

SYSTEMS INITIATIVE

SystemC High-Level Synthesis Standard

Version 2.0 (Public Review)

November 2026

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft,
subject to change.

Abstract

Abstract: High-level synthesis (HLS) tools translate behavioral or partially timed SystemC models into register-transfer level (RTL) implementations. This standard defines the minimum ISO C++17 and IEEE Std 1666 SystemC syntax subset to be supported by HLS tools, the standard input code structures that generate specific RTL structures, and the expected output structures in the generated RTL. It also defines a library of communication and synchronization components for HLS-based designs. With this standard, hardware designers can create SystemC designs that are portable among all conforming HLS tools. The standard does not restrict a tool's support for syntax beyond the subset, including vendor-specific design and library components.

Keywords: High-level Synthesis, Behavioral Synthesis, HLS, SystemC, Synthesizable Subset

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

Notices

Accellera Systems Initiative (Accellera) Standards documents are developed within Accellera and the Technical Committee of Accellera. Accellera develops its standards through a consensus development process, approved by its members and board of directors, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are members of Accellera and serve without compensation. While Accellera administers the process and establishes rules to promote fairness in the consensus development process, Accellera does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

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

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

The existence of an Accellera 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 an Accellera Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change due to developments in the state of the art and comments received from users of the standard. Every Accellera Standard is subjected to review periodically for revision and update. Users are cautioned to check to determine that they have the latest edition of any Accellera Standard.

In publishing and making this document available, Accellera is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is Accellera undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other Accellera 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 Accellera, Accellera will initiate action to prepare appropriate responses. Since Accellera 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, Accellera and the members of its Technical 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 Accellera Standards are welcome from any interested party, regardless of membership affiliation with Accellera. 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:

Accellera Systems Initiative.
8698 Elk Grove Blvd Suite 1, #114
Elk Grove, CA 95624
USA

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

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. Accellera shall not be responsible for identifying patents for which a license may be required by an Accellera standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Accellera is the sole entity that may authorize the use of Accellera-owned certification marks and/or trademarks to indicate compliance with the materials set forth herein.

Authorization to photocopy portions of any individual standard for internal or personal use must be granted by Accellera, provided that permission is obtained from and any required fee is paid to Accellera. To arrange for authorization please contact Lynn Garibaldi, Accellera Systems Initiative, 8698 Elk Grove Blvd Suite 1, #114, Elk Grove, CA 95624, phone (916) 670-1056, e-mail lynn@accellera.org. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained from Accellera.

Suggestions for improvements to the SystemC High-Level Synthesis Standard are welcome.

They should be sent to the Working Group's email reflector:

swg@lists.accellera.org

The current Working Group web page is:

https://www.accellera.org/activities/working-groups/systemc-synthesis

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

Introduction

High-Level Synthesis (HLS), also known as *Behavioral Synthesis*, allows designing at a level of abstraction higher than the register-transfer level (RTL) by automating the translation and optimization of a behavioral description, or a high-level model, into an RTL implementation. It transforms un-timed or partially timed functional models into fully timed RTL implementations.

Since the micro-architecture is generated automatically, designers can focus on designing and verifying the module functionality. Design teams create and verify their designs in less time because it eliminates the need to fully schedule and allocate design resources, as done with traditional RTL methods. This behavioral design flow increases design productivity, reduces errors, and speeds up verification.

A typical high-level synthesis process incorporates a number of complex stages. This process starts with a high-level language description of a module's behavior, including I/O actions and computational functionality. Several algorithmic optimizations are performed to reduce the complexity of a result and then the description is analyzed to determine the essential operations and the dataflow dependencies between them.

The other inputs to the high-level synthesis process typically include a target technology library and a set of directives that will influence the resulting architecture. These directives include, for example, timing constraints used by the algorithms of the synthesis tool as they create a cycle-by-cycle schedule of the required operations. In some flows, a set of low-level components is characterized at a target frequency for the selected fabrication process. This characterized library is used when allocation and binding occurs, in order to assign these operations to specific functional units, such as adders, multipliers, comparators, etc.

Finally, a state machine is generated that will control the resulting datapath to implement the desired functionality. The datapath and state machine outputs are RTL code optimized for use with conventional simulation and logic synthesis or physical synthesis tools. Details of the different stages of high-level synthesis are out of the scope of this document.

The growing popularity of SystemC for system and hardware design has spurred significant growth in the HLS industry in the recent past.

While there are multiple commercially available HLS tools that accept SystemC as an input language, the inherent difference between the description of a simulation model and that of a synthesis model presents the question of which constructs and semantics of SystemC should be supported by these tools and how they should be interpreted when generating the RTL design.

This standard is intended for use by hardware designers and HLS tool developers in a manner that allows creating HLS designs in SystemC that will be portable among all conforming HLS tools.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

Participants

The following working group members drove the Draft 2.0 effort:

- Thomas Arndt
- Stuart Swan
- Nathaniel Pinckney
- Dave Apte
- Ramanan Radhakrishnan
- Mike Meredith
- Rauf Salimi Khaligh, Vice-Chair
- Frederic Doucet, Chair

The following working group members drove the Draft 1.5 effort:

- Bob Condon
- Mark Johnstone
- Michael Pellauer
- Frederic Doucet
- Mike Meredith, Vice-Chair
- Andres Takach, Chair

The following working group members drove the Draft 1.4 effort:

- Ashfaq Khan
- Jos Verhaegh
- Tony Kirke
- Mike Meredith
- Benjamin Carrion Schafer
- Bob Condon
- Alan P. Su
- Stuart Swan
- Lucien Murray-Pitts
- Yusuke Iguchi
- Masachika Hamabe
- Mark Johnstone
- Frederic Doucet
- Andres Takach, Chair

The following working group members drove the Draft 1.1 ~ 1.3 effort:

- Mike Meredith
- Benjamin Carrion Schafer
- Alan P. Su
- Jos Verhaegh
- Andres Takach, Chair

The following working group members drove the Draft 1.0 effort:

