IEC 61691-7

Edition 1.0 2009-12

INTERNATIONAL STANDARD

IEEE Std 1666™

Behavioural languages –
Part 7: SystemC® Language Reference Manual

IEC 61691-7:2009(E) IEEE Std 1666-2005(E)

IEC Central Office
3, rue de Varembé
CH-1211 Geneva 20
Switzerland
Email: inmail@iec.ch
Web: www.iec.ch

Institute of Electrical and Electronics Engineers, Inc
3 Park Avenue
New York, NY10016-5997
USA
Email: stds.ipr@ieee.org
Web: www.ieee.org

About the IEC

The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications

The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigenda or an amendment might have been published.

- Catalogue of IEC publications: www.iec.ch/searchpub

The IEC on-line Catalogue enables you to search by a variety of criteria (reference number, text, technical committee,…). It also gives information on projects, withdrawn and replaced publications.

- IEC Just Published: www.iec.ch/online_news/justpub

Stay up to date on all new IEC publications. Just Published details twice a month all new publications released. Available on-line and also by email.

- Electropedia: www.electropedia.org

The world's leading online dictionary of electronic and electrical terms containing more than 20 000 terms and definitions in English and French, with equivalent terms in additional languages. Also known as the International Electrotechnical Vocabulary online.

- Customer Service Centre: www.iec.ch/webstore/custserv

If you wish to give us your feedback on this publication or need further assistance, please visit the Customer Service Centre FAQ or contact us:

Email: csc@iec.ch
Tel.: +41 22 919 02 11

IEC 61691-7

Edition 1.0 2009-12

INTERNATIONAL STANDARD

IEEE Std 1666™

Behavioural languages –
Part 7: SystemC® Language Reference Manual

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

PRICE CODE XH

ICS 25.040, 35.060

ISBN 978-0-7381-6284-3

CONTENTS

INTERNATIONAL ELECTROTECHNICAL COMMISSION

____

BEHAVIOURAL LANGUAGES –

Part 7: SystemC® Language Reference Manual

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
5) IEC itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.
6) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC 61691-7/IEEE Std 1666 has been processed through IEC technical committee 93: *Design automation*.

The text of this standard is based on the following documents:

IEEE Std	FDIS	Report on voting
1666 (2005)	93/279/FDIS	93/285/RVD

Full information on the voting for the approval of this standard can be found in the report on voting indicated in the above table.

A list of parts of the IEC 61691 series can be found on the IEC web site.

The committee has decided that the contents of this publication will remain unchanged until the maintenance result date indicated on the IEC web site under "http://webstore.iec.ch" in the data related to the specific publication. At this date, the publication will be

- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

IEC/IEEE Dual Logo International Standards

This Dual Logo International Standard is the result of an agreement between the IEC and the Institute of Electrical and Electronics Engineers, Inc. (IEEE). The original IEEE Standard was submitted to the IEC for consideration under the agreement, and the resulting IEC/IEEE Dual Logo International Standard has been published in accordance with the ISO/IEC Directives.

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

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

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

The existence of an IEC/IEEE Dual Logo International 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 IEC/IEEE Dual Logo International Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard.

Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

In publishing and making this document available, the IEC and IEEE are not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Neither the IEC nor IEEE is undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEC/IEEE Dual Logo International Standards or IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations – Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

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

Secretary, IEEE-SA Standards Board, 445 Hoes Lane, Piscataway, NJ 08854, USA and/or General Secretary, IEC, 3, rue de Varembé, PO Box 131, 1211 Geneva 20, Switzerland.

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

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

IEEE Standard SystemC® Language Reference Manual

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 28 March 2006

American National Standards Institute

Approved 6 December 2005

IEEE-SA Standards Board

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

Abstract: SystemC®[1] is defined in this standard. SystemC is an ANSI standard C++ class library for system and hardware design for use by designers and architects who need to address complex systems that are a hybrid between hardware and software. This standard provides a precise and complete definition of the SystemC class library so that a SystemC implementation can be developed with reference to this standard alone. The primary audiences for this standard are the implementors of the SystemC class library, the implementors of tools supporting the class library, and users of the class library.

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

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

IEEE introduction

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 modelinghardware and software together at multiple levels of abstraction. This capability is not available in traditional hardware description languages.

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

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

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

Notice to users

Errata

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

Interpretations

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

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE 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: This standard is not intended to ensure safety, security, health, or environmental protection in all circumstances. Implementers of the standard are responsible for determining appropriate safety, security, environmental, and health practices or regulatory requirements.

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

BEHAVIOURAL LANGUAGES –
Part 7: SystemC® Language Reference Manual

1. Overview

1.1 Scope

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

1.2 Purpose

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

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

1.3 Subsets

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

1.4 Relationship with C++

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

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

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

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

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

1.5 Guidance for readers

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

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

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

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

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

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

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

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

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

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

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

2. References

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

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

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

IEC 61691-4:2004, Behavioural languages - Part 4: Verilog® hardware description language ¦ IEEE Std 1364™-2001, IEEE Standard Verilog® Hardware Description Language.[4, 5, 6]

[3] IEC publications are available from the Sales Department of The International Electrotechnical Commission, Case Postale 131, 3, rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iec.ch). ISO publications are available from the ISO Central Secretariat, 1 chemin de la Voie-Creuse, CP 56, CH-1211, Genève 20, Switzerland/Suisse (http://www.iso.ch). ISO/IEC publications are also available in the United States from the Sales Department, American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (http://www.ansi.org/).

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

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

[6] IEEE Std 1364-2001 was adopted as IEC 61691-4:2004

3. Terminology and conventions used in this standard

3.1 Terminology

3.1.1 Shall, should, may, can

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

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

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

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

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

3.1.2 Implementation, application

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

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

3.1.3 Call, called from, derived from

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

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

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

3.1.4 Specific technical terms

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2 Syntactical conventions

3.2.1 Implementation-defined

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

3.2.2 Disabled

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

3.2.3 Ellipsis (...)

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

3.2.4 Class names

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

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

The class names are the following:

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

3.2.5 Embolded text

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

3.3 Semantic conventions

3.3.1 Class definitions and the inheritance hierarchy

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

3.3.2 Function definitions and side-effects

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

3.3.3 Functions whose return type is a reference or a pointer

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

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

3.3.3.1 Functions that return *this or an actual argument

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

3.3.3.2 Functions that return char*

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

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

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

a) sc_interface * // Returns a channel

b) sc_event& // Returns an event

c) sc_event_finder& // Returns an event finder

d) sc_time& // Returns a property of primitive channel sc_clock

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

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

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

a) sc_object *

b) sc_attr_base *

c) std::string& // Property of an attribute object

The functions concerned are the following:

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

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

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

a) std::vector< sc_object * > &

b) sc_attr_cltn *

The functions concerned are the following:

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

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

3.3.3.5 Functions sc_time_stamp and sc_signal::read

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

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

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

3.3.4 Namespaces and internal naming

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

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

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

3.3.5 Non-compliant applications and errors

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

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

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

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

3.4 Notes and examples

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

4. Elaboration and simulation semantics

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

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

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

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

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

4.1 Elaboration

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

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

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

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

4.1.1 Instantiation

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

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

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

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

The instantiation of a module also implies the construction of objects of class 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 have some member functions in common (see 5.15).

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

Example:

```
#include "systemc.h"

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

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

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

    SC_CTOR(Top)
    :   m1("m1"),                   // m1.name() returns "top.m1"
        s1("s1")                    // s1.m.name() returns "top.s1"
    {
        m2 = new Mod("m2");         // m2->name() returns "top.m2"
        f();
        S *s2 = new S("s2");        // s2->m.name() returns "top.s2"
    }
    void f() {
        Mod *m3 = new Mod("m3"); // Unusual coding style - not recommended
    }                                   // m3->name() returns "top.m3"
};

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

4.1.2 Process macros

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

SC_METHOD

SC_THREAD

SC_CTHREAD

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

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

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

4.1.3 Port binding and export binding

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

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

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

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

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

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

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

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

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

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

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

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

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

4.1.4 Setting the time resolution

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

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

4.2 Simulation

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

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

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

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

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

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

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

4.2.1 The scheduling algorithm

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

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

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

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

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

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

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

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

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

The scheduler starts by executing the initialization phase.

4.2.1.1 Initialization phase

Perform the following three steps in the order given:

a) Run the update phase as defined in 4.2.1.3 but without continuing to the delta notification phase.

b) Add every method and thread process instance in the object hierarchy to the set of runnable processes, but exclude those process instances for which the function **dont_initialize** has been called, and exclude clocked thread processes.

c) Run the delta notification phase, as defined in 4.2.1.4. At the end of the delta notification phase, go to the evaluation phase.

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

4.2.1.2 Evaluation phase

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

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

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

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

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

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

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

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

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

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

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

4.2.1.3 Update phase

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

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

4.2.1.4 Delta notification phase

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

a) Determine which process instances are sensitive to these events or time-outs.

b) Add all such process instances to the set of runnable processes.

c) Remove all such notifications and time-outs from the set of delta notifications and time-outs.

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

4.2.1.5 Timed notification phase

If pending timed notifications or time-outs exist:

a) Advance simulation time to the time of the earliest pending timed notification or time-out.

b) Determine which process instances are sensitive to the events notified and time-outs lapsing at this precise time.

c) Add all such process instances to the set of runnable processes.

d) Remove all such notifications and time-outs from the set of timed notifications and time-outs.

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

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

4.2.2 Cycles in the scheduling algorithm

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

a) An evaluation phase

b) An update phase

c) A delta notification phase

The initialization phase does not include a delta cycle.

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

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

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

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

4.3 Running elaboration and simulation

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

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

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

4.3.1 Function declarations

```
namespace sc_core {

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

4.3.2 Function sc_elab_and_sim

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

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

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

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

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

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

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

4.3.3 Functions sc_argc and sc_argv

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

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

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

4.3.4 Running under application control using functions sc_main and sc_start

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

4.3.4.1 Function sc_main

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

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

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

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

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

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

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

4.3.4.2 Function sc_start

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

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

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

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

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

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

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

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

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

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

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

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

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

4.3.5 Running under control of the kernel

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

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

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

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

4.4 Elaboration and simulation callbacks

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

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

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

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

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

The implementation shall provide the following two functions:

```
namespace sc_core {

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

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

4.4.1 before_end_of_elaboration

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

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

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

a) The instantiation of objects of class **sc_module**, **sc_port**, **sc_export**, **sc_prim_channel**

b) The instantiation of objects of other classes derived from class **sc_object**

c) Port binding

d) Export binding

e) The macros SC_METHOD, SC_THREAD, SC_CTHREAD, SC_HAS_PROCESS

f) The member **sensitive** and member functions **dont_initialize**, **set_stack_size**, and **reset_signal_is** of the class **sc_module**

g) Calls to event finder functions

h) Calls to function **sc_spawn** to create static spawned processes

i) Calls to member function **request_update** of class **sc_prim_channel** to create update requests (for example by calling member function **initialize** of class **sc_inout**)

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

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

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

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

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

4.4.2 end_of_elaboration

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

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

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

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

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

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

e) The member function **reset_signal_is** of the class **sc_module**

f) Calls to event finder functions

g) Calls to member function **notify** of the class **sc_event**

4.4.3 start_of_simulation

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

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

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

a) The instantiation of objects of classes derived from class **sc_object** but excluding classes **sc_module**, **sc_port**, **sc_export**, and **sc_prim_channel**

b) Calls to function **sc_spawn** to create dynamic spawned processes

c) Calls to member function **request_update** of class **sc_prim_channel** to create update requests (for example by calling member function **write** of class **sc_inout**)

d) Interface method calls using **operator->** and **operator[]** of class **sc_port**, provided that those calls do not attempt to perform actions prohibited outside simulation such as event notification

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

a) The instantiation of objects of class **sc_module**, **sc_port**, **sc_export**, **sc_prim_channel**

b) Port binding

c) Export binding

d) The macros SC_CTOR, SC_METHOD, SC_THREAD, SC_CTHREAD, SC_HAS_PROCESS

e) The member **sensitive** and member functions **dont_initialize**, **set_stack_size**, and **reset_signal_is** of the class **sc_module**

f) Calls to event finder functions

g) Calls to member function **notify** of the class **sc_event**

4.4.4 end_of_simulation

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

The purpose of member function **end_of_simulation** is to allow an application to perform housekeeping actions at the end of simulation. Examples include closing stimulus and response files and printing diagnostic messages. The intention is that an implementation that initiates elaboration and simulation under direct control of the kernel (in the absence of functions **sc_main** and **sc_start**) shall make the callbacks to **end_of_simulation** at the very end of simulation whether or not function **sc_stop** has been called.

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

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

a) The instantiation of objects of classes derived from class **sc_object**

b) Calls to function **sc_spawn** to create dynamic spawned processes

c) Calls to member function **request_update** of class **sc_prim_channel** to create update requests (for example by calling member function **write** of class **sc_inout**)

d) Calls to member function **notify** of the class **sc_event**

4.5 Other functions related to the scheduler

4.5.1 Function declarations

```
namespace sc_core {

    enum sc_stop_mode
    {
        SC_STOP_FINISH_DELTA ,
        SC_STOP_IMMEDIATE
    };

    extern void sc_set_stop_mode( sc_stop_mode mode );
    extern sc_stop_mode sc_get_stop_mode();
    void sc_stop();

    const sc_time& sc_time_stamp();
    const sc_dt::uint64 sc_delta_count();
    bool sc_is_running();
}
```

4.5.2 Function sc_stop, sc_set_stop_mode, and sc_get_stop_mode

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

```
    enum sc_stop_mode
    {
        SC_STOP_FINISH_DELTA ,
        SC_STOP_IMMEDIATE
    };

    extern void sc_set_stop_mode( sc_stop_mode mode );
    extern sc_stop_mode sc_get_stop_mode();
```

