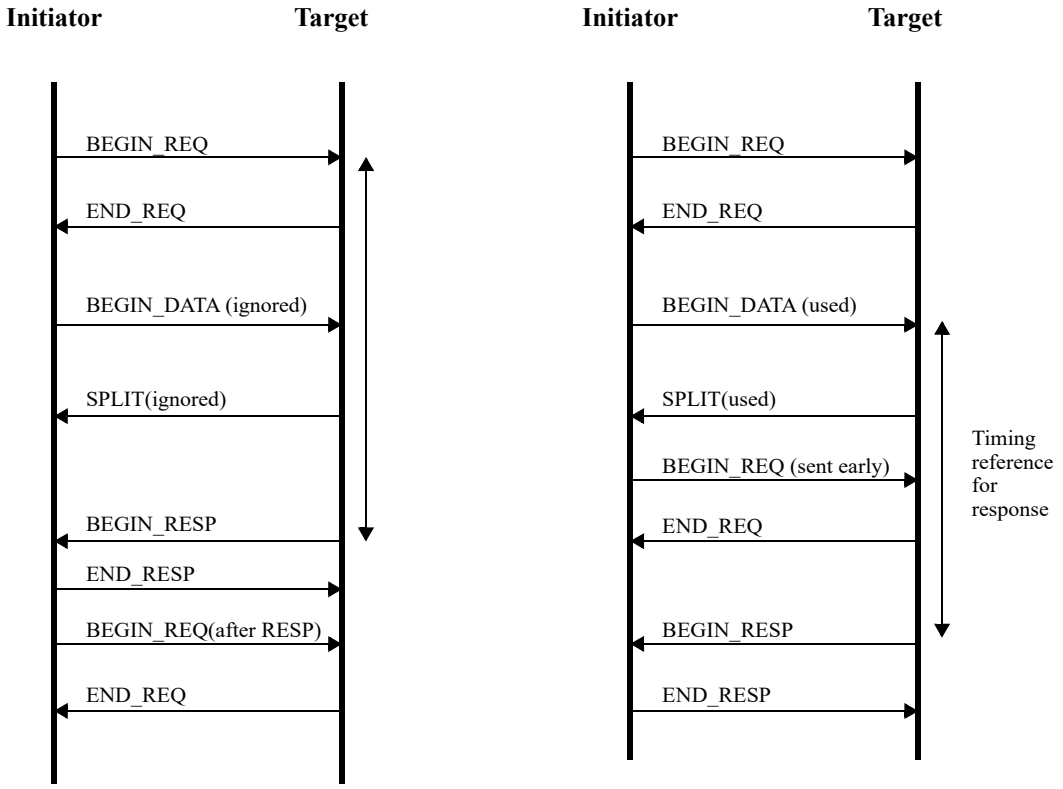

Figure 30—Ignorable phases

An example of an ignorable phase generated by an initiator would be a phase to mark the first beat of the data transfer from the initiator in the case of a write command (Figure 30). An interconnect component or target that recognized this phase could distinguish between the time at which the command and address become available and the start of the data transfer. A target that ignored this phase would have to use the BEGIN_REQ phase as its single timing reference for the availability of the command, address, and data.

An example of an ignorable phase generated by a target would be a phase to mark a split transaction. An initiator that recognized this phase could send the next request immediately on receiving the split phase, knowing that the target would be ready to process it. An initiator that ignored the split phase might wait until it had received a response to the first request before sending the second request.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

15.2.6 Base protocol timing parameters and flow control

a) With four phases, it is possible to model the request accept delay (or minimum initiation interval between sending successive transactions), the latency of the target, and the response accept delay. This kind of timing granularity is appropriate for the approximately-timed coding style (Figure 31).

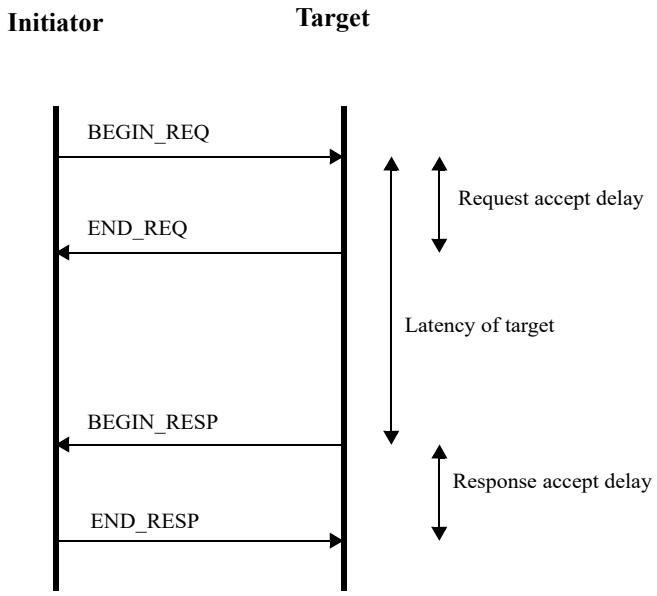

Figure 31—Approximately-timed timing parameters

b) For a write command, the BEGIN_REQ phase marks the time when the data becomes available for transfer from initiator through interconnect component to target. Notionally, the transition to the BEGIN_REQ phase corresponds to the start of the first beat of the data transfer. It is the responsibility of the downstream component to calculate the transfer time, and to send END_REQ back upstream when it is ready to receive the next transfer. It would be natural for the downstream component to delay the END_REQ until the end of the final beat of the data transfer, but it is not obliged to do so.

c) For a read command, the BEGIN_RESP phase marks the time when the data becomes available for transfer from target through interconnect component to initiator. Notionally, the transition to the BEGIN_RESP phase corresponds to the start of the first beat of the data transfer. It is the responsibility of the upstream component to calculate the transfer time, and to send END_RESP back downstream when it is ready to receive the next transfer. It would be natural for the upstream component to delay the END_RESP until the end of the final beat of the data transfer, but it is not obliged to do so.

d) For a read command, if a downstream component completes a transaction early by returning TLM_COMPLETED from **nb_transport_fw**, it is the responsibility of the upstream component to account for the data transfer time in some other way, if it wishes to do so (since it is not permitted to send END_RESP).

e) For the base protocol, an initiator or interconnect component shall not send a new transaction through a given socket with phase BEGIN_REQ until it has received END_REQ or BEGIN_RESP from the downstream component for the immediately preceding transaction or until the downstream component has completed the previous transaction by returning the value TLM_COMPLETED from **nb_transport_fw**. This is known as the *request exclusion rule*. For a multi-socket, the request

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

exclusion rule applies only to individual initiator-target interfaces bound to this socket (i.e., to each index individually).

f) For the base protocol, a target or interconnect component shall not respond to a new transaction through a given socket with phase BEGIN_RESP until it has received END_RESP from the upstream component for the immediately preceding transaction or until a component has completed the previous transaction over that hop by returning TLM_COMPLETED. This is known as the *response exclusion rule*. For a multi-socket, the response exclusion rule applies only to individual initiator-target interfaces bound to this socket (i.e., to each index individually) on the backward path.

g) All the rules governing phase transitions, including the request and response exclusion rules, shall be based on method call order alone, and shall not be affected by the value of the time argument (the timing annotation).

h) Successive transactions sent through a given socket using the non-blocking transport interface can be pipelined. By responding to each BEGIN_REQ (or BEGIN_RESP) with an END_REQ (or END_RESP), an interconnect component can permit any number of transaction objects to be in *flight* at the same time. By not responding immediately with END_REQ (or END_RESP), an interconnect component can exercise flow control over the stream of transaction objects coming from an initiator (or target).

i) This rule excluding the possibility of two outstanding requests or responses through a given socket shall only apply to the non-blocking transport interface, and shall have no direct effect on calls to **b_transport** (Figure 32 and Figure 33). (The rule may have an indirect effect on a call to **b_transport** in the case that **b_transport** itself calls **nb_transport_fw**.)

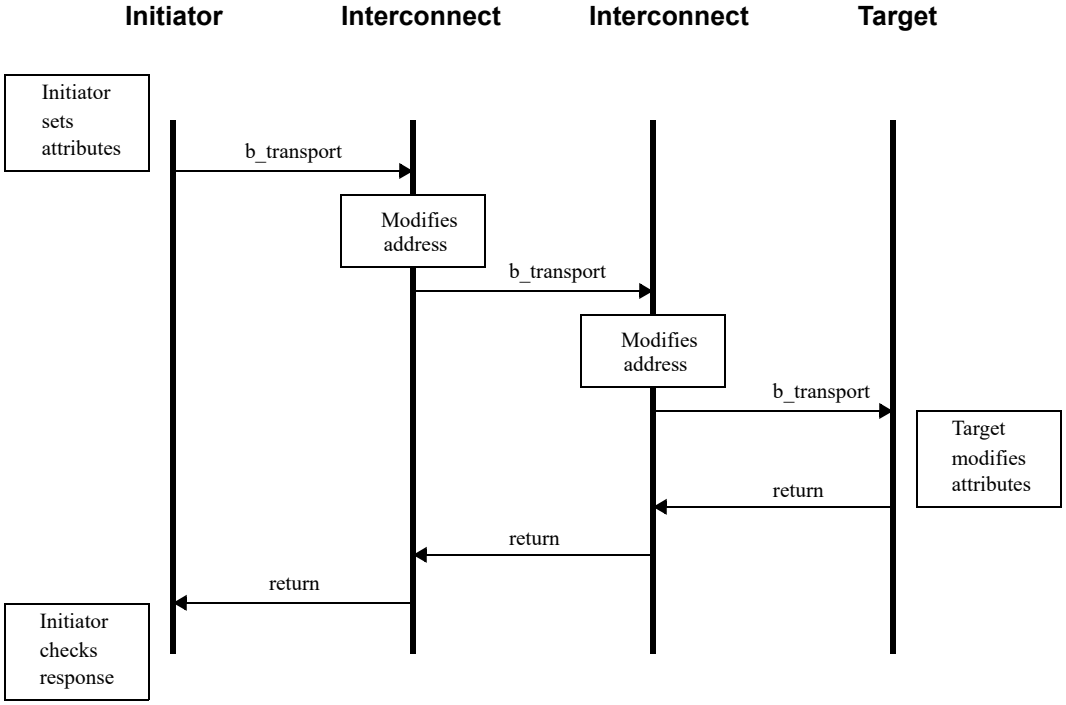

Figure 32—Causality with b_transport

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

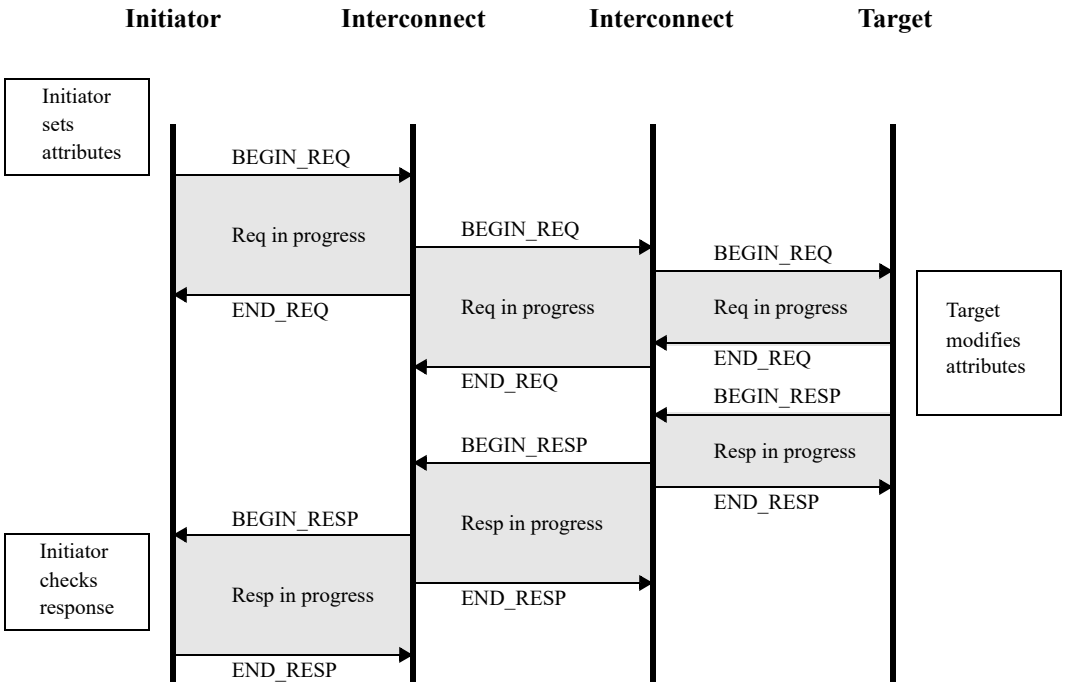

Figure 33—Causality with *nb_transport*

j) For a given transaction, BEGIN_REQ shall always start from an initiator and be propagated through zero or more interconnect components until it is received by a target. For a given transaction, an interconnect component is not permitted to send BEGIN_REQ to a downstream component before having received BEGIN_REQ from an upstream component.

k) For a given transaction, BEGIN_RESP shall always start from a target and be propagated through zero or more interconnect components until it is received by an initiator. For a given transaction, an interconnect component is not permitted to send BEGIN_RESP to an upstream component before having received BEGIN_RESP from a downstream component. This applies whether BEGIN_RESP is explicit or is implied by TLM_COMPLETED.

l) For a given transaction, an interconnect component may send END_REQ to an upstream component before having received END_REQ from a downstream component. Similarly, an interconnect component may send END_RESP to a downstream component before having received END_RESP from an upstream component. This applies whether END_REQ and END_RESP are explicit or implicit.

m) END_REQ and END_RESP are primarily for flow control between adjacent components. These two phases do not signal the validity of any standard generic payload attributes. Because these two phases are not propagated causally from end-to-end, they cannot reliably be used to signal the validity of extensions from initiator-to-target or target-to-initiator, but they can be used to signal the validity of extensions between two adjacent components.

n) Whether or not an interconnect component is permitted to send an extended phase before having received, the corresponding phase depends on the rules associated with the extended phase in question.

Example:

The following pseudo-code illustrates the interaction between timing annotation and the request and response exclusion rules:

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
void initiator_1_thread_process() {
    using namespace sc_core;
    using namespace tlm;

    // The initiator sends a request to be executed at +1000ns
    phase = BEGIN_REQ;
    delay = sc_time(1000.0, SC_NS);
    status = socket->nb_transport_fw(T1, phase, delay);
    assert(status == TLM_UPDATED && phase == END_REQ && delay == sc_time(1010.0, SC_NS));
    // END_REQ is returned immediately to be executed at +1010ns

    // Note that this is not a recommended coding style
    // With loosely-timed, the initiator would have called b_transport
    // With approximately-timed, the downstream component would have returned TLM_ACCEPTED
    // in order to synchronize, and the initiator would have been forced to wait for END_REQ

    // The initiator is allowed to send the next request immediately, to be executed at +1050ns
    phase = BEGIN_REQ;
    delay = sc_time(1050.0, SC_NS);
    status = socket->nb_transport_fw(T2, phase, delay);
    assert(status == TLM_UPDATED && phase == END_REQ && delay == sc_time(1060.0, SC_NS));

    // The initiator is technically allowed to send the next request at an earlier local time of +500ns,
    // although the decreased timing annotation is not a recommended coding style
    phase = BEGIN_REQ;
    delay = sc_time(500.0, SC_NS);
    status = socket->nb_transport_fw(T3, phase, delay);
    assert(status == TLM_UPDATED && phase == END_REQ && delay == sc_time(510.0, SC_NS));

    // The initiator now yields control, allowing other initiators to resume and simulation time to advance
    wait(...);
}
void initiator_2_thread_process() {
    using namespace sc_core;
    using namespace tlm;

    // Assume the calls below are appended to the transaction stream sent from the first initiator above

    // The second initiator sends a request to be executed at +10ns
    // The timing annotation as seen downstream has decreased from +510ns to +10ns
    // This is typical behavior for loosely-timed initiators
    phase = BEGIN_REQ;
    delay = sc_time(10.0, SC_NS);
    status = socket->nb_transport_fw(T4, phase, delay);
    assert(status == TLM_UPDATED && phase == END_REQ && delay == sc_time(30.0, SC_NS));
    // END_REQ is returned immediately to be executed at +30ns

    // The initiator sends the next request to be executed at +20ns, which overlaps with the previous request
    // This is technically allowed because the current phase of the hop is END_REQ,
    // but is not recommended
    phase = BEGIN_REQ;
    delay = sc_time(20.0, SC_NS);
    status = socket->nb_transport_fw(T5, phase, delay);
    assert(status == TLM_UPDATED && phase == END_REQ && delay == sc_time(40.0, SC_NS));
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    // END_REQ is returned immediately to be executed at +40ns

    // The initiator sends the next request to be executed at +0ns, which is before the previous two requests
    // This is technically allowed because the current phase of the hop is END_REQ,
    // but is not recommended
    phase = BEGIN_REQ;
    delay = sc_time(0.0, SC_NS);
    status = socket->nb_transport_fw(T6, phase, delay);
    assert(status == TLM_UPDATED && phase == END_REQ && delay == sc_time(60.0, SC_NS));
    // END_REQ is returned immediately to be executed at +60ns, overlapping the two previous requests
    // This is technically allowed, but is not recommended
    wait(...);
}
```

15.2.7 Base protocol rules concerning timing annotation

a) These rules should be read in conjunction with 11.2.4.

b) There are constraints on the way in which the implementations of **b_transport** and *nb_transport* are permitted to modify the time argument t such that the effective local time **sc_time_stamp() + t** is non-decreasing between function call and return (see 11.2.4.2).

c) For successive calls to and returns from *nb_transport* through a given socket for a given transaction, the sequence of effective local times shall be non-decreasing. The effective local time is given by the expression **sc_time_stamp() + t**, where t is the time argument to *nb_transport*. For this purpose, both calls to and returns from the function shall be considered as part of a single sequence. This applies on the forward and backward paths alike. The intent is that time should not run backward for a given transaction.

d) The preceding rule also applies between the call to and the return from **b_transport**. Again, see 11.2.4.2.

e) Moreover, for a given transaction object, as requests are propagated from initiator toward target and responses are propagated from target back toward initiator, the sequence of effective local times given by each successive transport method call and return shall be non-decreasing. Request propagation in this sense includes calls to **b_transport** and the BEGIN_REQ phase. Response propagation includes returns from **b_transport**, the BEGIN_RESP phase, and TLM_COMPLETED.

f) The effective local time may be increased by increasing the value of the timing annotation (the time argument), by advancing SystemC simulation time (**b_transport** only), or both.

g) For *different* transaction objects, there is no obligation that the effective local times of calls to **b_transport** and *nb_transport* shall be non-decreasing. Nonetheless, each initiator process is generally recommended to call **b_transport** and/or *nb_transport* in non-decreasing effective local time order. Otherwise, downstream components would infer that the out-of-order transactions had originated from separate initiators and would be free to choose the order in which those particular transactions were executed. However, transactions with out-of-order effective local times may arise wherever streams of transactions from different loosely-timed initiators converge.

h) For a given socket, an initiator is allowed to pass the same transaction object at different times through the blocking and non-blocking transport interfaces, the direct memory interface, and the debug transport interface. Also, an initiator is permitted to re-use the same trans*action* object for *different* transaction instances, all subject to the memory management rules of the generic payload (see 14.5).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

15.2.8 Base protocol rules concerning b_transport

a) **b_transport** calls are re-entrant. The implementation of **b_transport** can call **wait**, and meanwhile another call to **b_transport** can be made for a different transaction object from a different initiator with no constraint on the timing annotation.

b) In the case that there are multiple processes within the same initiator module, each process shall be regarded as being a separate initiator with respect to the transaction ordering rules. Specifically, there are no constraints on the ordering of **b_transport** calls made from different threads in the same module, regardless of whether those calls are made through the same initiator socket or through different sockets.

c) An interconnect or target can always deduce that multiple concurrent **b_transport** calls come from different initiator threads and from a different process to any concurrent **nb_transport_fw** calls, and therefore that there are no constraints on the mutual order of those calls. With **b_transport**, one request is allowed to overtake another.

d) It is forbidden to make a re-entrant call to **b_transport** for the same transaction object through the same socket.

Example:

The following pseudo-code fragments show a re-entrant b_transport call:

```
// Two initiator thread processes
void thread1() {
    socket->b_transport(T1, sc_core::sc_time(100.0, sc_core::SC_NS));
}

void thread2() {
    wait(10.0, sc_core::SC_NS);
    socket->b_transport(T2, sc_core::sc_time(50.0, sc_core::SC_NS)); // T2 overtakes T1
}