- Eike Grimpe
- Rocco Jonack

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

— Mike Meredith
— Fumiaki Nagao
— Andres Takach
— Yutaka Tamiya
— Minoru Tomobe
— Masamichi Kawarabayashi, Chair

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

Contents

1. Overview 14
   1.1 Scope 14
   1.2 Purpose 14
   1.3 Word usage 14
      1.3.1 Construct categories 15
   1.4 Conventions 15

2. Normative references 17
   2.1 Standard baseline versions 17
      2.1.1 Reference documents 17
      2.1.2 Other materials 17
   2.2 ISOC++ implementation compliance 17
      2.2.1 Implementation-defined behavior 17
      2.2.2 Undefined behavior 18
      2.2.3 Unspecified behavior 18
   2.3 SystemC LRM compliance 18
      2.3.1 Implementation-defined 18
      2.3.2 Undefined 18
      2.3.3 Error 18

3. Definitions, acronyms, and abbreviations 19

4. Translation units 20
   4.1 Translation units and their analysis 20
   4.2 Pre-processing directives 20

5. Modules 22
   5.1 Module definitions 22
   5.2 Data members for hardware interface 23
   5.3 Member functions 23
   5.4 Constructors 23
   5.5 Ignored member functions 24
   5.6 Not Supported member functions 24
   5.7 Port bindings 25
   5.8 Deriving modules 25
   5.9 Selecting the top of a design hierarchy 26

6. Processes 27
   6.1 SC_METHOD 27
      6.1.1 Combinational SC_METHOD 27
      6.1.2 Sequential SC_METHOD 28
   6.2 SC_THREAD and SC_CTHREAD 29
      6.2.1 Clock and reset 29
      6.2.2 Thread process body 30

7. Communication: signals, ports, channels and memories 34
   7.1 Basic signals 34
   7.2 Basic ports 34
   7.3 Interface, export and port 36
   7.4 Modular ports 36
      7.4.1 Syntactic requirements 36
      7.4.2 Synthesis rules 38

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

7.5 Memories ... 39
7.5.1 Array size and value type ... 39
7.5.2 Memories local to a module ... 40
7.5.3 Memory shared between modules ... 40
7.6 Arrays of signals and ports ... 42
7.6.1 Syntactic requirements ... 43
7.6.2 Synthesis rules ... 43
7.7 Unsupported pre-defined SystemC channels ... 44

8. Types ... 45
8.1 Fundamental types ... 45
8.1.1 Integer types ... 45
8.1.2 Type conversions ... 46
8.1.3 Operators ... 47
8.1.4 Floating point types ... 48
8.1.5 The void type ... 48
8.2 Compound types ... 48
8.3 SystemC datatypes ... 48
8.3.1 Limited precision integer types ... 49
8.3.2 Finite precision integer types ... 49
8.3.3 Finite precision fixed-point types ... 49
8.3.4 Logic and vector types ... 50
8.3.5 Common operators and functions ... 50

9. Declarations ... 59
9.1 Specifiers ... 59
9.1.1 Storage class specifiers ... 59
9.1.2 Function specifiers ... 60
9.1.3 The typedef specifier ... 60
9.1.4 The friend specifier ... 60
9.1.5 The constexpr specifier ... 60
9.1.6 The inline specifier ... 60
9.1.7 Type specifiers ... 60
9.2 Enumerations ... 61
9.3 Namespaces ... 61
9.4 The asm declaration ... 61
9.5 Linkage specifications ... 61
9.6 Attributes ... 61
9.6.1 Attribute syntax and semantics ... 61
9.6.2 Alignment specifier ... 61
9.6.3 Carries dependency attribute ... 61
9.6.4 Deprecated attribute ... 61
9.6.5 Noreturn attribute ... 62

10. Declarators ... 63
10.1 Type names ... 63
10.2 Ambiguity resolution ... 63
10.3 Kinds of declarators ... 63
10.3.1 Pointers ... 63
10.3.2 References ... 63
10.3.3 Pointers to nonstatic class members ... 63
10.3.4 Arrays ... 63
10.3.5 Function parameters ... 64
10.3.6 Default arguments ... 64
10.4 Function definition ... 64

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

10.4.1 Explicitly-defaulted functions ... 64
10.4.2 Deleted definitions ... 64
10.5 Structured binding declarations ... 64
10.6 Initializers ... 64
10.6.1 Aggregates ... 64
10.6.2 Character arrays ... 65
10.6.3 References ... 65
10.6.4 List-initialization ... 65

11. Expressions ... 66
11.1 Primary expressions ... 66
11.1.1 General ... 66
11.1.2 Lambda expressions ... 66
11.1.3 Fold expressions ... 66
11.2 Function call ... 66
11.3 Explicit type conversion ... 66
11.3.1 Type conversion operators ... 67
11.3.2 Functional notation ... 67
11.3.3 Cast notation ... 67
11.4 typeid ... 67
11.5 Unary expressions and operators ... 67
11.5.1 Unary operators ... 67
11.5.2 Increment and decrement ... 67
11.5.3 sizeof ... 68
11.5.4 new and delete ... 68
11.5.5 alignof ... 68
11.5.6 noexcept operator ... 68
11.6 Pointer-to-member operators ... 69
11.7 Multiplicative, additive, shift, relational and equality operators ... 69
11.8 Bitwise AND/XOR/OR and logical AND/OR operators ... 69
11.9 Conditional operator ... 69
11.10 Throwing an exception ... 69
11.11 Assignment and compound assignment operators ... 69
11.12 Comma operator ... 69
11.13 Constant expressions ... 69

12. Statements ... 70
12.1 Labeled statement ... 70
12.2 Expression statement ... 70
12.3 Compound statement ... 70
12.4 Selection statements ... 70
12.4.1 The if statement ... 70
12.4.2 The switch statement ... 70
12.5 Iteration statements ... 70
12.6 Jump statements ... 70
12.6.1 The break statement ... 71
12.6.2 The continue statement ... 71
12.6.3 The return statement ... 71
12.6.4 The goto statement ... 71
12.7 Declaration statement ... 71
12.8 Exception handling ... 71