```
void sc_stop();
```

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

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

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

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

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

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

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

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

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

4.5.3 Function sc_time_stamp

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

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

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

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

4.5.4 Function sc_delta_count

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

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

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

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

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

4.5.5 Function sc_is_running

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

```
bool sc_is_running();
```

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

5. Core language class definitions

5.1 Class header files

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

5.1.1 #include "systemc"

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

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

Example:

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

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

5.1.2 #include "systemc.h"

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

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

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

```
    #include "systemc"

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

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

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

5.2 sc_module

5.2.1 Description

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

5.2.2 Class definition

```
namespace sc_core {

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


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

        virtual const char* kind() const;

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

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

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

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

        sc_sensitive† sensitive;

        void dont_initialize();
        void set_stack_size( size_t );

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

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

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

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

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

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

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

const char* sc_gen_unique_name( const char* );

typedef sc_module sc_behavior;
typedef sc_module sc_channel;

}        // namespace sc_core
```

5.2.3 Constraints on usage

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

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

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

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

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

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

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

5.2.4 kind

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

5.2.5 SC_MODULE

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

Example:

```
// The following two class definitions are equally acceptable.

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

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

5.2.6 Constructors

sc_module(const sc_module_name&);
sc_module();

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

5.2.7 SC_CTOR

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

Example:

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

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

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

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

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

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

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

Example:

```
SC_MODULE(M)
{
    M(sc_module_name n, int a, int b)        // Additional constructor parameters
    : sc_module(n)
    {}
    ...
};
```

5.2.8 SC_HAS_PROCESS

Macro SC_CTOR includes definitions used by the macros SC_METHOD, SC_THREAD and SC_CTHREAD. These same definitions are introduced by the macro SC_HAS_PROCESS. If a process macro is invoked from the constructor body of a module but macro SC_CTOR is not used within the module class definition, macro SC_HAS_PROCESS shall be invoked within the class definition or the constructor body of the module. If a process macro is invoked from the **before_end_of_elaboration** or **end_of_elaboration** callbacks of a module but macro SC_CTOR is not used within the module class definition, macro SC_HAS_PROCESS shall be invoked within the class definition of the module or from that same callback.

Macro SC_HAS_PROCESS shall only be used within the class definition, constructor body, or member function body of a module. The name of the module class being constructed shall be passed as the argument to the macro.

NOTE—The use of the macros SC_CTOR and SC_HAS_PROCESS is not required in order to call the function **sc_spawn**.

Example:

```
SC_MODULE(M)
{
    M(sc_module_name n)              // SC_CTOR not used
    : sc_module(n)
    {
        SC_THREAD(T);                // Process macro
    }
    void T();

    SC_HAS_PROCESS(M);               // Necessary
    ...
};
```

5.2.9 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 4.1.2).

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

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 being called by the kernel.

Example:

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

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

5.2.10 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 cannot be *terminated*.

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.

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.

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 clocked thread 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.

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*.

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 prevent 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 pre-emptive, 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.

Example:

```
SC_MODULE(synchronous_module)
{
    sc_in<bool> 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 and reset_signal_is

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**.

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

Member function **reset_signal_is** of class **sc_module** shall determine the reset signal of a clocked thread process.

reset_signal_is 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 clocked thread 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** with a particular 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 **reset_signal_is** being called, the effect of calling **reset_signal_is** shall be undefined. It shall be an error to associate function **reset_signal_is** with a process instance that is not a clocked thread process.

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

A clocked thread process instance shall be reset when and only when the clock event to which the process instance is statically sensitive is notified and the reset signal is active. Resetting a clocked thread process instance shall consist of abandoning the current execution of the process instance, which shall have been suspended at a call to function **wait**, and calling the associated function again from the start of the function. A process instance being reset shall become runnable in the evaluation phase immediately following the delta notification phase or timed notification phase in which the clock event notification occurs. An active reset signal shall not cause the process to be reset in the absence of a clock event notification; in other words, the reset is synchronous with respect to the clock.

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_in<bool> clock;
sc_in<bool> reset;

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

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

5.2.13 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 5.2.16) or function **wait** (see 5.2.17).

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 a 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.

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.

5.2.14 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 prevent a particular unspawned process instance from being 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_signal<bool> A, B, C, D, E;

    SC_CTOR(Mod)
    {
        sensitive << A;          // Has no effect. Poor 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.
        dont_initialize();       // Method process M is not made runnable during initialization.
    }

    void f() { sensitive << D; }// An unusual coding style

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

5.2.15 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.

5.2.16 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**.

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.

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.

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

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

void **next_trigger**(*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 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.

void **next_trigger**(*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 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.

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.

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& , *sc_event_or_list*[†]&);
void **next_trigger**(double , sc_time_unit , *sc_event_or_list*[†]&);
void **next_trigger**(const sc_time& , const *sc_event_and_list*[†]&);

void **next_trigger**(double , sc_time_unit , *sc_event_and_list*[†]&);

> 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.

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_NS);   // Time-out after 1 nanosecond.
        else                next_trigger();           // Trigger on signal sig next time.
    }
    sc_signal<int> sig;
    sc_event e1, e2;
    ...
};
```

5.2.17 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.

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

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

void **wait**(*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 (see 5.8).

void **wait**(*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 (see 5.8).

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.

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

5.2.18 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 5.11).

```
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 5.11.3) 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_inout<int> P, Q, R; // Ports
    ...
};

SC_MODULE(Top1)
{
    sc_inout <int> A, B;
    sc_signal<int> 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_inout<int> S;
    sc_inout<int> *T;       // Pointer-to-port (an unusual coding style)
    sc_inout<int> U;
    SC_CTOR(M2) { T = new sc_inout<int>; }
    ...
};

SC_MODULE(Top2)
{
    sc_inout <int> D, E;
    sc_signal<int> 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
    }
    ...
};
```

5.2.19 before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation

See 4.4.

5.2.20 get_child_objects

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.

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.

Example:

```
int sc_main (int argc, char* argv[])
{
    Top_level_module top("top");

    std::vector<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_start();
    return 0;
}
```

5.2.21 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.

5.2.22 sc_behavior and sc_channel

typedef sc_module **sc_behavior**;
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_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_channel
{
    public:
        virtual void write(int addr, int  data);    // Interface methods implemented in channel
        virtual void read (int addr, int& data);

        sc_in<bool>  clock;                          // Ports
        sc_out<bool> wr, rd;
        sc_out<int>  addr_bus;
        sc_out<int>  data_out;
        sc_in <int>  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 {

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**, or **sc_prim_channel** 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.

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—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.

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 shall be 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_module
{
    A(sc_module_name) {}    // Calls sc_module()
};

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

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

struct Top: sc_module
{
    A a;
    C c;

    Top(sc_module_name n)
    : sc_module(n),         // Calls sc_module(sc_module_name&)
    a("a"),                 // Calls sc_module_name(char*) then calls A(sc_module_name)
    c("c") {}               // Calls sc_module_name(char*) then calls C(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& );

        // 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 5.2.13.

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.

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 5.11.3), 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—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.

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();
        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* );

    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

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 during 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 **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.

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**.

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**. 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 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 process handle to the spawned process instance.

NOTE 1—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.

NOTE 2—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.

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_module
{
    sc_signal<int> sig;
    void g();

    SC_CTOR(MyMod)
    {
        SC_THREAD(T);
    }
    int ret;
    void T()
    {
        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.
        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. The implementation shall make each call to **sc_spawn** immediately control enters the fork-join construct and shall spawn a separate process instance for each such call. In other words, 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 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, the commas, and SC_JOIN. If an application violates these rules, the effect shall be undefined.

Example:

```
SC_FORK
    sc_spawn( arguments ) ,
    sc_spawn( arguments ) ,
    sc_spawn( arguments )
SC_JOIN
```

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 shall not be associated with any underlying 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.

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_
};

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;

          const char* name() const;
          sc_curr_proc_kind proc_kind() const;
          const std::vector<sc_object*>& get_child_objects() 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;
};
sc_process_handle sc_get_current_process_handle();

}         // namespace sc_core
```

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 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 invalid 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 invalid 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 invalid 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.

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. The implementation is only obliged to keep the returned string valid 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 was spawned during the execution of the underlying process instance and any other application-defined objects derived from class **sc_object** created from the underlying process instance. 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.

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.

sc_object* **get_process_object**() const;

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 method process is never terminated, so member function **terminated** shall always return **false** for a method process. If the process handle is invalid, member function **terminated** shall return the value **false**.

When the underlying process instance terminates, an implementation may choose to invalidate any associated process handles but is not obliged to do so. In other words, when a process terminates, an implementation is neither obliged to keep the handle valid nor to invalidate the handle. If the process handle is valid, function **terminated** will return **true**, or if invalid, **terminated** will return **false**.

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. If member function **terminated_event** is called for a method process, the implementation shall generate a warning and the event shall never be notified. It shall be an error to call member function **terminated_event** for an invalid process handle and the event shall never be notified.

5.6.6 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. If the most recently created process instance was not within the current module, or if function **sc_get_current_process_handle** is called from one of the callbacks **before_end_of_elaboration** or **end_of_elaboration**, an implementation 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.

Example:

```
SC_MODULE(Mod)
{
    ...
    SC_CTOR(Mod)
    {
        SC_METHOD(Run);
        sensitive << in;
        sc_process_handle h1 = sc_get_current_process_handle();  // Returns a handle to process Run
    }
    void Run()
    {
        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
        }
    }
    ...
};
```

5.7 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:

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.h"

class if_class
: virtual public sc_interface
{
    public:
        virtual const sc_event& ev_func() const = 0;
        ...
};

class chan_class
: public if_class, public sc_prim_channel
{
    public:
        virtual const sc_event& ev_func() const { return an_event; }
        ...
    private:
        sc_event an_event;
};

template<int N = 1>
class port_class
: public sc_port<if_class,N>
{
    public:
        sc_event_finder& event_finder() const
        {
            return *new 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();
    ...

    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*† provide the **&** and **|** operators used to construct the event lists passed as arguments to the functions **wait** (see 5.2.16) and **next_trigger** (see 5.2.17).

5.8.2 Class definition

```
namespace sc_core {

class sc_event_and_list†
{
    public:
        sc_event_and_list†& operator& ( const sc_event& );

        // Other members
        implementation-defined
};

class sc_event_or_list†
{
    public:
        sc_event_or_list†& operator| ( const sc_event& );

        // Other members
        implementation-defined
};

}       // namespace sc_core
```

5.8.3 Constraints on usage

An application shall not explicitly create an object of class *sc_event_and_list*† or *sc_event_or_list*†.

Classes *sc_event_and_list*† and *sc_event_or_list*† are the return types of **operator&** and **operator|**, respectively, of class **sc_event**, and are parameter types of the functions **wait** and **next_trigger**.

5.8.4 Event lists

sc_event_and_list†& **operator&** (const sc_event&);
sc_event_or_list†& **operator|** (const sc_event&);

> A call to either operator shall add the event passed as an argument to the event list from which the operator is called.

5.9 sc_event

5.9.1 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.

5.9.2 Class definition

```
namespace sc_core {

class sc_event
{
    public:
        sc_event();
        ~sc_event();

        void notify();
        void notify( const sc_time& );
        void notify( double , sc_time_unit );
        void cancel();

        sc_event_or_list†&  operator| ( const sc_event& ) const;
        sc_event_and_list†& operator& ( const sc_event& ) const;

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

}       // namespace sc_core
```

5.9.3 Constraints on usage

Objects of class **sc_event** may be constructed during elaboration or simulation but events shall only be notified during simulation. It shall be an error to notify an event during elaboration

5.9.4 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**.

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.

NOTE—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.

NOTE 1—At most one pending notification can exist for any given event.

NOTE 2—Immediate notification cannot be cancelled.

5.9.5 Event lists

sc_event_or_list†& **operator|** (const sc_event&) const;
sc_event_and_list†& **operator&** (const sc_event&) const;

A call to either operator shall add the event passed as an argument to the event list from which the operator is called.

NOTE—Event lists are used as arguments to functions **wait** (see 5.2.17) and **next_trigger** (see 5.2.16).

5.9.6 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:

```
sc_event e;
e.notify(SC_ZERO_TIME);  // Delta notification
e.notify(1, SC_NS);      // Timed notification ignored due to pending delta notification
e.notify();              // Immediate notification cancels pending delta notification. e is notified

e.notify(2, SC_NS);      // Timed notification
e.notify(3, SC_NS);      // Timed notification ignored due to earlier pending timed notification
e.notify(1, 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
```

5.10 sc_time

5.10.1 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. For implementations using more than 64 bits, the return value of member function **value** need not be of type **sc_dt::uint64** (see member function **value** in 5.10.2).

5.10.2 Class definition

```
namespace sc_core {

enum sc_time_unit {SC_FS = 0, SC_PS, SC_NS, SC_US, SC_MS, SC_SEC};

class sc_time
{
    public:
        sc_time();
        sc_time( double , sc_time_unit );
        sc_time( const sc_time& );

        sc_time& operator= ( const sc_time& );

        sc_dt::uint64 value() const;
        double to_double() const;
        double to_seconds() const;
        const std::string to_string() 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 );

        void print( std::ostream& = std::cout ) const;
};

const sc_time operator+ ( const sc_time&, const sc_time& );
const sc_time operator- ( const sc_time&, const sc_time& );

const sc_time operator* ( const sc_time&, double );
const sc_time operator* ( double, const sc_time& );
const sc_time operator/ ( const sc_time&, double );
double 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();

}         // namespace sc_core
```

5.10.3 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.

The constructor for **sc_time** shall scale and round the given time value to the nearest multiple of the time resolution. Whether the value is rounded up or down is implementation-defined. The default constructor shall create an object having a time value of zero.

The values of enum **sc_time_unit** shall be taken to have their standard physical meanings, for example, SC_FS = femtosecond = 10E-15 seconds.

The function **sc_get_time_resolution** shall return the time resolution.