// Implementation of b_transport in the target
void b_transport( TRANS& trans, sc_core::sc_time& t ) {
    wait(t);
    execute(trans);                                    // T1 executed at 100ns, T2 executed at 60ns
    t = sc_core::SC_ZERO_TIME;
}
```

15.2.9 Base protocol rules concerning request and response ordering

The intent of the following rules is to ensure that when an initiator sends a series of pipelined requests to a particular target, those requests will be executed at the target in the same order as they were sent from the initiator. Because the generic payload transaction stores neither the identity of the initiator nor the identity of the target, the initiator can only be inferred from the identity of the incoming socket, and the target can only be inferred from the values of the address and command attributes. The execution order of requests sent to non-overlapping addresses is not guaranteed.

a) The base protocol permits incoming requests or responses arriving through different sockets to be mutually delayed, interleaved, or executed in any order. For example, an interconnect component may assign a higher priority to requests arriving through a particular target socket or having a particular data length, allowing them to overtake lower priority requests. As another example, an interconnect component may re-order responses arriving back through different initiator sockets to match the order in which the corresponding requests were originally received.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

b) Request routing shall be deterministic and shall depend only on the address and command attributes of the transaction object. (These are the only attributes common to the transport, DMI, and debug transport interfaces.) The address map shall not be modified while there are transactions in progress.

c) If an initiator or interconnect component sends multiple concurrent requests with overlapping addresses on the forward path, those requests shall be routed through the same initiator socket. *Multiple concurrent requests* means requests for which the corresponding responses have not yet been received from the target. *Overlapping addresses* means that at least one byte in the data arrays of the transaction objects shares the same address. Read and write requests to the same address may be routed through different sockets provided they are not concurrent.

d) If an interconnect component (or a target) receives multiple concurrent requests with overlapping addresses through the same target socket by means of incoming calls to **nb_transport_fw**, those requests shall be sent forward (or executed, respectively) in the same order as they were received. *The same order* means the same interface method call order. (Note that if the interface method call order and the effective local time order of a set of transactions were to differ, any component receiving those transactions would be permitted to execute them in any order, regardless of the address. Also note that this rule holds even if the requests in question originate from different initiators.)

e) Note that the preceding rule does not apply for incoming **b_transport** calls, for which there are no constraints on the order of multiple concurrent requests. On the other hand, if incoming **nb_transport_fw** calls are converted to outgoing **b_transport** calls, the **b_transport** calls must be serialized to enforce the ordering rules of the *nb_transport* calls.

f) On the other hand, an interconnect component or target is permitted to re-order multiple concurrent requests that were received through different target sockets, or are sent through different initiator sockets, or whose addresses do not overlap, or for incoming **b_transport** calls.

g) Responses may be re-ordered. There is no guarantee that responses will arrive back at an initiator in the same order as the corresponding requests were sent.

h) Note that it would be technically possible with the base protocol to use an ignorable extension that allowed an interconnect component to re-order multiple concurrent requests, in which case the initiator that added the extension must be able to tolerate out-of-order execution at the target. On the other hand, an extension that forced responses to arrive back in the same order as requests were sent would not be an ignorable extension and, hence, would not be permitted by the base protocol.

15.2.10 Base protocol rules for switching between b_transport and *nb_transport*

a) Each thread within an initiator or an interconnect component is permitted to switch between calling **b_transport** and **nb_transport_fw** for different transaction objects. The intent is to permit an initiator to make occasional switches between the loosely-timed and approximately-timed coding styles. An initiator that interleaves calls to **b_transport** and **nb_transport_fw** should have low expectations with regard to timing accuracy.

b) Every interconnect component and target is obliged to support both the blocking and non-blocking transport interfaces, and to maintain any internal state information such that it is accessible from both interfaces. This applies to incoming interface method calls received through the same socket or through different sockets.

c) A thread within an initiator or an interconnect component shall not call both **b_transport** and **nb_transport_fw** for the same transaction instance. Note that a thread may call both **b_transport** and **nb_transport_fw** for the same transaction object provided that object represents a different transaction instance on each occasion.

d) It is recommended that a thread within an initiator or interconnect component should not call **b_transport** if there is still a transaction in progress from a previous **nb_transport_fw** call from that same thread, that is, when there is a previous transaction with a non-zero reference count. Otherwise, a downstream component could wrongly deduce that the two transactions had come from separate initiators.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

e) The convenience socket **simple_target_socket** provides an example of how a base protocol target can support both the blocking and the non-blocking transport interfaces while only being required to implement one of **b_transport** and **nb_transport_fw** (see 16.2.2).

15.2.11 Other base protocol rules

a) A given transaction object shall not be sent through multiple parallel sockets or along multiple parallel paths simultaneously. Each transaction instance shall take a unique well-defined path through a set of components and sockets that shall remain fixed for the lifetime of the transaction instance and is common to the transport, direct memory, and debug transport interfaces. Of course, different transactions sent through a given socket may take different paths; that is, they may be routed differently. Also, note that a component may choose dynamically whether to act as an interconnect component or as a target.

b) An upstream component should not know and should not need to know whether it is connected to an interconnect component or directly to a target. Similarly, a downstream component should not know and should not need to know whether it is connected to an interconnect component or directly to an initiator.

c) For a write transaction (TLM_WRITE_COMMAND), a response status of TLM_OK_RESPONSE shall indicate that the write command has completed successfully at the target. The target is obliged to set the response status before returning from **b_transport**, before sending BEGIN_RESP along the backward or return path, or before returning TLM_COMPLETED. In other words, an interconnect component is not permitted to signal the completion of a write transaction without having had confirmation of successful completion from the target. The intent of this rule is to guarantee the coherency of the storage within the target simulation model.

d) For a read transaction (TLM_READ_COMMAND), a response status of TLM_OK_RESPONSE shall indicate that the read command has completed and the generic payload data array has been modified by the target. The target is obliged to set the response status before returning from **b_transport**, before sending BEGIN_RESP along the backward or return path, or before returning TLM_COMPLETED.

15.2.12 Summary of base protocol transaction ordering rules

Table 58 gives a summary of the transaction ordering rules for the base protocol. For a full description of the rules, refer to the clauses above.

The base protocol ordering rules are a union of the rules from three categories: rules of the core transport interfaces concerning timing annotation, rules specific to the base protocol concerning causality and phases, and rules specific to the base protocol that ensure that pipelined requests are executed at the target in the order expected by the initiator.

Table 58—Base protocol transaction ordering rules

Circumstance	Ordering rule
Effective local time order different from interface method call order	Recipient may execute or route transactions in any order. Takes precedence over all other rules
Successive transport method calls and returns for the same transaction through the same socket	Effective local time order shall be non-decreasing
Successive transport method calls from the same initiator process	Effective local time order is recommended to be non-decreasing

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Table 58—Base protocol transaction ordering rules *(continued)*

Circumstance	Ordering rule
Successive transport method calls from the same initiator process	Recommended not to call **b_transport** if previous *nb_transport* transaction is still alive
Successive transport method calls for different transactions through the same socket	No obligation on effective local time order, but recommended to be non-decreasing if incoming transaction stream was non-decreasing
With *nb_transport* only, two requests or two responses outstanding through the same socket	Forbidden
Transactions incoming through different sockets	No obligations on the order in which they are executed or routed on
Multiple concurrent requests with overlapping addresses	If routed forward, shall be sent through the same socket
Multiple concurrent requests with overlapping addresses incoming through the same socket using *nb_transport*	Shall be executed or routed forward in the same order as they were received
Multiple concurrent requests incoming using **b_transport**	No obligations on the order in which they are executed or routed forward
Multiple concurrent responses	No obligations on the order in which they are executed or routed back

15.2.13 Guidelines for creating base-protocol-compliant components

15.2.13.1 Overview

Subclause 15.2.13 contains a set of guidelines for creating base protocol components. This is just a brief restatement of some rules presented more fully elsewhere in this document, and is provided for convenience.

15.2.13.2 Guidelines for creating a base protocol initiator

a) Instantiate one initiator socket of class **tlm_initiator_socket** (or a derived class) for each connection to a memory-mapped bus.

b) Allow the **tlm_initiator_socket** to take the default value **tlm_base_protocol_types** for the template TYPES argument.

c) Implement the member functions **nb_transport_bw** and **invalidate_direct_mem_ptr**. (An initiator can avoid the need to implement these member functions explicitly by instantiating the convenience socket **simple_initiator_socket**.)

d) Set every attribute of each generic payload transaction object before passing it as an argument to **b_transport** or **nb_transport_fw**, remembering in particular to reset the response status and DMI hint attributes before the call. (The byte enable length attribute need not be set in the case where the byte enable pointer attribute is 0, and extensions need not be used.)

e) When using the generic payload extension mechanism, check that any extensions are ignorable by the target and any interconnect component.

f) Obey the base protocol rules concerning phase sequencing, flow control, timing annotation, and transaction ordering.

g) On completion of the transaction (or after receiving BEGIN_RESP), check the value of the response status attribute.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

15.2.13.3 Guidelines for creating an initiator that calls *nb_transport*

a) Before passing a transaction as an argument to **nb_transport_fw**, set a memory manager for the transaction object and call the member function **acquire** of the transaction. Call the member function **release** when the transaction is complete.

b) When calling **nb_transport_fw**, set the phase argument to BEGIN_REQ or END_RESP according to state of the transaction. Do not send BEGIN_REQ before having received (or inferred) the END_REQ from the previous transaction.

c) When making a series of calls to **nb_transport_fw** for a given transaction, check that the effective local times (simulation time + timing annotation) form a non-decreasing sequence of values.

d) Respond appropriately to the incoming phase values END_REQ and BEGIN_RESP whether received on the backward path (a call to **nb_transport_bw**), the return path (TLM_UPDATED returned from **nb_transport_fw**), or implicitly (for example, TLM_COMPLETED returned from **nb_transport_fw**). Incoming phase values of BEGIN_REQ and END_RESP would be illegal. Treat all other incoming phase values as being ignorable.

15.2.13.4 Guidelines for creating a base protocol target

a) Instantiate one target socket of class **tlm_target_socket** (or a derived class) for each connection to a memory-mapped bus.

b) Allow the **tlm_target_socket** to take the default value **tlm_base_protocol_types** for the template TYPES argument.

c) Implement the member functions **b_transport**, **nb_transport_fw**, **get_direct_mem_ptr**, and **transport_dbg**. (A target can avoid the need to implement every member function explicitly by using the convenience socket **simple_target_socket**.)

d) In the implementations of the member functions **b_transport** and **nb_transport_fw**, inspect and act on the value of every attribute of the generic payload with the exception of the response status, the DMI hint, and any extensions. Rather than implementing the full functionality of the generic payload, a target may choose to respond to a given attribute by generating an error response. Set the value of the response status attribute to indicate the success or failure of the transaction.

e) Obey the base protocol rules concerning phase sequencing, flow control, timing annotation, and transaction ordering.

f) In the implementation of **get_direct_mem_ptr**, either return the value **false**, or inspect and act on the values of the command and address attributes of the generic payload and set the return value and all the attributes of the DMI descriptor appropriately (class **tlm_dmi**).

g) In the implementation of **transport_dbg**, either return the value 0, or inspect and act on the values of the command, address, data length, and data pointer attributes of the generic payload.

h) For each interface, the target may inspect and act on any ignorable extensions in the generic payload, but is not obliged to do so.

15.2.13.5 Guidelines for creating a target that calls *nb_transport*

a) When calling **nb_transport_bw**, set the phase argument to END_REQ or BEGIN_RESP according to state of the transaction. Do not send BEGIN_RESP before having received (or inferred) END_RESP from the previous transaction.

b) When making a series of calls to **nb_transport_bw** for a given transaction, check that the effective local times (simulation time + timing annotation) form a non-decreasing sequence of values.

c) Respond appropriately to the incoming phase values BEGIN_REQ and END_RESP, whether received on the forward path (a call to **nb_transport_fw**), the return path (TLM_UPDATED returned from **nb_transport_bw**), or implicitly (for example, TLM_COMPLETED returned from

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

nb_transport_bw). Incoming phase values of END_REQ and BEGIN_RESP would be illegal. Treat all other incoming phase values as being ignorable.

d) In the implementation of **nb_transport_fw**, when needing to keep a pointer or reference to a transaction object beyond the return from the member function, call the member function **acquire** of the transaction. Call the member function **release** when the transaction object is finished.

15.2.13.6 Guidelines for creating a base protocol interconnect component

a) Instantiate one initiator or target socket of class **tlm_initiator_socket** or **tlm_target_socket** (or derived classes) for each connection to a memory-mapped bus.

b) Allow each socket to take the default value **tlm_base_protocol_types** for the template TYPES argument.

c) Implement the member functions **nb_transport_bw** and **invalidate_direct_mem_ptr** for each initiator socket, and the member functions **b_transport**, **nb_transport_fw**, **get_direct_mem_ptr**, and **transport_dbg** for each target socket. (The need to implement every member function explicitly can be avoided by using the convenience sockets.)

d) Pass on incoming interface method calls as appropriate on both the forward and backward paths, honoring the request and response exclusion rules, the transaction ordering rules, and the rule that no further calls are allowed following TLM_COMPLETED. Do not pass on ignorable phases. The implementations of the member functions **get_direct_mem_ptr** and **transport_dbg** may return the values **false** and 0, respectively, without forwarding the transaction object.

e) In the implementation of the transport interfaces, the only generic payload attributes modifiable by an interconnect component are the address, DMI hint, and extensions. Do not modify any other attributes. A component needing to modify any other attributes should construct a new transaction object, and thereby become an initiator in its own right.

f) Decode the generic payload address attribute on the forward path and modify the address attribute if necessary according to the location of the target in the system memory map. This applies to the transport, direct memory, and debug transport interfaces.

g) In the implementation of the transport interfaces, obey the base protocol rules concerning phase sequencing, flow control, timing annotation, and transaction ordering.

h) In the implementation of **get_direct_mem_ptr**, do not modify the DMI descriptor attributes on the forward path. Do modify the DMI pointer, DMI start address and end address, and DMI access attributes appropriately on the return path.

i) In the implementation of **invalidate_direct_mem_ptr**, modify the address range arguments before passing the call along the backward path.

j) In the implementation of **nb_transport_fw**, when needing to keep a pointer or reference to a transaction object beyond the return from the function, call the member function **acquire** of the transaction. Call the member function **release** when the transaction object is finished.

k) For each interface, the interconnect component may inspect and act on any ignorable extensions in the generic payload but is not obliged to do so. If the transaction needs to be extended further, make sure any extensions are ignorable by the other components. Honor the generic payload memory management rules for extensions.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

16. TLM-2.0 utilities

16.1 Overview

The utilities comprise a set of classes that are provided for convenience and for consistent coding style. The utilities do not belong to the interoperability layer, so use of the utilities is not a requirement for interoperability.

16.2 Convenience sockets

16.2.1 Introduction

16.2.1.1 Overview

There is a family of convenience sockets, each socket implementing some additional functionality to make component models easier to write. The convenience sockets are derived from the classes **tlm_initiator_socket** and **tlm_target_socket**. They are not part of the TLM-2.0 interoperability layer but are to be found in the namespace tlm_utils.

16.2.1.2 Summary of standard and convenience socket types

The convenience sockets are summarized in Table 59 as follows:

— **Register callbacks?** The socket provides member functions to register callbacks for incoming interface method calls, rather than having the socket be bound to an object that implements the corresponding interfaces.

— **Multi-ports?** The socket class template provides number-of-bindings and binding policy template arguments such that a single initiator socket can be bound to multiple target sockets and vice versa.

— **b - nb conversion?** The target socket is able to convert incoming calls to **b_transport** into **nb_transport_fw** calls, and vice versa. A '-' indicates an initiator socket.

— **Tagged?** Incoming interface method calls are tagged with an id to indicate the socket through which they arrived.

Table 59—Socket types

Class	Register callbacks?	Multi-ports?	b - nb conversion?	Tagged?
tlm_initiator_socket	no	yes	—	no
tlm_target_socket	no	yes	no	no
simple_initiator_socket	yes	no	—	no
simple_initiator_socket_tagged	yes	no	—	yes
simple_target_socket	yes	no	yes	no
simple_target_socket_tagged	yes	no	yes	yes
passthrough_target_socket	yes	no	no	no
passthrough_target_socket_tagged	yes	no	no	yes
multi_passthrough_initiator_socket	yes	yes	—	yes
multi_passthrough_target_socket	yes	yes	no	yes

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

16.2.1.3 Permitted socket bindings

The bindings permitted between the standard and convenience socket types are summarized in Table 60 and organized into four quarters. Each binding is of the form From(To) or From.bind(To), where From and To are sockets of the given type.

Table 60—Permitted socket bindings

To	tlm-init	simple-init	multi-init		tlm-targ	simple-targ	multi-targ
From	Hierarchical child-to-parent binding				Initiator-to-target binding		
tlm-init	1				1	1	N:1
simple-init	1				1	1	N:1
multi-init			1		1:M	1:M	N:M
	Reverse binding operators				Hierarchical parent-to-child binding		
tlm-targ	1*	1*			1	1	
simple-targ	1*	1*					
multi-targ							1

Key:

tlm-init	tlm_initiator_socket
simple-init	simple_initiator_socket or passthrough_initiator_socket
multi-init	multi_passthrough_initiator_socket
tlm-targ	tlm_target_socket
simple-targ	simple_target_socket or simple_target_socket_tagged or passthrough_target_socket or passthrough_target_socket_tagged
multi-targ	multi_passthrough_target_socket
1*	The binding is from inititiator to target, despite the method call being in the direction target.bind(initiator).

16.2.2 Simple sockets

16.2.2.1 Introduction

The *simple sockets* are so-called because they are intended to be simple to use. They are derived from the interoperability layer sockets **tlm_initiator_socket** and **tlm_target_socket**, so can be bound directly to sockets of those types.

Instead of having to bind a socket to an object that implements the corresponding interface, each simple socket provides member functions for registering callback methods. Those callbacks are in turn called whenever an incoming interface method call arrives. Callback member functions may be registered for each of the interfaces supported by the socket.

The user of a simple socket may register a callback for every interface method but is not obliged to do so. In particular, for the simple target socket, the user need only register one of **b_transport** and **nb_transport_fw**, in which case incoming calls to the unregistered member function will be converted automatically to calls to the registered member function. This conversion process is non-trivial, and is dependent on the rules of the base protocol being respected by the initiator and target. Hence, although the class template **simple_target_socket** takes the protocol type as a template parameter, there exists a dependency between this class and the base protocol that is only exposed if the application makes use of the

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

conversion between blocking and non-blocking transport calls. Specifically, class **simple_target_socket** uses the values of type **tlm_phase_enum**. An application requiring such a conversion for a protocol other than the base protocol would be obliged to create a new convenience socket, possibly derived from **simple_target_socket**. The **passthrough_target_socket** is a variant of the **simple_target_socket** that does not support conversion between blocking and non-blocking calls.

16.2.2.2 Class definition

```
namespace tlm_utils {

template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types
>
class simple_initiator_socket :  public tlm::tlm_initiator_socket<BUSWIDTH, TYPES>
{
public:
    typedef typename TYPES::tlm_payload_type    transaction_type;
    typedef typename TYPES::tlm_phase_type      phase_type;
    typedef tlm::tlm_sync_enum                  sync_enum_type;

    simple_initiator_socket();
    explicit simple_initiator_socket( const char* n );

    void register_nb_transport_bw(
        MODULE* mod,
        sync_enum_type (MODULE::*cb)(transaction_type&, phase_type&, sc_core::sc_time&));

    void register_invalidate_direct_mem_ptr(
        MODULE* mod,
        void (MODULE::*cb)(sc_dt::uint64, sc_dt::uint64));
};

template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types
>
class simple_target_socket : public tlm::tlm_target_socket<BUSWIDTH, TYPES>
{
public:
    typedef typename TYPES::tlm_payload_type    transaction_type;
    typedef typename TYPES::tlm_phase_type      phase_type;
    typedef tlm::tlm_sync_enum                  sync_enum_type;

    simple_target_socket();
    explicit simple_target_socket( const char* n );

    tlm::tlm_bw_transport_if<TYPES> * operator ->();

    void register_nb_transport_fw(
        MODULE* mod,
        sync_enum_type (MODULE::*cb)(transaction_type&, phase_type&, sc_core::sc_time&));
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    void register_b_transport(
        MODULE* mod,
        void (MODULE::*cb)(transaction_type&, sc_core::sc_time&));

    void register_transport_dbg(
        MODULE* mod,
        unsigned int (MODULE::*cb)(transaction_type&));

    void register_get_direct_mem_ptr(
        MODULE* mod,
        bool (MODULE::*cb)(transaction_type&, tlm::tlm_dmi&));
};

template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types
>
class passthrough_target_socket :   public tlm::tlm_target_socket<BUSWIDTH, TYPES>
{
public:
    typedef typename TYPES::tlm_payload_type    transaction_type;
    typedef typename TYPES::tlm_phase_type      phase_type;
    typedef tlm::tlm_sync_enum                  sync_enum_type;

    passthrough_target_socket();
        explicit passthrough_target_socket( const char* n );

    void register_nb_transport_fw(
        MODULE* mod,
        sync_enum_type (MODULE::*cb)(transaction_type&, phase_type&, sc_core::sc_time&));

    void register_b_transport(
        MODULE* mod,
        void (MODULE::*cb)(transaction_type&,  sc_core::sc_time&));

    void register_transport_dbg(
        MODULE* mod,
        unsigned int (MODULE::*cb)(transaction_type&));

    void register_get_direct_mem_ptr(
        MODULE* mod,
        bool (MODULE::*cb)(transaction_type&, tlm::tlm_dmi&));
};

} // namespace tlm_utils
```

16.2.2.3 Header file

The class definitions for the simple sockets shall be in the header files **tlm_utils/simple_initiator_socket.h**, **tlm_utils/simple_target_socket.h**, and **tlm_utils/passthrough_target_socket.h**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

16.2.2.4 Rules

a) Each constructor shall call the constructor of the corresponding base class passing through the **char*** argument, if there is one. In the case of the default constructors, the **char*** argument of the base class constructor shall be set to sc_gen_unique_name ("simple_initiator_socket"), sc_gen_unique_name ("simple_target_socket"), or sc_gen_unique_name ("passthrough_target_socket"), respectively.

b) A **simple_initiator_socket** can be bound to a **simple_target_socket** or a **passthrough_target_socket** by calling the member function **bind** or **operator()** of either socket, with precisely the same effect. In either case, the forward path lies in the direction from the initiator socket to the target socket.

c) A **simple_initiator_socket** can be bound to a **tlm_target_socket**, and a **tlm_initiator_socket** can be bound to a **simple_target_socket** or to a **passthrough_target_socket**.

d) A **simple_initiator_socket**, **simple_target_socket** or **passthrough_target_socket** can only implement incoming interface method calls by registering callbacks, not by being bound hierarchically to another socket on a child module. On the other hand, a **simple_initiator_socket** of a child module can be bound hierarchically to a **tlm_initiator_socket** of a parent module, and a **tlm_target_socket** of a parent module can be bound hierarchically to a **simple_target_socket** or **passthrough_target_socket** of a child module.

e) A target is not obliged to register a **b_transport** callback with a simple target socket provided it has registered an **nb_transport_fw** callback, in which case an incoming **b_transport** call will automatically cause the target to call the member function registered for **nb_transport_fw**. In this case, the member function registered for **nb_transport_fw** shall implement with the rules of the base protocol (see 16.2.2.5).

f) A target is not obliged to register an **nb_transport_fw** callback with a simple target socket provided it has registered a **b_transport** callback, in which case an incoming **nb_transport_fw** call will automatically cause the target to call the member function registered for **b_transport** and subsequently to call **nb_transport_bw** on the backward path.

g) If a target does not register either a **b_transport** or an **nb_transport_fw** callback with a simple target socket, this will result in a run-time error if and only if the corresponding member function is called.

h) A target should register **b_transport** and **nb_transport_fw** callbacks with a passthrough target socket. Not doing so will result in a run-time error if and only if the corresponding member function is called.

i) A target is not obliged to register a **transport_dbg** callback with a simple target socket or a passthrough target socket, in which case an incoming **transport_dbg** call shall return with a value of 0.

j) A target is not obliged to register a **get_direct_mem_ptr** callback with a simple target socket or a passthrough target socket, in which case an incoming **get_direct_mem_ptr** call shall return with a value of false.

k) An initiator should register an **nb_transport_bw** callback with a simple initiator socket. Not doing so will result in a run-time error if and only if the member function **nb_transport_bw** is called.

l) An initiator is not obliged to register an **invalidate_direct_mem_ptr** callback with a simple initiator socket, in which case an incoming **invalidate_direct_mem_ptr** call shall be ignored.

16.2.2.5 Simple target socket b/nb conversion

a) In the case that a member function **b_transport** or **nb_transport_fw** is called through a socket of class **simple_target_socket** but no corresponding callback is registered, the simple target socket will act as an adapter between the two interfaces. In this case, and in no other case, the implementation of **simple_target_socket** shall have an explicit dependency on the values of type **tlm_phase_enum**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

b) When the simple target socket acts as an adapter, it shall honor the rules of the base protocol both from the point of view of the initiator and from the point of view of the implementation of the member function **b_transport** or **nb_transport_fw** in the target (see 15.2).

c) The socket shall pass through the given transaction object without modification and shall not construct a new transaction object.

d) In the case that only the **nb_transport_fw** callback has been registered by the target, the initiator is not permitted to call **nb_transport_fw** while there is an earlier **b_transport** call from the initiator still in progress. This is a limitation of the current implementation of the simple target socket.

e) Figure 34 shows the case where an initiator calls **nb_transport_fw**, but the target only registers a **b_transport** callback with the simple target socket. The initiator sends BEGIN_REQ, to which the socket returns TLM_ACCEPTED. The socket then calls **b_transport**, and on return sends BEGIN_RESP back to the initiator, to which the initiator returns TLM_COMPLETED. Since it is not permissible in SystemC to call a blocking member function directly from a non-blocking member function, the socket is obliged to call **b_transport** from a separate internal thread process, not directly from **nb_transport_fw**.

f) Figure 34 shows just one possible scenario. On the final transition, the initiator could have returned TLM_ACCEPTED, in which case the socket would expect to receive a subsequent END_RESP from the initiator. Also, the target could have called **wait** from within **b_transport**.

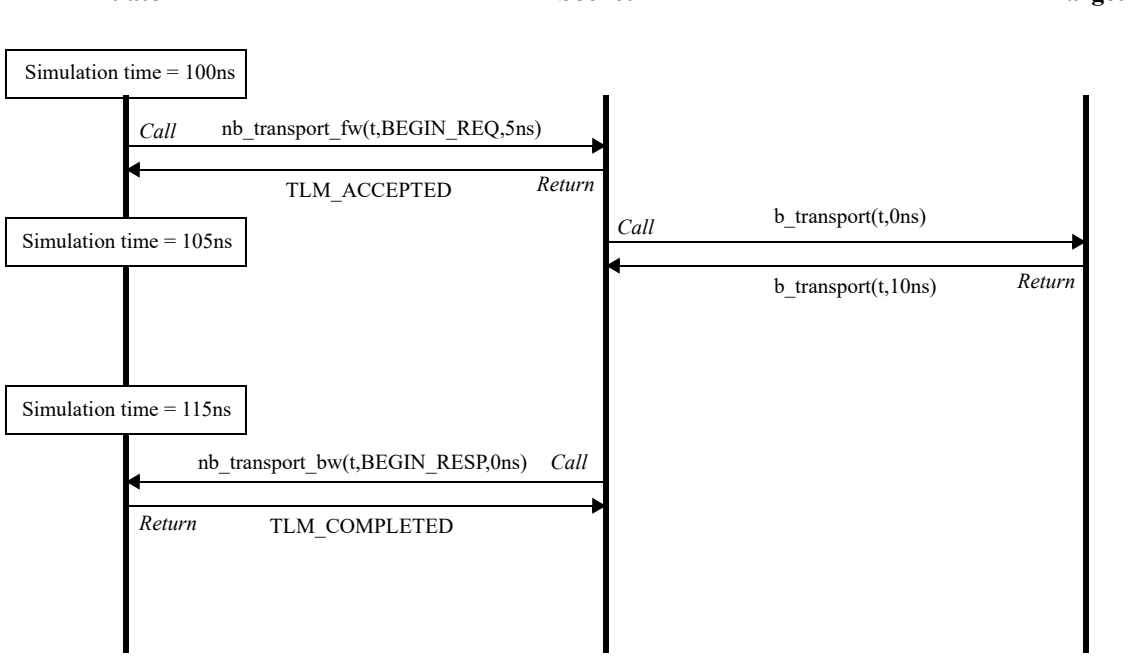

Figure 34—Simple target socket nb/b adapter

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

g) Figure 35 shows the case where an initiator calls **b_transport**, but the target only registers an **nb_transport_fw** callback with the simple target socket. The initiator calls **b_transport**, then the socket and the target handshake using *nb_transport* and obeying the rules of the base protocol. The target may or may not send the END_REQ phase; it may jump straight to the BEGIN_RESP phase. The socket returns TLM_COMPLETED from the call to **nb_transport_bw** for the BEGIN_RESP phase.

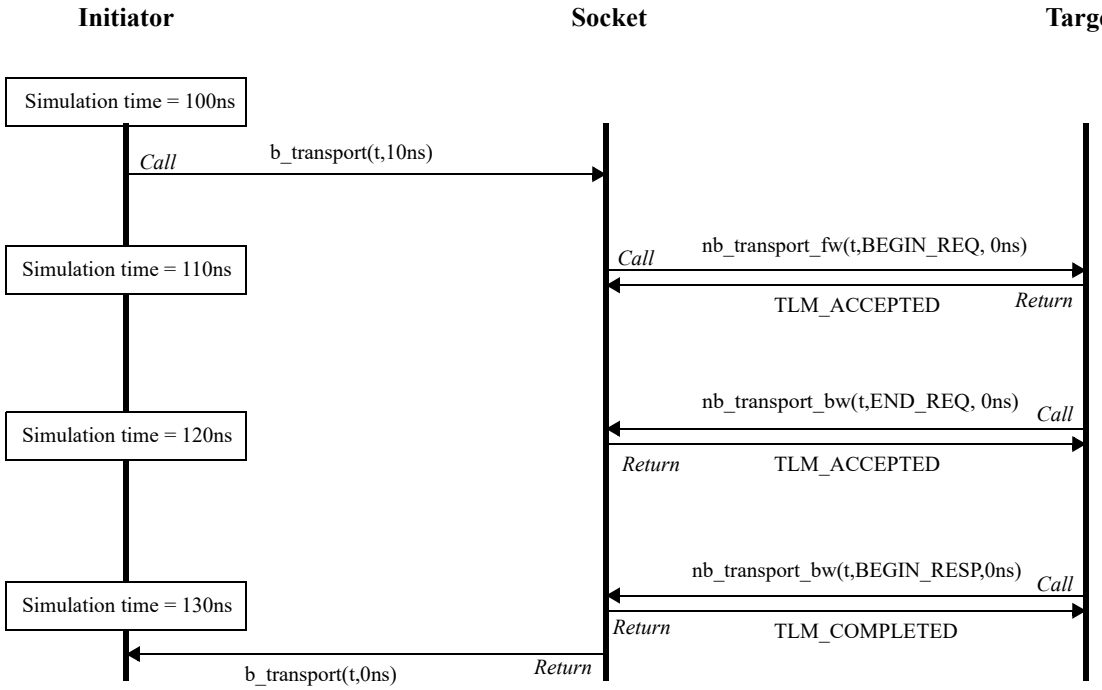

Figure 35—Simple target socket b/nb adapter

Example:

```
#include <tlm>
#include "tlm_utils/simple_initiator_socket.h"          // Header files from utilities
#include "tlm_utils/simple_target_socket.h"

struct Initiator : sc_core::sc_module {
    tlm_utils::simple_initiator_socket<Initiator, 32, tlm::tlm_base_protocol_types> socket;

    SC_CTOR(Initiator)
    : socket("socket") {                                // Construct and name simple socket
        socket.register_nb_transport_bw(this, &Initiator::nb_transport_bw);
                                                        // Register callbacks with simple socket
        socket.register_invalidate_direct_mem_ptr(this, &Initiator::invalidate_direct_mem_ptr);
    }

    virtual tlm::tlm_sync_enum nb_transport_bw(
            tlm::tlm_generic_payload &trans, tlm::tlm_phase &phase, sc_core::sc_time &delay){
        return tlm::TLM_COMPLETED;                      // Placeholder implementation
    }
    virtual void invalidate_direct_mem_ptr(sc_dt::uint64 start_range, sc_dt::uint64 end_range) {
    }                                                   // Placeholder implementation
};
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
struct Target : sc_core::sc_module {                        // Target component
    tlm_utils::simple_target_socket<Target, 32, tlm::tlm_base_protocol_types> socket;

    SC_CTOR(Target)
    : socket("socket") {                                    // Construct and name simple socket
        socket.register_nb_transport_fw(this, &Target::nb_transport_fw);
                                                            // Register callbacks with simple socket
        socket.register_b_transport(this, &Target::b_transport);
        socket.register_get_direct_mem_ptr(this, &Target::get_direct_mem_ptr);
        socket.register_transport_dbg(this, &Target::transport_dbg);
    }

    virtual void b_transport(tlm::tlm_generic_payload &trans, sc_core::sc_time &delay) {
    }                                                       // Placeholder implementation

    virtual tlm::tlm_sync_enum nb_transport_fw(
            tlm::tlm_generic_payload &trans, tlm::tlm_phase &phase, sc_time &delay) {
        return tlm::TLM_ACCEPTED;                           // Placeholder implementation
    }

    virtual bool get_direct_mem_ptr(tlm::tlm_generic_payload &trans, tlm::tlm_dmi &dmi_data) {
        return false;                                       // Placeholder implementation
    }

    virtual unsigned int transport_dbg(tlm::tlm_generic_payload &r) {
        return 0;                                           // Placeholder implementation
    }
};

SC_MODULE(Top)
{
    Initiator *initiator;
    Target    *target;
    SC_CTOR(Top) {
        initiator  = new Initiator("initiator");
        target     = new Target("target");
        initiator->socket.bind( target->socket );           // Bind initiator socket to target socket
    }
};
```

16.2.3 Tagged simple sockets

16.2.3.1 Introduction

The tagged simple sockets are a variation on the simple sockets that tag incoming interface method calls with an integer id that allows the callback to identify through which socket the incoming call arrived. This is useful in the case where the same callback member function is registered with multiple initiator sockets or multiple target sockets. The id is specified when the callback is registered, and gets inserted as an extra first argument to the callback member function.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

16.2.3.2 Header file

The class definitions for the tagged simple sockets shall be in the same header files as the corresponding simple sockets, that is **tlm_utils/simple_initiator_socket.h**, **tlm_utils/simple_target_socket.h**, and **tlm_utils/passthrough_target_socket.h**.

16.2.3.3 Class definition

```
namespace tlm_utils {

template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types
>
class simple_initiator_socket_tagged : public tlm::tlm_initiator_socket<BUSWIDTH, TYPES>
{
public:
    typedef typename TYPES::tlm_payload_type    transaction_type;
    typedef typename TYPES::tlm_phase_type      phase_type;
    typedef tlm::tlm_sync_enum                  sync_enum_type;

    simple_initiator_socket_tagged();
    explicit simple_initiator_socket_tagged( const char* n );

    void register_nb_transport_bw(
        MODULE* mod,
        sync_enum_type (MODULE::*cb)(int, transaction_type&,  phase_type&,   sc_core::sc_time&),
        int id);

    void register_invalidate_direct_mem_ptr(
        MODULE* mod,
        void (MODULE::*cb)(int, sc_dt::uint64, sc_dt::uint64),
        int id);
};

template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types
>
class simple_target_socket_tagged : public tlm::tlm_target_socket<BUSWIDTH, TYPES>
{
public:
    typedef typename TYPES::tlm_payload_type            transaction_type;
    typedef typename TYPES::tlm_phase_type              phase_type;
    typedef tlm::tlm_sync_enum                          sync_enum_type;
    typedef tlm::tlm_fw_transport_if<TYPES>             fw_interface_type;
    typedef tlm::tlm_bw_transport_if<TYPES>             bw_interface_type;
    typedef tlm::tlm_target_socket<BUSWIDTH, TYPES> base_type;

    simple_target_socket_tagged();
    explicit simple_target_socket_tagged( const char* n );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    tlm::tlm_bw_transport_if<TYPES> * operator ->();

    void register_nb_transport_fw(
        MODULE* mod,
        sync_enum_type (MODULE::*cb)(int id, transaction_type&, phase_type&, sc_core::sc_time&),
        int id);

    void register_b_transport(
        MODULE* mod,
        void (MODULE::*cb)(int id, transaction_type&, sc_core::sc_time&),
        int id);

    void register_transport_dbg(
    MODULE* mod,
        unsigned int (MODULE::*cb)(int id, transaction_type&),
        int id);

    void register_get_direct_mem_ptr(
        MODULE* mod,
        bool (MODULE::*cb)(int id, transaction_type&, tlm::tlm_dmi&),
         int id);
};

template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types
>
class passthrough_target_socket_tagged :  public tlm::tlm_target_socket<BUSWIDTH, TYPES>
{
public:
    typedef typename TYPES::tlm_payload_type        transaction_type;
    typedef typename TYPES::tlm_phase_type          phase_type;
    typedef tlm::tlm_sync_enum                      sync_enum_type;

    passthrough_target_socket_tagged();
    explicit passthrough_target_socket_tagged( const char* n );

    void register_nb_transport_fw(
        MODULE* mod,
        sync_enum_type (MODULE::*cb)(int id, transaction_type&, phase_type&, sc_core::sc_time&),
        int id);

    void register_b_transport(
        MODULE* mod,
        void (MODULE::*cb)(int id, transaction_type&, sc_core::sc_time&),
        int id);

    void register_transport_dbg(
        MODULE* mod,
        unsigned int (MODULE::*cb)(int id, transaction_type&),
        int id);

    void register_get_direct_mem_ptr(
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        MODULE* mod,
        bool (MODULE::*cb)(int id, transaction_type&, tlm::tlm_dmi&),
        int id);
};

} // namespace tlm_utils
```

16.2.3.4 Rules

a) Each constructor shall call the constructor of the corresponding base class passing through the **char*** argument, if there is one. In the case of the default constructors, the **char*** argument of the base class constructor shall be set to sc_gen_unique_name ("simple_initiator_socket_tagged"), sc_gen_unique_name ("simple_target_socket_tagged"), or sc_gen_unique_name ("passthrough_target_socket_tagged"), respectively.

b) Apart from the int id tag, the tagged simple sockets behave in the same way as the untagged simple sockets.

c) A given callback member function can be registered with multiple sockets instances using different tags. This is the purpose of the tagged sockets.

d) The int id tag is specified as the final argument of the member functions used to register the callbacks. The socket shall prepend this tag as the first argument of the corresponding callback member function. Note that the order of the id tag argument differs between the registration member function and the callback member function. See the example below.

e) A tagged simple socket is not a multi-socket. A tagged simple socket cannot be bound to multiple sockets on other components (see 16.2.4).

Example:

```
struct my_target : sc_core::sc_module {
    tlm_utils::simple_target_socket_tagged<my_target> socket1;
    tlm_utils::simple_target_socket_tagged<my_target> socket2;

    SC_CTOR(my_target)
    : socket1("socket1"), socket2("socket2") {
        socket1.register_b_transport(this, &my_target::b_transport, 1);      // Registered with id = 1
        socket2.register_b_transport(this, &my_target::b_transport, 2);      // Registered with id = 2
    }
    void b_transport(int id, Transaction &trans, sc_core::sc_time &delay); // May be called with id = 1 or
                                                                           // id = 2
    ...
};
```

16.2.4 Multi-sockets

16.2.4.1 Introduction

The multi-sockets are a variation on the tagged simple sockets that permit a single socket to be bound to multiple sockets on other components. In contrast to the tagged simple sockets, which identify through which socket an incoming call arrives, a multi-socket callback is able to identify from which socket on another component an incoming interface method call arrives, using the multi-port index number as the tag. Unlike the other convenience sockets, the multi-sockets also support hierarchical child-to-parent socket binding on both the initiator and the target side.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

16.2.4.2 Header file

The class definitions for the multi-sockets shall be in the header files **tlm_utils/multi_passthrough_initiator_socket.h** and **tlm_utils/multi_passthrough_target_socket.h**.

16.2.4.3 Class definition

```
namespace tlm_utils {

template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types,
    unsigned int N=0,
    sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
>
class multi_passthrough_initiator_socket : public multi_init_base< BUSWIDTH, TYPES, N, POL>
{
public:
    typedef typename TYPES::tlm_payload_type                transaction_type;
    typedef typename TYPES::tlm_phase_type                  phase_type;
    typedef tlm::tlm_sync_enum                              sync_enum_type;
    typedef multi_init_base<BUSWIDTH, TYPES, N, POL>        base_type;
    typedef typename base_type::base_target_socket_type     base_target_socket_type;

    multi_passthrough_initiator_socket();
    multi_passthrough_initiator_socket(const char* name);
    ~multi_passthrough_initiator_socket();

    void register_nb_transport_bw(
        MODULE* mod,
        sync_enum_type (MODULE::*cb)(int, transaction_type&, phase_type&, sc_core::sc_time&));

    void register_invalidate_direct_mem_ptr(
        MODULE* mod,
        void (MODULE::*cb)(int, sc_dt::uint64, sc_dt::uint64));

    // Override virtual functions of the tlm_initiator_socket:
    virtual tlm::tlm_bw_transport_if<TYPES>& get_base_interface();
    virtual const tlm::tlm_bw_transport_if<TYPES>& get_base_interface() const;

    virtual sc_core::sc_export<tlm::tlm_bw_transport_if<TYPES> >& get_base_export();
    virtual const sc_core::sc_export<tlm::tlm_bw_transport_if<TYPES> >& get_base_export() const;

    virtual void bind(base_target_socket_type& s);
    void operator() (base_target_socket_type& s);

    // SystemC standard callback
     //  multi_passthrough_initiator_socket::before_end_of_elaboration must be called from
    // any derived class
    void before_end_of_elaboration();

    // Bind multi initiator socket to multi initiator socket (hierarchical bind)
    virtual void bind(base_type& s);
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    void operator() (base_type& s);

    tlm::tlm_fw_transport_if<TYPES>* operator[](int i);
    unsigned int size();
};

template <
    typename MODULE,
    unsigned int BUSWIDTH = 32,
    typename TYPES = tlm::tlm_base_protocol_types,
    unsigned int N=0,
    sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND
>
class multi_passthrough_target_socket : public multi_target_base< BUSWIDTH, TYPES, N, POL>
{
public:
    typedef typename TYPES::tlm_payload_type    transaction_type;
    typedef typename TYPES::tlm_phase_type      phase_type;
    typedef tlm::tlm_sync_enum                  sync_enum_type;

    typedef sync_enum_type
                        (MODULE::*nb_cb)(int, transaction_type&, phase_type&, sc_core::sc_time&);
    typedef void        (MODULE::*b_cb)(int, transaction_type&, sc_core::sc_time&);
    typedef unsigned int (MODULE::*dbg_cb)(int, transaction_type& txn);
    typedef bool        (MODULE::*dmi_cb)(int, transaction_type& txn, tlm::tlm_dmi& dmi);

    typedef multi_target_base<BUSWIDTH, TYPES, N, POL>    base_type;
    typedef typename base_type::base_initiator_socket_type  base_initiator_socket_type;
    typedef typename base_type::initiator_socket_type       initiator_socket_type;

    multi_passthrough_target_socket();
    multi_passthrough_target_socket(const char* name);
    ~multi_passthrough_target_socket();

    void register_nb_transport_fw       (MODULE* mod, nb_cb cb);
    void register_b_transport           (MODULE* mod, b_cb cb);
    void register_transport_dbg         (MODULE* mod, dbg_cb cb);
    void register_get_direct_mem_ptr    (MODULE* mod, dmi_cb cb);

    // Override virtual functions of the tlm_target_socket:
    virtual tlm::tlm_fw_transport_if<TYPES>& get_base_interface();
    virtual const tlm::tlm_fw_transport_if<TYPES>& get_base_interface() const;
    virtual sc_core::sc_export<tlm::tlm_fw_transport_if<TYPES> >& get_base_export();
    virtual const sc_core::sc_export<tlm::tlm_fw_transport_if<TYPES> >& get_base_export() const;

    // SystemC standard callback
    //  multi_passthrough_target_socket::end_of_elaboration must be called from any derived class
    void end_of_elaboration();

    virtual void bind(base_type& s);
    void operator() (base_type& s);

    tlm::tlm_bw_transport_if<TYPES>* operator[] (int i);
    unsigned int size();
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
};

} // namespace tlm_utils
```

16.2.4.4 Rules

a) The base classes *multi_init_base* and *multi_target_base* are implementation-defined, and should not be used directly by applications.

b) Each constructor shall call the constructor of the corresponding base class passing through the **char*** argument, if there is one. In the case of the default constructors, the **char*** argument of the base class constructor shall be set to sc_gen_unique_name ("multi_passthrough_initiator_socket"), or sc_gen_unique_name ("multi_passthrough_target_socket"), respectively.

c) Class **multi_passthrough_initiator_socket** and class **multi_passthrough_target_socket** each act as multi-sockets; that is, a single initiator socket can be bound to multiple target sockets, and a single target socket can be bound to multiple initiator sockets. The two class templates have template parameters specifying the number of bindings and the port binding policy, which are used within the class implementation to parameterize the associated **sc_port** template instantiation.

d) A single **multi_passthrough_initiator_socket** can be bound to many **tlm_target_sockets** and/or many **simple_target_sockets** and/or many **passthrough_target_sockets** and/or many **multi_passthrough_target_sockets**. Many **tlm_initiator_sockets** and/or **simple_initiator_sockets** and/or **multi_passthrough_initiators_sockets** can be bound to a single **multi_passthrough_target_socket**.

e) A **multi_passthrough_initiator_socket** can be bound hierarchically to exactly one other **multi_passthrough_initiator_socket**. A **multi_passthrough_target_socket** can be bound hierarchically to exactly one other **multi_passthrough_target_socket**. Other than these two specific cases, a multi-socket cannot be bound hierarchically to another socket. The multiple binding capabilities of multi-sockets do not apply to hierarchical binding but only apply when binding one or more initiator sockets to one or more target sockets.

f) The implementation of each **operator()** shall achieve its effect by calling the corresponding virtual member function **bind**.

g) The binding operators can only be used in the direction initiator-socket-to-target-socket. In other words, unlike classes **tlm_target_socket** and **simple_target_socket**, class **multi_passthrough_target_socket** does not have operators to bind a target socket to an initiator socket.

h) In the case of hierarchical binding (Figure 36), an initiator multi-socket of a child module shall be bound to an initiator multi-socket of a parent module, and a target multi-socket of a parent module bound to a target multi-socket of a child module. This is consistent with the initiator-to-target binding direction rule given above.

i) If an object of class **multi_passthrough_initiator_socket** or **multi_passthrough_target_socket** is bound multiple times, then the **operator[]** can be used to address the corresponding object to which the socket is bound. The index value is determined by the order in which the member function **bind** or **operator()** were called to bind the sockets. This same index value is used to determine the id tag passed to a callback.

j) For example, consider a **multi_passthrough_initiator_socket** bound to two separate targets. The calls **socket[0]->nb_transport_fw(...)** and **socket[1]->nb_transport_fw()** would address the two targets, and incoming member function **nb_transport_bw()** calls from those two targets would carry the tags 0 and 1, respectively.

k) Member function **size** shall return the number of socket instances to which the current multi-socket has been bound. As for SystemC multi-ports, if **size** is called during elaboration and before the callback **end_of_elaboration**, the value returned is implementation-defined because the time at which port binding is completed is implementation-defined.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

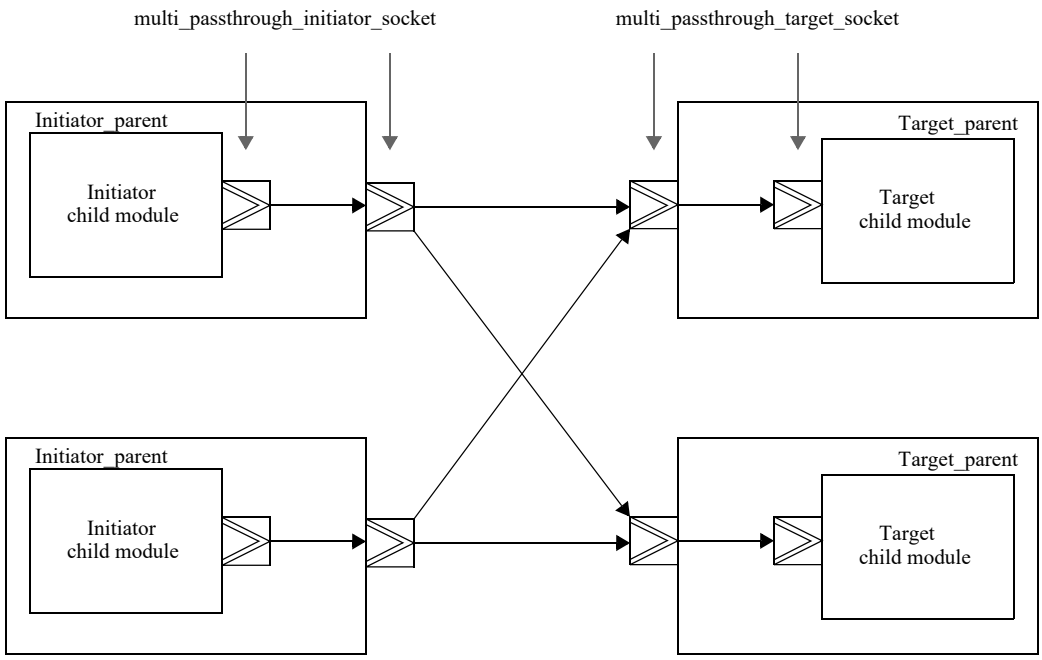

Figure 36—Hierarchical binding of multi-sockets

l) In the absence of hierarchical binding to a multi-socket on a child module, a target should register **b_transport** and **nb_transport_fw** callbacks with a target multi-socket. Not doing so will result in a run-time error if and only if the corresponding member function is called.

m) In the absence of hierarchical binding to a multi-socket on a child module, a target is not obliged to register a **transport_dbg** callback with a target multi-socket, in which case an incoming **transport_dbg** call shall return with a value of 0.

n) In the absence of hierarchical binding to a multi-socket on a child module, a target is not obliged to register a **get_direct_mem_ptr** callback with a target multi-socket, in which case an incoming **get_direct_mem_ptr** call shall return with a value of false.

o) In the absence of hierarchical binding to a multi-socket on a child module, an initiator should register an **nb_transport_bw** callback with an initiator multi-socket. Not doing so will result in a run-time error if and only if the member function **nb_transport_bw** is called.

p) In the absence of hierarchical binding to a multi-socket on a child module, an initiator is not obliged to register an **invalidate_direct_mem_ptr** callback with an initiator multi-socket, in which case an incoming **invalidate_direct_mem_ptr** call shall be ignored.

Example:

```
// Initiator component with a multi-socket
struct Initiator : sc_core::sc_module {
    tlm_utils::multi_passthrough_initiator_socket<Initiator> socket;

    SC_CTOR(Initiator) : socket("socket") {
        // Register callback member functions with socket
        socket.register_nb_transport_bw(this, &Initiator::nb_transport_bw);
        socket.register_invalidate_direct_mem_ptr(this, &Initiator::invalidate_direct_mem_ptr);
    }
    ...
};
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
struct Initiator_parent : sc_core::sc_module {
    tlm_utils::multi_passthrough_initiator_socket<Initiator_parent> socket;
    Initiator *initiator;

    SC_CTOR(Initiator_parent) : socket("socket") {
        initiator = new Initiator("initiator");
        // Hierarchical binding of initiator socket on child to initiator socket on parent
        initiator->socket.bind(socket);
    }
    ...
};

struct Target : sc_core::sc_module {
    tlm_utils::multi_passthrough_target_socket<Target> socket;

    SC_CTOR(Target) : socket("socket") {
        // Register callback member functions with socket
        socket.register_nb_transport_fw(this, &Target::nb_transport_fw);
        socket.register_b_transport(this, &Target::b_transport);
        socket.register_get_direct_mem_ptr(this, &Target::get_direct_mem_ptr);
        socket.register_transport_dbg(this, &Target::transport_dbg);
    }
    ...
};

// Target component with a multi-socket
struct Target_parent : sc_core::sc_module {
    tlm_utils::multi_passthrough_target_socket<Target_parent> socket;
    Target *target;