13. Namespaces ... 72
13.1 Namespace definition ... 72

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

13.1.1 Unnamed namespaces ... 72
13.1.2 Namespace member definitions ... 72
13.2 Namespace alias ... 72
13.3 The using declaration ... 72
13.4 Using directive ... 73

14. Classes ... 74
14.1 Class names ... 74
14.2 Class members ... 74
14.3 Member functions ... 74
14.3.1 Non-static member functions ... 74
14.3.2 The this pointer ... 74
14.4 Static members ... 74
14.4.1 Static member functions ... 74
14.4.2 Static data members ... 74
14.5 Unions ... 74
14.6 Bit-fields ... 74
14.7 Nested class declarations ... 75
14.8 Local class declarations ... 75
14.9 Nested type names ... 75
14.10 Derived classes ... 75
14.10.1 Multiple base classes ... 75
14.10.2 Member name lookup ... 75
14.10.3 Virtual functions ... 75
14.10.4 Abstract classes ... 75
14.11 Member access control ... 75
14.11.1 Access specifiers ... 75
14.11.2 Accessibility of base classes and base class members ... 75
14.11.3 Friends ... 75
14.11.4 Protected member access ... 75
14.11.5 Access to virtual functions ... 76
14.11.6 Multiple access ... 76
14.11.7 Nested classes ... 76
14.12 Special member functions ... 76
14.12.1 Constructors ... 76
14.12.2 Temporary objects ... 76
14.12.3 Conversions ... 76
14.12.4 Destructors ... 76
14.12.5 Free store ... 76
14.12.6 Initialization ... 76
14.12.7 Copying and moving class objects ... 77

15. Overloading ... 78
15.1 Overloadable declarations ... 78
15.2 Declaration matching ... 78
15.3 Overload resolution ... 78
15.3.1 Candidate functions and argument lists ... 78
15.3.2 Viable functions ... 79
15.3.3 Best viable function ... 79
15.4 Address of overloaded function ... 79
15.5 Overloaded operators ... 79
15.5.1 Unary operators ... 79
15.5.2 Binary operators ... 79
15.5.3 Assignment ... 79
15.5.4 Function call ... 80

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

15.5.5 Subscripting ... 80
15.5.6 Class member access ... 80
15.5.7 Increment and decrement ... 80
15.6 Built-in operators ... 80

16. Templates ... 81
16.1 Template parameters ... 81
16.2 Names of template specializations ... 81
16.3 Template arguments ... 81
16.3.1 Template type arguments ... 81
16.3.2 Template non-type arguments ... 81
16.3.3 Template template arguments ... 81
16.4 Type equivalence ... 81
16.5 Template declarations ... 81
16.5.1 Class templates ... 81
16.5.2 Member templates ... 82
16.5.3 Variadic templates ... 82
16.5.4 Friends ... 82
16.5.5 Class template partial specializations ... 82
16.5.6 Function templates ... 82
16.5.7 Alias templates ... 82
16.6 Name resolution ... 82
16.6.1 Locally declared names ... 83
16.6.2 Dependent names ... 83
16.6.3 Non-dependent names ... 83
16.6.4 Dependent name resolution ... 83
16.6.5 Friend names declared within a class template ... 83
16.7 Template instantiation and specialization ... 83
16.7.1 Implicit instantiation ... 83
16.7.2 Explicit instantiation ... 84
16.7.3 Explicit specialization ... 84
16.8 Function template specializations ... 84
16.8.1 Explicit template argument specification ... 84
16.8.2 Template argument deduction ... 84
16.8.3 Overload resolution ... 84
16.9 Deduction guides ... 84

17. Libraries ... 85
17.1 Standard C and C++ libraries ... 85
17.1.1 Standard console I/O ... 85
17.1.2 Standard file streaming ... 85
17.1.3 Type traits ... 85
17.1.4 Standard C++ array ... 85
17.2 SystemC functions and types ... 87
17.2.1 Tracing ... 87
17.2.2 `set_stack_size` ... 87
17.2.3 `sc_gen_unique_name` ... 87
17.2.4 `before_end_of_elaboration` and `end_of_elaboration` ... 87
17.2.5 `dont_initialize` ... 87
17.2.6 `next_trigger` ... 87
17.2.7 `print` and `dump` ... 87
17.2.8 `kind` ... 87
17.2.9 `sc_report_handler` ... 87

18. The SystemC HLS Library ... 89

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

18.1 Overview 89
18.2 Simulation and Synthesis Views 90
18.3 Message Passing with Flow Control 91
18.4 Message Passing without Back-Pressure 93
18.5 Rendezvous Synchronization 94
18.6 FIFO 95
18.7 Direct Port Access 98
18.8 Simulation and Verification Features 99
18.8.1 Tracing 99
18.8.2 Transaction Recording 99
18.8.3 Latency and Capacity Back-Annotation 100
18.8.4 Random Stall Injection 100
18.9 TLM Fast-Simulation View 100
18.10 Clock Management 101
18.11 Diagnostics 101

Annex A: References 103

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

1. Overview

1.1 Scope

This standard defines the following aspects of high-level synthesis of SystemC:

a) The minimum ISO C++17 [1] and IEEE Std 1666 [2] SystemC syntax subset to be supported by HLS tools

b) Standard input code structures to generate specific RTL structures

c) Expected output structures in the generated RTL code

d) A library of communication and synchronization components for HLS-based designs

The intent of this version of the standard is to describe a minimum initial subset which can be supported by tools. It is not meant to restrict synthesis support for syntax beyond this subset, including additional vendor-specific design/library components.

1.2 Purpose

In this standard document, the Synthesis Working Group (SWG) of the Accellera Systems Initiative has defined a subset of SystemC that is appropriate for synthesis. This is intended to be useful for hardware designers to accelerate the modeling process with SystemC and for EDA tool developers to develop SystemC compliant high-level synthesis (HLS) tools. The synthesizable subset of SystemC will be defined within the existing ISO C++ and IEEE Std 1666 SystemC specifications. Hence, familiarity with these standards is presumed.