5.10.4 Functions and operators

All arithmetic, relational, equality, and assignment operators declared in 5.10.2 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.

```
sc_dt::uint64 value() const;
double to_double() const;
double to_seconds() const;
```

These functions shall return the underlying representation of the time value, first converting the value to a **double** in each of the two cases **to_double** and **to_seconds**, and then also scaling the resultant value to units of 1 second in the case of **to_seconds**.

```
const std::string to_string() const;
void print( std::ostream& = std::cout ) const;
std::ostream& operator<< ( std::ostream& , const sc_time& );
```

These 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.

5.10.5 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_event e;
e.notify(SC_ZERO_TIME);   // Delta notification
wait(SC_ZERO_TIME);       // Delta time-out
```

5.11 sc_port

5.11.1 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 5.12).

5.11.2 Class definition

```
namespace sc_core {

enum sc_port_policy
{
    SC_ONE_OR_MORE_BOUND ,                    // Default
    SC_ZERO_OR_MORE_BOUND ,
    SC_ALL_BOUND
};

class sc_port_base
: public sc_object { implementation-defined };

template <class IF, int N = 1, sc_port_policy POL = SC_ONE_OR_MORE_BOUND>
class sc_port
: public sc_port_base
{
    public:
        sc_port();
        explicit sc_port( const char* );
        virtual ~sc_port();

        virtual const char* kind() const;

        void operator() ( IF& );
        void operator() ( sc_port<IF,N>& );

        void bind( IF& );
        void bind( sc_port<IF,N>& );

        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;

    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_port( const sc_port<IF,N>& );
        sc_port<IF,N>& operator= ( const sc_port<IF,N>& );
};

}       // namespace sc_core
```

5.11.3 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 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:

```
sc_port<IF>                              // Bound to exactly 1 channel instance
sc_port<IF,0>                            // Bound to 1 or more channel instances
                                         // with no upper limit
sc_port<IF,3>                            // Bound to 1, 2 or 3 channel instances
sc_port<IF,0,SC_ZERO_OR_MORE_BOUND>      // Bound to 0 or more channel instances
                                         // with no upper limit
sc_port<IF,1,SC_ZERO_OR_MORE_BOUND>      // Bound to 0 or 1 channel instances
sc_port<IF,3,SC_ZERO_OR_MORE_BOUND>      // Bound to 0, 1, 2 or 3 channel instances
sc_port<IF,3,SC_ALL_BOUND>               // 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 4.1.3).

5.11.4 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 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.

5.11.5 Constructors

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_object** to set the string name of the instance in the module hierarchy.

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.

5.11.6 kind

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

5.11.7 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&);
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&**.

void **operator()** (sc_port<IF,N>&);
void **bind**(sc_port<IF,N>&);

> 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.

Example:

```
SC_MODULE(M)
{
    sc_inout<int> P, Q, R, S;  // Ports
    sc_inout<int> *T;          // Pointer-to-port (not a recommended coding style)

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

SC_MODULE(Top)
{
    sc_inout <int> A, B;
    sc_signal<int> C, D;
    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
    }
    ...
};
```

5.11.8 Member functions for bound ports and port-to-port binding

The member functions described in this subclause 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 this subclause) 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.

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 this subclause may change during elaboration and the final order is implementation-defined, but 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.

5.11.8.1 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.

5.11.8.2 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_interface
{
    virtual int read() const = 0;
};

struct chan
: iface, sc_prim_channel
{
    virtual int read() const;
};

int chan::read() const { ... }

SC_MODULE(modu)
{
    sc_port<iface> P;

    SC_CTOR(modu)
    {
        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
    }
};
```

5.11.8.3 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 slave_interface
: virtual public sc_interface
{
    public:
        virtual void slave_write(int addr, int data) = 0;
        virtual void slave_read (int addr, int& data) = 0;
};

class bus_channel
: public bus_interface, public sc_module
{
    public:
        ...
        sc_port<slave_interface, 0> slave_port;     // Multiport for attaching slaves to bus

        SC_CTOR(bus_channel)
        {
            SC_THREAD(action);
        }
    private:
        void action()
        {
            for (int i = 0; i < slave_port.size(); i++)         // Function size() returns number of slaves
                slave_port[i]->slave_write(0,0);               // Operator[] indexes slave port
         }
};
```

```
class memory
: public slave_interface, public sc_module
{
    public:
        virtual void slave_write(int addr, int  data);
        virtual void slave_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.slave_port(ram0);
        bus.slave_port(ram1);
        bus.slave_port(ram2);
        bus.slave_port(ram3);                    // One multiport bound to four memory channels
    }
};
```

5.11.8.4 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_in<bool> clock;

    void before_end_of_elaboration()
    {
        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
...
```

5.11.9 before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation

See 4.4.

5.12 sc_export

5.12.1 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.

5.12.2 Class definition

```
namespace sc_core {

class sc_export_base
: public sc_object { implementation-defined };

template<class IF>
class sc_export
: public sc_export_base
{
    public:
        sc_export();
        explicit sc_export( const char*  );
        virtual ~sc_export();

        virtual const char* kind() const;

        void operator() ( IF& );
        void bind( IF& );
        operator IF& ();

        IF* operator-> ();
        const IF* operator-> () const;

        virtual sc_interface* get_interface();
        virtual const sc_interface* get_interface() 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
```

5.12.3 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 4.1.3).

5.12.4 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.

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 5.11.4).

5.12.5 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.

5.12.6 kind

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

5.12.7 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&);
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.

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_interface
{
    virtual void print() = 0;
};

struct Chan: sc_channel, i_f
{
    SC_CTOR(Chan) {}
    void print() { std::cout << "I'm Chan, name=" << name() << std::endl; }
};

struct Caller: sc_module
{
    sc_port<i_f> p;
    ...
};

struct Bottom: sc_module
{
    sc_export<i_f> xp;
    Chan ch;
    SC_CTOR(Bottom) : ch("ch")
    {
        xp.bind(ch);                    // Bind export xp to channel ch
    }
};

struct Middle: sc_module
{
    sc_export<i_f> xp;
    Bottom* b;
    SC_CTOR(Middle)
    {
        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_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
        }
    };
```

5.12.8 Member functions for bound exports and export-to-export binding

The member functions described in this subclause 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 during 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 this subclause) 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.

5.12.8.1 operator-> and operator IF&

IF* **operator->** ();
const IF* **operator->** () const;
operator IF& ();

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.

5.12.8.2 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.

5.12.9 before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation

See 4.4.

5.13 sc_interface

5.13.1 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.

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.

5.13.2 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
```

5.13.3 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:

a) It shall be publicly derived directly or indirectly from class **sc_interface**

b) If directly derived from class **sc_interface**, it shall use the **virtual** specifier

c) It shall not be derived directly or indirectly from class **sc_object**

An interface proper should typically obey the following rules:

a) It should contain one or more pure virtual functions

b) It should not be derived from any other class that is not itself an interface proper

c) It should not contain any function declarations or function definitions apart from the pure virtual functions

d) 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.

5.13.4 register_port

virtual void **register_port**(sc_port_base& , const char*);

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_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;
}
```

5.13.5 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_interface
{
    virtual int read() = 0;
};

class my_ch
: public my_if, public sc_module
{
    public:
        virtual int read() { return m_val; }
        virtual const sc_event& default_event() const { return m_ev; }
    private:
        int m_val;
        sc_event m_ev;
        ...
};
```

5.14 sc_prim_channel

5.14.1 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).

5.14.2 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();
        virtual void update();

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

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

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

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

}       // namespace sc_core
```

5.14.3 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.

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.

5.14.4 Constructors

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.

5.14.5 kind

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

5.14.6 request_update and update

void **request_update**();

Member function **request_update** shall cause the scheduler to queue an update request for the specific primitive channel instance making the call (see 4.2.1.3).

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**. An application may override this member function in a primitive channel. The definition of this function in class **sc_prim_channel** itself does nothing.

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

If the application violates the two 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** 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.

5.14.7 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 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.

5.14.8 before_end_of_elaboration, end_of_elaboration, start_of_simulation, end_of_simulation

See 4.4.

Example:

```
struct my_if
: virtual sc_interface                        // An interface proper
{
    virtual int  read() = 0;
    virtual void write(int) = 0;
};

struct my_prim
: sc_prim_channel, my_if                      // A primitive channel
```

```
{
    my_prim()                                   // Default constructor
    :
        sc_prim_channel( 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_event m_write_event;
    int m_new_val, m_cur_val;
};
```

5.15 sc_object

5.15.1 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.

Attributes may be added to each **sc_object**.

NOTE—An implementation may permit multiple top-level **sc_object**s (see 4.3).

5.15.2 Class definition

```
namespace sc_core {

class sc_object
{
    public:
        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;
        sc_object* get_parent_object() const;

        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*);
        virtual ~sc_object();
};

const std::vector<sc_object*>& sc_get_top_level_objects();
sc_object* sc_find_object( const char* );

}       // namespace sc_core
```

5.15.3 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 during 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 shall 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.

5.15.4 Constructors and hierarchical names

```
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"**.

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** is reached. The hierarchical name shall end with the string name of the **sc_object** itself.

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) The lower-case letters a-z

b) The upper-case letters A-Z

c) The decimal digits 0-9

d) The 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** 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.

The constructor shall build a hierarchical name from the string name (either passed in as an argument or the default name **"object"**) 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 **sc_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 **sc_object**. In other words, the final string name shall have the original string name as a prefix.

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.

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.

5.15.5 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.h"
SC_MODULE(Mod)
{
    sc_port<sc_signal_in_if<int> > p;

    SC_CTOR(Mod)             // p.name() returns "top.mod.p"
    : p("p")                 // p.basename() returns "p"
    {}                       // p.kind() returns "sc_port"
};

SC_MODULE(Top)
{
    Mod *mod;                // mod->name() returns "top.mod"
    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;
}
```

5.15.6 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.

5.15.7 Functions for object hierarchy traversal

The four functions in this subclause return information that supports the traversal of the object hierarchy. An implementation shall allow each of these four 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.

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.

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_objects**.

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_object* obj)              // Traverse the entire object subhierarchy
                                                 // below a given object
{
    std::vector<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_object*> tops = 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_object* obj = sc_find_object("foo.foobar");   // Find an object given its hierarchical name

sc_module* m;
if (m = dynamic_cast<sc_module*>(obj))           // Test whether the given object is a module
    ...                                          // The given object is a module

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
```

5.15.8 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 5.18).

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:

```
sc_signal<int> sig;
...
// Add an attribute to an sc_object
sc_attribute<int> a("number", 1);
sig.add_attribute(a);

// 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 sc_attr_base

5.16.1 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.

5.16.2 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
```

5.16.3 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.

5.17 sc_attribute

5.17.1 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**.

5.17.2 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
```

5.17.3 Template parameters

The argument passed to template **sc_attribute** shall be of a *copy-constructible* type.

5.17.4 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.

5.18 sc_attr_cltn

5.18.1 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.

5.18.2 Class definition

```
namespace sc_core {

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
```

5.18.3 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**.

5.18.4 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:

```
sc_signal<int> sig;
...

// Iterate through all the attributes of an sc_object
sc_attr_cltn& c = sig.attr_cltn();
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;
}
```

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.

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**).

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();
    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 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 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, and is derived from a further interface proper **sc_signal_write_if**.

6.3.2 Class definition

```
namespace sc_core {

template <class T>
class sc_signal_write_if
{
    public:
        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 write

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.

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.

6.4.2 Class definition

```
namespace sc_core {

template <class T>
class sc_signal
: public sc_signal_inout_if<T>, 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 T& read() const;
        operator const T& () const;

        virtual void write( const T& );
        sc_signal<T>& operator= ( const T& );
        sc_signal<T>& operator= ( const sc_signal<T>& );

        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>& );
};

template <class T>
inline std::ostream& operator<< ( std::ostream&, const sc_signal<T>& );

}       // 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 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. It shall be an error to write a given signal instance from more than one process instance. A signal may be written during elaboration to initialize the value of the signal.

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, 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"))

explicit **sc_signal**(const char* name_);

This constructor shall call the base class constructor from its initializer list as follows:

sc_prim_channel(name_)

Both constructors shall initialize the value of the signal by calling the default constructor for type **T** from their initializer lists.

6.4.6 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_signal**, and shall perform the following error check. It is an error if more than one port of type **sc_signal_inout_if** is 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 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**.

operator=

The behavior of **operator=** shall be equivalent to the following definitions:

```
sc_signal<T>& operator= ( const T& arg ) { write( arg ); return *this; }
sc_signal<T>& operator= ( const sc_signal<T>& 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 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;

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; 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>
inline std::ostream& **operator<<** (std::ostream& , const sc_ signal<T>&);

> **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_signal<int> sig;

    SC_CTOR(M)
    {
        SC_THREAD(writer);
        SC_THREAD(reader);
        SC_METHOD(writer2);
        sensitive << sig;                  // Sensitive to the default event
    }
    void writer()
    {
        wait(50, SC_NS);
        sig.write(1);
        sig.write(2);
        wait(50, SC_NS);
        sig = 3;                           // Calls operator= ( const T& )
    }
    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>** and **sc_signal<sc_dt::sc_logic>** 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 conveniently access the member functions of the signal through the port.

6.5 sc_signal<bool> and sc_signal<sc_dt::sc_logic>

6.5.1 Description

Classes **sc_signal<bool>** and **sc_signal<sc_dt::sc_logic>** are predefined primitive channels that provide additional member functions appropriate for two-valued signals.

6.5.2 Class definition

```
namespace sc_core {

template <>
class sc_signal<bool>
: 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 void write( const bool& );
        sc_signal<bool>& operator= ( const bool& );
        sc_signal<bool>& operator= ( const sc_signal<bool>& );

         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>& );
};


template <>
class sc_signal<sc_dt::sc_logic>
: public sc_signal_inout_if<sc_dt::sc_logic>, 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>& operator= ( const sc_dt::sc_logic& );
        sc_signal<sc_dt::sc_logic>& operator= ( const sc_signal<sc_dt::sc_logic>& );

        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<sc_dt::sc_logic>& );
};

}         // 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 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 the new value of the signal is **false** or **'0'**.