    SC_CTOR(Target_parent) : socket("socket") {
        target = new Target("target");
        // Hierarchical binding of target socket on parent to target socket on child
        socket.bind(target->socket);
    }
};

SC_MODULE(Top) {
    Initiator_parent *initiator1;
    Initiator_parent *initiator2;
    Target_parent *target1;
    Target_parent *target2;

    SC_CTOR(Top) {
        // Instantiate two initiator and two target components
        initiator1 = new Initiator_parent("initiator1");
        initiator2 = new Initiator_parent("initiator2");
        target1 = new Target_parent("target1");
        target2 = new Target_parent("target2");

        // Bind two initiator multi-sockets to two target multi-sockets
        initiator1->socket.bind(target1->socket);
        initiator1->socket.bind(target2->socket);
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        initiator2->socket.bind(target1->socket);
        initiator2->socket.bind(target2->socket);
    }
};
```

16.3 Quantum keeper

16.3.1 Introduction

Temporal decoupling permits SystemC processes to run ahead of simulation time for an amount of time known as the time quantum (see Clause 12 and Figure 37).

The utility class **tlm_quantumkeeper** provides a set of member functions for managing and interacting with the time quantum. When using temporal decoupling, use of the quantum keeper is recommended in order to maintain a consistent coding style. However, it is straightforward in principle to implement temporal decoupling directly in SystemC. Whether or not the utility class **tlm_quantumkeeper** is used, all temporally decoupled models should reference the global quantum maintained by the class **tlm_global_quantum**.

Class **tlm_quantumkeeper** is in namespace **tlm_utils**.

For a general description of temporal decoupling, see 10.3.3.

For a description of timing annotation, see 11.2.4.

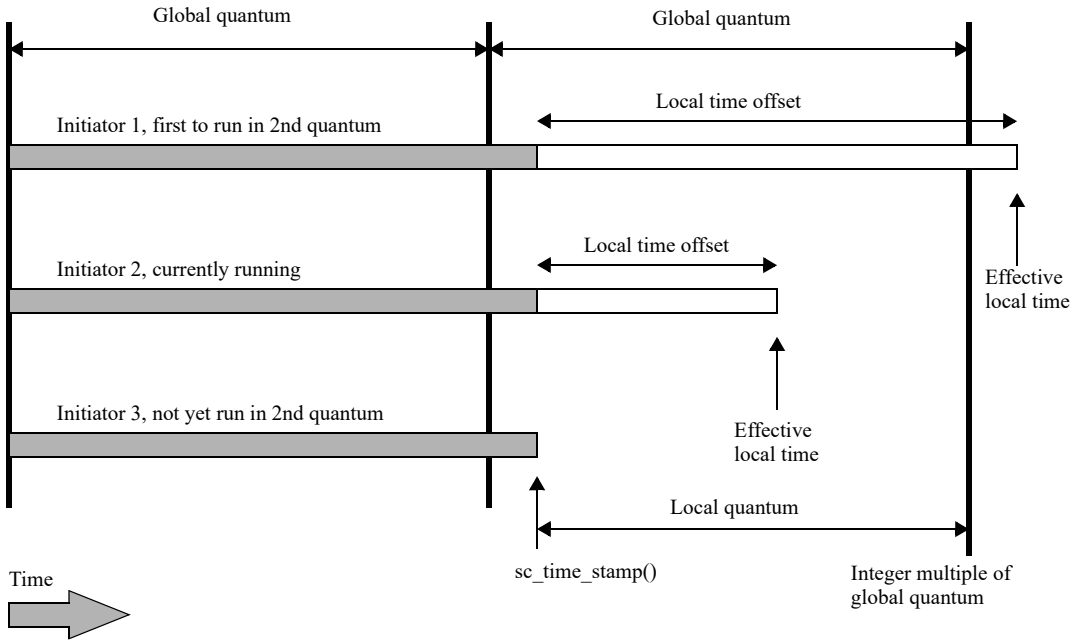

Figure 37—QuantumKeeper terminology

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

16.3.2 Header file

The class definitions for the quantum keeper shall be in the header file **tlm_utils/tlm_quantumkeeper.h**.

16.3.3 Class definition

```
namespace tlm_utils {

class tlm_quantumkeeper
{
public:
    static void set_global_quantum( const sc_core::sc_time& );
    static const sc_core::sc_time& get_global_quantum();

    tlm_quantumkeeper();
    virtual ~tlm_quantumkeeper();

    virtual void inc( const sc_core::sc_time& );
    virtual void set( const sc_core::sc_time& );
    virtual sc_core::sc_time get_current_time() const;
    virtual sc_core::sc_time get_local_time();

    virtual bool need_sync() const;
    virtual void sync();

    void set_and_sync(const sc_core::sc_time& t)
    {
        set(t);
        if (need_sync())
            sync();
    }

    virtual void reset();

protected:
    virtual sc_core::sc_time compute_local_quantum();
};

} // namespace tlm_utils
```

16.3.4 General guidelines for processes using temporal decoupling

a) For maximum simulation speed, all initiators should use temporal decoupling, and the number of other runnable SystemC processes should be zero or minimized.

b) In an ideal scenario, the only runnable SystemC processes will belong to temporally decoupled initiators, and each process will run ahead to the end of its time quantum before yielding to the SystemC kernel.

c) A temporally decoupled initiator is not obliged to use a time quantum if communication with other processes is explicitly synchronized. Where a time quantum is used, it should be chosen to be less than the typical communication interval between initiators; otherwise, important process interactions may be lost, and the model may be broken.

d) *Yield* means call **wait** in the case of a thread process, or return from the function in the case of a method process.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

e) Temporal decoupling runs in the context of the standard SystemC simulation kernel, so events can be scheduled, processes suspended and resumed, and loosely-timed models can be mixed with other coding styles.

f) There is no obligation for every initiator to use temporal decoupling. Processes with and without temporal decoupling can be mixed. However, any process that is not temporally decoupled is likely to become a simulation speed bottleneck.

g) Each temporally decoupled initiator may accumulate any local processing delays and communication delays in a local variable, referred to in this clause as the *local time offset*. It is recommended that the quantum keeper should be used to maintain the local time offset.

h) Calls to the member function **sc_time_stamp** will return the simulation time as it was at or near the start of the current time quantum.

i) The local time offset is unknown to the SystemC scheduler. When using the transport interfaces, the local time offset should be passed as an argument to the member function **b_transport** or *nb_transport* methods.

j) Use of the *nb_transport* method with temporal decoupling and the quantum keeper is not ruled out, but it is not usually advantageous because the speed advantage to be gained from temporal decoupling would be nullified by the high degree of inter-process communication inherent in the approximately-timed coding style.

k) The order in which processes resume within the quantum is under the control of the SystemC scheduler and, by the rules of SystemC, is indeterminate. In the absence of any explicit synchronization mechanism, if a variable is modified by one such process and read by another, the value to be read will be indeterminate. The new value may become available in the current quantum or the next quantum, assuming it only changes relatively infrequently compared to the quantum length, and the application would need to be tolerant of precisely when the new value becomes available. If this is not the case, the application should guard the variable access with an appropriate synchronization mechanism.

l) Any access to a variable or object from a temporally decoupled process will give the value it had at the start of the current time quantum unless it has been modified by the current process or by another temporally decoupled process that has already run in the current quantum. In particular, any **sc_signal** accessed from a temporally decoupled process will have the same value it had at the start of the current time quantum. This is a consequence of the fact that conventional SystemC simulation time (as returned by **sc_time_stamp**) does not advance within the quantum.

16.3.5 tlm_quantumkeeper

a) The constructor shall set the local time offset to SC_ZERO_TIME but shall not call the virtual member function **compute_local_quantum**. Because the constructor does not calculate the local quantum, an application should call the member function **reset** immediately after constructing a quantum keeper object.

b) The implementation of class **tlm_quantumkeeper** shall not create a static object of class **sc_time**, but the constructor may create objects of class **sc_time**. This implies that an application may call function **sc_core::sc_set_time_resolution** before, and only before, constructing the first quantum keeper object.

c) Member function **set_global_quantum** shall set the value of the global quantum to the value passed as an argument, but it shall not modify the local quantum. Member function **get_global_quantum** shall return the current value of the global quantum. After calling **set_global_quantum**, it is recommended to call the member function **reset** to recalculate the local quantum.

d) Member function **get_local_time** shall return the value of the local time offset.

e) Member function **get_current_time** shall return the value of the effective local time, that is, **sc_time_stamp() + local_time_offset**.

f) Member function **inc** shall add the value passed as an argument to the local time offset.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

g) Member function **set** shall set the value of the local time offset to the value passed as an argument.

h) Member function **need_sync** shall return the **value** true if and only if the local time offset is greater than the local quantum.

i) Member function **sync** shall call **wait(local_time_offset)** to suspend the process until simulation time equals the effective local time, and shall then call member function **reset**.

j) Member function **set_and_sync** is a convenience method to call **set**, **need_sync**, and **sync** in sequence. It should not be overridden.

k) Member function **reset** shall call the member function **compute_local_quantum** and shall set the local time offset back to SC_ZERO_TIME.

l) Member function **compute_local_quantum** of class **tlm_quantumkeeper** shall call the member function **compute_local_quantum** of class **tlm_global_quantum**, but it may be overridden in order to calculate a smaller value for the local quantum.

m) The class **tlm_quantumkeeper** should be considered the default implementation for the quantum keeper. Applications may derive their own quantum keeper from class **tlm_quantumkeeper** and override the member function **compute_local_quantum**, but this is unusual.

n) When the local time offset is greater than or equal to the local quantum, the process should yield to the kernel. It is strongly recommended that the process does this by calling the member function **sync**.

o) There is no mechanism to enforce synchronization at the end of the time quantum. It is the responsibility of the initiator to check **need_sync** and call **sync** as needed.

p) Member function **b_transport** may itself yield such that the value of **sc_time_stamp** can be different before and after the call. The value of the local time offset and any timing annotations are always expressed relative to the current value of **sc_time_stamp**. On return from **b_transport** or **nb_transport_fw**, it is the responsibility of the initiator to set the local time offset of the quantum keeper by calling the member function **set**, and then to check for synchronization by calling the member function **need_sync**.

q) If an initiator needs to synchronize before the end of the time quantum; that is, if an initiator needs to suspend execution so that simulation time can catch up with the local time, it may do so by calling the member function **sync** or by explicitly waiting on an event. This gives any other processes the chance to execute, and it is known as synchronization-on-demand.

r) Making frequent calls to **sync** will reduce the effectiveness of temporal decoupling.

Example:

```
struct Initiator : sc_core::sc_module {                    // Loosely-timed initiator
    tlm_utils::simple_initiator_socket<Initiator> init_socket;
    tlm_utils::tlm_quantumkeeper m_qk;                     // The quantum keeper

    SC_CTOR(Initiator) : init_socket("init_socket") {
        SC_THREAD(thread);                                 // The initiator process
        ...
        m_qk.set_global_quantum(sc_core::sc_time(1.0, sc_core::SC_US)); // Replace the global quantum
        m_qk.reset();                                      // Re-calculate the local quantum
    }

    void thread() {
        tlm::tlm_generic_payload trans;
        sc_core::sc_time delay;
        trans.set_command(tlm::TLM_WRITE_COMMAND);
        trans.set_data_length(4);
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
        for (int i = 0; i < RUN_LENGTH; i += 4) {
        int word = i;
        trans.set_address(i);
        trans.set_data_ptr((unsigned char *)(&word));

            delay = m_qk.get_local_time();              // Annotate b_transport with local time
            init_socket->b_transport(trans, delay);
            qk.set(delay);                              // Update qk with time consumed by target

            m_qk.inc(sc_core::sc_time(100.0, sc_core::SC_NS)); // Further time consumed by initiator
            if (m_qk.need_sync())
                m_qk.sync();                            // Check local time against quantum
        }
    }
    ...
};
```

16.4 Payload event queue

16.4.1 Introduction

A payload event queue (PEQ) is a class that maintains a queue of SystemC event notifications, where each notification carries an associated transaction object. Each transaction is written into the PEQ annotated with a delay, and each transaction emerges from the back of the PEQ at a time calculated from the current simulation time plus the annotated delay.

Two payload event queues are provided as utilities. As well as being useful in their own right, the PEQ is of conceptual relevance in understanding the semantics of timing annotation with the approximately-timed coding style. However, it is possible to implement approximately-timed models without using the specific payload event queues given here. In an approximately-timed model, it is often appropriate for the recipient of a transaction passed using *nb_transport* to put the transaction into a PEQ with the annotated delay. The PEQ will schedule the timing point associated with the *nb_transport* call to occur at the correct simulation time.

Transactions are inserted into a PEQ by calling the member function **notify** of the PEQ, passing a delay as an argument. There is also a member function **notify** that schedules an immediate notification. The delay is added to the current simulation time (**sc_time_stamp**) to calculate the time at which the transaction will emerge from the back end of the PEQ. The scheduling of the events is managed internally using a SystemC timed event notification, exploiting the property of class **sc_event** that if the member function **notify** is called while there is a notification pending, the notification with the earliest simulation time will remain while the other notification gets cancelled.

Transactions emerge in different ways from the two PEQ variants. In the case of **peq_with_get**, the member function **get_event** returns an event that is notified whenever a transaction is ready to be retrieved. Member function **get_next_transaction** should be called repeatedly to retrieve any available transactions one at a time.

In the case of **peq_with_cb_and_phase**, a callback member function is registered as a constructor argument, and that member function is called as each transaction emerges. This particular PEQ carries both a transaction object and a phase object with each notification, and both are passed as arguments to the callback member function.

For an example, see 15.1.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

16.4.2 Header file

The class definitions for the two payload event queues shall be in the header files **tlm_utils/peq_with_get.h** and **tlm_utils/peq_with_cb_and_phase.h**.

16.4.3 Class definition

```
namespace tlm_utils {

template <class PAYLOAD>
class peq_with_get : public sc_core::sc_object
{
public:
    typedef PAYLOAD transaction_type;

    peq_with_get(const char* name);

    void notify( transaction_type& trans, const sc_core::sc_time& t );
    void notify( transaction_type& trans );

    transaction_type* get_next_transaction();
    sc_core::sc_event& get_event();
    void cancel_all();
};

template<typename OWNER, typename TYPES=tlm::tlm_base_protocol_types>
class peq_with_cb_and_phase : public sc_core::sc_object
{
public:
    typedef typename TYPES::tlm_payload_type     tlm_payload_type;
    typedef typename TYPES::tlm_phase_type       tlm_phase_type;
    typedef void (OWNER::*cb)(tlm_payload_type&, const tlm_phase_type&);

    peq_with_cb_and_phase(OWNER* , cb );
    peq_with_cb_and_phase(const char* , OWNER* , cb);
    ~peq_with_cb_and_phase();

    void notify ( tlm_payload_type& , const tlm_phase_type& , const sc_core::sc_time& );
    void notify ( tlm_payload_type& , const tlm_phase_type& );
    void cancel_all();
};

} // namespace tlm_utils
```

16.4.4 Rules

a) Member function **notify** shall insert a transaction into the PEQ. The transaction shall emerge from the PEQ at time **t1 + t2**, where **t1** is the value returned from **sc_time_stamp()** at the time **notify** is called, and **t2** is the value of the **sc_time** argument to **notify**. In the case of immediate notification, the transaction shall emerge in the current evaluation phase of the SystemC scheduler.

b) Transactions may be queued in any order and emerge in the order given by the previous rule. Transactions do not necessarily emerge in the order in which they were inserted.

c) There is no limit to the number of transactions that may be in the PEQ at any given time.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

d) If several transactions are queued to emerge at the same time, they shall all emerge in the same evaluation phase (that is, the same delta cycle) in the order in which they were inserted.

e) Member function **cancel_all** shall immediately remove all queued transactions from the PEQ, effectively restoring the PEQ to the state it had immediately after construction. This is the only way to remove transactions from a PEQ.

f) The **PAYLOAD** template argument to class **peq_with_get** shall be the name of the transaction type used by the PEQ.

g) Member function **get_event** shall return a reference to an event that is notified when the next transaction is ready to emerge from the PEQ. If more than one transaction is ready to emerge in the same evaluation phase (that is, in the same delta cycle), the event is notified once only.

h) Member function **get_next_transaction** shall return a pointer to a transaction object that is ready to emerge from the PEQ, and shall remove the transaction object from the PEQ. If a transaction is not retrieved from the PEQ in the evaluation phase in which the corresponding event notification occurs, it will still be available for retrieval on a subsequent call to **get_next_transaction** at the current time or at a later time.

i) If there are no more transactions to be retrieved in the current evaluation phase, **get_next_transaction** shall return a null pointer.

j) The **TYPES** template argument to class **peq_with_cb_and_phase** shall be the name of the protocol traits class containing the transaction and phase types used by the PEQ.

k) The **OWNER** template argument to class **peq_with_cb_and_phase** shall be the type of the class of which the PEQ callback member function is a member. This will usually be the parent module of the PEQ instance.

l) The **OWNER*** argument to the constructor **peq_with_cb_and_phase** shall be a pointer to the object of which the PEQ callback member function is a member. This will usually be the parent module of the PEQ instance.

m) The **cb** argument to the constructor **peq_with_cb_and_phase** shall be the name of the PEQ callback member function, which shall be a member function.

n) The implementation of class **peq_with_cb_and_phase** shall call the PEQ callback member function whenever a transaction object is ready to emerge from the PEQ. The first argument of the callback is a reference to the transaction object and the second argument a reference to the phase object, as passed to the corresponding member function **notify**.

o) The implementation shall call the PEQ callback member function from a SystemC method process, so the callback member function shall be non-blocking.

p) The implementation shall only call the PEQ callback member function once for each transaction. After calling the PEQ callback member function, the implementation shall remove the transaction object from the PEQ. The PEQ callback member function may be called multiple times in the same evaluation phase.

16.5 Instance-specific extensions

16.5.1 Introduction

The generic payload contains an array of pointers to extension objects such that each transaction object can contain at most one instance of each extension type. This mechanism alone does not directly permit multiple instances of the same extension to be added to a given transaction object. This clause describes a set of utilities that provide instance-specific extensions, that is, multiple extensions of the same type added to a single transaction object.

An instance-specific extension type is created using a class template **instance_specific_extension**, used in a similar manner to class **tlm_extension**. Unlike **tlm_extension**, applications are not required or permitted to

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

implement virtual member functions **clone** and **copy_from**. The access methods are restricted to **set_extension**, **get_extension**, **clear_extension**, and **resize_extensions**. Automatic deletion of instance-specific extensions is not supported, so a component calling **set_extension** should also call **clear_extension**. As for class **tlm_extension**, member function **resize_extensions** need only be called if a transaction object is constructed during static initialization.

An instance-specific extension is accessed using an object of type **instance_specific_extension_accessor**. This class provides **operator()** that returns a proxy object through which the access methods can be called. Each object of type **instance_specific_extension_accessor** gives access to a distinct set of extension objects, even when used with the same transaction object.

In the class definition in 16.5.3, terms in italics are implementation-defined names that should not be used directly by an application.

16.5.2 Header file

The class definitions for the instance-specific extensions shall be in the header file **tlm_utils/instance_specific_extensions.h**.

16.5.3 Class definition

```
namespace tlm_utils {

template <typename T>
class instance_specific_extension : public implementation-defined {
public:
    virtual ~instance_specific_extension();
};

template<typename U>
class proxy {
public:
    template <typename T> T* set_extension( T* );
    template <typename T> void get_extension( T*& ) const;
    template <typename T> void clear_extension( const T* );
    void resize_extensions();
};

class instance_specific_extension_accessor {
public:
    instance_specific_extension_accessor();

    template<typename T>  proxy< implementation-defined >&  operator() ( T& );
};

} // namespace tlm_utils
```

Example:

```
struct my_extn : tlm_utils::instance_specific_extension<my_extn> {
    int num;                                                    // User-defined extension attribute
};

struct Interconnect : sc_core::sc_module {
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    tlm_utils::simple_target_socket<Interconnect> targ_socket;
    tlm_utils::simple_initiator_socket<Interconnect> init_socket;
    ...
    tlm_utils::instance_specific_extension_accessor accessor;
    static int count;

    virtual tlm::tlm_sync_enum nb_transport_fw(
            tlm::tlm_generic_payload &trans, tlm::tlm_phase &phase, sc_core::sc_time &delay) {
        my_extn *extn;
        accessor(trans).get_extension(extn);                    // Get existing extension
        if (extn) {
            accessor(trans).clear_extension(extn);              // Delete existing extension
        } else {
            extn = new my_extn;
            extn->num = count++;
            accessor(trans).set_extension(extn);                // Add new extension
        }
        return init_socket->nb_transport_fw(trans, phase, delay);
    }
    ...
};

SC_MODULE(Top) {
...
    SC_CTOR(Top) {
        // Transaction object passes through two instances of Interconnect
        interconnect1 = new Interconnect("interconnect1");
        interconnect2 = new Interconnect("interconnect2");
        interconnect1->init_socket.bind(interconnect2->targ_socket);
        ...
    }
};
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

17. TLM-1 message passing interface and analysis ports

17.1 Overview

The TLM-1 message passing interface comprises the blocking and non-blocking put, get, peek, transport, write, and analysis interfaces, the **tlm_fifo** channel, the analysis port, and the analysis fifo. The TLM-1 transport interface is distinct from the TLM-2.0 transport interface.

17.2 Put, get, peek, and transport interfaces

17.2.1 Description

The TLM-1 message passing interfaces are fundamentally different from the TLM-2 core interfaces in the following sense: Whereas the TLM-2 core interfaces pass a transaction object by reference and the lifetime of that transaction object may span multiple interface method calls, the TLM-1 interfaces implement message-passing semantics. With TLM-1 message passing semantics, the intent is that there should be no shared memory between caller and callee, and that the message-passing abstraction implemented by TLM-1 interface method calls should hide any internal state changes in one SystemC module from any other module. The TLM-1 blocking interfaces realize synchronous message passing, and the TLM-1 non-blocking interfaces realize asynchronous message passing.

TLM-1 message-passing, which equates to transaction-passing, is unidirectional. The TLM-1 bidirectional transport interface can be considered to be composed of two unidirectional message channels that pass separate messages in opposing directions. At an abstract level, the intent is that the recipient of a transaction (whether that transaction is passed using **put** or **get**) should receive the exact same value as was sent. Message passing systems would typically implement this requirement using strict pass-by-value semantics. However, TLM-1 uses so-called effective pass-by-value semantics, whereby although a transaction may in some cases be passed by reference, neither the caller nor the callee is permitted to modify the transaction object once it has been assigned by the sender.

TLM-1 imposes a further constraint to help ensure the integrity of the message-passing semantics. The data type of the transaction object should support deep copy semantics such that if a copy is taken using C++ initialization (copy constructor) or assignment, then a subsequent modification to the copy should not modify the original transaction object. In other words, if the transaction object contains any pointers or references to shared memory outside of itself, this standard does not specify the ownership, lifetime, access, or update rules for such shared memory locations. Hence, the responsibility for the use of any such shared memory lies entirely with the application and should be carefully documented.

In what follows, the term *sender* means the caller in the case of member function **put**, or the callee in the case of member function **get** or **peek**, and the term *recipient* means the callee in the case of **put**, or the caller in the case of **get** of **peek**. In the case of member function **transport**, the caller is the sender of the request and the recipient of the response, whereas the callee is the recipient of the request and the sender of the response.

The transaction object is the object passed as an argument to the member function **put**, **nb_put**, **nb_get**, **nb_peek**, or **transport**, or as the return value from the member function **get**, **peek**, or **transport**.