There are a wide variety of resources available to assist users in modeling with SystemC, including the IEEE Std 1666 Language Reference Manual (LRM), user guides of the HLS tools, and a number of books on SystemC modeling. This document is intended to fill a gap by defining a standard for creating synthesizable hardware description in SystemC, allowing a smooth transition between abstract modeling in SystemC and synthesizable description. Familiarity with hardware design and HLS may also be helpful in understanding this standard.

1.3 Word usage

The word *shall* indicates mandatory requirements strictly to be followed to conform to the standard and from which no deviation is permitted (*shall* equals *is required to*; *shall not* equals *is not permitted to*).

The word *should* indicates a certain course of action is preferred, but is not a mandatory requirement; or (in the negative form, *should not*) that a certain course of action is permitted, but such usage is discouraged (*should* equals *is recommended that*).

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

The word *application* is a C++ program written by an end user that uses the SystemC, TLM-1, and TLM-2.0 class libraries, i.e., it uses classes, functions, or macros defined in the SystemC LRM.

The word *implementation* means any specific implementation of the full SystemC, TLM-1, and TLM-2.0 class libraries, as defined in the SystemC LRM, of which only the public interface need be exposed to the application.

The word *design* is used to mean a SystemC design written by an end user that describes hardware in conformance with this standard.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

A synthesis tool is said to *accept* a SystemC construct if it allows that construct to be a legal input; it is said to *interpret* the construct (or to provide an *interpretation* of the construct) if it accepts that construct and produces a corresponding synthesis result.

The word *synthesis tool* is used to mean a high-level synthesis tool that *accepts* and *interprets* a *design* in conformance with this standard.

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

The term *class* is used to cover the C++ keywords *class* or *struct*.

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

A *synthesis refinement* in this document imposes a restriction or alteration upon some other standard (e.g., the SystemC LRM or the ISOC++ standard) in order to subset the otherwise supported methods to describe behavior to a set that can be implemented in hardware.

The word *deprecated* is used to describe a feature that is superseded by a better, safer, easier to use alternative. Use of the *deprecated* feature is strongly discouraged and future standards *may* make such use illegal.

1.3.1 Construct categories

The constructs in this standard *shall* be categorized as:

Supported
A *synthesis tool* *shall* interpret the construct.

Ignored
A *synthesis tool* *shall* accept the construct, but *may* choose not to interpret the construct. The mechanism, if any, by which a *synthesis tool* notifies (warns) the user of such constructs is not defined in this standard.

Not Supported
A *synthesis tool* *may* choose not to accept the construct. The behavior of the *synthesis tool* upon encountering such a construct is not defined in this standard. For example, a *synthesis tool* *may* choose to fail upon encountering such a construct; alternatively, it *may* choose to accept/interpret such a construct. Even if a *synthesis tool* accepts/interprets some of the constructs that are ***Not Supported***, a *design* that uses such constructs runs the risk of losing portability. However, it is also possible that, in a future revision of this standard, some of the constructs that are ***Not Supported*** now will be ***Supported***.

Supported with Restrictions
A *synthesis tool* *shall* interpret the construct with certain restrictions. This means, instances of the construct which are within the restrictions, as set forth by this standard, are ***Supported***; while instances that violate these restrictions are either ***Ignored*** or ***Not Supported***. Unless explicitly categorized as ***Ignored***, the violating instances are ***Not Supported***. A construct that is ***Supported with Restrictions*** is also said to have ***Restricted Support***.

1.4 Conventions

This document uses the following conventions:

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

1) The body of the text of this standard uses *italics* to denote SystemC or C++ reserved words (e.g., *sensitive* and *SC_MODULE*).

2) The body of the text of this standard also uses *italics underlined* to highlight definitions (e.g., *application* or *design*) or to visually reinforce key terms (e.g., *shall*, *should not*, and *synthesis refinement*).

3) The body of the text of this standard uses ***bold italics*** to visually reinforce construct categories (e.g., ***Supported*** and ***Ignored***).

4) The text of the SystemC examples and code fragments is represented in a fixed-width font.

5) A note provides an informative expansion of certain key concepts. Notes are intended to assist in understanding of a construct, but are not intended as a restriction or enhancement of a *synthesis tool*.

6) The examples that appear in this document under "*Example:*" are for the sole purpose of demonstrating the syntax and semantics of SystemC for synthesis. It is not the intent of this standard to demonstrate, recommend, or emphasize coding styles that are more (or less) efficient in generating an equivalent hardware representation. In addition, it is not the intent of this standard to present examples that represent a compliance test suite or a performance benchmark, even though these examples are compliant to this standard (except as noted otherwise).

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

2. Normative references

2.1 Standard baseline versions

2.1.1 Reference documents

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

- ISO/IEC 14882:2017 C++ Standard (C++17) [1]
  - available at https://webstore.ansi.org/standards/iso/isoiec148822017
- IEEE Standard 1666-2023 for Standard SystemC Language Reference Manual [2]
  - available at https://ieeexplore.ieee.org/servlet/opac?punumber=10246123

The *ISO/IEC 14882:2017 C++ Standard* is hereinafter called *ISOC++* in this document.

The *IEEE Standard 1666-2023 for Standard SystemC* is hereinafter called *SystemC LRM* in this document.

The *IEEE Standard 1666-2011 for Standard SystemC Language Reference Manual* is available from https://ieeexplore.ieee.org/servlet/opac?punumber=6581814

2.1.2 Other materials

The website for the C++ Standard is https://isocpp.org/std/the-standard

C++ standard workgroup webpages documenting changes:

- from C++11 to C++14: https://isocpp.org/files/papers/p1319r0.html
- from C++14 to C++17: https://isocpp.org/files/papers/p0636r0.html

The *IEEE Standard 1666-2023/Cor1-2025 for Standard SystemC Language, Reference Manual - Corrigendum 1* is available from https://ieeexplore.ieee.org/servlet/opac?punumber=10932725

Informal useful references from the CPP Reference website:

- C++14 https://en.cppreference.com/w/cpp/14
- C++17 https://en.cppreference.com/w/cpp/17

2.2 ISOC++ implementation compliance

ISOC++ 4.1 (Implementation compliance) applies to the synthesizable subset defined in this document.