Example:

```
sc_signal<bool> clk;
...
void thread_process()
{
    for (;;)
    {
        if (clk.posedge())
            wait(clk.negedge_event());
        ...
    }
}
```

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>
class sc_buffer
: public sc_signal<T>
{
    public:
        sc_buffer();
        explicit sc_buffer( const char* );

        virtual void write( const T& );

        sc_buffer<T>& operator= ( const T& );
        sc_buffer<T>& operator= ( const sc_signal<T>& );
        sc_buffer<T>& operator= ( const sc_buffer<T>& );

        virtual const char* kind() const;

    protected:
        virtual void update();

    private:
        // Disabled
        sc_buffer( const sc_buffer<T>& );
};

}       // 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<T>( 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<T>( name_ )
```

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>& operator= ( const T& arg ) { write( arg ); return *this; }

sc_buffer<T>& operator= ( const sc_signal<T>& arg ) { write( arg.read() ); return *this; }

sc_buffer<T>& operator= ( const sc_buffer<T>& 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_buffer<int> buf;

    SC_CTOR(M)
    {
        SC_THREAD(writer);
        SC_METHOD(reader);
        sensitive << buf;
    }
    void writer()
    {
        buf.write(1);
        wait(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& );

        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<bool>( 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.

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 conveniently access 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
: public sc_port<sc_signal_in_if<T>,1>
{
    public:
        sc_in();
        explicit sc_in( const char* );
        virtual ~sc_in();

        void bind ( const sc_signal_in_if<T>& );
        void operator() ( const sc_signal_in_if<T>& );

        void bind ( sc_port<sc_signal_in_if<T>, 1>& );
        void operator() ( sc_port<sc_signal_in_if<T>, 1>& );

        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**.

Member function **bind** and **operator()** shall each 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()
```

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 Function 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) 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();

        void bind ( const sc_signal_in_if<bool>& );
        void operator() ( const sc_signal_in_if<bool>& );

        void bind ( sc_port<sc_signal_in_if<bool>, 1>& );
        void operator() ( sc_port<sc_signal_in_if<bool>, 1>& );

        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();

        void bind ( const sc_signal_in_if<sc_dt::sc_logic>& );
        void operator() ( const sc_signal_in_if<sc_dt::sc_logic>& );

        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>& );

        void bind ( sc_port<sc_signal_inout_if<sc_dt::sc_logic>, 1>& );
        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).

6.10 sc_inout

6.10.1 Description

Class **sc_inout** is a specialized port class for use with signals. It provides functions to conveniently access 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
```

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 Function 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) 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.

6.10.6 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;
        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;
        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>
{
    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>
{
    public:
        sc_signal_resolved();
        explicit sc_signal_resolved( const char* );
        virtual ~sc_signal_resolved();

        virtual void register_port( sc_port_base&, const char* );

        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_dt::sc_logic>( 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<sc_dt::sc_logic>( name_ )
```

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 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 1.
3) Repeat 2 until only a single value remains. This is the resolved value.

Table 1—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 1 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"**.

Example:

```
SC_MODULE(M)
{
    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, SC_NS);
        sig = sc_dt::SC_LOGIC_0;        // Time=10 ns, written values=0        sig=0
        wait(20, SC_NS);
        sig = sc_dt::SC_LOGIC_Z;        // Time=30 ns, written values=Z,Z      sig=Z
    }
    void T2()
    {
        wait(20, SC_NS);
        sig = sc_dt::SC_LOGIC_Z;        // Time=20 ns, written values=0,Z      sig=0
        wait(30, SC_NS);
        sig = sc_dt::SC_LOGIC_0;        // Time=50 ns, written values=Z,0,1    sig=X
    }
    void T3()
    {
        wait(40, 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>** 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();

        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>** 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>& );
        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 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>& );
        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> >
{
    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();

    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_dt::sc_lv<W> > ( 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> >** 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();

        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> >** 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>& );

        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 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>& );
        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 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>
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
{
    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.

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. A *fifo* is 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
```

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 an error if more than one port of type **sc_fifo_out_if** is bound to a given fifo.

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; }
```

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.

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_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_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 conveniently access certain member functions of the fifo to which the port is bound.

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 conveniently access certain member functions of the fifo to which the port is bound.

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; }
        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_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, SC_NS);
                status = fifo_out.nb_write(u);          // Non-blocking write
            } while (!status);
        }
    }
};

SC_MODULE(M2)
{
    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)
{
    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 primitive 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.

NOTE—Although **sc_mutex** is derived from **sc_prim_channel**, **sc_mutex** does not use the request update mechanism.

6.27.2 Class definition

```
namespace sc_core {

class sc_mutex
: public sc_mutex_if, public sc_prim_channel
{
    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_prim_channel(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_prim_channel(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.

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
```

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 primitive 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.

NOTE—Although **sc_semaphore** is derived from **sc_prim_channel**, **sc_semaphore** does not use the request update mechanism.

6.29.2 Class definition

```
namespace sc_core {

class sc_semaphore
: public sc_semaphore_if, public sc_prim_channel
{
    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_prim_channel(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_prim_channel(name_)

Both constructors shall set the *semaphore value* to the value of the **int** parameter, which shall be non-negative.

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 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.

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();
        explicit sc_event_queue( sc_module_name );
        ~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();

The default constructor shall call the base class constructor from its initializer list as follows:

sc_module(sc_gen_unique_name("event_queue")) explicit **sc_event_queue**(sc_module_name);

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"**.

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 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_event_queue EQ;

SC_CTOR(Mod)
{
        SC_THREAD(T);
        SC_METHOD(M);
            sensitive << EQ;
            dont_initialize();
}

void T()
{
        EQ.notify(2, SC_NS);             // M runs at time 2ns
        EQ.notify(1, SC_NS);             // M runs at time 1ns, 1st or 2nd delta cycle
        EQ.notify(SC_ZERO_TIME);         // M runs at time 0ns
        EQ.notify(1, SC_NS);             // M runs at time 1ns, 2nd or 1st delta cycle
}
```

7. 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 lower-case 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 lower-case 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.

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 2.

Table 2—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
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

This subclause specifies some common characteristics of the SystemC data types such as common operators and functions. This subclause 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.

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.

7.2.1 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 principal, 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 3.3.5).

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.

7.2.2 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.

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 7.10.4).

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_int<16> sci = 2;
sc_bigint<16> bi = 20;
float f = 2.5;
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
```

7.2.3 Base class default word length

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 **end** method shall only be called for the length context currently at the top of the context stack. A length context is also deactivated and removed from the stack when it goes out of scope. 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.

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_length_param length10(10);
sc_length_context cntxt10(length10);                // length10 now in context
sc_int_base int_array[2];                           // Array of 10-bit integers
sc_core::sc_signal<sc_int_base> S1;                 // Signal of 10-bit integer
{
    sc_length_param length12(12);
    sc_length_context cntxt12(length12,SC_LATER);    // cntxt12 deferred
    sc_length_param length14(14);
    sc_length_context cntxt14(length14,SC_LATER);   // cntxt14 deferred
    sc_uint_base var1;                              // length 10
    cntxt12.begin();                                // Bring length12 into context
    sc_uint_base var2;                              // length 12
    cntxt14.begin();                                // Bring length14 into context
    sc_uint_base var3;                              // length 14
    cntxt14.end();                                  // end cntx14, cntx12 restored
    sc_bv_base var4;                                // length 12
}                                                   // cntxt12 out of scope, cntx10 restored
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.

7.2.4 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**.

7.2.5 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.

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_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**.

7.2.6 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.

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, 7.9.8.3). A part-select of an lvalue may be used as an rvalue or 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_int<8> I2 = 2;                    // "0b00000010"
I2.range(3,2) = I2.range(1,0);       // "0b00001010"
sc_int<8> I3 = I2.range(3,0);        // "0b00001010"
                                     // Zero-extended to 8 bits
sc_bv<8> b1 = "0b11110000";
b1.range(5,2) = b1.range(2,5);       // "0b11001100"
                                     // Reversed bit-order between position 5 and 2
```

NOTE 1—A part-select cannot be used to reverse the bit-order of a limited-precision integer type.

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.

7.2.7 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 a)

c) Part-selects of a)

d) Concatenations of a) and/or b) and/or 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 3.3.5).

Example:

The following concatenations are well-formed:

```
sc_uint<8> U1 = 2;                              // "0b00000010"
sc_uint<2> U2 = 1;                              // "0b01"
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_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.

7.2.8 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.

7.2.9 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.

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.

7.2.10 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.

7.2.11 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.

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.

The permitted representations are identified with a symbol from the enumerated type **sc_numrep** as specified in Table 3.

Table 3—String literal representation

sc_numrep	Prefix	Magnitude format
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 3. The character ‘+’ or ‘-’ may optionally be placed before the prefix for decimal and “sign & 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 3. 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 3 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_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_BIN,true); // The contents of S1 will be the string "0b0100"
sc_int<10> I2;                          // 10-bit integer
I2 = "0d478";                           // Decimal equivalent of "0b0111011110"
std::string S2 = I2.to_string(SC_CSD,false); // The contents of S2 will be the string "1000-000-0"
sc_uint<8> I3;                          // 8-bit unsigned integer
I3 = "0x7";                             // Zero-extended to 8-bit value "0x07"
std::string S3 = I3.to_string(SC_HEX);  // The contents of S3 will be the string "0x007"
sc_lv<16> lv;                           // 16-bit logic vector
lv = "0xff";                            // Sign-extended to 16-bit value "0xffff"
std::string S4 = lv.to_string(SC_HEX);  // The contents of S4 will be the string "0x0ffff"
sc_bv<8> bv;                            // 8-bit bit vector
bv = "11110000";                        // Bit-pattern string
std::string S5 = bv.to_string(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.

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 methods that may be called by an implementation to perform concatenation operations.

7.4.1.1 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( unsigned long* dst_p , int low_i ) const;
        virtual bool concat_get_data( unsigned long* 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
```

7.4.1.2 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.

7.4.1.3 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**(unsigned long* 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**.

virtual bool **concat_get_data**(unsigned long* 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**.

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.

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 );
```

```
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
const sc_int_base operator++ ( int );      // Postfix
sc_int_base& operator-- ();                // Prefix
const 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 methods
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
const std::string to_string( sc_numrep numrep = SC_DEC ) const;
```

```
        const std::string to_string( sc_numrep numrep , bool w_prefix ) const;

        // Other methods
        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 7.2.1.

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**.

7.5.2.7 Explicit type conversion

const std::string **to_string**(sc_numrep numrep = SC_DEC) const;
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 7.2.11. 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 4 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 4—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.

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 7.2.10).

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 7.2.10).

int **length**() const;

> Member function **length** shall return the word length (see 7.2.4).

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 );
        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
const sc_uint_base operator++ ( int );      // Postfix
sc_uint_base& operator-- ();                // Prefix
const 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 methods
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
const std::string to_string( sc_numrep numrep = SC_DEC ) const;
```

```
          const std::string to_string( sc_numrep numrep , bool w_prefix ) const;

          // Other methods
          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.

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 7.2.1.

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

const std::string **to_string**(sc_numrep numrep = SC_DEC) const;
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 7.2.11. 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 5 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 5—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
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 7.2.10).

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 7.2.10).

int **length**() const;

> Member function **length** shall return the word length (see 7.2.4).

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**.

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 );
    sc_int<W>& operator= ( uint64 a );
    sc_int<W>& operator= ( double a );

    // Prefix and postfix increment and decrement operators
    sc_int<W>& operator++ ();                 // Prefix
    const sc_int<W> operator++ ( int );       // Postfix
    sc_int<W>& operator-- ();                 // Prefix
    const 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 7.2.1.

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 7.2.1.

7.5.4.6 Arithmetic and bitwise operators

Operations specified in Table 6 are permitted. The following applies:

- **n** represents an object of type **sc_int**.
- **i** represents an object of integer type **int_type**.

Table 6—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

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.

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 );
        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
        const sc_uint<W> operator++ ( int );        // Postfix
        sc_uint<W>& operator-- ();                  // Prefix
        const 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**.

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 7.2.1.

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 7.2.1.

7.5.5.6 Arithmetic and bitwise operators

Operations specified in Table 7 are permitted. The following applies:

— **U** represents an object of type **sc_uint**.

— **u** represents an object of integer type **uint_type**.

Table 7—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.

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 methods
        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†
{
    friend class sc_int_base;

    public:
        // Copy constructor
        sc_int_bitref†( const sc_int_bitref†& a );

        // Assignment operators
        sc_int_bitref†& operator= ( const sc_int_bitref_r†& b );
        sc_int_bitref†& operator= ( const sc_int_bitref†& b );
        sc_int_bitref†& operator= ( bool b );
        sc_int_bitref†& operator&= ( bool b );
        sc_int_bitref†& operator|= ( bool b );
        sc_int_bitref†& operator^= ( bool b );

        // Other methods
        void scan( std::istream& is = std::cin );

    private:
        sc_int_bitref†();
};

// -------------------------------------------------------------

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 methods
        void print( std::ostream& os = std::cout ) const;

    protected:
        sc_uint_bitref_r†();

    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 methods
        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_int_base i, int n) {
    return i[n];        // Unsafe: returned bit-select references local variable
}
```

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 to prevent their use 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 7.2.10).

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 7.2.10). 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 7.2.4).