17.2.2 Class definition

```
namespace tlm {

template<class  T>
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
class tlm_tag {};

// Uni-directional blocking interfaces
template < typename T >
class tlm_blocking_put_if : public virtual sc_core::sc_interface
{
public:
    virtual void put( const T &t ) = 0;
};

template < typename T >
class tlm_blocking_get_if : public virtual sc_core::sc_interface
{
public:
    virtual T get( tlm_tag<T> *t = 0 ) = 0;
    virtual void get( T &t ) { t = get(); }
};

template < typename T >
class tlm_blocking_peek_if : public virtual sc_core::sc_interface
{
public:
    virtual T peek( tlm_tag<T> *t = 0 ) const = 0;
    virtual void peek( T &t ) const { t = peek(); }
};

// Uni-directional non blocking interfaces
template < typename T >
class tlm_nonblocking_put_if : public virtual sc_core::sc_interface
{
public:
    virtual bool nb_put( const T &t ) = 0;
    virtual bool nb_can_put( tlm_tag<T> *t = 0 ) const = 0;
    virtual const sc_core::sc_event &ok_to_put( tlm_tag<T> *t = 0 ) const = 0;
};

template < typename T >
class tlm_nonblocking_get_if : public virtual sc_core::sc_interface
{
public:
    virtual bool nb_get( T &t ) = 0;
    virtual bool nb_can_get( tlm_tag<T> *t = 0 ) const = 0;
    virtual const sc_core::sc_event &ok_to_get( tlm_tag<T> *t = 0 ) const = 0;
};

template < typename T >
class tlm_nonblocking_peek_if : public virtual sc_core::sc_interface
{
public:
    virtual bool nb_peek( T &t ) const = 0;
    virtual bool nb_can_peek( tlm_tag<T> *t = 0 ) const = 0;
    virtual const sc_core::sc_event &ok_to_peek( tlm_tag<T> *t = 0 ) const = 0;
};
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
// Uni-directional combined blocking and non blocking interfaces
template < typename T >
class tlm_put_if :
    public virtual tlm_blocking_put_if< T > ,
    public virtual tlm_nonblocking_put_if< T > {};

template < typename T >
class tlm_get_if :
    public virtual tlm_blocking_get_if< T > ,
    public virtual tlm_nonblocking_get_if< T > {};

template < typename T >
class tlm_peek_if :
    public virtual tlm_blocking_peek_if< T > ,
    public virtual tlm_nonblocking_peek_if< T > {};

// Uni-directional combined get-peek interfaces
template < typename T >
class tlm_blocking_get_peek_if :
    public virtual tlm_blocking_get_if<T> ,
    public virtual tlm_blocking_peek_if<T> {};

template < typename T >
class tlm_nonblocking_get_peek_if :
    public virtual tlm_nonblocking_get_if<T> ,
    public virtual tlm_nonblocking_peek_if<T> {};

template < typename T >
class tlm_get_peek_if :
    public virtual tlm_get_if<T> ,
    public virtual tlm_peek_if<T> ,
    public virtual tlm_blocking_get_peek_if<T> ,
    public virtual tlm_nonblocking_get_peek_if<T>  {};

// Bidirectional blocking transport interface
template < typename REQ , typename RSP >
class tlm_transport_if : public virtual sc_core::sc_interface
{
public:
    virtual RSP transport( const REQ& ) = 0;
    virtual void transport( const REQ& req , RSP& rsp ) { rsp = transport( req ); }
};

} // namespace tlm
```

17.2.3 Blocking versus non-blocking interfaces

a) Member functions **put**, **get**, **peek**, and **transport** are blocking interface methods.

b) Member functions **nb_put**, **nb_can_put**, **ok_to_put**, **nb_get**, **nb_can_get**, **ok_to_get**, **nb_peek**, **nb_can_peek**, and **ok_to_peek** are non-blocking interface methods.

c) Blocking interface methods may call **wait**, directly or indirectly.

d) Blocking interface methods shall not be called from a method process.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

e) Non-blocking interface methods shall not call **wait**, directly or indirectly.

f) Non-blocking interface methods may be called from a thread process or from a method process.

17.2.4 Blocking interface methods

a) Consecutive calls to the member function **put** and consecutive calls to the member function **get** (through the same port or export) shall represent distinct transaction instances, regardless of whether or not the same transaction object is passed on each occasion. In other words, on each call to **put**, the caller shall pass the next transaction in sequence, and on each return from **get**, the callee shall return the next transaction in sequence.

b) Member function **put** shall not return until the recipient has accepted the transaction object, that is, until the transaction object has been executed, copied, or passed downstream by the recipient. In other words, on return from the member function **put**, the caller can assume that the callee has completed whatever processing of the transaction object was appropriate at that stage. The transaction may be executed within the body of the member function **put**, or the member function **put** may take a copy of the transaction object for subsequent processing. In either case, the caller may attempt to send the next transaction immediately.

c) Member function **get** shall not return until the next transaction object is ready to be returned. In other words, on return from the member function **get**, the caller can assume that the callee is returning a valid transaction object ready to be processed, and the caller may attempt to get the next transaction immediately.

d) Member function **peek** shall not return until the next transaction object is ready to be returned. However, unlike the member function **get**, the member function **peek** shall not remove the transaction from the callee. In other words, consecutive calls to **peek** (through the same port or export and with no intervening call to **get**) shall return the same transaction object representing the same transaction instance. Similarly, a call to **peek** followed by a single call to **get** shall each return the same transaction object representing the same transaction instance.

e) Member function **transport** shall combine two unidirectional transactions: a request object sent from caller to callee and a response object sent from callee to caller. The implementation of the member function **transport** shall be semantically equivalent to a call to **put** that passes the request object followed by a call to **get** that returns the corresponding response object. The response object returned by the callee shall represent the response of the callee to the given request object. In other words, the entire round-trip transaction shall be executed in a single call to the member function **transport**.

17.2.5 Non-blocking interface methods

a) The non-blocking interface methods each depend on being able to determine whether or not the callee is ready to accept (in the case of **nb_put**) or to return (in the case of **nb_get** and **nb_peek**) the next transaction immediately, that is, as part of the execution of the current non-blocking method call. If the callee is able to respond immediately, then the non-blocking member function (**nb_put**, **nb_can_put**, **nb_get**, **nb_can_get**, **nb_peek**, or **nb_can_peek**) shall return the value **true**. Otherwise, the non-blocking method shall return the value **false**, and shall not accept or return the next transaction.

b) If the interface method **nb_put**, **nb_get**, or **nb_peek** return the value **true**, they shall each behave as would the corresponding blocking member function **put**, **get**, or **peek,** respectively, except that they shall return immediately without calling **wait**.

c) If the interface method **nb_put**, **nb_can_put**, **nb_get**, **nb_can_get**, **nb_peek**, or **nb_can_peek** return the value **false**, they shall each return immediately without accepting or returning a transaction. In other words, they shall not modify the state of the callee with respect to sending or receiving transactions using the particular port or export through which they are called.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

d) The interface methods **nb_put** and **nb_can_put** shall each return the same value (**true** or **false**) if called in place of one another at a given time through a given port or export. The return value shall not depend on the value of the transaction object passed as an argument.

e) Similarly, the interface methods **nb_get**, **nb_can_get**, **nb_peek**, and **nb_can_peek** shall each return the same value (**true** or **false**) if called in place of one another at a given time through a given port or export. In other words, it is not permitted that a callee would return **true** for **nb_get** but **false** for **nb_can_get**, and vice versa, or **true** for **nb_can_get** but **false** for **nb_can_peek**, and vice versa.

f) The interface methods **ok_to_put**, **ok_to_get**, and **ok_to_peek** shall each return an **sc_event** that is notified by the callee whenever the callee becomes ready to accept or to return the next transaction. The intent is that the notification of this event can act as a cue to the caller to attempt to put, get, or peek the next transaction by calling one of the corresponding non-blocking interface methods through the same port or export. However, the caller cannot assume that a non-blocking interface method would necessarily return the value **true** immediately following the notification of one of these events: The caller is still obliged to check the value returned from the non-blocking interface method.

g) There is no obligation that the corresponding blocking and non-blocking interface methods have consistent behavior when called through the same port or export in place of one another, although it is recommended that they do. For example, in circumstances where **put** would return immediately, a call to **nb_put** or **nb_can_put** should normally return **true**, although it is not obliged to do so. Similarly, in circumstances where **put** would call **wait**, a call to **nb_put** or **nb_can_put** should normally return **false**, although again it is not obliged to do so.

17.2.6 Argument passing and transaction lifetime

a) The argument of type **tlm_tag<T>*** may be used by the caller to differentiate between template instances in the case where there exist multiple instantiations of a TLM-1 interface that differ only in their transaction type. The caller is not obliged to provide a value for this argument except as required by the C++ language rules to disambiguate the method call. The body of the interface method shall not use this argument.

b) In the case of the interface methods **put**, **nb_put**, and **transport**, which pass a transaction object as a const reference argument (of type **const T&**, where T is a class template parameter), the caller shall initialize or assign the value of the transaction object passed as the actual argument (the one-and-only argument to **put** and **nb_put** and the first argument to transport) with a value representing the transaction instance being sent before executing the method call.

c) In the case of the interface methods **get**, **peek**, **nb_get**, **nb_peek**, and **transport**, which return a transaction object though a non-const reference argument (of type **T&**, where T is a class template parameter), the callee shall (for **get**, **peek**, **transport**) or may (for **nb_get**, **nb_peek**) assign a value to the formal argument representing the transaction instance being returned.

d) In either case, subsequent to initializing or assigning a value to the actual or formal argument, respectively, neither caller nor callee shall modify the value of this transaction object (directly or indirectly) until after the return from the interface method call, bearing in mind that in some cases the method call may block and that there may exist concurrent SystemC processes (within the caller or the callee) with access to the actual or formal argument that may execute while the interface method itself is suspended.

e) In the case of the interface methods **get**, **peek**, and **transport** that have a non-void return type, the transaction object is returned by value, so the issue of modifying the transaction object during the method call does not arise.

f) The lifetime of a transaction object passed to a TLM-1 interface method is determined by the rules of the C++ language, remembering that a transaction object is either passed as a reference argument to an interface method call or is returned by value from an interface method call.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

g) A transaction object passed as an argument to a TLM-1 interface method shall represent a valid transaction instance from the point when the corresponding actual or formal argument is initialized or assigned a value representing the transaction instance to the point following the return from the interface method call. For example, in the case of **transport**, the request object is valid from the point when it is initialized or assigned a value prior to the call to **transport** to the point following the return from **transport**, and the response object is valid from the point when the formal argument is assigned a value within the implementation of **transport** to the point following the return from **transport**. (This standard does not define any obligations regarding the validity of the transaction object beyond this point. In other words, the transaction object is valid on return from the interface method call, but it is up to each application to determine the fate of the transaction object from that point on.)

h) The recipient of a transaction may take a copy of the transaction object for subsequent processing, in which case the application is responsible for destroying the copy when it is no longer needed.

17.2.7 Constraints on the transaction data type

a) The intent of the recommendation below is to help ensure the integrity of the message-passing semantics between sender and recipient, and to exclude the possibility of communication through shared memory.

b) If a transaction object contains pointers or references, the recipient of the transaction should not modify the contents of the storage accessible through those pointers or references.

c) The data type of the transaction object should have deep copy semantics such that if the recipient takes a copy of the actual or formal argument that represents the transaction (using C++ initialization or assignment), any subsequent modification to the copy should not modify the original.

d) The above constraint could be met in various ways. For example, the transaction object data type could implement copy-on-write semantics such that the original transaction and the copy both have internal pointers to an area of shared memory, but any assignment to either object causes a separate copy to be made.

e) The application shall provide a destructor for any transaction class that has non-trivial destruction semantics (such as a transaction with non-trivial deep copy semantics). The application is responsible for destroying each transaction object when it is no longer required.

f) If the transaction object data type does not adhere to the above constraints, for example, if it creates shallow copies of pointers, then the application is responsible for following an appropriate communication protocol to help ensure message-passing semantics.

17.3 TLM-1 fifo interfaces

17.3.1 Description

Class **tlm_fifo_debug_if** is an interface proper that provides debug access to a **tlm_fifo**. Class **tlm_fifo_put_if** and **tlm_fifo_get_if** are interfaces proper that combine the **tlm_fifo_debug_if** with the **tlm_put_if** and the **tlm_get_peek_if**, respectively. Each of these three interfaces is implemented by the predefined channel **tlm_fifo**.

17.3.2 Class definition

```
namespace tlm {

// Fifo debug interface
template< typename T >
class tlm_fifo_debug_if : public virtual sc_core::sc_interface
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
{
public:
    virtual int used() const = 0;
    virtual int size() const = 0;
    virtual void debug() const = 0;

    virtual bool nb_peek( T & , int n ) const = 0;
    virtual bool nb_poke( const T & , int n = 0 ) = 0;
};

// Fifo interfaces
template < typename T >
class tlm_fifo_put_if :
    public virtual tlm_put_if<T> ,
    public virtual tlm_fifo_debug_if<T> {};

template < typename T >
class tlm_fifo_get_if :
    public virtual tlm_get_peek_if<T> ,
    public virtual tlm_fifo_debug_if<T> {
public:
        using tlm_get_peek_if<T>::nb_peek;
        using tlm_fifo_debug_if<T>::nb_peek
};
} // namespace tlm
```

17.3.3 Member functions

The following member functions are all pure virtual functions. The descriptions refer to the expected definitions of the functions when overridden in a channel that implements this interface. The precise semantics will be channel-specific.

Member function **used** shall return the number of items currently available to be retrieved from the fifo using **get** or **nb_get**.

Member function **size** shall return the current size of the fifo, that is, the maximum number of items that the fifo can hold at any instant.

Member function **debug** shall print diagnostic information pertaining to the current state of the fifo to standard output.

Member function **nb_peek** shall return a reference to the item at a given position in the fifo, where position 0 holds the item that will next be retrieved by **get** or **nb_get**, and position **used**()-1 holds the item most recently inserted by **put** or **nb_put**. If there is no item at the given position or the given position is negative, **nb_peek** shall return the value **false**, or otherwise **true**.

Member function **nb_poke** shall overwrite the item at a given position in the fifo with another item passed as an argument, where position 0 holds the item that will next be retrieved by **get** or **nb_get**, and position **used**()-1 holds the item most recently inserted by **put** or **nb_put**. If there is no item at the given position or the given position is negative, **nb_peek** shall return the value **false**, or otherwise **true**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

17.4 tlm_fifo

17.4.1 Description

Class **tlm_fifo** is a predefined primitive channel intended to model the behavior of a fifo, that is, a first-in-first-out buffer. Each TLM fifo has a number of slots for storing items. The number of slots is set when the object is constructed but a TLM fifo may be resized after construction and may be unbounded. The primary differences between classes **tlm_fifo** and **sc_fifo** are that first, the **tlm_fifo** implements the TLM-1 message-passing interface as opposed to the SystemC fifo interface, and second, a **tlm_fifo** may be resized or unbounded rather than having a fixed size. In the following description, the term *fifo* refers to an object of class **tlm_fifo**.

Each fifo shall implement the semantics of the **tlm_put_if**, **tlm_get_if**, **tlm_peek_if**, and **tlm_fifo_debug_if** as described above by storing the sequence of transaction objects passed through those interfaces in a single first-in-first-out buffer. Calls to **get** or **nb_get** shall retrieve transactions from the fifo in the same sequence in which transactions were previously inserted into the fifo using calls to **put** or **nb_put**. Whenever a transaction object is retrieved from the fifo through a call to **get** or **nb_get**, the fifo shall not retain any internal copy of or reference to the retrieved transaction object.

Class **tlm_fifo** shall implement delta cycle semantics as described in 17.4.6.

17.4.2 Class definition

```
namespace tlm {

template <typename T>
class tlm_fifo :
    public virtual tlm_fifo_get_if<T>,
    public virtual tlm_fifo_put_if<T>,
    public sc_core::sc_prim_channel
{
public:
    explicit tlm_fifo( int size_ = 1 );
    explicit tlm_fifo( const char* name_, int size_ = 1 );
    virtual ~tlm_fifo();

    T get( tlm_tag<T> *t = 0 );
    bool nb_get( T& );
    bool nb_can_get( tlm_tag<T> *t = 0 ) const;
    const sc_core::sc_event &ok_to_get( tlm_tag<T> *t = 0 ) const;

    T peek( tlm_tag<T> *t = 0 ) const;
    bool nb_peek( T& ) const;
    bool nb_can_peek( tlm_tag<T> *t = 0 ) const;
    const sc_core::sc_event &ok_to_peek( tlm_tag<T> *t = 0 ) const;

    void put( const T& );
    bool nb_put( const T& );
    bool nb_can_put( tlm_tag<T> *t = 0 ) const;
    const sc_core::sc_event& ok_to_put( tlm_tag<T> *t = 0 ) const;

    void nb_expand( unsigned int n = 1 );
    void nb_unbound( unsigned int n = 16 );
    bool nb_reduce( unsigned int n = 1 );
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    bool nb_bound( unsigned int n );

    bool nb_peek( T & , int n ) const;
    bool nb_poke( const T & , int n = 0 );

    int used() const;
    int size() const;
    void debug() const;

  const char* kind() const;
} // namespace tlm
```

17.4.3 Template parameter T

The typename argument passed to template **tlm_fifo** shall be either a C++ type for which the predefined semantics for assignment are adequate (for example, a fundamental type or a pointer) or a type T that obeys each of the following rules:

a) If the default assignment semantics are inadequate to assign the state of the object, the following assignment operator should be defined for the type T. The implementation shall use this operator to copy the value being written into a fifo slot or the value being read out of a fifo slot.

```
const T& operator= ( const T& );
```

b) If any constructor for type T exists, a default constructor for type T shall be defined.

NOTE 1—The assignment operator is not obliged to assign the complete state of the object, although it should typically do so. For example, diagnostic information may be associated with an object that is not to be propagated through the fifo.

NOTE 2—The SystemC data types proper (sc_dt::sc_int, sc_dt::sc_logic, and so forth) all conform to the above rule set.

NOTE 3—It is legal to pass type sc_module* through a fifo, although this would be regarded as an abuse of the module hierarchy and thus bad practice.

17.4.4 Constructors and destructor

explicit **tlm_fifo**(int size_ = 1);

This constructor shall call the base class constructor from its initializer list as follows:

```
sc_prim_channel( sc_gen_unique_name( "fifo" ) )
```

explicit **tlm_fifo**(const char* name_, int size_ = 1);

This constructor shall call the base class constructor from its initializer list as follows:

```
sc_prim_channel( name_ )
```

Both constructors shall initialize the number of slots in the fifo using the value given by the parameter size_. This value may be positive, negative, or zero. A positive value shall indicate a bounded fifo of the given size, and a negative value shall indicate an unbounded fifo. A bounded fifo may become full, an unbounded fifo cannot become full, and the behavior of a fifo with a size equal to 0 shall be undefined.

virtual ~**tlm_fifo**();

The destructor shall delete the first-in-first-out buffer and shall delete all transaction objects.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

17.4.5 Member functions

The member functions of the interfaces **tlm_put_if**, **tlm_get_if**, and **tlm_peek_if** shall be implemented as described in 17.2 with the addition of the delta cycle semantics described in 17.4.6.

void **nb_expand**(unsigned int n = 1);

Member function **nb_expand** shall increase the size of a bounded fifo by the value passed as an argument (new_size = previous_size + n), and shall cause the event returned by **ok_to_put** to be notified in the immediately following update phase of the fifo. If the fifo is unbounded, the behavior of **nb_expand** shall be undefined.

void **nb_unbound**(unsigned int n = 16);

Member function **nb_unbound** shall cause the fifo to become unbounded, and shall cause the event returned by **ok_to_put** to be notified in the immediately following update phase of the fifo, regardless of whether or not the fifo was previously unbounded. If the value passed as an argument is greater than the current size of the fifo, **nb_unbound** shall resize the fifo to that value; otherwise, the size of the fifo shall remain unaltered.

bool **nb_reduce**(unsigned int n = 1);

Member function **nb_reduce** shall return **true** and shall reduce the size of a bounded fifo by the value passed as an argument provided that the new size is not less than the number of items currently in the fifo (new_size = max(previous_size - n, used)). If the proposed size is less than the number of items currently in the fifo **nb_reduce** shall return **false** and shall reduce the size to the number of items currently in the fifo. If the fifo is unbounded, **nb_reduce** shall return **false** without modifying the size.

bool **nb_bound**(unsigned int n);

Member function **nb_bound** shall cause the fifo to become bounded regardless of whether or not the fifo was previously bounded. The size of the fifo shall be set to the value passed as an argument or to the number of items currently in the fifo, whichever is the greater (new_size = max(n, used)). **nb_bound** shall return **true** if and only if the new size of the fifo is equal to the value passed as an argument.

bool **nb_peek**(T & , int n) const;

Member function **nb_peek** shall return the item at a given position in the fifo, where position 0 holds the item that will next be retrieved by **get** or **nb_get**, and position **used**()-1 holds the item most recently inserted by **put** or **nb_put**. If there is no item at the given position or the given position is negative, **nb_peek** shall return the value **false**, or otherwise **true**.

bool **nb_poke**(const T & , int n = 0);

Member function **nb_poke** shall overwrite the item at a given position in the fifo with another item passed as an argument, where position 0 holds the item that will next be retrieved by **get** or **nb_get**, and position **used**()-1 holds the item most recently inserted by **put** or **nb_put**. If there is no item at the given position or the given position is negative, **nb_peek** shall return the value **false**, or otherwise **true**.

int **used**() const;

Member function **used** shall return the number of items currently available to be retrieved from the fifo using **get**, **nb_get**, or **nb_peek** or modified using **nb_poke**. Member functions **put** and **nb_put** may cause the value of **used** to be increased in the next update phase, member functions **get** and **nb_get** may cause the value of **used** to be decreased immediately, and member functions **peek**, **nb_peek**, and **nb_poke** shall not change the value of **used**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

int **size**() const;

Member function **size** shall return the current size of the fifo, that is, the maximum number of items that the fifo can hold at any instant. The fifo may be re-sized. A non-negative **size** shall indicate that the fifo is bounded, that is, can become full. For non-negative size, member functions **put**, **nb_put**, **get**, and **nb_get** shall not change the value of **size**. A negative **size** shall indicated that the fifo is unbounded, in which case the actual value of **size** is undefined.

void **debug**() const;

Member function **debug** shall print diagnostic information pertaining to the current state of the fifo to standard output. The detail of the diagnostic information is undefined.

const char* **kind**() const;

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

17.4.6 Delta cycle semantics

Any transactions inserted into the fifo by calls to **put** or **nb_put** shall only become available to **get**, **nb_get**, **peek**, or **nb_peek** in the next delta cycle, although they shall have an immediate effect on any subsequent calls to **put** or **nb_put** in the current evaluation phase with respect to the fifo becoming full.

Any vacated slots created in the fifo by a calls to **get** or **nb_get** shall only become available to **put** or **nb_put** in the next delta cycle, although they shall have an immediate effect on any subsequent calls to **get** or **nb_get** in the current evaluation phase.

In other words, calls to **put**, **nb_put**, **get**, or **nb_get** in a given evaluation phase shall cause relevant state variables to be modified in the update phase of the primitive channel such that the effect only becomes visible in the immediately following evaluation phase. This shall include the notification of the events returned by the member functions **ok_to_put**, **ok_to_get**, and **ok_to_peek**.

For example, a sequence of calls to **put** on an empty fifo within a given evaluation phase may cause the fifo to become full and **put** to block, even though **nb_get** would still return **false**. Similarly, a sequence of calls to **get** on a full fifo within a given evaluation phase may cause the fifo to become empty and **get** to block, even though **nb_put** would still return **false**.

The member functions **nb_peek** and **nb_poke** of class **tlm_fifo_debug_if** do not have access to any transactions inserted into the fifo by calls to **put** or **nb_put** in the current evaluation phase, nor do they have access to any transctions already removed from the fifo by calls to **get** or **nb_get** in the current evaluation phase.