2.2.1 Implementation-defined behavior

ISOC++ 3.12

Simulation of the *design* is based on C++ compilers on specific computer platforms. In most cases, such platforms have converged on certain default implementation-defined behaviors and *synthesis tools* *shall* either adhere to the implementation-defined behavior of the C++ compiler used for simulation of the *design* or issue a warning that a different implementation-defined behavior is being followed.

An example of an implementation-defined behavior is the bit-width of fundamental integer types.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

2.2.2 Undefined behavior

ISOC++ 3.27

The presence of undefined behavior *may* lead to differences between simulation and synthesis and differences in results from different *synthesis tools*. Unless this standard provides a *synthesis refinement* that provides a definition, such undefined behavior is ***Not Supported***.

In some cases, undefined behavior is assumed to be not present in the *design*. One specific example is division by zero (0). *Synthesis tools* *may* assume that such a condition will not be present during simulation of the *design* and treat such a condition as a don't care.

2.2.3 Unspecified behavior

ISOC++ 3.28

The presence of unspecified behavior *may* lead to differences between simulation and synthesis results and differences in results from different *synthesis tools*.

An example is the order of evaluation of arguments to a function is not specified. In ISOC++, the implementation is not required to document which behavior occurs.

2.3 SystemC LRM compliance

This section describes some general terms that are used in the SystemC LRM that have similarities with terms used in ISOC++.

2.3.1 Implementation-defined

SystemC LRM 3.2.1

In general, most of the implementation-defined items in the SystemC LRM do not affect synthesis. One item that does affect synthesis is the type for limited precision (SystemC LRM 7). In that case, this standard only supports one implementation as specified in Section 8.3 on SystemC datatypes.

2.3.2 Undefined

The description in Section 2.2.2 applies when the word *undefined* is used in the context of the behavior not being defined.

2.3.3 Error

The word *error* is used in the SystemC LRM (3.3.6) to mean that the implementation is required to report SC_ERROR through the report handler, normally during elaboration or simulation. This differs from *undefined*, where an implementation may or may not issue an error or a warning. *Synthesis tools* assume that such conditions will not be present and treat such condition as a don't care, unless the *synthesis tools* are able to prove that the error is present, in which case they *may* issue an error appropriately.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

3. Definitions, acronyms, and abbreviations

Intentionally left blank for this public review release.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

4. Translation units

4.1 Translation units and their analysis

The text of a *design* is, as described by ISOC++ Sections 5.1 and 5.2, kept in units called source files. A source file together with all included sources, less any lines skipped by conditional inclusion directives (ISOC++ 19.1), is known as a translation unit.

A translation unit *shall* be specified using the basic source character set, as described by ISOC++ Section 5.3. Alternative tokens (ISOC++ Section 5.5) are ***Not Supported***. Furthermore, instance names using Universal-Character-Name as described in ISOC++ Sections 5.3 and 5.10 are *synthesis tool*-dependent and, therefore, ***Not Supported***.

Using multiple translation units to describe a *design* is ***Supported***. This standard, however, does not specify how a user will provide the translation-unit information to the *synthesis tool*.

The use of the keyword *extern* to refer to a declaration in a different scope within the same translation unit or in a different translation unit is ***Supported***. However, the use of libraries (pre-compiled binary files) is ***Not Supported***, and a *design* *shall* contain all the source files required for synthesis (with the exception of SystemC implementation and C/C++ native library files).

The use of *extern* string-literals for linkage specification (e.g., extern “C”, extern “C++”) is ***Not Supported*** (ISOC++ Section 10.5).

The functions *main* and *sc_main* are ***Ignored***.

4.2 Pre-processing directives

The full set of C/C++ preprocessing directives is ***Supported*** (refer to ISOC++ Clause 19).

The `__has_include` preprocessor expression (ISOC++ 19.1) is ***Supported***. This allows conditionally including headers based on their availability.

A *synthesis tool* *shall* recognize pragma directives (#pragma). It *may* ignore or process pragma directives.

A *synthesis tool* *shall* predefine the following macro names:

Macro name	Description
__STDC__	The value is implementation-dependent.
__cplusplus	The value is 201703L (ISOC++ 19.8).
SC_SYNTHESIS	The version of the synthesis subset, in a year and month format such as 202611L.
__SYNTHESIS__	The value is implementation-dependent.
SC_HLS_DEFAULT_SYNTH_VIEW_TAG	The synthesis view tag (see 18.2).

Note:

The expected use of the __SYNTHESIS__ and SC_SYNTHESIS macros is to guard non-synthesizable code.

The version contained in SC_SYNTHESIS is meant to select different versions of the same behavior that

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

depend on updates to the synthesis subset.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

5. Modules

This section specifies the SystemC modules used in a *design*. In this document `sc_module` and module are used interchangeably. A module models a hardware block and a system is described by creating a hierarchy of submodules stemming from a parent module.

This section does not limit the optimizations that can be performed on the model, the scope of which depends on the *synthesis tool* itself. Irrespective of the level of optimizations performed, the result of the synthesis *shall* have the same functionality as the input model, but *may* have different sequential timed behavior.

5.1 Module definitions

A SystemC module, as defined in the SystemC LRM, *may* contain the port-level interfaces, any required internal storage elements, and any required behavior for that module.

The ***Supported*** possibilities for module definition are as follows.

- Use of the `SC_MODULE` macro;
- Derivation of a *class* from `sc_module`.

In addition to the above, templated module definition has ***Restricted Support***.