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†;

    public:
        // Copy constructor
        sc_int_subref_r†( const sc_int_subref_r†& a );
        // Destructor
        virtual ~sc_int_subref_r†();

        // Capacity
        int length() const;

        // Reduce methods
        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
        const std::string to_string( sc_numrep numrep = SC_DEC ) const;
        const std::string to_string( sc_numrep numrep , bool w_prefix ) const;

        // Other methods
        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:
        // Copy constructor
        sc_int_subref†( const sc_int_subref†& a );

        // Assignment operators
        sc_int_subref†& operator= ( int_type v );
        sc_int_subref†& operator= ( const sc_int_base& a );
        sc_int_subref†& operator= ( const sc_int_subref_r†& a );
        sc_int_subref†& operator= ( const sc_int_subref†& a );
        template< class T >
        sc_int_subref†& operator= ( const sc_generic_base<T>& a );
        sc_int_subref†& operator= ( const char* a );
        sc_int_subref†& operator= ( unsigned long a );
        sc_int_subref†& operator= ( long a );
        sc_int_subref†& operator= ( unsigned int a );
        sc_int_subref†& operator= ( int a );
        sc_int_subref†& operator= ( uint64 a );
        sc_int_subref†& operator= ( double a );
        sc_int_subref†& operator= ( const sc_signed& );
        sc_int_subref†& operator= ( const sc_unsigned& );
        sc_int_subref†& operator= ( const sc_bv_base& );
        sc_int_subref†& operator= ( const sc_lv_base& );

        // Other methods
        void scan( std::istream& is = std::cin );

    protected:
        sc_int_subref†();

};

// ------------------------------------------------------------

class sc_uint_subref_r†
{
    friend class sc_uint_base;
    friend class sc_uint_subref†;

    public:
        // Copy constructor
        sc_uint_subref_r†( const sc_uint_subref_r†& a );

        // Destructor
        virtual ~sc_uint_subref_r();

        // Capacity
        int length() const;

        // Reduce methods
        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
        const std::string to_string( sc_numrep numrep = SC_DEC ) const;
        const std::string to_string( sc_numrep numrep , bool w_prefix ) const;

        // Other methods
        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& );
        sc_uint_subref†& operator= ( const sc_unsigned& );
        sc_uint_subref†& operator= ( const sc_bv_base& );
        sc_uint_subref†& operator= ( const sc_lv_base& );
        // Other methods
        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 7.2.6.

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_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 7.2.1.

Assignment operators for rvalue integer part-selects shall be declared as private to prevent their use 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.
>
> 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

const std::string **to_string**(sc_numrep numrep = SC_DEC) const;
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 7.2.11. 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 7.2.10).

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 7.2.10).

int **length**() const;

Member function **length** shall return the word length of the part-select (see 7.2.4).

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 the following:

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 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 );
        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++ ();
        const sc_signed operator++ ( int );

        // Decrement operators.
        sc_signed& operator-- ();
        const 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
        const std::string to_string( sc_numrep numrep = SC_DEC ) const;
        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;

        // Reduce methods
        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 7.2.1.

7.6.3.6 Explicit type conversion

const std::string **to_string**(sc_numrep numrep = SC_DEC) const;
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 7.2.11. 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 8, Table 9, and Table 10 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 8—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
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

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.

Table 9—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

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.

Table 10—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

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.

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 7.2.10).

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 7.2.10).

int **length**() const;

Member function **length** shall return the word length (see 7.2.4).

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 );
        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++ ();
const sc_unsigned operator++ ( int );

// Decrement operators
sc_unsigned& operator-- ();
const 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;
double to_double() const;

// Explicit conversion to character string
const std::string to_string( sc_numrep numrep = SC_DEC ) const;
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 methods
        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 7.2.1.

7.6.4.6 Explicit type conversion

const std::string **to_string**(sc_numrep numrep = SC_DEC) const;
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 7.2.11. 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 11, Table 12, and Table 13 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 11—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

Table 11—sc_unsigned arithmetic operations *(continued)*

Expression	Return type	Operation
U / s	sc_signed	division of sc_unsigned by signed
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

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.

Table 12—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
S >> U	sc_signed	sc_signed right-shift
U >>= i	sc_unsigned&	sc_unsigned assign right-shift
~U	sc_unsigned	sc_unsigned bitwise complement

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 one. The bit 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_unsigned** operand. The bit on the left-hand side of the result shall be set to zero.

Table 13—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

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 7.2.10).

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 7.2.10).

int **length**() const;

Member function **length** shall return the word length (see 7.2.4).

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 );
        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.

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 7.2.1.

NOTE—Most of the 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 7.2.1.

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 );
        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
```

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 7.2.1.

NOTE—Most of the 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 7.2.1.

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†;

    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 methods
        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 methods
        void scan( std::istream& is = std::cin );

    protected:
        sc_signed_bitref†();
};

// ----------------------------------------------------------------

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 methods
        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†& );
        sc_unsigned_bitref†& operator= ( bool );

        sc_unsigned_bitref†& operator&= ( bool );
        sc_unsigned_bitref†& operator|= ( bool );
        sc_unsigned_bitref†& operator^= ( bool );


        // Other methods
        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_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 to prevent their use 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.

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 7.2.10).

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 7.2.10). 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 7.2.4).

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;

        // 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
        const std::string to_string( sc_numrep numrep = SC_DEC ) const;
        const std::string to_string( sc_numrep numrep, bool w_prefix ) const;

        // Reduce methods
        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 methods
        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 );
        sc_signed_subref†& operator= ( unsigned int a );
        sc_signed_subref†& operator= ( int a );
        sc_signed_subref†& operator= ( uint64 a );
        sc_signed_subref†& operator= ( int64 a );
        sc_signed_subref†& operator= ( double a );
        sc_signed_subref†& operator= ( const sc_int_base& a );
        sc_signed_subref†& operator= ( const sc_uint_base& a );

        // Other methods
        void scan( std::istream& is = std::cin );

    private:
        // Disabled
        sc_signed_subref†();
};

// -----------------------------------------------------------------
```

```
class sc_unsigned_subref_r†
: public sc_value_base†
{
    friend class sc_signed;
    friend class sc_unsigned;

    public:
        // Copy constructor
        sc_unsigned_subref_r†( const sc_unsigned_subref_r†& a );

        // Destructor
        virtual ~sc_unsigned_subref_r†();

        // 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
        const std::string to_string( sc_numrep numrep = SC_DEC ) const;
        const std::string to_string( sc_numrep numrep , bool w_prefix ) const;

        // Reduce methods
        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 methods
        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 methods
        void scan( std::istream& is = std::cin );

    protected:
        sc_unsigned_subref†();
};

}       // namespace sc_dt
```

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 7.2.6.

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_signed s, int pos) {
    return s(pos+7,pos);        // Unsafe: returned part-select references local variable
}
```

NOTE—The left-hand index of a finite-precision integer part-select may be less than the right-hand index. The bit order in the part-select is then the reverse of that in the original integer.

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 7.2.1.

Assignment operators for rvalue integer part-selects shall be declared as private to prevent their use by an application.

7.6.8.5 Implicit type conversion

sc_signed_subref_r†:: **operator sc_unsigned** () const;
sc_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

const std::string **to_string**(sc_numrep numrep = SC_DEC) const;
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 7.2.11. 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**.

7.6.8.7 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 7.2.10).

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 7.2.10).

int **length**() const;

> Member function **length** shall return the word length of the part-select (see 7.2.4).

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;
        sc_unsigned operator~ () const;

        // Explicit conversion to character string
        const std::string to_string( sc_numrep numrep = SC_DEC ) const;
        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 methods
        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 methods
        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 7.2.7.

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>** 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 7.2.1.

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

const std::string **to_string**(sc_numrep numrep = SC_DEC) const;
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 7.2.11. 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 7.2.10).

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 7.2.10).

int **length**() const;

Member function **length** shall return the word length of the concatenation (see 7.2.4).

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.

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, 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 lower-case 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 14, Table 15, Table 16, and Table 17.

Table 14—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'

Table 15—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 16—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 17—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 );
        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 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.

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

Operations specified in Table 18 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**.

Table 18—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

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.

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 7.2.10).

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 7.2.10).

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_logic> A;
A = '0';                                // Error: ambiguous conversion
A = static_cast<sc_logic>('0');         // Correct but not recommended
A = SC_LOGIC_0;                         // Recommended representation of logic 0
```

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 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 );

        // 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 methods
        int length() const;

        // Explicit conversions to character string
        const std::string to_string() const;
        const std::string to_string( sc_numrep ) const;
        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;
        bool is_01() const;

        // Other methods
        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);
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 7.2.1.

7.9.3.6 Explicit type conversion

const std::string **to_string**() const;
const std::string **to_string**(sc_numrep) const;
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 7.2.11. 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_bv_base B(4);                        // 4-bit vector
B = "0xf";                              // Each bit set to logic 1
std::string S1 = B.to_string(SC_BIN,false); // The contents of S1 will be the string "01111"
std::string S2 = B.to_string(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 7.2.9.

7.9.3.7 Bitwise and comparison operators

Operations specified in Table 19 and Table 20 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.

Table 19—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

Table 19—sc_bv_base bitwise operations *(continued)*

Expression	Return type	Operation
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

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_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.

Table 20—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

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 7.2.10).

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 7.2.10).

int **length**() const;

Member function **length** shall return the word length (see 7.2.4).

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;

    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;

    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 methods
    int length() const;

    // Explicit conversions to character string
    const std::string to_string() const;
    const std::string to_string( sc_numrep ) const;
    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;
    bool is_01() const;

    // Other methods
    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.

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 7.2.1.

7.9.4.6 Explicit type conversion

const std::string **to_string**() const;
const std::string **to_string**(sc_numrep) const;
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 7.2.11. 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_lv_base L(4);                            // 4-bit vector
L = "0xf";                                  // Each bit set to logic 1
std::string S1 = L.to_string(SC_BIN,false); // The contents of S1 will be the string "01111"
std::string S2 = L.to_string(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**.

Member functions that return the integer equivalent of the bit representation shall be provided to satisfy the requirements of subclause 7.2.9. 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 21 and Table 22 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 21—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
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

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.

Table 22—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

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 7.2.10).

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 7.2.10).

int **length**() const;

Member function **length** shall return the word length (see 7.2.4).

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 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 );

        // 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 7.2.1. 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 7.2.1. 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.

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 );
        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 7.2.1. 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 7.2.1. 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.

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
        const sc_logic operator~ () const;

        // Implicit conversion to sc_logic
        operator const sc_logic() const;

        // Explicit conversions
        bool is_01() const;
        bool to_bool() const;
        char to_char() const;

        // Common methods
        int length() const;

        // Other methods
        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 );
        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 methods
        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 7.2.6.

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_bitref<sc_bv_base> get_bit_n(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 to prevent its use by an application.

7.9.7.5 Implicit type conversion

operator const 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**.

7.9.7.7 Bitwise and comparison operators

Operations specified in Table 23 are permitted. The following applies:

B represents an object of type *sc_bitref_r*†*<T>* (or any derived class).

Table 23—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

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.

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 7.2.10).

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 7.2.10). 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 7.2.4).

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 );

        // 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 methods
        int length() const;

        // Explicit conversions to character string
        const std::string to_string() const;
        const std::string to_string( sc_numrep ) const;
        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;
        bool is_01() const;

        // Other methods
        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

        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 methods
        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 7.2.6.

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_bv_base> get_byte(sc_bv_base bv, int pos) {
    return bv(pos+7,pos);      // Unsafe: returned part-select references local variable
}
```

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 7.2.1. 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 to prevent its use by an application.

7.9.8.5 Explicit type conversion

```
const std::string to_string() const;
const std::string to_string( sc_numrep ) const;
const std::string to_string( sc_numrep , bool ) const;
```

Member function **to_string** shall convert to an **std::string** representation, as described in 7.2.11. 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 7.2.9. 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.

7.9.8.6 Bitwise and comparison operators

Operations specified in Table 24 and Table 27 are permitted for all vector part-selects. Operations specified in Table 25 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 24—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

Table 25—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

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.

Table 26—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

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.

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 7.2.10).

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 7.2.10).

int **length**() const;

Member function **length** shall return the word length of the part-select (see 7.2.4).

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;

        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 methods
        int length() const;

        // Explicit conversions to character string
        const std::string to_string() const;
        const std::string to_string( sc_numrep ) const;
        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;
        bool is_01() const;

        // Other methods
        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 );
        sc_concref†<T1,T2>& operator= ( const sc_concref†<T1,T2>& a );
        sc_concref†<T1,T2>& operator= ( const char* a );
        sc_concref†<T1,T2>& operator= ( const bool* a );
        sc_concref†<T1,T2>& operator= ( const sc_logic* a );
        sc_concref†<T1,T2>& operator= ( const sc_unsigned& a );
        sc_concref†<T1,T2>& operator= ( const sc_signed& a );
        sc_concref†<T1,T2>& operator= ( const sc_uint_base& a );
        sc_concref†<T1,T2>& operator= ( const sc_int_base& a );
        sc_concref†<T1,T2>& operator= ( unsigned long a );
        sc_concref†<T1,T2>& operator= ( long a );
        sc_concref†<T1,T2>& operator= ( unsigned int a );
        sc_concref†<T1,T2>& operator= ( int a );
        sc_concref†<T1,T2>& operator= ( uint64 a );
        sc_concref†<T1,T2>& operator= ( int64 a );

        // Bitwise rotations
        sc_concref†<T1,T2>& lrotate( int n );
        sc_concref†<T1,T2>& rrotate( int n );

        // Bitwise reverse
        sc_concref†<T1,T2>& reverse();

        // Bit selection
        sc_bitref†<sc_concref†<T1,T2>> operator[] ( int i );

        // Part selection
        sc_subref†<sc_concref†<T1,T2>> operator() ( int hi , int lo );

        sc_subref†<sc_concref†<T1,T2>> range( int hi , int lo );

        // Other methods
        void scan( std::istream& = std::cin );

    private:
        // Disabled
        sc_concref†();
};

// 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 );

}       // 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 7.2.7. 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_bv_base,sc_bv_base> pad(sc_bv_base& bv, char pchar) {
    const 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 7.2.1. 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 to prevent its use by an application.

7.9.9.5 Explicit type conversion

```
const std::string to_string() const;
const std::string to_string( sc_numrep ) const;
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 7.2.11. 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.

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 7.2.9. 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 27 and Table 29 are permitted for all vector concatenations; operations specified in Table 28 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 27—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

Table 28—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

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.