Example:

```
struct Top : sc_core::sc_module {
    typedef tlm_fifo<int> fifo_t;

    fifo_t *fifo;

    Top(sc_core::sc_module_name _name) {
        fifo = new fifo_t(2);                    // Construct bounded fifo with size of 2
        SC_THREAD(T1);
        SC_THREAD(T2);
    }

    sc_dt::uint64 delta;
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    void T1() {
        sc_assert(fifo->size() == 2);
        for (int i = 0; i < 4; i++)
            fifo->put(i);                           // The third call will block

        fifo->nb_expand(2);                         // Increase fifo size
        sc_assert(fifo->size() == 4);

        for (int i = 4; i < 8; i++)
            fifo->put(i);

        sc_assert(fifo->nb_reduce(3));              // Decrease fifo size
        sc_assert(fifo->size() == 1);

        for (int i = 8; i < 12; i++)
            fifo->put(i);                           // The second call will block

        fifo->nb_unbound();                         // Make fifo unbounded
        sc_assert(fifo->size() < 0);

        delta = sc_delta_count();

        for (int i = 101; i <= 104; i++)
            fifo->put(i);

        sc_assert(sc_delta_count() == delta);
        sc_assert(fifo->used() == 0);
    }

    void T2() {
        for (int i = 0; i < 12; i++)
            sc_assert(fifo->get() == i);

        sc_assert(fifo->get() == 101);
        sc_assert(fifo->used() == 3);
        sc_assert(sc_delta_count() == delta + 1);

        sc_assert(fifo->get() == 102);
        sc_assert(fifo->used() == 2);
        sc_assert(sc_delta_count() == delta + 1);

        sc_assert(fifo->get() == 103);
        sc_assert(fifo->used() == 1);
        sc_assert(sc_delta_count() == delta + 1);

        sc_assert(fifo->get() == 104);
        sc_assert(fifo->used() == 0);
        sc_assert(sc_delta_count() == delta + 1);
    }
};
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

17.5 Analysis interface and analysis ports

17.5.1 Overview

Analysis ports are intended to support the distribution of transactions to multiple components for analysis, meaning tasks such as checking for functional correctness or collecting functional coverage statistics. The key feature of analysis ports is that a single port can be bound to multiple channels or *subscribers* such that the port itself replicates each call to the interface method **write** with each subscriber. An analysis port can be bound to zero or more subscribers or other analysis ports, and can be unbound.

Class **tlm_analysis_port** is derived from class **sc_object**, not from class **sc_port**, so an analysis port is not technically a port. Analysis ports can be instantiated, deleted, bound, and unbound dynamically during simulation.

Each subscriber implements the member function **write** of the **tlm_analysis_if**. The member function is passed a **const** reference to a transaction, which a subscriber may process immediately. Otherwise, if the subscriber wishes to extend the lifetime of the transaction, it is obliged to take a deep copy of the transaction object, at which point the subscriber effectively becomes the initiator of a new transaction and is thus responsible for the memory management of the copy.

Analysis ports should not be used in the main operational pathways of a model, but only where data is tapped off and passed to the side for analysis. Interface **tlm_analysis_if** is derived from **tlm_write_if**. The latter interface is not specific to analysis, and may be used for other purposes. For example, see 16.4.

The **tlm_analysis_fifo** is simply an infinite **tlm_fifo** that implements the **tlm_analysis_if** to write a transaction to the fifo.

17.5.2 Class definition

```
namespace tlm {

// Write interface
template <typename T>
class tlm_write_if : public virtual sc_core::sc_interface {
public:
    virtual void write( const T& ) = 0;
};

// Analysis interface
template <typename T>
class tlm_analysis_if : public virtual tlm_write_if<T>
{
};

// Analysis port
template < typename T>
class tlm_analysis_port : public sc_core::sc_object , public virtual tlm_analysis_if<T>
{
public:
    tlm_analysis_port();
    tlm_analysis_port( const char * );

    // bind and () work for both interfaces and analysis ports,
    // since analysis ports implement the analysis interface
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

```
    virtual void bind( tlm_analysis_if<T> &  );
    void operator() ( tlm_analysis_if<T> & );
    virtual bool unbind( tlm_analysis_if<T> & );

    void write( const T & );
};

// Analysis fifo - an unbounded tlm_fifo
template<typename T>
class tlm_analysis_fifo :
    public tlm_fifo<T> ,
    public virtual tlm_analysis_if<T> ,

public:
    tlm_analysis_fifo( const char *nm ) : tlm_fifo<T>( nm, –16 ) {}
    tlm_analysis_fifo() : tlm_fifo<T>( –16 ) {}

    void write( const T &t ) { nb_put( t ); }
};

} // namespace tlm
```

17.5.3 Rules

a) **tlm_write_if** and **tlm_analysis_if** (and their delayed variants) are unidirectional, non-negotiated, non-blocking transaction-level interfaces, meaning that the callee has no choice but to accept immediately the transaction passed as an argument.

b) The constructor shall pass any character string argument to the constructor belonging to the base class **sc_object** to set the string name of the instance in the module hierarchy.

c) Member function **bind** shall register the subscriber passed as an argument with the analysis port instance so that any call to the member function **write** shall be passed on to the registered subscriber. Multiple subscribers may be registered with a single analysis port instance.

d) The implementation of **operator()** shall achieve its effect by calling the virtual member function **bind**.

e) There may be zero subscribers registered with any given analysis port instance, in which case calls to the member function **write** shall not be propagated.

f) Member function **unbind** shall reverse the effect of the member function **bind**; that is, the subscriber passed as an argument shall be removed from the list of subscribers to that analysis port instance.

g) Member functions **bind** and **unbind** can be called during elaboration or can be called dynamically during simulation.

h) Member function **write** of class **tlm_analysis_port** shall call the member function **write** of every subscriber registered with that analysis port instance, passing on the argument as a **const** reference.

i) Member function **write** is non-blocking. It shall not call **wait**.

j) Member function **write** shall not modify the transaction object passed as a **const** reference argument, nor shall it modify any data associated with the transaction object (such as the data and byte enable arrays of the generic payload).

k) If the implementation of the member function **write** in a subscriber is unable to process the transaction before returning control to the caller, the subscriber shall be responsible for taking a deep copy of the transaction object and for managing any memory associated with that copy thereafter.

l) The constructors of class **tlm_analysis_fifo** shall each construct an unbounded **tlm_fifo**.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

m) Member function **write** of class **tlm_analysis_fifo** shall call the member function **nb_put** of the base class **tlm_fifo**, passing on their argument to **nb_put**.

Example:

```
struct Trans {                                  // Analysis transaction class
    int i;
};

struct Subscriber : sc_core::sc_object, tlm::tlm_analysis_if<Trans> {
    Subscriber(const char *n) : sc_core::sc_object(n) {}
    virtual void write(const Trans &t)
    {
        std::cout << "Hello, got" << t.i << std::endl; // Implementation of the write member function
    }
};

SC_MODULE(Child) {
    tlm::tlm_analysis_port<Trans> ap;

    SC_CTOR(Child) : ap("ap") {
        SC_THREAD(thread);
    }

    void thread() {
        Trans t = {999};
        ap.write(t);          // Interface method call to the write member function of the analysis port
    }
};

SC_MODULE(Parent) {
    tlm::tlm_analysis_port<Trans> ap;
    Child *child;

    SC_CTOR(Parent) : ap("ap") {
        child = new Child("child");
        child->ap.bind(ap);                     // Bind analysis port of child to analysis port of parent
    }
};

SC_MODULE(Top) {
    Parent *parent;
    Subscriber *subscriber1;
    Subscriber *subscriber2;

    SC_CTOR(Top) {
        parent = new Parent("parent");
        subscriber1 = new Subscriber("subscriber1");
        subscriber2 = new Subscriber("subscriber2");
        parent->ap.bind(*subscriber1);          // Bind analysis port to two separate subscribers
        parent->ap.bind(*subscriber2);          // This is the key feature of analysis ports
    }
};
```


Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Annex A

(informative)

Glossary

This glossary contains brief, informal descriptions for a number of terms and phrases used in this standard. Where appropriate, the complete, formal definition of each term or phrase is given in the main body of the standard. Each glossary entry contains either the clause number of the definition in the main body of the standard or an indication that the term is defined in ISO/IEC 14882:2017.

abstract class: A class that has or inherits at least one pure virtual function that is not overridden by a non-pure virtual function. (C++ term)

adapter: A module that connects a transaction level interface to a pin level interface (in the general sense of the word *interface*) or that connects together two transaction level interfaces, often at different abstraction levels. An adapter may be used to convert between two sockets specialized with different protocol types. *See also:* **bridge; transactor**. (See 14.2.3.)

application: A C++ program, written by an end user, that uses the SystemC or TLM-2.0 class libraries, that is, uses classes, calls functions, uses macros, and so forth. An application may use as few or as many features of C++ as is seen fit and as few or as many features of SystemC as is seen fit. (See 3.1.2.)

approximately-timed: A modeling style for which there exists a one-to-one mapping between the externally observable states of the model and the states of some corresponding detailed reference model such that the mapping preserves the sequence of state transitions but not their precise timing. The degree of timing accuracy is undefined. *See also:* **cycle-approximate**. (See 10.3.5.)

argument: An expression in the comma-separated list bounded by the parentheses in a function call (or macro or template instantiation), also known as an actual argument. *See also:* **parameter**. (C++ term)

attach: To associate an attribute with an object by calling member function **add_attribute** of class **sc_object**. (See 5.16.8.)

attribute (of a transaction): Data that is part of and carried with the transaction and is implemented as a member of the transaction object. These may include attributes inherent in the bus or protocol being modeled, and attributes that are artifacts of the simulation model (a timestamp, for example). (See 11.2.3 and 14.7.)

automatic deletion: A generic payload extension marked for automatic deletion will be deleted at the end of the transaction lifetime, that is, when the transaction reference count reaches 0. (See 14.21.4.)

backward path: The calling path by which a target or interconnect component makes interface method calls back in the direction of another interconnect component or the initiator. (See 10.4.)

base class sub-object: A sub-object whose type is the base class type of a given object. *See also:* **sub-object**. (C++ term)

base protocol: A protocol traits class consisting of the generic payload and tlm_phase types, along with an associated set of protocol rules that together help ensure maximal interoperability between transaction-level models. (See 15.2.)

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

base-protocol-compliant: Obeying all the rules of the TLM-2.0 base protocol. (See 9.2.)

bidirectional interface: A TLM-1 transaction level interface in which a pair of transaction objects, the request and the response, are passed in opposite directions, each being passed according to the rules of the unidirectional interface. For each transaction object, the transaction attributes are strictly read-only in the period between the first timing point and the end of the transaction lifetime. (See 17.2.1.)

binding, bound: An asymmetrical association created during elaboration between a port or export on the one hand and a channel (or another port or export) on the other. If a port (or export) is bound to a channel, a process can make an interface method call through the port to a method defined in the channel. Ports can be bound by name or by position. Exports can only be bound by name. *See also:* **interface method call.** (See 4.2.4.)

bit-select: A class that references a single bit within a multiple-bit data type or an instance of such a class. Bit-selects are defined for each SystemC numeric type and vector class. Bit-selects corresponding to lvalues and rvalues of a particular type are distinct classes. (See 7.2.6.)

bit vector: A class that is derived from class **sc_bv_base**, or an instance of such a class. A bit vector implements a multiple-bit data type, where each bit is represented by the symbol “0” or “1”. (See 7.1.)

blocking: Permitted to call the member function **wait**. A blocking function may consume simulation time or perform a context switch and, therefore, shall not be called from a method process. A blocking interface defines only blocking functions.

blocking transport interface: A blocking interface of the TLM-2.0 standard that contains a single member function **b_transport**. Beware that there still exists a blocking transport member function named **transport**, part of TLM-1. (See 11.2.2.)

body: A compound statement immediately following the parameter declarations and constructor initializer (if any) of a function or constructor, and containing the statements to be executed by the function. (C++ term)

bridge: A component connecting two segments of a communication network together. A bus bridge is a device that connects two similar or dissimilar memory-mapped buses together. *See also:* **adapter; transaction bridge; transactor**. (See 10.4 and 14.21.3.)

buffer: An instance of class **sc_buffer**, which is a primitive channel derived from class **sc_signal**. A buffer differs from a signal in that an event occurs on a buffer whenever a value is written to the buffer, regardless of whether the write causes a value change. An event only occurs on a signal when the value of the signal changes. (See 6.6.1.)

call: The term *call* is taken to mean that a function is called either directly or indirectly by calling an intermediate function that calls the function in question. (See 3.1.3.)

caller: In a function call, the sequence of statements from which the given function is called. The referent of the term may be a function, a process, or a module. This term is used in preference to initiator to refer to the caller of a function as opposed to the initiator of a transaction.

callee: In a function call, the function that is called by the caller, or the module in which that function is defined. The referent of the term may be a function or a module. This term is used in preference to target to refer to the function body as opposed to the target of a transaction.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

callback: A member function overridden within a class in the module hierarchy that is called back by the kernel at certain fixed points during elaboration and simulation. The callback functions are before_end_of_elaboration, end_of_elaboration, start_of_simulation, and end_of_simulation. (See 4.5.)

channel: A class that implements one or more interfaces or an instance of such a class. A channel may be a hierarchical channel or a primitive channel, or if neither of these, it is strongly recommended that a channel at least be derived from class **sc_object**. Channels serve to encapsulate the definition of a communication mechanism or protocol. (See 3.1.4.)

child: An instance that is within a given module. Module A is a *child* of module B if module A is *within* module B. (See 3.1.4 and 5.16.1.)

class template: A pattern for any number of classes whose definitions depend on the template parameters. The compiler treats every member function of the class as a function template with the same parameters as the class template. A function template is itself a pattern for any number of functions whose definitions depend on the template parameters. (C++ term)

clock: An instance of class **sc_clock**, which is a predefined primitive channel that models the behavior of a periodic digital clock signal. Alternatively, a clock can be modeled as an instance of the class **sc_signal<bool>**. (See 6.7.1.)

clocked thread process: A thread process that is resumed only on the occurrence of a single explicit clock edge. A clocked thread process is created using the SC_CTHREAD macro. There are no dynamic clocked threads. (See 5.2.8 and 5.2.12.)

combined interfaces: Pre-defined groups of core interfaces used to parameterize the socket classes. There are four combined interfaces: the blocking and non-blocking forward and backward interfaces. (See 10.6 and 13.1.)

complete object: An object that is not a sub-object of any other object. If a complete object is of class type, it is also called a *most derived object*. (C++ term)

component: An instance of a SystemC module. This standard recognizes three kinds of component; the initiator, interconnect component, and target. (See 10.4.)

concatenation: An object that references the bits within multiple objects as if they were part of a single aggregate object. (See 7.2.8.)

contain: The inverse relationship to *within* between two modules. Module A *contains* module B if module B is *within* module A. (See 3.1.4.)

convenience socket: A socket class, derived from **tlm_initiator_socket** or **tlm_target_socket**, that implements some additional functionality and is provided for convenience. Several convenience sockets are provided as utilities. (See 16.2.)

conversion function: A member function of the form **operator type_id** that specifies a conversion from the type of the class to the type **type_id**. *See also:* **user-defined conversion**. (C++ term)

copy-constructible type: A type T for which T(t) is equivalent to t and &t denotes the address of t. This includes fundamental types as well as certain classes. (C++ term)

core interface: One of the specific transaction level interfaces defined in this standard, including the blocking and non-blocking transport interface, the direct memory interface, and the debug transport

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

interface. Each core interface is an *interface proper*. The core interfaces are distinct from the generic payload API. (See Clause 9.)

custom-protocol-compliant: Using the TLM-2.0 standard sockets (or classes derived from these) specialized with a traits class other than **tlm_base_protocol_types** and using the TLM-2.0 generic payload. (See 9.2.)

cycle-accurate: A modeling style in which it is possible to predict the state of the model in any given cycle at the external boundary of the model and thus to establish a one-to-one correspondence between the states of the model and the externally observable states of a corresponding RTL model in each cycle, but which is not required to re-evaluate explicitly the state of the entire model in every cycle or to represent explicitly the state of every boundary pin or internal register. This term is only applicable to models that have a notion of cycles. (See 10.3.8.)

cycle-approximate: A model for which there exists a one-to-one mapping between the externally observable states of the model and the states of some corresponding cycle-accurate model such that the mapping preserves the sequence of state transitions but not their precise timing. The degree of timing accuracy is undefined. This term is only applicable to models that have a notion of cycles.

cycle count accurate, cycle count accurate at transaction boundaries: A modeling style in which it is possible to establish a one-to-one correspondence between the states of the model and the externally observable states of a corresponding RTL model as sampled at the timing points marking the boundaries of a transaction. A cycle count accurate model is not required to be cycle-accurate in every cycle, but it is required to predict accurately both the functional state and the number of cycles at certain key timing points as defined by the boundaries of the transactions through which the model communicates with other models.

data member: An object declared within a class definition. A non-static data member is a sub-object of the class. A static data member is not a sub-object of the class but has static storage duration. Outside of a constructor or member function of the class or of any derived class, a data member can only be accessed using the dot **.** and arrow **->** operators. (C++ term)

declaration: A C++ language construct that introduces a name into a C++ program and specifies how the C++ compiler is to interpret that name. Not all declarations are definitions. For example, a class declaration specifies the name of the class but not the class members, while a function declaration specifies the function parameters but not the function body. *See also:* **definition**. (C++ term)

definition: The complete specification of a variable, function, type, or template. For example, a class definition specifies the class name and the class members, and a function definition specifies the function parameters and the function body. *See also:* **declaration**. (C++ term)

delta cycle: A control loop within the scheduler that consists of one evaluation phase followed by one update phase. The delta cycle mechanism serves to help ensure the deterministic simulation of concurrent processes by separating and alternating the computation (or evaluation) phase and the communication (or update) phase. (See 4.3.3.)

delta notification: A notification created as the result of a call to function **notify** with a zero time argument. The event is notified one delta cycle after the call to function **notify**. (See 4.3.2 and 5.10.6.)

delta notification phase: The control step within the scheduler during which processes are made runnable as a result of delta notifications. (See 4.3.2.5.)

during elaboration, during simulation: The phrases *during elaboration* and *during simulation* are used to indicate that an action may or may not happen at these times. The meaning of these phrases is closely tied to the definition of the elaboration and simulation callbacks. For example, a number of actions that are

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

permitted *during elaboration* are explicitly forbidden from the **end_of_elaboration** callback. (See 3.1.4 and 4.5.)

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

dynamic sensitivity: The set of events or time-outs that would cause a process to be resumed or triggered, as created by the most recent call to the member function **wait** (in the case of a thread process) or the member function **next_trigger** (in the case of a method process). *See also:* **sensitivity**. (See 4.3.)

effective local time: The current time within a temporally decoupled initiator. effective_local_time = sc_time_stamp() + local_time_offset. (See 11.2.4.2.)

elaboration: The execution phase during which the module hierarchy is created and ports are bound. The execution of a C++ application consists of elaboration followed by simulation. (See Clause 4.)

error: An obligation on the implementation to generate a diagnostic message using the report-handling mechanism (function **report** of class **sc_report_handler**) with a severity of SC_ERROR. (See 3.2.5.)

evaluation phase: The control step within the scheduler during which processes are executed. The evaluation phase is complete when the set of runnable processes is empty. *See also:* **delta cycle**. (See 4.3.2.3.)

event: An object of class **sc_event**. An event provides the mechanism for synchronization between processes. The member function **notify** of class **sc_event** causes an event to be notified at a specific point in time. (The notification of an event is distinct from an object of type sc_event. The former is a dynamic occurrence at a unique point in time, and the latter is an object that can be notified many times during its lifetime.) *See also:* **notification**. (See 3.1.4~~,~~ and 5.10.)

event expression: A list of events or event lists, separated by either operator& or operator|, and passed as an argument to either the member function **wait** or **next_trigger**. (See 5.9.)

event finder: A member function of a port class that returns an event within a channel instance to which the port is bound. An event finder can only be called when creating static sensitivity. (See 5.7.)

event list: An object of type **sc_event_and_list** or **sc_event_or_list** that may be passed as an argument to member function **wait** or **next_trigger**. (See 5.8.)

exclusion rule: A rule of the base protocol that prohibits a request or a response from being sent through a socket if there is already a request or a response (respectively) in progress through that socket. The base protocol has two exclusion rules, the request exclusion rule and the response exclusion rule, which act independently of one another. (See 15.2.6.)

export: An instance of class **sc_export**. An export specifies an interface provided by a module. During simulation, a port forwards method calls to the channel to which the export was bound. An export forwards method calls down and into a module instance. (See 3.1.4 and 5.13.)

extension: A user-defined object added to and carried around with a generic payload transaction object, or a user-defined class that extends the set of values that are assignment compatible with the tlm_phase type. An ignorable extension may be used with the base protocol, but a non-ignorable or mandatory extension requires the definition of a new protocol traits class. (See 14.21.)

fifo: An instance of class **sc_fifo**, which is a primitive channel that models a first-in-first-out buffer. Alternatively, a fifo can be modeled as a module. (See 6.23.)

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

finite-precision fixed-point type: A class that is derived from class **sc_fxnum** or an instance of such a class. A finite-precision fixed-point type represents a signed or unsigned fixed-point value at a precision limited only by its specified word length, integer word length, quantization mode, and overflow mode. (See 7.1.)

finite-precision integer: A class that is derived from class **sc_signed**, class **sc_unsigned**, or an instance of such a class. A finite-precision integer represents a signed or unsigned integer value at a precision limited only by its specified word length. (See 7.1.)

forward path: The calling path by which an initiator or interconnect component makes interface method calls forward in the direction of another interconnect component or the target. (See 10.4.)

generic payload: A specific set of transaction attributes and their semantics together defining a transaction payload that may be used to achieve a degree of interoperability between loosely-timed and approximately-timed models for components communicating over a memory-mapped bus. The same transaction class is used for all modeling styles. (See Clause 14.)

global quantum: The default time quantum used by every quantum keeper and temporally decoupled initiator. The intent is that all temporally decoupled initiators should typically synchronize on integer multiples of the global quantum, or more frequently on demand. (See Clause 12.)

hierarchical binding: Binding a socket on a child module to a socket on a parent module, or a socket on a parent module to a socket on a child module, passing transactions up or down the module hierarchy. (See 16.2.4.)

hierarchical channel: A class that is derived from class **sc_module** and that implements one or more interfaces or, more informally, an instance of such a class. A hierarchical channel is used when a channel requires its own ports, processes, or module instances. *See also:* **channel**). (See 3.1.4 and 5.2.23.)

hierarchical name: The unique name of an instance within the module hierarchy. The hierarchical name is composed from the string names of the parent-child chain of module instances starting from a top-level module and terminating with the string name of the instance being named. The string names are concatenated and separated with the dot character. (See 5.3.4 and 5.16.4.)

hop: One initiator socket bound to one target socket. The path from an initiator to a target may consist of multiple hops, each hop connecting two adjacent components. The number of hops between an initiator and a target is always one greater than the number of interconnect components along that path. For example, if an initiator is connected directly to a target with no intervening interconnect components, the number of hops is one. (See 10.4.)

ignorable extension: A generic payload extension that may be ignored by any component other than the component that set the extension. An ignorable extension is not required to be present. Ignorable extensions are permitted by the base protocol. (See 14.21.1.2.)

ignorable phase: A phase, created by the macro DECLARE_EXTENDED PHASE, that may be ignored by any component that receives the phase and that cannot demand a response of any kind. Ignorable phases are permitted by the base protocol. (See 15.2.5.)

immediate notification: A notification created as the result of a call to function with an empty argument list. Any process sensitive to the event becomes runnable immediately. (See 4.3.2 and 5.10.6.)

implementation: A specific concrete implementation of the full SystemC and TLM-2.0 class libraries, only the public shell of which need be exposed to the application (for example, parts may be precompiled and distributed as object code by a tool vendor). *See also:* **kernel.** (See 3.1.2.)