The restriction is that, for the top-level modules, only modules that have been instantiated (as defined in ISOC++ 17.7.1 and 17.7.2) or have been explicitly specialized (as defined in ISOC++ 17.7.3) are ***Supported***.

Template specialization and partial-specialization of modules is also ***Supported***.

Note:

In C++, template classes and functions are not instantiated until they are instantiated either

- implicitly, as defined in ISOC++ 17.7.1, or
- explicitly, as defined in ISOC++ 17.7.2.

In C++, template classes and functions can be explicitly specialized (as defined in ISOC++ 17.7.3) using the `template<>` declaration (i.e., the class is fully specialized).

Examples of explicit instantiation and explicit specialization are shown below. The most common form for synthesis is the use of implicit instantiation for submodules. Note that explicit specializations are less common and only indicated when there is an advantage in providing a specialized version of the module (the example below is not an illustration of such a case).

```
template <int N>
SC_MODULE(design) {
    sc_in <int>   SC_NAMED(a);
    sc_out<int>   SC_NAMED(c);

    void add() { c = a + N;}

    SC_CTOR(design) {
      SC_METHOD(add);
      sensitive << a;
    }
};
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

```
// EXPLICIT INSTANTIATION:
template class design<5>;

template<>
SC_MODULE(design<3>) // EXPLICIT SPECIALIZATION
{
    sc_in <int> SC_NAMED(a);
    sc_out<int> SC_NAMED(c);

    void add() { c = a + 3; }

    SC_CTOR(design) {
        SC_METHOD(add);
        sensitive << a;
    }
};
```

5.2 Data members for hardware interface

A module's member specification consists of a set of declarations and definitions. These data members are classified into **interface data members** and **non-interface data members**.

Interface data members define the externally visible interface of a module and are used to transfer data into and out of the module.

- **Ports**: SystemC ports, declared using the `sc_in` and `sc_out` constructs, describe the signal-level interface at the module boundary. The bidirectional port `sc_inout` is ***Not Supported***, see Section 7.2.
- **Signals**: Signals, declared using the `sc_signal` construct, enable communication between processes within a module and between modules through their ports.
- **Memories**: Modules can also communicate via shared memories. A memory shared between modules is modeled as an array passed to the constructors of the communicating modules.

All these mechanisms are ***Supported with Restrictions***, as detailed in Section 7. Any other type of data member used for inter-process or inter-module communication is ***Not Supported***.

Non-interface data members are ***Supported with Restrictions*** as specified in Sections 9, 10 and 14.

5.3 Member functions

Module member functions are ***Supported with Restrictions*** as described in Section 6 for processes and Section 10 for all other functions.

5.4 Constructors

Module constructors are ***Supported*** through the use of

- the `SC_CTOR` macro, or
- a constructor function with a `const sc_module_name&` argument, or
- a constructor function with a `const sc_module_name&` argument and array references.

Every module declaration *shall* contain at least one declaration or definition of a constructor member function.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

The module constructor is used for design elaboration, to instantiate and connect the modules, start and initialize the processes.

To that end, within the constructor of an `SC_MODULE` the following is ***Supported***:

a) Constructor calls for interface ports and internal signals `sc_in`, `sc_out` and `sc_signal` in the constructor initializer list
b) Constructor calls for `SC_MODULE` sub-module hierarchy:
   — initialization of sub-module data members in the constructor initializer list
   — initialization of sub-module pointer data members using the `new` operator
c) Port bindings between the sub-modules ports and signals or top-module ports, using their `bind()` or `operator()` functions
d) Creation of processes using `SC_THREAD`, `SC_CTHREAD` and `SC_METHOD`, and sensitivity specification for these processes, as described in Section 6.
e) Initialization of constant data members in the constructor initializer list.
f) Initialization of array pointer or references in the constructor initializer list.

Within the constructor body, control-flow statements are ***Supported with Restrictions***: the expressions used in decisions and bounds *shall* be constant expressions. Function calls within a constructor are ***Supported with Restrictions***, as described in Section 11.2. An additional constraint applies: the body of any function called from within a constructor *shall* be inlined at the call site.

Within the constructor body, the following is ***Not Supported***:

— writing to data members, other than for the initialization explicitly specified above
— calls to `wait()` statement
— calls to the `read()` or `write()` functions of a port or signal.

Once the design structure is elaborated, in a hardware sense, the design structure shall never change. To that end, overwriting pointers or references to `sc_module` or memory array variables after constructor initialization is ***Not Supported***.

Note:

Initialization of an `SC_MODULE` data member should be performed within the reset portion of an `SC_THREAD`, `SC_CTHREAD`, or sequential `SC_METHOD` in order to have the initialization be performed at reset time.

5.5 Ignored member functions

The following member functions are ***Ignored***:

— `void set_stack_size( size_t );`
— `virtual void start_of_simulation();`
— `virtual void end_of_simulation();`

Additionally, all module destructors are ***Ignored***.

5.6 Not Supported member functions

The following functions are ***Not Supported*** by the synthesis tool:

— `virtual void before_end_of_elaboration();`
— `virtual void end_of_elaboration();`

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

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

The copy constructor and the assignment operator (ISOC++ 15.8) are disabled for `sc_module` by the SystemC LRM 5.2.2 and are therefore ***Not Supported***.

5.7 Port bindings

Port binding, as described in SystemC LRM 4.2.4 is ***Supported***.

Calling the binding function for a port that is a data member of a module, from the constructor of the parent module, is ***Supported***.

Note that positional binding (as described in SystemC LRM 5.2.19) is ***Not Supported***.

5.8 Deriving modules

Deriving modules, as defined in SystemC LRM 5.2.3, is ***Supported***.

Example:

```
// Deriving a module:
SC_MODULE(BaseModule) {
    sc_in<bool>    SC_NAMED(reset);
    sc_in<bool>    SC_NAMED(clock);
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

```
    BaseModule(const sc_module_name& n) : sc_module(n) {}
};

class DerivedModule : public BaseModule {
    void newProcess() { /* ... */ }

public:
    DerivedModule(const sc_module_name& n) : BaseModule(n)
    {
        SC_CTHREAD(newProcess, this->clock.pos());
        reset_signal_is(this->reset, true);
    }
};
```

5.9 Selecting the top of a design hierarchy

The mechanism for specifying the top-level module(s) of a *design* is *synthesis tool* dependent and is not defined in this standard.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

6. Processes

The use of the *SC_THREAD*, *SC_CTHREAD*, and *SC_METHOD* constructs to create processes is ***Supported with Restrictions***.