Table 29—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

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.

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 7.2.10).

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 7.2.10).

int **length**() const;

Member function **length** shall return the word length of the concatenation (see 7.2.4).

7.9.9.8 Function 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.

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

This subclause describes the fixed-point types and the operations and conventions imposed by these types.

7.10.1 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 30 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 30 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 30 for examples of this case.

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:

$$[0\ \ ,2^{(\mathrm{iwl})} - 2^{-(\mathrm{wl}-\mathrm{iwl})}]$$

Table 30—Examples of fixed-point formats

Index	wl	iwl	Fixed-point representation[*]	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]

[*] **x** is an arbitrary binary digit, 0, or 1. **s** is a sign-extended digit, 0, or 1,

7.10.2 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 7.10.9.1 and 7.10.9.9).

7.10.3 Fixed-point data types

This subclause describes the classes that are provided to represent fixed-point values.

7.10.3.1 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.

7.10.3.2 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.

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 shall not exceed 53 bits.

7.10.4 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**, **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 31. 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.

Table 31—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

Table 31—Fixed-point arithmetic and bitwise functions *(continued)*

Expression	Operation
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 32, Table 33, Table 34, and Table 35.

The addition operations specified in Table 32 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**, **double**, **sc_signed**, **sc_unsigned**, **sc_int_base**, **sc_uint_base**, **sc_fxval**, **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 32—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

The addition operations specified in Table 33 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**, **double**, **sc_signed**, **sc_unsigned**, **sc_int_base**, **sc_uint_base**, **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 33—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 34 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**, **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 34—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

The addition operations specified in Table 35 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**, **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 35—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 32, Table 33, Table 34, and Table 35.

7.10.5 Bit and part selection

Bit and part selection shall be supported for the fixed-point types, as described in 7.2.5 and 7.2.6. 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.

7.10.6 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.

7.10.7 Fixed-point word length and mode

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 7.2.3. 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 36.

Table 36—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 7.11.5 and 7.11.6).

A global floating-point cast context stack shall manage floating-point cast contexts using the same semantics as the length context stack described in 7.2.3.

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_fxtype_params fxt(32,16);
sc_fxtype_context fcxt(fxt);

sc_fix A,B,res;                // wl = 32, iwl = 16
A = 10.0;
B = 0.1;
res = A * B;                   // res = .999908447265625

sc_fxcast_switch fxs(SC_OFF);
sc_fxcast_context fccxt(fxs);
sc_fix C,D;                    // Floating-point behavior
C = 10.0;
D = 0.1;
res = C * D;                   // res = 1
```

7.10.7.1 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.

7.10.7.2 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:

const sc_fxval **value()** const;

The following function is defined for every limited-precision fixed-point object and shall return its current value:

const sc_fxval_fast **value()** const;

7.10.8 Conversions to character string

Conversion to character string of the fixed-point types shall be supported by the **to_string** method, as described in 7.3.

The **to_string** method 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 uniquely represent the value 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 shall be determined by the integer and fractional widths (iwl, fwl) of the variable (with sign or zero extension as needed).

Example:

```
sc_fixed<7,4> a = -1.5;
a.to_string(SC_DEC);              // -1.5
a.to_string(SC_BIN);              // 0b1110.100
sc_fxval b = -1.5;
b.to_string(SC_BIN);              // 0b10.1
sc_fixed<4,6> c = 20;
c.to_string(SC_BIN,false,SC_F);   // 010100
c.to_string(SC_BIN,false,SC_E);   // 0101e+2
```

7.10.8.1 String shortcut methods

Four shortcut methods to the **to_string** method shall be provided for frequently used combinations of arguments. The shortcut methods are listed in Table 37.

Table 37—Shortcut methods

Shortcut method	Number representation
to_dec()	SC_DEC
to_bin()	SC_BIN
to_oct()	SC_OCT
to_hex()	SC_HEX

The shortcut methods shall use the default string formatting.

Example:

```
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"
```

7.10.8.2 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.

7.10.9 Finite word-length effects

The following subclauses describe the overflow and quantization modes of SystemC.

7.10.9.1 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 38 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 38—Overflow modes

Overflow mode	Name
Saturation	SC_SAT
Saturation to zero	SC_SAT_ZERO
Symmetrical saturation	SC_SAT_SYM
Wrap-around [*]	SC_WRAP
Sign magnitude wrap-around[*]	SC_WRAP_SM

[*] with 0 or n_bits saturated bits (n_bits > 0). The default value for n_bits is 0.

In the following subclauses, each of the overflow modes 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.

7.10.9.2 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 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(lN)*	*R*	*R*	*R*	*R*	*R*	*R*	*R*	*R*	*lR*

The following flags and symbols are used in the template just given and in Table 39:

- *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.

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 39 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.

Table 39—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.		
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 ^lNo ^ ! 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.		

7.10.9.3 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 40:

- *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.

Table 40 shows how an unsigned fixed-point number shall be cast in case there is an overflow for each of the possible overflow modes.

Table 40—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.	

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.

7.10.9.4 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 1 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.

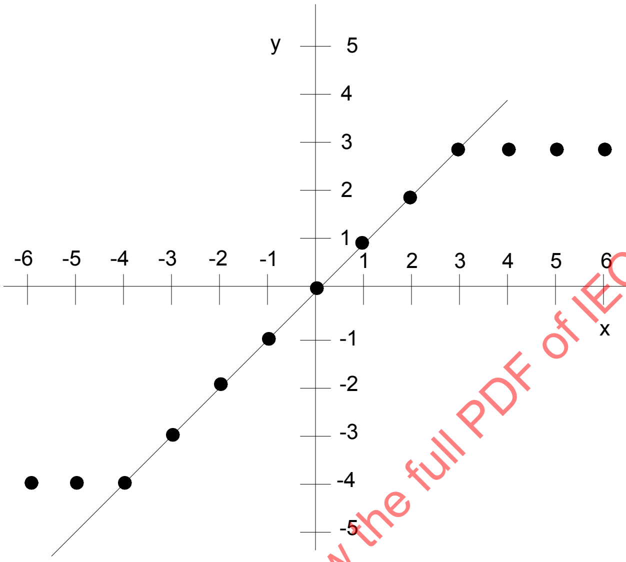

Figure 1—Saturation for signed numbers

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.

7.10.9.5 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 2 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.

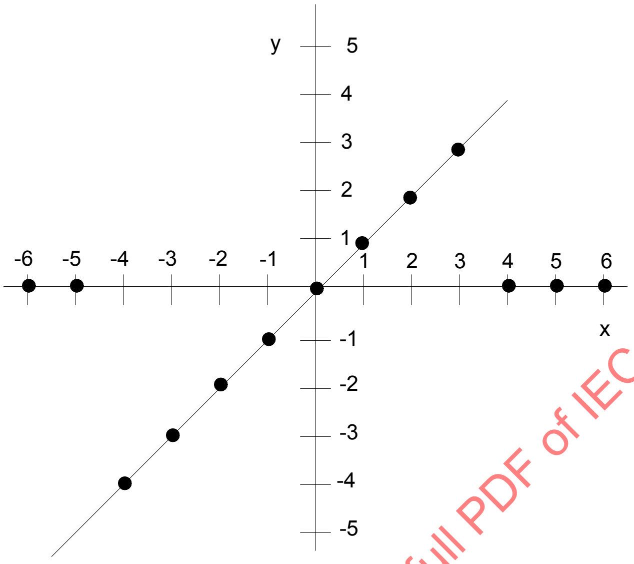

Figure 2—Saturation to zero for signed numbers

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)**

7.10.9.6 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 3 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.

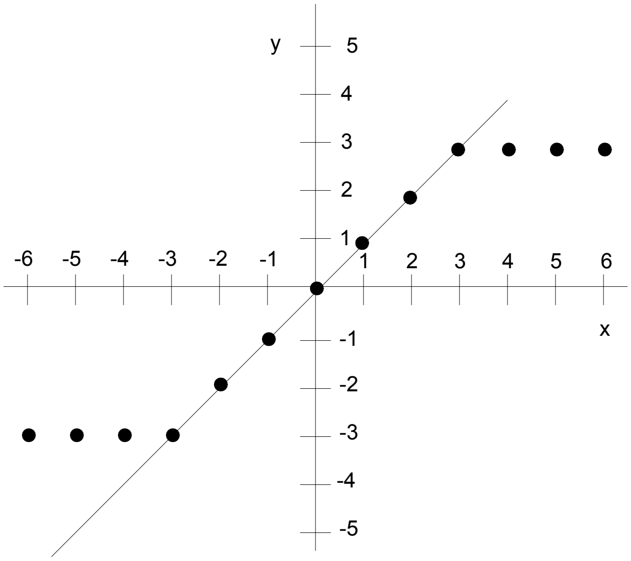

Figure 3—Symmetrical saturation for signed numbers

Examples (signed, 3-bit number):

after symmetrical saturation: **0110 (6)**

after symmetrical 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.

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)**

7.10.9.7 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 4 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.

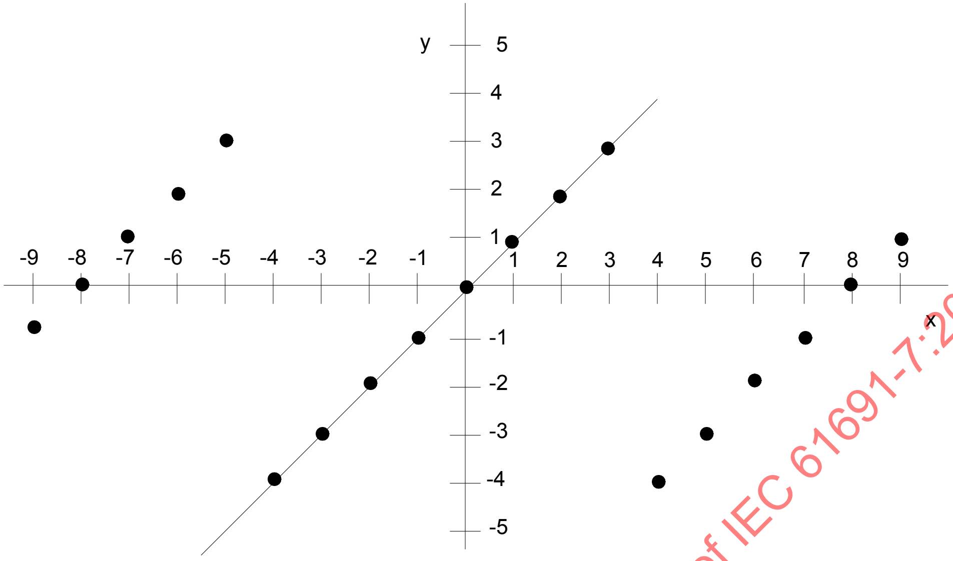

Figure 4—Wrap-around with n_bits = 0 for signed numbers

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 5 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.

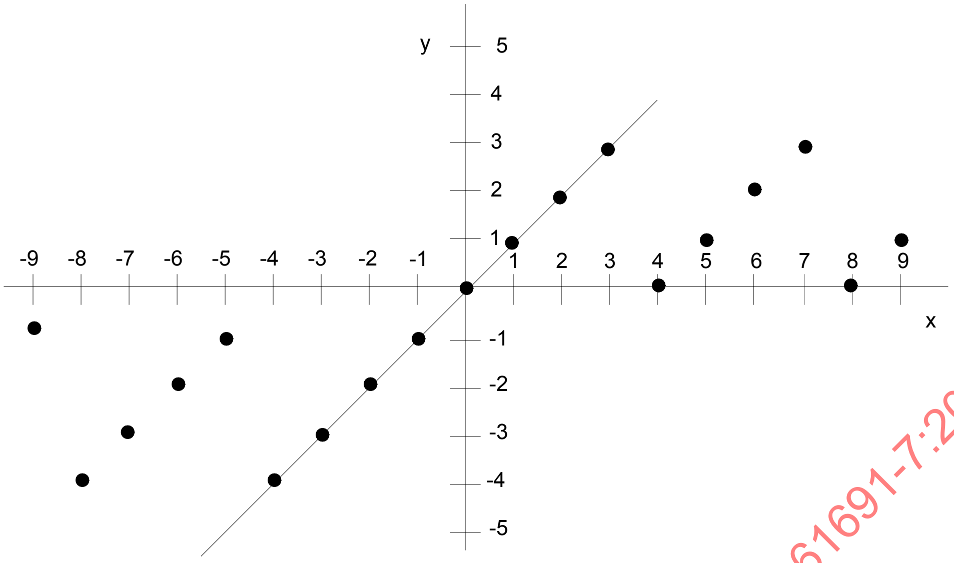

Figure 5—Wrap-around with n_bits = 1 for signed numbers

Examples (signed, 3-bit number):

before wrapping around with 1 bit: **0101 (5)**

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 3 bits at the MSB side are set to 1; the remaining bits are copied.

7.10.9.8 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 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.

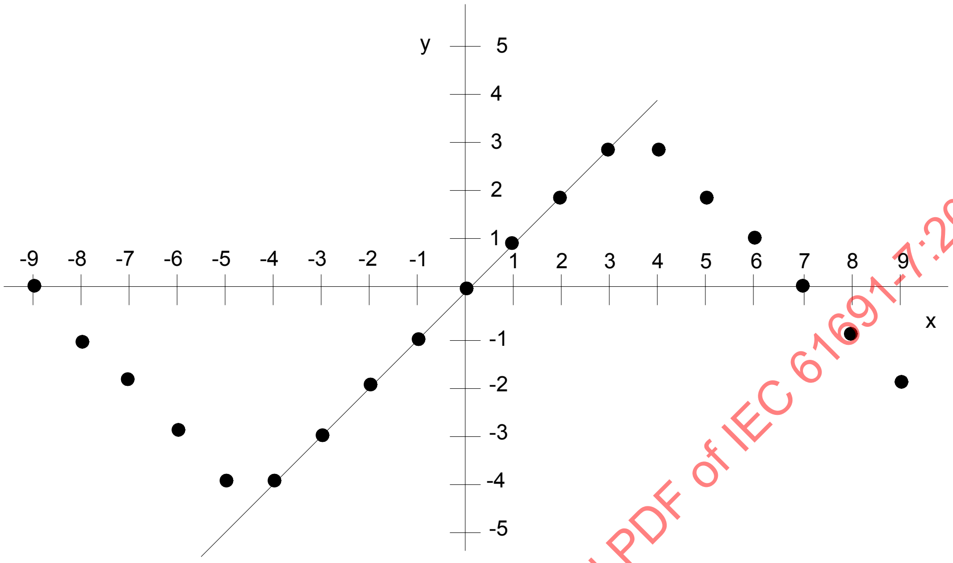

Figure 6—Sign Magnitude Wrap-Around with n_bits = 0

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 6, 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 41.

Table 41—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
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.

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.

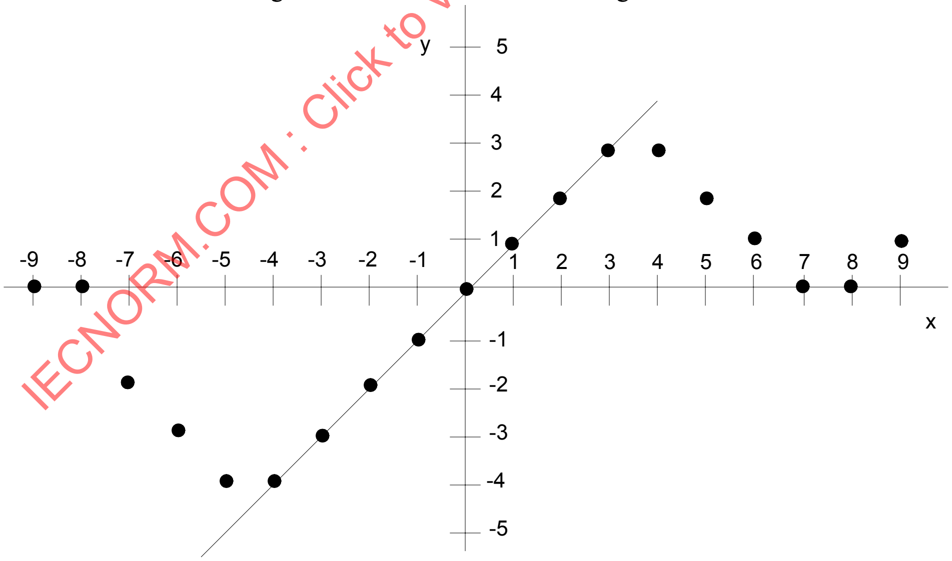

Figure 7—Sign magnitude wrap-around with n_bits = 1

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 7, 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 42.

Table 42—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
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

7.10.9.9 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 43 shall be provided. The default quantization mode shall be SC_TRN.

Table 43—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

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 of the 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.

7.10.9.10 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 44:

- *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 44 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.

The second column in Table 44 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 44—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.

7.10.9.11 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 45:

- *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.
- *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 45 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 45 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.

Table 45—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.

NOTE—For all rounding modes, overflow can occur. One extra bit on the MSB side is needed to represent the result in full precision.

7.10.9.12 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 towards plus infinity, as shown in Figure 8.

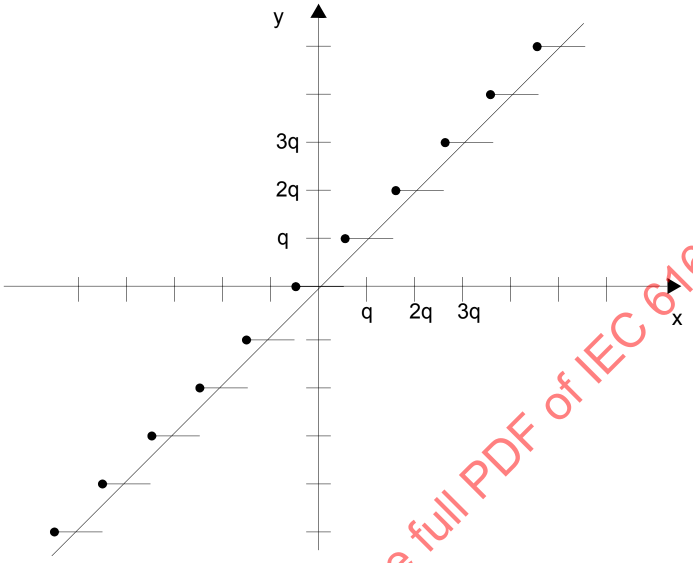

Figure 8—Rounding to plus infinity

In Figure 8, 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)**

7.10.9.13 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 towards 0, as shown in Figure 9. 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.

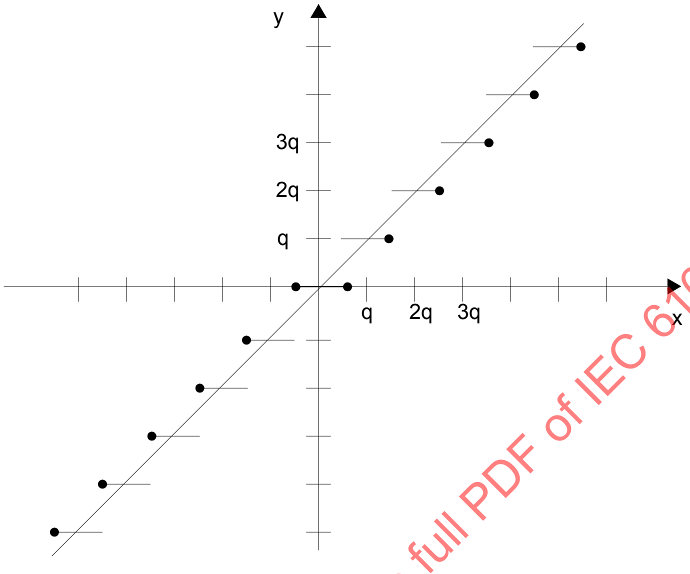

Figure 9—Rounding to Zero

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)**

7.10.9.14 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 towards minus infinity, as shown in Figure 10, by omitting the redundant bits on the LSB side.

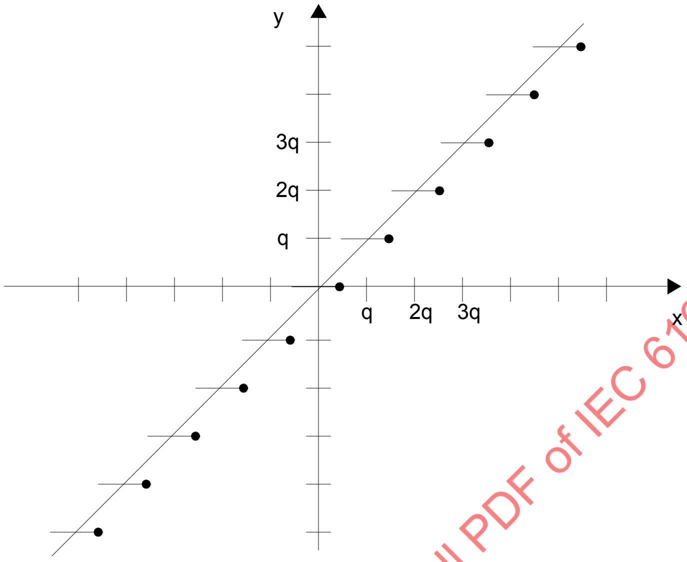

Figure 10—Rounding to minus infinity

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)**

7.10.9.15 SC_RND_INF

Rounding shall be performed if the two nearest representable numbers are at equal distance.

For positive numbers, there shall be rounding towards plus infinity if the LSB of the remaining bits is 1 and towards minus infinity if the LSB of the remaining bits is 0, as shown in Figure 11.

For negative numbers, there shall be rounding towards minus infinity if the LSB of the remaining bits is 1 and towards plus infinity if the LSB of the remaining bits is 0, as shown in Figure 11.

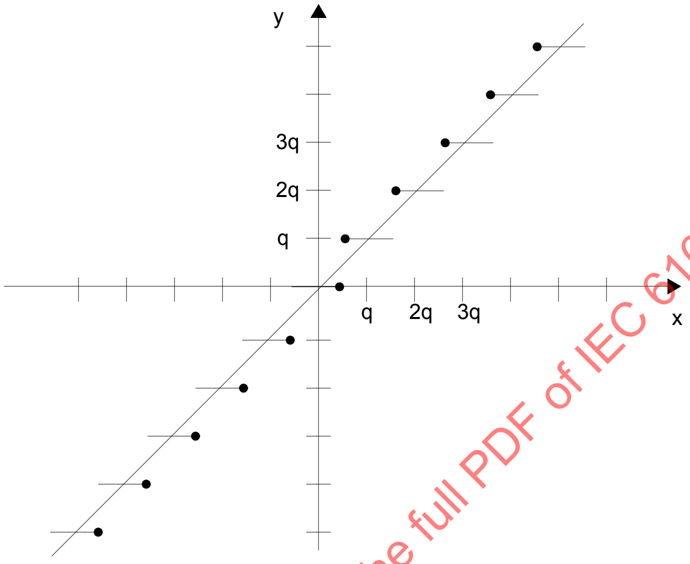

Figure 11—Rounding to infinity

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)**

7.10.9.16 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 towards plus infinity if the LSB of the remaining bits is 1. There shall be rounding towards minus infinity if the LSB of the remaining bits is 0. The characteristics are shown in Figure 12.

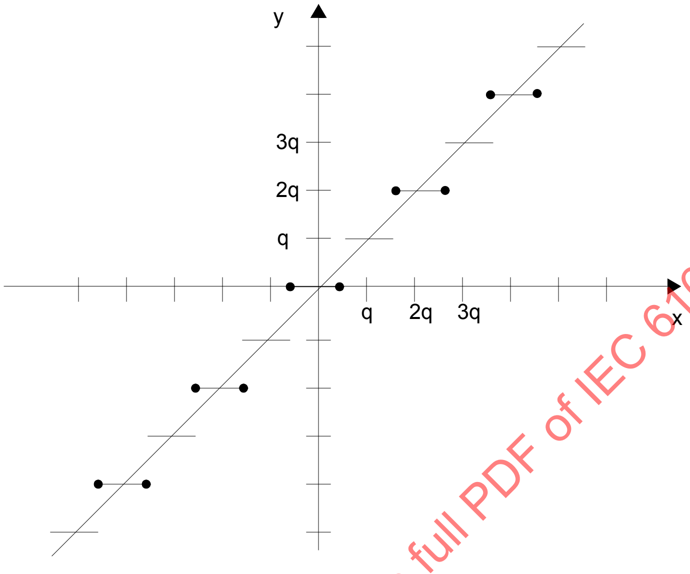

Figure 12—Convergent rounding

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 towards 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 towards 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)** before convergent rounding: **000100110.01011 (38.34375)**

after convergent rounding: **000100110.0110 (38.375)**

7.10.9.17 SC_TRN

SC_TRN shall be the default quantization mode. The result shall be rounded towards 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 13.

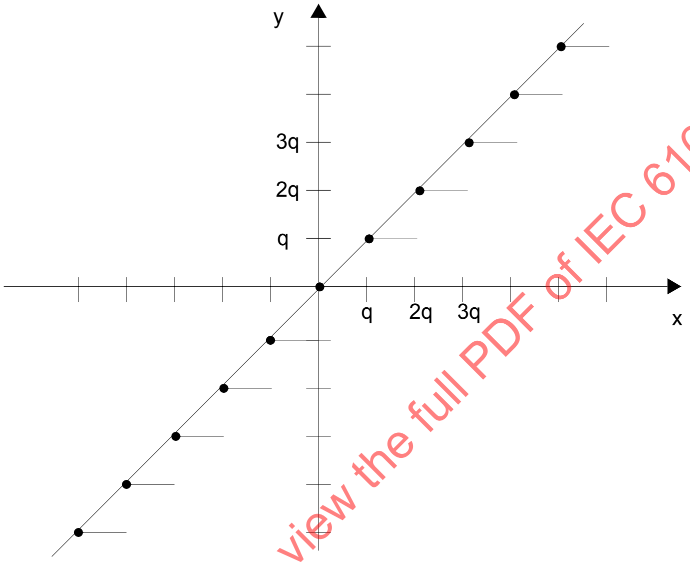

Figure 13—Truncation

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)**

7.10.9.18 SC_TRN_ZERO

For positive numbers, this quantization mode shall correspond to SC_TRN. For negative numbers, the result shall be rounded towards 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 14.

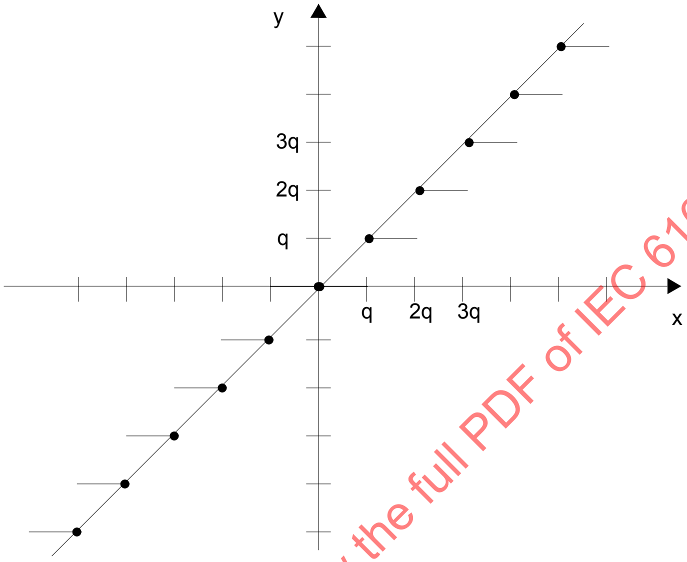

Figure 14—Truncation to zero

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)**

7.10.10 sc_fxnum

7.10.10.1 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.

7.10.10.2 Class definition