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

implement: To create a channel that provides a definition for every pure virtual function declared in the interface from which it is derived. (See 5.14.1.)

implicit conversion: A C++ language mechanism whereby a standard conversion or a user-defined conversion is called implicitly under certain circumstances. User-defined conversions are only applied implicitly where they are unambiguous, and at most one user-defined conversion is applied implicitly to a given value. *See also:* **user-defined conversion**. (C++ term)

initialization phase: The first phase of the scheduler, during which every process is executed once until it suspends or returns. (See 4.3.2.2.)

initializer list: The part of the C++ syntax for a constructor definition that is used to initialize base class sub-objects and data members. (Related to the C++ term *mem-initializer-list*)

initiator: A module that can initiate transactions. The initiator is responsible for initializing the state of the transaction object, and for deleting or reusing the transaction object at the end of the transaction's lifetime. In the case of the TLM-1 interfaces, the term *initiator* as defined here may not be strictly applicable, so the terms *caller* and *callee* may be used instead for clarity. (See 10.4.)

initiator socket: A class containing a port for interface method calls on the forward path and an export for interface method calls on the backward path. A socket overloads the SystemC binding operators to bind both the port and the export. (See 13.2.)

interconnect component: A module that accesses a transaction object, but it does not act as an initiator or a target with respect to that transaction. An interconnect component may or may not be permitted to modify the attributes of the transaction object, depending on the rules of the payload. An arbiter or a router would typically be modeled as an interconnect component, the alternative being to model it as a target for one transaction and an initiator for a separate transaction. (See 10.4.)

instance: A particular case of a given category. For example, a module *instance* is an object of a class derived from class **sc_module**. Within the definition of the core language, an *instance* is typically an object of a class derived from class **sc_object** and has a unique hierarchical name. (See 3.1.4.)

instantiation: The creation of a new object. For example, a module instantiation creates a new object of a class derived from class **sc_module**. (See 4.2.2.)

integer: A limited-precision integer or a finite-precision integer. (See 7.2.2.)

interface: A class derived from class **sc_interface**. An interface proper is an interface, and in the object-oriented sense a channel is also an interface. However, a channel is not an interface proper. (See 3.1.4.)

interface method call (IMC): A call to an interface method. An interface method is a member function declared within an interface. The IMC paradigm provides a level of indirection between a method call and the implementation of the method within a channel such that one channel can be substituted with another without affecting the caller. (See 4.2.4 and 5.12.1.)

interface proper: An abstract class derived from class **sc_interface** but not derived from class **sc_object**. An interface proper declares the set of methods to be implemented within a channel and to be called through a port. An interface proper contains pure virtual function declarations, but typically it contains no function definitions and no data members. (See 3.1.4 and 5.14.1.)

interoperability: The ability of two or more transaction level models from diverse sources to exchange information using the interfaces defined in this standard. The intent is that models that implement common memory-mapped bus protocols in the programmers view use case should be interoperable without the need

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

for explicit adapters. Furthermore, the intent is to reduce the amount of engineering effort needed to achieve interoperability for models of divergent protocols or use cases, although it is expected that adapters will be required in general. (See Clause 9.)

interoperability layer: The subset of classes in this standard that are necessary for interoperability. The interoperability layer comprises the TLM-2.0 core interfaces, the initiator and target sockets, the generic payload, **tlm_global_quantum** and **tlm_phase**. Closely related to the base protocol. (See Clause 9.)

kernel: The core of any SystemC implementation including the underlying elaboration and simulation engines. The kernel honors the semantics defined by this standard but may also contain implementation-specific functionality outside the scope of this standard. *See also:* **implementation.** (See Clause 4.)

lifetime (of an object): The lifetime of an object starts when storage is allocated and the constructor call has completed, if any. The lifetime of an object ends when storage is released or immediately before the destructor is called, if any. (C++ term)

lifetime (of a transaction): The period of time that starts when the transaction becomes valid and ends when the transaction becomes invalid. Because it is possible to pool or re-use transaction objects, the lifetime of a transaction object may be longer than the lifetime of the corresponding transaction. For example, a transaction object could be a stack variable passed as an argument to multiple *put* calls of the TLM-1 interface.

limited-precision fixed-point type: A class that is derived from class **sc_fxnum_fast**, or an instance of such a class. A limited-precision fixed-point type represents a signed or unsigned fixed-point value at a precision limited by its underlying native C++ floating-point representation and its specified word length, integer word length, quantization mode, and overflow mode. (See 7.1.)

limited-precision integer: A class that is derived from class **sc_int_base**, class **sc_uint_base**, or an instance of such a class. A limited-precision integer represents a signed or unsigned integer value at a precision limited by its underlying native C++ representation and its specified word length. (See 7.1.)

local quantum: The amount of simulation time remaining before the initiator is required to synchronize. Typically, the local quantum equals the current simulation time subtracted from the next largest integer multiple of the global quantum, but this calculation can be overridden for a given quantum keeper. (See 16.3.)

local time offset: Time as measured relative to the most recent quantum boundary in a temporally decoupled initiator. The timing annotation arguments to the **b_transport** and *nb_transport* methods are local time offsets. effective_local_time = sc_time_stamp() + local_time_offset. (See 16.3.)

logic vector: A class that is derived from class **sc_lv_base** or an instance of such a class. A logic vector implements a multiple bit data type, where each bit is represented by a four-valued logic symbol “0”, “1”, “X”, or “Z”. (See 7.1.)

loosely-timed: A modeling style that represents minimal timing information sufficient only to support features necessary to boot an operating system and to manage multiple threads in the absence of explicit synchronization between those threads. A loosely-timed model may include timer models and a notional arbitration interval or execution slot length. Some users adopt the practice of inserting random delays into loosely-timed descriptions in order to test the robustness of their protocols, but this practice does not change the basic characteristics of the modeling style. (See 10.3.3.)

lvalue: An object reference whose address can be taken. The left-hand operand of the built-inassignment operator must be a non-const lvalue. (C++ term)

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

mandatory extension: A generic payload extension that is required to be present on every transaction that is sent through a socket of a given user-defined protocol type. (See 14.21.1.3.)

member function: A function declared within a class definition, excluding friend functions. Outside of a constructor or member function of the class or of any derived class, a non-static member function can only be accessed using the dot **.** and arrow **->** operators. *See also:* **method**. (C++ term)

memory manager: A user-defined class that performs memory management for a generic payload transaction object. A memory manager must provide a **free** method, called when the reference count of the transaction reaches 0. (See 14.5.)

method: A function that implements the behavior of a class. This term is synonymous with the C++ term *member function*. In SystemC, the term *method* is used in the context of an *interface method call*. Throughout this standard, the term *member function* is used when defining C++ classes (for conformance to the C++ standard), and the term *method* is used in more informal contexts and when discussing interface method calls.

method process: A process that executes in the thread of the scheduler and is called (or triggered) by the scheduler at times determined by its sensitivity. An unspawned method process is created using the SC_METHOD macro, a spawned method process by calling the function **sc_spawn**. (See 5.2.8 and 5.2.10.)

module: A class that is derived from class **sc_module** or, more informally, an instance of such a class. A SystemC application is composed of modules, each module instance representing a hierarchical boundary. A module can contain instances of ports, processes, primitive channels, and other modules. (See 3.1.4 and 5.2.)

module hierarchy: The set of all instances created during elaboration and linked together using the mechanisms of module instantiation, port instantiation, primitive channel instantiation, process instantiation, and port binding. The module hierarchy is a subset of the object hierarchy. (See 3.1.4 and Clause 4.)

multiport: A port that may be bound to more than one channel or port instance. A multiport is used when an application wishes to bind a port to a set of addressable channels and the number of channels is not known until elaboration. (See 4.2.4 and 5.12.3.)

multi-socket: One of a family of convenience sockets that can be bound to multiple sockets belonging to other components. An initiator multi-socket can be bound to more than one target socket, and more than one initiator socket can be bound to a single target multi-socket. When calling interface methods through multi-sockets, the destinations are distinguished using the subscript operator. (See 16.2.4.)

mutex: An instance of class **sc_mutex**, which is a predefined channel that models a mutual exclusion communication mechanism. (See 6.27.1.)

nb_transport: The member functions **nb_transport_fw** and **nb_transport_bw**. In this document, the italicized term *nb_transport* is used to describe both methods in situations where there is no need to distinguish between them. (See 11.2.3.4.)

non-abstract class: A class that is not an abstract class. (C++ term)

non-blocking: Not permitted to call the member function **wait**. A non-blocking function is expected to return without consuming simulation time or performing a context switch and, therefore, may be called from a thread process or from a method process. A non-blocking interface defines only non-blocking functions.

non-blocking transport interface: A non-blocking interface of the TLM-2.0 standard. There a two such interfaces, containing member function named **nb_transport_fw** and **nb_transport_bw**. (See 10.3.9.)

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

non-ignorable extension: A generic payload extension that, if present, every component receiving the transaction is obliged to act on. (See 14.21.1.3.)

notification: The act of scheduling the occurrence of an event as performed by the member function **notify** of class **sc_event**. There are three kinds of notification: immediate notification, delta notification, and timed notification. *See also:* **event**. (See 4.3.2 and 5.10.6.)

notified: An event is said to be *notified* at the control step of the scheduler in which the event is removed from the set of pending events and any processes that are currently sensitive to that event are made runnable. Informally, the event *occurs* precisely at the point when it is notified. (See 4.3.)

numeric type: A finite-precision integer, a limited-precision integer, a finite-precision fixed-point type, or a limited-precision fixed-point type. (See 7.1.)

object: A region of storage. Every object has a type and a lifetime. An object created by a definition has a name, whereas an object created by a new expression is anonymous. (C++ term)

object hierarchy: The set of all objects of class **sc_object**. Each object has a unique hierarchical name. Objects that do not belong to the module hierarchy may be created and destroyed dynamically during simulation. (See 3.1.4 and 5.16.1.)

occurrence: The notification of an event. Except in the case of immediate notification, a call to the member function **notify** of class **sc_event** will cause the event to *occur* in a later delta cycle or at a later point in simulation time. Of a time-out: a time-out *occurs* when the specified time interval has elapsed. (See 5.10.1.)

opposite path: The path in the opposite direction to a given path. For the forward path, the opposite path is the forward return path or the backward path. For the backward path, the opposite path is the forward path or the backward return path. (See 10.4.)

overload: To create two or more functions with the same name declared in the same scope and that differ in the number or type of their parameters. (C++ term)

override: To create a member function in a derived class has the same name and parameter list as a member function in a base class. (C++ term)

parameter: An object declared as part of a function declaration or definition (or macro definition or template parameter), also known as a formal parameter. *See also:* **argument**. (C++ term)

parent: The inverse relationship to *child*. Module A is the *parent* of module B if module B is a *child* of module A. (See 3.1.4 and 5.16.1.)

part-select: A class that references a contiguous subset of bits within a multiple-bit data type or an instance of such a class. Part-selects are defined for each SystemC numeric and vector class. Part-selects corresponding to lvalues and rvalues of a particular type are distinct classes. (See 7.2.8.)

paused: The state of simulation after the scheduler has been exited following a call to **sc_pause**.

payload event queue (PEQ): A class that maintains a queue of SystemC event notifications, where each notification carries an associated transaction object. Transactions are put into the queue annotated with a delay, and each transaction pops out of the back of queue at the time it was put in plus the given delay. Useful when combining the non-blocking interface with the approximately-timed coding style. (See 16.4.)

pending: The state of an event for which a notification has been posted; that is, the member function **notify** has been called, but the event has not yet been notified.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

phase: A period in the lifetime of a transaction. The phase is passed as an argument to the non-blocking transport method. Each phase transition is associated with a timing point. The timing point may be delayed by an amount given by the time argument to *nb_transport*. (See 11.2.3.6.)

phase transition: A transition from one phase to another, where the phase is represented by the value of the phase argument to the non-blocking transport method. Each call to *nb_transport*, and each return from *nb_transport* with a return value of TLM_UPDATED, marks a phase transition. The base protocol does not permit calls to *nb_transport* where the value of the phase argument is unchanged from the previous state. (See 15.2.4.)

port: A class that is derived from class **sc_port** or, more informally, an instance of such a class. A port is the primary mechanism for allowing communication across the boundary of a module. A port specifies an interface required by a module. During simulation, a port forwards method calls made from a process within a module to the channel to which the port was bound when the module was instantiated. A port forwards method calls up and out of a module instance. (See 3.1.4 and 5.12.)

portless channel access: Calling the member functions of a channel directly and not through a port or export. (See 5.12.1.)

primitive channel: A class that is derived from class **sc_prim_channel** and implements one or more interfaces or, more informally, an instance of such a class. A primitive channel has access to the update phase of the scheduler but cannot contain ports, processes, or module instances. (See 3.1.4 and 5.15.)

process: A process instance belongs to an implementation-defined class derived from class **sc_object**. Each process instance has an associated function that represents the behavior of the process. A process may be a static process, a dynamic process, a spawned process, or an unspawned process. The process is the primary means of describing a computation. *See also:* **dynamic process; spawned process; static process; unspawned process**. (See 3.1.4.)

process handle: An object of class **sc_process_handle** that provides safe access to an underlying spawned or unspawned process instance. A process handle can be valid or invalid. A process handle continues to exist in the invalid state even after the associated process instance has been destroyed. (See 3.1.4 and 5.6.)

programmers view (PV): The use case of the software programmer who requires a functionally accurate, loosely-timed model of the hardware platform for booting an operating system and running application software.

protocol traits class: A class containing a **typedef** for the type of the transaction object and the phase type, which is used to parameterize the combined interfaces, and effectively defines a unique type for a protocol. (See 14.2.3 and 14.2.4.)

proxy class: A class whose only purpose is to extend the readability of certain statements that would otherwise be restricted by the semantics of C++. An example is to allow an **sc_int** variable to be used as if it were a C++ array of bool. Proxy classes are only intended to be used for the temporary (unnamed) value returned by a function. A proxy class constructor shall not be called explicitly by an application to create a named object. (See 7.2.6.)

quantum: In temporal decoupling, the amount a process is permitted to run ahead of the current simulation time. (See 10.3.3, 11.2.2.7, and Clause 12.)

quantum keeper: A utility class used to store the local time offset from the current simulation time, which it checks against a local quantum. (See 16.3.)

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

request: For the base protocol, the stage during the lifetime of a transaction when information is passed from the initiator to the target. In effect, the request transports generic payload attributes from the initiator to the target, including the command, the address, and for a write command, the data array. (The transaction is actually passed by reference and the data array by pointer.)

resolved signal: An instance of class **sc_signal_resolved** or **sc_signal_rv**, which are signal channels that may be written to by more than one process, with conflicting values being resolved within the channel. (See 6.13.1.)

response: For the base protocol, the stage during the lifetime of a transaction when information is passed from the target back to the initiator. In effect, the response transports generic payload attributes from the target back to the initiator, including the response status, and for a read command, the data array. (The transaction is actually passed by reference and the data array by pointer.)

resume: To cause a thread or clocked thread process to continue execution starting with the executable statement immediately following the member function **wait** at which it was suspended, dependent on the sensitivity of the process. (See 5.2.11.) Also, a member function of class **sc_process_handle** that cancels the effect of a previous call to **suspend**. (See 5.6.6.2.)

return path: The control path by which the call stack of a set of interface method calls is unwound along either the forward path or the backward path. The return path for the forward path can carry information from target to initiator, and the return path for the backward path can carry information from initiator to target. (See 10.4.)

rvalue: A value that does not necessarily have any storage or address. An rvalue of fundamental type can only appear on the right-hand side of an assignment. (C++ term)

scheduled: The state of an event or of a process as a result of a call to the member function **notify**, **next_trigger**, or **wait**. An event can be *scheduled* to occur, or a process can be *scheduled* to be triggered or resumed, either in a later delta cycle or at a later simulation time.

scheduler: The part of the kernel that controls simulation and is thus concerned with advancing time, making processes runnable as events are notified, executing processes, and updating primitive channels. (See Clause 4.)

sensitivity: The set of events or time-outs that would cause a process to be resumed or triggered. Sensitivity make take the form of static sensitivity or dynamic sensitivity. (See 4.3.)

signal: An instance of class **sc_signal**, which is a primitive channel intended to model relevant aspects of the behavior of a simple wire as appropriate for digital hardware simulation. (See 3.1.4 and 6.4.)

signature: The information about a function relevant to overload resolution, such as the types of its parameters and any qualifiers. (C++ term)

simple socket: One of a family of convenience sockets that are simple to use because they allow callback methods to be registered directly with the socket object rather than the socket having to be bound to another object that implements the required interfaces. The simple target socket avoids the need for a target to implement both blocking and non-blocking transport interfaces by providing automatic conversion between the two. (See 16.2.2.)

simulation: The execution phase that consists of the execution of the scheduler together with the execution of user-defined processes under the control of the scheduler. The execution of a SystemC application consists of elaboration followed by simulation. (See Clause 4.)

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

socket: *See:* **initiator socket**; **target socket**.

spawned process: A process instance created by calling the function **sc_spawn**. *See also:* **process**. (See 3.1.4 and 5.5.6.)

specialized port: A class derived from template class **sc_port** that passes a particular type as the first argument to template **sc_port**, and which provides convenience functions for accessing ports of that specific type. (See 6.8.)

standard error response: The behavior prescribed by this standard for a generic payload target that is unable to execute a transaction successfully. A target should either (1) execute the transaction successfully, (2) set the response status attribute to an error response, or (3) call the SystemC report handler. (See 14.17.2.)

statement: A specific category of C++ language construct that is executed in sequence, such as the *if statement*, *switch statement*, *for statement*, and *return statement*. A C++ expression followed by a semicolon is also a statement. (C++ term)

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

static sensitivity: The set of events or time-outs that would cause a process to be resumed or triggered, as created using the data member **sensitive** of class **sc_module** (in the case of an unspawned process) or using class **sc_spawn_options** (in the case of a spawned process). *See also:* **sensitivity; spawned process; unspawned process**. (See 4.3.)

sticky extension: A generic payload extension object that is not deleted (either automatically or explicitly) at the end of life of the transaction object, and thus remains with the transaction object when it is pooled. Sticky extensions are not deleted by the memory manager. (See 14.5.)

string name: A name passed as an argument to the constructor of an instance to provide an identity for that object within the module hierarchy. The string names of instances having a common parent module will be unique within that module and that module only. *See also:* **hierarchical name**. (See 5.3 and 5.16.4.)

sub-object: An object contained within another object. A sub-object of a class may be a data member of that class or a base class sub-object. (C++ term)

suspend: To cause a process to cease execution by having the associated function call **wait**. Also, a member function of class **sc_process_handle** that causes a process to remain suspended and whose effect can be cancelled by calling **resume**. Also, to cause a process to cease execution by calling member function **suspend** of a process handle associated with the process itself. (See 5.6.6.2.)

synchronize: To yield such that other processes may run, or when using temporal decoupling, to yield and wait until the end of the current time quantum. (See 10.3.3.)

synchronization-on-demand: The action of a temporally decoupled process when it yields control back to the SystemC scheduler so that simulation time may advance and other processes run in addition to the synchronization points that may occur routinely at the end of each quantum. (See 10.3.4.)

synchronous reset state: The state entered by a process instance when its reset signal becomes active or when **sync_reset_on** is called. When in the synchronous reset state, a process instance is reset each time it is resumed. (See 5.6.6.4.)

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

tagged socket: One of a family of convenience sockets that add an int id tag to every incoming interface method call in order to identify the socket (or element of a multi-socket) through which the transaction arrived. (See 16.2.3.4.)

target: A module that represents the final destination of a transaction, able to respond to transactions generated by an initiator, but not itself able to initiate new transactions. For a write operation, data is copied from the initiator to one or more targets. For a read operation, data is copied from one target to the initiator. A target may read or modify the state of the transaction object. In the case of the TLM-1 interfaces, the term *target* as defined here may not be strictly applicable, so the terms *caller* and *callee* may be used instead for clarity. (See 10.4.)

target socket: A class containing a port for interface method calls on the backward path and an export for interface method calls on the forward path. A socket also overloads the SystemC binding operators to bind both port and export. (See 10.4.)

temporal decoupling: The ability to allow one or more initiators to run ahead of the current simulation time in order to reduce context switching and thus increase simulation speed. (See 10.3.3.)

terminated: The state of a thread or clocked thread process when the associated function executes to completion or executes a return statement and thus control returns to the kernel or after the process has been killed. Calling function **wait** does not terminate a thread process. *See also:* **clocked thread process; method process; thread process.** (See 5.2.11 and 5.6.5.)

thread process: A process that executes in its own thread and is called once only by the scheduler during initialization. A thread process may be suspended by the execution of a member function **wait**, in which case it will be resumed under the control of the scheduler. An unspawned thread process is created using the SC_THREAD macro, a spawned thread process by calling the function **sc_spawn**. *See also:* **spawned process; unspawned process**. (See 5.2.8 and 5.2.11.)

time-out: The thing that causes a process to resume or trigger as a result of a call to the member function **wait** or **next_trigger** with a time-valued argument. The process that called the member function will resume or trigger after the specific time has elapsed, unless it has already resumed or triggered as a result of an event being notified. (See 4.3 and 4.3.2.)

timed notification: A notification created as the result of a call to function notify with a non-zero time argument. (See 4.3.2 and 5.10.6.)

timed notification phase: The control step within the scheduler during which processes are made runnable as a result of timed notifications. (See 4.3.2.6.)

timing annotation: The **sc_time** argument to the **b_transport** and *nb_transport* methods. A timing annotation is a local time offset. The recipient of a transaction is required to behave as if it had received the transaction at effective_local_time = sc_time_stamp() + local_time_offset. (See 11.2.3.12 and 11.2.4.)

timing point: A significant time within the lifetime of a transaction. A loosely-timed transaction has two timing points corresponding to the call to and return from **b_transport**. An approximately-timed base protocol transaction has four timing points, each corresponding to a phase transition.

TLM-1: The first major version of the OSCI Transaction Level Modeling standard. TLM-1 was released in 2005. (See Clause 9.)

TLM-2.0: The second major version of the OSCI Transaction Level Modeling standard. TLM-2.0 was first released in 2008 and the OSCI TLM-2.0 LRM was released in 2009. (See Clause 9.)

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

TLM-2.0-compliant implementation: An implementation that provides all of the TLM-2.0 classes described in this standard with the semantics described in this standard, including both the TLM-2.0 interoperability layer and the TLM-2.0 utilities. (See 9.2.)

top-level module, top-level object: A module or object that is not instantiated within any other module or process. Top-level modules are either instantiated within **sc_main**, or in the absence of **sc_main**, are identified using an implementation-specific mechanism. (See 3.1.4 and 5.16.1.)

traits class: In C++ programming, a class that contains definitions such as typedefs that are used to specialize the behavior of a primary class, typically by having the traits class passed as a template argument to the primary class. The default template parameter provides the default traits for the primary class. (See 14.2.3 and 14.2.4.)

transaction: An abstraction for an interaction or communication between two or more concurrent processes. A transaction carries a set of attributes and is bounded in time, meaning that the attributes are only valid within a specific time window. The timing associated with the transaction is limited to a specific set of timing points, depending on the type of the transaction. Processes may be permitted to read or modify attributes of the transaction, depending on the protocol.

transaction bridge: A component that acts as the target for an incoming transaction and as the initiator for an outgoing transaction, usually for the purpose of modeling a bus bridge. *See also:* **bridge**. (See 10.4.)

transaction instance: A unique instance of a transaction. A transaction instance is represented by one transaction object, but the same transaction object may be re-used for several transaction instances.

transaction level (TL): The abstraction level at which communication between concurrent processes is abstracted away from pin wiggling to transactions. This term does not imply any particular level of granularity with respect to the abstraction of time, structure, or behavior. (See 10.2.)

transaction object: The object that stores the attributes associated with a transaction. The type of the transaction object is passed as a template argument to the core interfaces.

transaction level model, transaction level modeling (TLM): A model at the transaction level and the act of creating such a model, respectively. Transaction level models typically communicate using function calls, as opposed to the style of setting events on individual pins or nets as used by RTL models. (See 10.2.)

transactor: A module that connects a transaction level interface to a pin level interface (in the general sense of the word *interface*) or that connects together two or more transaction level interfaces, often at different abstraction levels. In the typical case, the first transaction level interface represents a memory-mapped bus or other protocol, and the second interface represents the implementation of that protocol at a lower abstraction level. However, a single transactor may have multiple transaction level or pin level interfaces. *See also:* **adapter; bridge**.

transparent component: A interconnect component with the property that all incoming interface method calls are propagated immediately through the component without delay and without modification to the arguments or to the transaction object (extensions excepted). The intent of a transparent component is to allow checkers and monitors to pass ignorable phases. (See 15.2.5.)

transport interface: The one and only bidirectional core interface in TLM-1. The transport interface passes a request transaction object from caller to callee, and it returns a response transaction object from callee to caller. TLM-2.0 adds separate blocking and non-blocking transport interfaces. (See 10.3.9.)

trigger: To cause the member function associated with a method process instance to be called by the scheduler, dependent on its sensitivity. *See also:* **method process; sensitivity**. (See 5.2.10.)

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

undefined: The absence of any obligations on the implementation. Where this standard states that a behavior or a result is *undefined*, the implementation may or may not generate an error or a warning. (See 3.3.6.)

unidirectional interface: A TLM-1 transaction level interface in which the attributes of the transaction object are strictly read-only in the period between the first timing point and the end of the transaction lifetime. Effectively, the information represented by the transaction object is strictly passed in one direction either from caller to callee or from callee to caller. In the case of **void put(const T& t)**, the first timing point is marked by the function call. In the case of **void get(T& t)**, the first timing point is marked by the return from the function. In the case of **T get()**, strictly speaking there are two separate transaction objects, and the return from the function marks the degenerate end-of-life of the first object and the first timing point of the second. (See 17.2.1.)

unspawned process: A process created by invoking one of the three macros SC_METHOD, SC_THREAD, or SC_CTHREAD during elaboration. *See also:* **process**. (See 3.1.4 and 4.2.3.)

untimed: A modeling style in which there is no explicit mention of time or cycles, but which includes concurrency and sequencing of operations. In the absence of any explicit notion of time as such, the sequencing of operations across multiple concurrent threads must be accomplished using synchronization primitives such as events, mutexes, and blocking FIFOs. Some users adopt the practice of inserting random delays into untimed descriptions in order to test the robustness of their protocols, but this practice does not change the basic characteristics of the modeling style. (See 10.3.2.)

update phase: The control step within the scheduler during which the values of primitive channels are updated. The update phase consists of executing the member function **update** for every primitive channel that called the member function **request_update** during the immediately preceding evaluation phase. (See 4.3.2.4.)

user: The creator of an application, as distinct from an implementor, who creates an implementation. A user may be a person or an automated process such as a computer program. (See 3.1.2.)

user-defined conversion: Either a conversion function or a non-explicit constructor with exactly one parameter. *See also:* **conversion function; implicit conversion**. (C++ term)

utilities: A set of classes of the TLM-2.0 standard that are provided for convenience only and are not strictly necessary to achieve interoperability between transaction-level models. (See Clause 16.)

valid: The state of a process handle, or of an object passed to or returned from a function by pointer or by reference, during any period in which the handle or object is not deleted and its value or behavior remains accessible to the application. A process handle is *valid* when it is associated with a process instance. (See 3.3.4 and 5.6.1.)

variable-precision fixed-point type: Classes **sc_fxval** and **sc_fxval_fast** represent variable-precision fixed-point values that do not model overflow and quantization effects but are used as operand types and return types by many fixed-point operations. These types are not typically used directly by an application. (See 7.1.)

vector: *See:* **bit vector**; **logic vector**. (See 7.1 and 8.5.)

warning: An obligation on the implementation to generate a diagnostic message using the report-handling mechanism (function **report** of class **sc_report_handler**) with a severity of SC_WARNING. (See 3.3.6.)