The use of *sc_spawn()* to create processes is ***Not Supported***.

None of the uses of *sc_process_handle* are synthesizable, hence such usage is ***Not Supported***.

Consequently, the use of *suspend()*, *resume()*, *enable()*, *disable()*, *kill()*, *reset()*, and *throw_it()* is also ***Not Supported***.

6.1 SC_METHOD

In a SystemC *design*, the body of an *SC_METHOD* is executed whenever its sensitivity condition is met. The logic inferred by a *synthesis tool* for an *SC_METHOD* depends on both the sensitivity condition and the form of the *SC_METHOD* body.

As a *synthesis refinement*, all paths through an *SC_METHOD* *shall* either be combinational or sequential. An *SC_METHOD* with a mix of combinational and sequential paths is ***Not Supported***.

For *SC_METHOD*, *reset_signal_is* and *async_reset_signal_is* are ***Not Supported***. Modeling of reset behavior is done using the coding styles described below.

6.1.1 Combinational SC_METHOD

An *SC_METHOD* can be used to describe combinational logic, with the *SC_METHOD* sensitive to any change in the signals in its sensitivity list.

As a *synthesis refinement*, the sensitivity list here *shall* be static and include all signals that are read in the body of the method. This is done to avoid accidental latching, which would lead to synthesis and simulation mismatches.

Example of combinational SC_METHOD:

```
SC_MODULE(design) {
    sc_in <int> SC_NAMED(a);
    sc_in <int> SC_NAMED(b);
    sc_in <int> SC_NAMED(c);
    sc_out<int> SC_NAMED(f);

    SC_CTOR(design) {
        SC_METHOD(comb);
        sensitive << a << b << c;
    }

    void comb() {
        f.write(a.read() + b.read() + c.read());
    }
};
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

6.1.2 Sequential SC_METHOD

The body of a sequential *SC_METHOD* *shall* consist of two mutually exclusive branches based on the clock and reset conditions.

6.1.2.1 Sequential SC_METHOD with synchronous reset

A sequential *SC_METHOD* with synchronous reset *shall* be sensitive to only a positive or negative edge of a clock signal.

Example of sequential SC_METHOD with synchronous reset:

```
SC_MODULE(design) {
    sc_in <bool>    SC_NAMED(clk);
    sc_in <bool>    SC_NAMED(rst);
    ...

    SC_CTOR(design) {
        SC_METHOD(dff);
        sensitive << clk.pos();
    }

    void dff() {
        if (rst.read() == 0) {
            ... // reset signals
        } else {
            ... // assign signals
        }
    }
};
```

6.1.2.2 Sequential SC_METHOD with asynchronous reset

A sequential *SC_METHOD* with asynchronous reset *shall* be sensitive to only a positive or negative edge of a clock signal and a positive or negative edge of a reset signal.

Example of sequential SC_METHOD with asynchronous reset:

```
SC_MODULE(design) {
    sc_in <bool>    SC_NAMED(clk);
    sc_in <bool>    SC_NAMED(rst);
    ...

    SC_CTOR(design) {
        SC_METHOD(dff);
        sensitive << clk.pos() << rst.neg();
    }

    void dff() {
        if (rst.read() == 0) {
            ... // reset signals
        } else {
            ... // assign signals
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

```
        }
    }
};
```

6.2 SC_THREAD and SC_CTHREAD

SystemC allows the usage of the *SC_THREAD* and *SC_CTHREAD* macros to create unspawned processes which run from the start of simulation and are expected to run until the end of simulation (see Section 6.2.2).

SC_CTHREAD and *SC_THREAD* are ***Supported with Restrictions***. They are ***Supported***, as specified in the SystemC LRM, when the following restrictions regarding the process sensitivity and process body are met (in addition to the other restrictions mentioned throughout this standard, e.g., regarding break/goto statements).

Note:

In earlier versions of SystemC, *SC_THREAD* and *SC_CTHREAD* differed in their reset behavior.

Currently, the two differ in how the clock is specified and in the forms of *wait()* allowed by the SystemC LRM, but for synthesis purposes, they have the same expressiveness.

6.2.1 Clock and reset

The sensitivity of the process *shall* be specified in the constructor of the *SC_MODULE* enclosing the process. A process *shall* be statically sensitive to exactly one clock edge and *shall* have at least one reset specification.

A process *shall* have at most one synchronous reset specification and at most one asynchronous reset specification.

Different *SC_(C)THREAD*s *may* be sensitive to different clocks and resets, i.e., an *SC_MODULE* *may* contain multiple clocks.

The use of *reset_signal_is* and *async_reset_signal_is* to specify reset sensitivity is ***Supported***.

Example of SC_THREAD:

```
SC_MODULE(design) {
    sc_in <bool>    SC_NAMED(clk);
    sc_in <bool>    SC_NAMED(rst);
    ...

    SC_CTOR(design) {
        SC_THREAD(thread_process);
        sensitive << clk.pos();
        async_reset_signal_is(rst, false); // active low asynchronous reset
    }

    void thread_process() {
        ...
    }
};
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

Example of SC_CTHREAD:

```
SC_MODULE(design) {
    sc_in <bool>    SC_NAMED(clk);
    sc_in <bool>    SC_NAMED(rst);
    ...