```
namespace sc_dt {

class sc_fxnum
{
    friend class sc_fxval;

    friend class sc_fxnum_bitref†;
    friend class sc_fxnum_subref†;
    friend class sc_fxnum_fast_bitref†;
    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 const sc_fxval operator op ( const sc_fxnum& , tp ); \
            friend const 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 , dummy ) \
            friend const 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 , 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 )

        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 const sc_fxval operator<< ( const sc_fxnum& , int );
friend const 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 , 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& ) \
    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 , 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
    const sc_fxval operator++ ( int );
    const sc_fxval operator-- ( int );
    sc_fxnum& operator++ ();
    sc_fxnum& operator-- ();

    // Bit selection
    const sc_fxnum_bitref† operator[] ( int ) const;
    sc_fxnum_bitref† operator[] ( int );

    // Part selection
    const sc_fxnum_subref† operator() ( int , int ) const;
    sc_fxnum_subref† operator() ( int , int );
    const sc_fxnum_subref† range( int , int ) const;
    sc_fxnum_subref† range( int , int );
    const sc_fxnum_subref† operator() () const;
    sc_fxnum_subref† operator() ();
    const sc_fxnum_subref† 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;
    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
        const std::string to_string() const;
        const std::string to_string( sc_numrep ) const;
        const std::string to_string( sc_numrep , bool ) const;
        const std::string to_string( sc_fmt ) const;
        const std::string to_string( sc_numrep , sc_fmt ) const;
        const std::string to_string( sc_numrep , bool , sc_fmt ) const;
        const std::string to_dec() const;
        const std::string to_bin() const;
        const std::string to_oct() const;
        const std::string to_hex() const;

        // Query value
        bool is_neg() const;
        bool is_zero() const;
        bool quantization_flag() const;
        bool overflow_flag() const;
        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
```

7.10.10.3 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**.

7.10.10.4 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 7.10.4.

7.10.10.5 Implicit type conversion

operator **double**() const;

Operator **double** shall provide implicit type conversion from **sc_fxnum** to double.

7.10.10.6 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.

const std::string **to_string**() const;
const std::string **to_string**(sc_numrep) const;
const std::string **to_string**(sc_numrep , bool) const;
const std::string **to_string**(sc_fmt) const;
const std::string **to_string**(sc_numrep , sc_fmt) const;
const std::string **to_string**(sc_numrep , bool , sc_fmt) const;
const std::string **to_dec**() const;
const std::string **to_bin**() const;
const std::string **to_oct**() const;
const std::string **to_hex**() const;

These member functions shall perform the conversion to a **string** representation, as described in 7.2.11, 7.10.8, and 7.10.8.1.

7.10.11 sc_fxnum_fast

7.10.11.1 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.

7.10.11.2 Class definition

namespace sc_dt {

class **sc_fxnum_fast**
{
    friend class sc_fxval_fast;

    friend class *sc_fxnum_bitref*†;
    friend class *sc_fxnum_subref*†;
    friend class *sc_fxnum_fast_bitref*†;
    friend class *sc_fxnum_fast_subref*†;

```
public:
    // Unary operators
    const sc_fxval_fast operator- () const;
    const sc_fxval_fast operator+ () const;

    // Binary operators
    #define DECL_BIN_OP_T( op , tp ) \
        friend const sc_fxval_fast operator op ( const sc_fxnum_fast& , tp ); \
        friend const 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 , dummy ) \
        friend const 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 , 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 const sc_fxval operator<< ( const sc_fxnum_fast& , int );
    friend const 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 ) \
        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 , 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 , 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
const sc_fxval_fast operator++ ( int );
const sc_fxval_fast operator-- ( int );
sc_fxnum_fast& operator++ ();
sc_fxnum_fast& operator-- ();
```

```
// Bit selection
const sc_fxnum_bitref† operator[] ( int ) const;
sc_fxnum_bitref† operator[] ( int );

// Part selection
const sc_fxnum_fast_subref† operator() ( int , int ) const;
sc_fxnum_fast_subref† operator() ( int , int );
const sc_fxnum_fast_subref† range( int , int ) const;
sc_fxnum_fast_subref† range( int , int );
const sc_fxnum_fast_subref† operator() () const;
sc_fxnum_fast_subref† operator() ();
const sc_fxnum_fast_subref† 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
const std::string to_string() const;
const std::string to_string( sc_numrep ) const;
const std::string to_string( sc_numrep , bool ) const;
const std::string to_string( sc_fmt ) const;
const std::string to_string( sc_numrep , sc_fmt ) const;
const std::string to_string( sc_numrep , bool , sc_fmt ) const;
const std::string to_dec() const;
const std::string to_bin() const;
const std::string to_oct() const;
const std::string to_hex() const;

// Query value
bool is_neg() const;
bool is_zero() const;
bool quantization_flag() const;
bool overflow_flag() const;
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
```

7.10.11.3 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**.

7.10.11.4 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 7.10.4.

7.10.11.5 Implicit type conversion

```
operator double() const;
```

Operator **double** shall provide implicit type conversion from **sc_fxnum_fast** 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.

```
const std::string to_string() const;
const std::string to_string( sc_numrep ) const;
const std::string to_string( sc_numrep , bool ) const;
const std::string to_string( sc_fmt ) const;
const std::string to_string( sc_numrep , sc_fmt ) const;
const std::string to_string( sc_numrep , bool , sc_fmt ) const;
const std::string to_dec() const;
const std::string to_bin() const;
const std::string to_oct() const;
const std::string to_hex() const;
```

> These member functions shall perform the conversion to a **string** representation, as described in 7.2.11, 7.10.8, and 7.10.8.1.

7.10.12 sc_fxval

7.10.12.1 Description

Class **sc_fxval** is the variable-precision fixed-point type. It may hold the value of any of the fixed-point types and performs 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.

7.10.12.2 Class definition

```
namespace sc_dt {

class sc_fxval
{
    public:
        // Constructors and destructor
        sc_fxval();
        sc_fxval( int );
        sc_fxval( unsigned int );
        sc_fxval( long );
        sc_fxval( unsigned long );
        sc_fxval( double );
        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& );
        sc_fxval( int64 );
        sc_fxval( uint64 );
        sc_fxval( const sc_int_base& );
        sc_fxval( const sc_uint_base& );
        sc_fxval( const sc_signed& );
        sc_fxval( const sc_unsigned& );
        ~sc_fxval();

        // Unary operators
        const 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 const sc_fxval operator op ( const sc_fxval& , tp ); \
            friend const 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 , dummy ) \
    friend const 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 , 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 const sc_fxval operator<< ( const sc_fxval& , int );
friend const 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 ) \
    DECL_REL_OP_T( op , long ) \
    DECL_REL_OP_T( op , unsigned long ) \
    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 , 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
const sc_fxval operator++ ( int );
const sc_fxval operator-- ( int );
sc_fxval& operator++ ();
sc_fxval& 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
        const std::string to_string() const;
        const std::string to_string( sc_numrep ) const;
        const std::string to_string( sc_numrep , bool ) const;
        const std::string to_string( sc_fmt ) const;
        const std::string to_string( sc_numrep , sc_fmt ) const;
        const std::string to_string( sc_numrep , bool , sc_fmt ) const;
        const std::string to_dec() const;
        const std::string to_bin() const;
        const std::string to_oct() const;
        const std::string to_hex() const;

        // Methods
        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
```

7.10.12.3 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.

7.10.12.4 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.

7.10.12.5 Operators

The operators that shall be defined for **sc_fxval** are given in Table 46.

Table 46—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 7.10.4.

7.10.12.6 Implicit type conversion

operator **double**() const;

> Operator **double** can be used for implicit type conversion to the C++ type **double**.

7.10.12.7 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.

const std::string **to_string**() const;
const std::string **to_string**(sc_numrep) const;
const std::string **to_string**(sc_numrep , bool) const;
const std::string **to_string**(sc_fmt) const;
const std::string **to_string**(sc_numrep , sc_fmt) const;
const std::string **to_string**(sc_numrep , bool , sc_fmt) const;
const std::string **to_dec**() const;
const std::string **to_bin**() const;
const std::string **to_oct**() const;
const std::string **to_hex**() const;

> These member functions shall perform the conversion to a **string** representation, as described in 7.2.11, 7.10.8, and 7.10.8.1.

7.10.13 sc_fxval_fast

7.10.13.1 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.

7.10.13.2 Class definition

```
namespace sc_dt {

class sc_fxval_fast
{
    public:
        sc_fxval_fast();
        sc_fxval_fast( int );
        sc_fxval_fast( unsigned int );
        sc_fxval_fast( long );
        sc_fxval_fast( unsigned long );
        sc_fxval_fast( double );
        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& );
        sc_fxval_fast( int64 );
        sc_fxval_fast( uint64 );
        sc_fxval_fast( const sc_int_base& );
        sc_fxval_fast( const sc_uint_base& );
        sc_fxval_fast( const sc_signed& );
        sc_fxval_fast( const sc_unsigned& );
        ~sc_fxval_fast();

        // Unary operators
        const sc_fxval_fast  operator- () const;
        const sc_fxval_fast& operator+ () const;

         // Binary operators
        #define DECL_BIN_OP_T( op , tp ) \
            friend const sc_fxval_fast operator op ( const sc_fxval_fast& , tp ); \
            friend const 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& )

        #define DECL_BIN_OP( op , dummy ) \
            friend const 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 , 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 const sc_fxval_fast operator<< ( const sc_fxval_fast& , int );
friend const 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 , 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& ) \
    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 , 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
        const sc_fxval_fast operator++ ( int );
        const 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
        const std::string to_string() const;
        const std::string to_string( sc_numrep ) const;
        const std::string to_string( sc_numrep , bool ) const;
        const std::string to_string( sc_fmt ) const;
        const std::string to_string( sc_numrep , sc_fmt ) const;
        const std::string to_string( sc_numrep , bool, sc_fmt ) const;
        const std::string to_dec() const;
        const std::string to_bin() const;
        const std::string to_oct() const;
        const std::string to_hex() const;

        // Other methods
        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_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.

7.10.13.4 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.

7.10.13.5 Operators

The operators that shall be defined for **sc_fxval_fast** are given in Table 47.

Table 47—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**.

7.10.13.6 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.

```
const std::string to_string() const;
const std::string to_string( sc_numrep ) const;
const std::string to_string( sc_numrep , bool ) const;
const std::string to_string( sc_fmt ) const;
const std::string to_string( sc_numrep , sc_fmt ) const;
const std::string to_string( sc_numrep , bool, sc_fmt ) const;
const std::string to_dec() const;
const std::string to_bin() const;
const std::string to_oct() const;
const std::string to_hex() const;
```

These member functions shall perform the conversion to a **string** representation, as described in 7.2.11, 7.10.8, and 7.10.8.1.

7.10.14 sc_fix

7.10.14.1 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.

7.10.14.2 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& );

        #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( 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
const sc_fix operator~ () const;

// Binary bitwise operators
friend const sc_fix operator& ( const sc_fix& , const sc_fix& );
friend const sc_fix operator& ( const sc_fix& , const sc_fix_fast& );
friend const sc_fix operator& ( const sc_fix_fast& , const sc_fix& );
friend const sc_fix operator| ( const sc_fix& , const sc_fix& );
friend const sc_fix operator| ( const sc_fix& , const sc_fix_fast& );
friend const sc_fix operator| ( const sc_fix_fast& , const sc_fix& );
friend const sc_fix operator^ ( const sc_fix& , const sc_fix& );
friend const sc_fix operator^ ( const sc_fix& , const sc_fix_fast& );
friend const sc_fix operator^ ( const sc_fix_fast& , const sc_fix& );

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 , 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& )

    const sc_fxval operator++ ( int );
    const sc_fxval operator-- ( int );
    sc_fix& operator++ ();
    sc_fix& operator-- ();
};

}       // namespace sc_dt
```

7.10.14.3 Constraints on usage

The word length shall be greater than zero. The number of saturated bits, if specified, shall not be less than zero.

7.10.14.4 Public constructors

The constructor arguments may specify the fixed-point type parameters, as described in 7.10.1. 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 7.10.7.

7.10.14.5 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 7.10.4.

7.10.14.6 Bitwise operators

Bitwise operators for all combinations of operands of type **sc_fix** and **sc_fix_fast** shall be defined, as described in 7.10.4.

7.10.15 sc_ufix

7.10.15.1 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.

7.10.15.2 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 ); \
            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( 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
const sc_ufix operator~ () const;

// Binary bitwise operators
friend const sc_ufix operator& ( const sc_ufix& , const sc_ufix& );
friend const sc_ufix operator& ( const sc_ufix& , const sc_ufix_fast& );
friend const sc_ufix operator& ( const sc_ufix_fast& , const sc_ufix& );
friend const sc_ufix operator| ( const sc_ufix& , const sc_ufix& );
friend const sc_ufix operator| ( const sc_ufix& , const sc_ufix_fast& );
friend const sc_ufix operator| ( const sc_ufix_fast& , const sc_ufix& );
friend const sc_ufix operator^ ( const sc_ufix& , const sc_ufix& );
friend const sc_ufix operator^ ( const sc_ufix& , const sc_ufix_fast& );
friend const sc_ufix operator^ ( const sc_ufix_fast& , const sc_ufix& );

// 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 , 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
    const sc_fxval operator++ ( int );
    const sc_fxval operator-- ( int );
    sc_ufix& operator++ ();
    sc_ufix& 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.

7.10.15.4 Public constructors

The constructor arguments may specify the fixed-point type parameters, as described in 7.10.1. 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 7.10.7.

7.10.15.5 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 7.10.4.

7.10.15.6 Bitwise operators

Bitwise operators for all combinations of operands of type **sc_ufix** and **sc_ufix_fast** shall be defined, as described in 7.10.4.

7.10.16 sc_fix_fast

7.10.16.1 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.

7.10.16.2 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& );
```