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

within: The relationship that exists between an instance and a module if the constructor of the instance is called from the constructor of the module, and also provided that the instance is not *within* a nested module. (See 3.1.4.)

yield: Return control to the SystemC scheduler. For a thread process, to yield is to call **wait**. For a method process, to yield is to return from the function.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Annex B

(informative)

Introduction to SystemC

This annex is informative and is intended to aid the reader in the understanding of the structure and intent of the SystemC class library.

The SystemC class library supports the functional modeling of systems by providing classes to represent the following:

- The hierarchical decomposition of a system into modules
- The structural connectivity between those modules using ports and exports
- The scheduling and synchronization of concurrent processes using events and sensitivity
- The passing of simulated time
- The separation of computation (processes) from communication (channels)
- The independent refinement of computation and communication using interfaces
- Hardware-oriented data types for modeling digital logic and fixed-point arithmetic

Loosely speaking, SystemC allows a user to write a set of C++ functions (processes) that are executed under control of a scheduler in an order that mimics the passage of simulated time and that are synchronized and communicate in a way that is useful for modeling electronic systems containing hardware and embedded software. The processes are encapsulated in a module hierarchy that captures the structural relationships and connectivity of the system. Inter-process communication uses a mechanism, the interface method call, that facilities the abstraction and independent refinement of system-level interfaces.

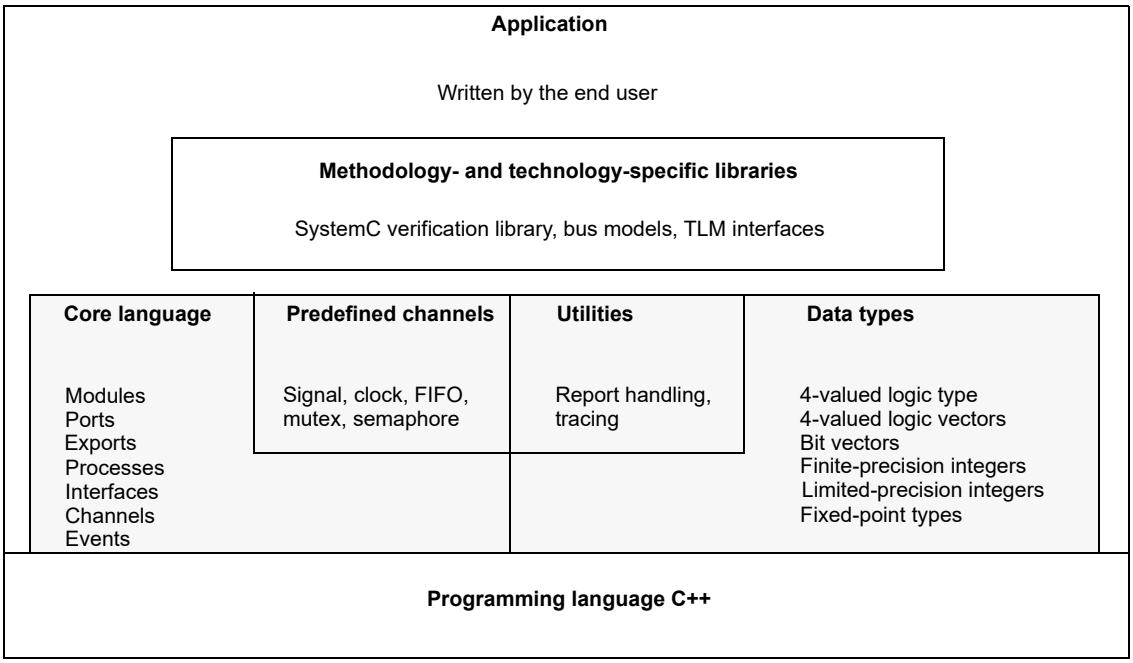

Figure B-1—SystemC language architecture

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

The architecture of a SystemC application is shown in Figure B-1. The shaded blocks represent the SystemC class library itself. The layer shown immediately above the SystemC class library represents standard or proprietary C++ libraries associated with specific design or verification methodologies or specific communication channels and is outside the scope of this standard.

The classes of the SystemC library fall into four categories: the core language, the SystemC data types, the predefined channels, and the utilities. The core language and the data types may be used independently of one another, although they are more typically used together.

At the core of SystemC is a simulation engine containing a process scheduler. Processes are executed in response to the notification of events. Events are notified at specific points in simulated time. In the case of time-ordered events, the scheduler is deterministic. In the case of events occurring at the same point in simulation time, the scheduler is non-deterministic. The scheduler is non-preemptive (see 4.3.2).

The *module* is the basic structural building block. Systems are represented by a module hierarchy consisting of a set of modules related by instantiation. A module can contain the following:

- Ports (see 5.12)
- Exports (see 5.13)
- Channels (see 5.2 and 5.15)
- Processes (see 5.2.10 and 5.2.11)
- Events (see 5.10)
- Instances of other modules (see 4.2.2)
- Other data members
- Other member functions

Modules, ports, exports, channels, interfaces, events, and times are implemented as C++ classes.

The execution of a SystemC application consists of *elaboration*, during which the module hierarchy is created, followed by *simulation*, during which the scheduler runs. Both elaboration and simulation involve the execution of code both from the application and from the *kernel*. The kernel is the part of a SystemC class library implementation that provides the core functionality for elaboration and the scheduler.

Instances of ports, exports, channels, and modules can only be created during elaboration. Once created during elaboration, this hierarchical structure remains fixed for the remainder of elaboration and simulation (see Clause 4). Process instances can be created statically during elaboration (see 5.2.8) or dynamically during simulation (see 5.5). Modules, channels, ports, exports, and processes are derived from a common base class **sc_object**, which provides methods for traversing the module hierarchy. Arbitrary attributes (name-value pairs) can be attached to instances of **sc_object** (see 5.16).

Instances of ports, exports, channels, and modules can only be created within modules. The only exception to this rule is top-level modules.

Processes are used to perform computations and hence to model the functionality of a system. Although notionally concurrent, processes are actually scheduled to execute in sequence. Processes are C++ functions registered with the kernel during elaboration (static processes) or during simulation (dynamic processes), and called from the kernel during simulation.

The *sensitivity* of a process identifies the set of events that would cause the scheduler to execute that process should those events be notified. Both *static* and *dynamic* sensitivity are provided. Static sensitivity is created at the time the process instance is created, whereas dynamic sensitivity is created during the execution of the function associated with the process during simulation. A process may be sensitive to named events or to events buried within channels or behind ports and located using an *event finder*. Furthermore, dynamic

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

sensitivity may be created with a *time-out*, meaning that the scheduler executes the process after a given time interval has elapsed (see 4.3.2 and 5.2.14 through 5.2.18).

Channels serve to encapsulate the mechanisms through which processes communicate and hence to model the communication aspects or protocols of a system. Channels can be used for inter-module communication or for inter-process communication within a module.

Interfaces provide a means of accessing channels. An interface proper is an abstract class that declares a set of pure virtual functions (interface methods). A channel is said to *implement* an interface if it defines all of the methods (that is, member functions) declared in that interface. The purpose of interfaces is to exploit the object-oriented type system of C++ in order that channels can be refined independently from the modules that use them. Specifically, any channel that implements a particular interface can be interchanged with any other such channel in a context that names that interface type.

The methods defined within a channel are typically called through an interface. A channel may implement more than one interface, and a single interface may be implemented by more than one channel.

Interface methods implemented in channels may create dynamic sensitivity to events contained within those same channels. This is a typical coding idiom and results in a so-called blocking method in which the process calling the method is suspended until the given event occurs. Such methods can only be called from certain kinds of processes known as thread processes (see 5.2.10 and 5.2.11).

Because processes and channels may be encapsulated within modules, communication between processes (through channels) may cross boundaries within the module hierarchy. Such boundary crossing is mediated by ports and exports, which serve to forward method calls from the processes within a module to channels to which those ports or exports are bound. A port specifies that a particular interface is required by a module, whereas an export specifies that a particular interface is provided by a module. Ports allow interface method calls within a module to be independent of the context in which the module is instantiated in the sense that the module need have no explicit knowledge of the identity of the channels to which its ports are bound. Exports allow a single module to provide multiple instances of the same interface.

Ports belonging to specific module instances are bound to channel instances during elaboration. The port binding policy can be set to control whether a port need be bound, but the binding cannot be changed subsequently. Exports are bound to channel instances that lie within or below the module containing the export. Hence, each interface method call made through a port or export is directed to a specific channel instance in the elaborated module hierarchy—the channel instance to which that port is bound.

Ports can only forward method calls *up* or *out* of a module, whereas exports can only forward method calls *down* or *into* a module. Such method calls always originate from processes within a module and are directed to channels instantiated elsewhere in the module hierarchy.

Ports and exports are instances of a templated class that is parameterized with an interface type. The port or export can only be bound to a channel that implements that particular interface or one derived from it (see 5.12 through 5.14).

There are two categories of channel: hierarchical channels and primitive channels. A hierarchical channel is a module. A primitive channel is derived from a specific base class (**sc_prim_channel**) and is not a module. Hence, a hierarchical channel can contain processes and instances of modules, ports, and other channels, whereas a primitive channel can contain none of these. It is also possible to define channels derived from neither of these base classes, but every channel implements one or more interfaces.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

A primitive channel provides unique access to the update phase of the scheduler, enabling the very efficient implementation of certain communication schemes. This standard includes a set of predefined channels, together with associated interfaces and ports, as follows:

sc_signal (see 6.4)

sc_buffer (see 6.6)

sc_clock (see 6.7)

sc_signal_resolved (see 6.13)

sc_signal_rv (see 6.17)

sc_fifo (see 6.23)

sc_mutex (see 6.27)

sc_semaphore (see 6.29)

sc_event_queue (see 6.29)

Class **sc_signal** provides the semantics for creating register transfer level or pin-accurate models of digital hardware. Class **sc_fifo** provides the semantics for point-to-point FIFO-based communication appropriate for models based on networks of communicating processes. Classes **sc_mutex** and **sc_semaphore** provide communication primitives appropriate for software modeling.

This standard includes a set of data types for modeling digital logic and fixed-point arithmetic, as follows:

sc_int<> (see 7.5.4)

sc_uint<> (see 7.5.5)

sc_bigint<> (see 7.6.5)

sc_biguint<> (see 7.6.6)

sc_logic (see 7.9.2)

sc_lv<> (see 7.9.6)

sc_bv<> (see 7.9.6)

sc_fixed<> (see 7.10.19)

sc_ufixed<> (see 7.10.20)

Classes **sc_int** and **sc_uint** provide signed and unsigned limited-precision integers with a word length limited by the C++ implementation. Classes **sc_bigint** and **sc_biguint** provide finite-precision integers. Class **sc_logic** provides four-valued logic. Classes **sc_bv** and **sc_lv** provide two- and four-valued logic vectors. Classes **sc_fixed** and **sc_ufixed** provide signed and unsigned fixed-point arithmetic.

The classes **sc_report** and **sc_report_handler** provide a general mechanism for error handling that is used by the SystemC class library itself and is also available to the user. Reports can be categorized by severity and by message type, and customized actions can be set for each category of report, such as writing a message, throwing an exception, or aborting the program (see 8.2 and 8.3).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Annex C

(informative)

Deprecated features

This annex contains a list of deprecated features. A *deprecated* feature is a feature that was present in version 2.0.1 of the OSCI open source proof-of-concept SystemC implementation but is not part of this standard. Deprecated features may or may not remain in the Accellera Systems Initiative implementation in the future. The user is strongly discouraged from using deprecated features because an implementation is not obliged to support such features. An implementation may issue a warning on the first occurrence of each deprecated feature but is not obliged to do so.

a) Functions **sc_cycle** and **sc_initialize** (Use **sc_start** instead)

b) Class **sc_simcontext** (Replaced by functions **sc_delta_count**, **sc_is_running**, **sc_get_top_level_objects**, and **sc_find_object**, and by member functions **get_child_objects** and **get_parent_object**)

c) Type **sc_process_b** (Replaced by class **sc_process_handle**)

d) Function **sc_get_curr_process_handle** (Replaced by function **sc_get_current_process_handle**)

e) Member function **notify_delayed** of class **sc_event** (Use **notify(SC_ZERO_TIME)** instead)

f) Non-member function **notify** (Use member function **notify** of class **sc_event** instead)

g) Member function **timed_out** of classes **sc_module** and **sc_prim_channel**

h) **operator**, and **operator<<** of class **sc_module** for positional port binding (Use **operator()** instead)

i) **operator()** of class **sc_module** for positional port binding when called more than once per module instance (Use named port binding instead)

j) Constructors of class **sc_port** that bind the port at the time of construction of the port object

k) **operator()** of class **sc_sensitive** (Use **operator<<** instead)

l) Classes **sc_sensitive_pos** and **sc_sensitive_neg** and the corresponding data members of class **sc_module** (Use the event finders pos and neg instead)

m) Member function **end_module** of class **sc_module**

n) Default time units and all the associated functions and constructors, including:

   1) Function **sc_simulation_time**
   2) Function **sc_set_default_time_unit**
   3) Function **sc_get_default_time_unit**
   4) Function **sc_start(double)**
   5) Constructor **sc_clock(const char*, double, double, double, bool)**

o) Member function **trace** of classes **sc_object**, **sc_signal**, **sc_clock**, and **sc_fifo** (Use **sc_trace** instead)

p) Member function **add_trace** of classes **sc_in** and **sc_inout** (Use **sc_trace** instead)

q) Member function **get_data_ref** of classes **sc_signal** and **sc_clock** (Use member function **read** instead)

r) Member function **get_new_value** of class **sc_signal**

s) Typedefs **sc_inout_clk** and **sc_out_clk** (Use **sc_out<bool>** instead)

t) Typedef **sc_signal_out_if**

u) Constant SC_DEFAULT_STACK_SIZE (Function **set_stack_size** is not deprecated)

v) Constant SC_MAX_NUM_DELTA_CYCLES

w) Constant SYSTEMC_VERSION (Function **sc_version** is not deprecated)

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

x) Support for the **wif** and **isdb** trace file formats (The **vcd** trace file format is not deprecated)

y) Member function **sc_set_vcd_time_unit** of class **vcd_trace_file**

z) Function **sc_trace_delta_cycles**

aa) Function **sc_trace** for writing enumeration literals to the trace file (Other **sc_trace** functions are not deprecated)

ab) Type **sc_bit** (Use type **bool** instead)

ac) Macro SC_CTHREAD, except for the case where the second argument is an event finder, which is still supported

ad) Global and local watching for clocked threads (Use function **reset_signal_is** instead)

ae) The reporting mechanism based on integer ids and the corresponding member functions of class **sc_report**, namely, **register_id**, **get_message**, **is_suppressed**, **suppress_id**, **suppress_infos**, **suppress_warnings**, **make_warnings_errors**, and **get_id**. (Replaced by a reporting mechanism using string message types)

af) Utility classes **sc_string**, **sc_pvector**, **sc_plist**, **sc_phash**, and **sc_ppq**

ag) Macro DECLARE_EXTENDED_PHASE (From TLM-2.0.1. Use TLM_DECLARE_EXTENDED_PHASE instead)

ah) Macro SC_HAS_PROCESS

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

Annex D

(informative)

Changes between IEEE Std 1666-2011 and IEEE Std 1666-2023

This annex lists the more significant changes between the 2011 and 2023 versions of this SystemC standard.

1) Requirement that implementations and tools conforming to this standard use at least the C++17 programming language (i.e., ISO/IEC 14882:2017).
2) Restriction that an application shall not pass a null pointer as argument to functions that use const char* as parameter.
3) Restriction on sc_core::sc_start(float) that it can no longer be called with a negative value.
4) The definitions of sc_core::sc_argc and sc_core::sc_argv have been refined to make them consistent with the use of argc and argv in a C++ main function.
5) New functions to suspend and unsuspend the simulation kernel:

   sc_core::sc_suspend_all
   sc_core::sc_unsuspend_all
   sc_core::sc_unsuspendable
   sc_core::sc_suspendable
   sc_core::sc_prim_channel::async_attach_suspending
   sc_core::sc_prim_channel::async_detach_suspending
6) New class sc_core::sc_stage_callback_if provides an interface to enable user-defined callbacks during elaboration or simulation stages that are otherwise not available to the application. New functions:

   sc_core::sc_stage_callback_if::stage_callback
   sc_core::sc_register_stage_callback
   sc_core::sc_unregister_stage_callback
7) New function sc_core::sc_delta_count_at_current_time returns the number of delta cycles for the current time.
8) The function sc_core::sc_get_status is required to be thread-safe to enable the creation of multi-threaded implementations and/or applications. This function now returns sc_core::SC_SUSPENDED when the simulation kernel is suspended.
9) The constructor macro SC_CTOR now supports additional constructor parameters.
10) Macro SC_HAS_PROCESS is not required anymore and has been deprecated.
11) New macro SC_NAMED offers a convenient approach to declare a variable with a matching name.
12) For daggered class sc_core::sc_sensitive, the operator<< now supports passing a collection of elements to be added to the static sensitivity of the process instance. Typical collections are elements in an sc_core::sc_vector or C arrays of ports, exports, channels, or events.
13) Class sc_core::sc_process_handle now defines the member function basename that returns the string name of the underlying process instance.
14) Class sc_core::sc_event introduces the member function triggered to determine whether an event has recently been triggered. In addition, a static member none is provided in case a reference to an event is required that is never notified.
15) The enumeration type sc_core::sc_time_unit now supports attosecond (sc_core::SC_AS), zeptosecond (sc_core::SC_ZS), and yoctosecond (sc_core::SC_YS).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

16) Class sc_core::sc_time supports a constructor passing the time as string argument. Modulo operator% has been added. Improved definition when sc_core::sc_time values are below sc_core::sc_get_time_resolution and sc_core::sc_time values are above sc_core::sc_max_time. Other new functions:

    sc_core::sc_time::from_value
    sc_core::sc_time::from_seconds
    sc_core::sc_time::from_string
    sc_core::sc_time::to_string

17) Added definition that the time resolution can be changed after calling sc_core::sc_max_time. However, sc_core::sc_max_time cannot be changed after calling sc_core::sc_time::to_double, sc_core::sc_time::to_seconds, sc_core::sc_time::to_string, sc_core::sc_time::print, or operator<< on any sc_core::sc_time object, including objects created by sc_core::sc_max_time.

18) New functions for class sc_core::sc_port_base and sc_core::sc_export_base to get access to the interface or the interface type:

    sc_core::sc_port_base::get_interface
    sc_core::sc_port_base::get_interface_type
    sc_core::sc_export_base::get_interface
    sc_core::sc_export_base::get_interface_type

19) Removed the const qualifier from a function return type if the function returns by value.

20) sc_object has now a virtual destructor.

21) Class sc_core::sc_object defines new member functions to get access the hierarchy context. New functions:

    sc_core::sc_object::get_current_sc_object
    sc_core::sc_object::get_hierarchy_scope

22) New class sc_core::sc_hierarchy_scope provides an interface to allow an application to place objects of type sc_core::sc_object in an object hierarchy outside the current hierarchical scope. New functions:

    sc_core::sc_hierarchy_scope::get_root

23) Functions to register and un-register hierarchical names:

    sc_core::sc_register_hierarchical_name
    sc_core::sc_unregister_hierarchical_name

24) Definition of the constant SC_DEFAULT_WRITER_POLICY. By default, the value of SC_DEFAULT_WRITER_POLICY shall be equal to SC_ONE_WRITER for backward compatibility with previous versions of this standard. The default writer policy for class sc_core::sc_signal, sc_core::sc_signal_resolved, and sc_core::sc_buffer is set to SC_DEFAULT_WRITER_POLICY. An implementation may define another value of SC_DEFAULT_WRITER_POLICY.

25) New constructor for class sc_core::sc_signal, sc_core::sc_signal_resolved, and sc_core::sc_buffer to set an initial value to the channel, without triggering an event and a subsequent delta cycle.

26) Fixed interface class inheritance for sc_signal.

27) New static object sc_core::sc_unbound can be used in an application to make an unbound (open) connection to a port. In addition, the function sc_core::sc_tie::value(const &T) can be used to tie a port to a specified value of type T.

28) New functions and operators for class sc_dt::sc_bitref_r improve support for Boolean conversions:

    sc_dt::sc_bitref_r::operator bool
    sc_dt::sc_bitref_r::operator!

29) Restriction on the bit order in the part-selects (which cannot be reversed).

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

30) New daggered classes sc_dt::sc_fxnum_bitref_r, sc_dt::sc_fxnum_subref_r, sc_dt::sc_fxnum_fast_bitref_r, and sc_dt::sc_fxnum_fast_subref_r represent a read-only bit selected from an sc_dt::sc_fxnum or sc_dt::sc_fxnum_fast, respectively.

31) New definition supports tracing of types sc_core::sc_event and sc_core::sc_time using sc_core::sc_trace.

32) New function get_handler returns the current report handler.

33) sc_report_handler::get_log_file_name() no longer returns a null pointer.

34) SC_DEFAULT_INFO_ACTIONS, SC_DEFAULT_WARNING_ACTIONS, SC_DEFAULT_ERROR_ACTIONS, and SC_DEFAULT_FATAL_ACTIONS are now defined as enum values in the namespace sc_core.

35) sc_assert is now explicitly calling an implementation-defined function sc_core::sc_assertion_failed, which holds the [[noreturn]] attribute.

36) New functions sc_core::sc_vector::emplace_back and sc_core::sc_vector::emplace_back_with_name support making incremental additions to sc_core::sc_vector. Additions are allowed only if the sc_core::sc_vector is not locked. The policy for locking sc_core::sc_vector is defined during vector initialization, by specifying the sc_core::sc_vector_init_policy, which supports two modes: locking after initialization (SC_VECTOR_LOCK_AFTER_INIT) or locking after elaboration (SC_VECTOR_LOCK_AFTER_ELABORATION). Once the sc_vector is locked, it cannot be unlocked.

37) New template-free base class tlm::tlm_base_socket_if declares pure virtual functions that should be overridden in any derived socket class.

38) Constraints added to functions tlm::tlm_generic_payload::set_extension and tlm::tlm_generic_payload::set_auto_extension disallow attaching multiple extensions of the same type to the same generic payload.

39) Clarification on the request/response exclusion rule for TLM-2 multi-sockets.

40) Fixed missing arrow in nb_transport call sequence diagram (Figure 29).

41) TLM-1 tlm::tlm_fifo_get_if C++ disambiguation.

42) The exact numerical value for all enum definitions has been removed and is now implementation-defined.

43) General update of language/words to be more inclusive.

Copyright © 2023 IEEE. All rights reserved.
Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.

IEEE SA
STANDARDS ASSOCIATION

RAISING THE WORLD'S STANDARDS

Connect with us on:

Twitter: twitter.com/ieeesa
Facebook: facebook.com/ieeesa
LinkedIn: linkedin.com/groups/1791118
Beyond Standards blog: beyondstandards.ieee.org
YouTube: youtube.com/ieeesa

standards.ieee.org
Phone: +1 732 981 0060

IEEE

Authorized licensed use limited to: david james. Downloaded on May 31,2026 at 01:27:46 UTC from IEEE Xplore. Restrictions apply.