    SC_CTOR(design) {
        SC_CTHREAD(thread_process, clk.pos());
        async_reset_signal_is(rst, false); // active low asynchronous reset
    }

    void thread_process() {
        ...
    }
};
```

6.2.2 Thread process body

In a SystemC *application*, it is a common coding idiom to include an infinite loop containing a call to the *wait()* function within a thread process in order to prevent the process from terminating prematurely, at the same time allowing co-operative scheduling during simulation by suspending the process. The thread process body of a *design* *shall* also follow this structure. Among the available constructs to suspend a process, a *design* *shall* only use a call to *wait()*, where the wait condition is the clock edge to which the thread process is sensitive, as specified in the module constructor. That is, the only ***Supported*** waits are the following.

— *wait()*
— *wait(int)*: where the integer argument is statically determinable.

Other forms of suspending a thread process are ***Not Supported***.

The behavior of a thread process consists of reset behavior, which executes on reset, and operational behavior, which executes after reset.

The body of the thread process in a *design* *shall* follow the form:

```
    [<optional reset behavior>]
    [<optional operational behavior>]
    <infinite loop> subject to the restrictions below
```

The reset and operational behavior *shall* be separated by a call to the *wait()* function. Any behavior encountered in the thread process prior to encountering the first call to the *wait()* function *shall* be considered by the *synthesis tool* as reset behavior. Any behavior encountered in the thread process after encountering the first call to the *wait()* function *shall* be considered by the *synthesis tool* as operational behavior.

The first call to the *wait()* function *may* occur before the infinite loop, or it *may* occur within the infinite loop. In the case where the first call to the *wait()* function occurs within the infinite loop, the behavior within the loop body prior to the first call to the *wait()* function *shall* be considered by the *synthesis tool* to be both reset behavior and operational behavior.

Multiple calls to the *wait()* function *may* occur within the body of the thread process, both before the infinite loop, and within the body of the infinite loop. The behavior encountered before the first call to the *wait()*

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

function *shall* be the reset behavior; any behavior encountered after the first call to the *wait()* function *shall* be the operational behavior.

Within the body of the thread process, the first call to the *wait()* function *shall* be statically determinable. The first call to the *wait()* function *may* be within conditional constructs only when the condition is statically determinable (e.g., in the case where the condition is based on a template parameter). Subsequent calls to the *wait()* function (in the operational behavior) *may* occur within conditional constructs.

Reset values *shall* be statically determinable constants.

The following forms of infinite loop are ***Supported***, others are ***Not Supported***.

```
while (1) { }
```

```
while (true) { }
```

```
do { } while (1);
```

```
do { } while (true);
```

```
for ( ; ; ) { }
```

Example 1 of SC_THREAD/SC_CTHREAD body:

```
void dut::thread_process() {
    // Reset behavior goes here

    wait(); // This wait separates reset behavior from operational behavior.

    // You can have operational behavior here

    while(1) {
        // The infinite loop prevents the operational behavior
        // from terminating.

        // Repeated operational behavior.

        wait(); // This wait allows suspension of the process.
    }
}
```

Example 2 of SC_THREAD/SC_CTHREAD body:

```
void dut::thread_process() {
    // Reset behavior goes here.

    // You cannot have operational behavior here, because there
    // is no wait() to separate them from reset behavior.
    while(1) {
        // The infinite loop prevents the process from terminating.
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

```
            // Behavior here is both reset behavior and operational behavior.

            wait(); // This wait separates reset behavior from  operational
                    // behavior and also allows suspension of the process.

            // Operational behavior.
        }
    }
```

Note:

Reset behavior is behavior that is executed when reset is asserted. As Section 5.4 indicates, writing to data members, signals, or ports of a module from inside the *SC_MODULE* constructor is not reset behavior.

Default constructors of data members of the *SC_MODULE* that initialize the data members (e.g., the default constructor for *sc_int* initializes the value to 0), are not part of the reset behavior. If the behavior of the *design* depends on those non-reset initial values, then synthesis results might differ from simulation results.

Example:

```
class X {
    public:
    X() {
            m_1 = 0;
    }
    ...
    private:
    int m_1;
};

class XChild : public X {
    ...
};

SC_MODULE(Module) {
    sc_signal<X>        SC_NAMED(xSig);
    sc_signal<XChild>   SC_NAMED(xChildSig);
    sc_in_clk           SC_NAMED(clk);
    sc_in<bool>         SC_NAMED(rst);

    SC_CTOR(Module)     // Warning: Synthesis will not invoke default
                        //          constructors for xSig and xChildSig!
    {
        SC_CTHREAD(proc, clk.pos());
        reset_signal_is(rst, true);
    }
    void proc() {
        // Reset clause
        X x_tmp; // OK: Invoke default constructor for X.
        xSig = x_tmp; // OK: Initialize xSig with x_tmp.
        xChildSig = XChild(); // OK: Initialize xChildSig by the  default
                              //     constructor.
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

```
        wait();
        while (true) {
            // Main loop
            ...
        }
    }
};
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

7. Communication: signals, ports, channels and memories

This section specifies how processes can communicate, either within a module or between modules.

Within a module, processes *shall* only communicate with each other through

- data members of type `sc_signal`,
- data members of a struct or class type that contains data members of type `sc_signal`, provided that the processes communicate only by reading and writing those signals, directly or through member functions of the struct or class, or
- *array variables* mapped to *memories* for synthesis.

Between modules, communication *shall* only be through:

- `sc_in` and `sc_out` ports which are to be bound to ports in the parent module or to `sc_signal`s, or
- *array variables* mapped to *memories* for synthesis.

Non-constant variables of other types *shall* be read and written by a single process.

Other cases of access to non-constant module data members from outside the module are ***Not Supported***.

7.1 Basic signals

The `sc_signal` primitive has ***Restricted Support***:

- with the `WRITER_POLICY` defaulted or explicitly set to `SC_ONE_WRITER` is ***Supported***
- with the `SC_MANY_WRITERS` policy is ***Not Supported***.

Furthermore, only the following member functions are ***Supported***.

- `void write( const T& )`
- `const T& read( ) const`
- `operator const T& () const`
- `sc_signal()`
- `sc_signal( const char* )`
- `operator= ( const T& )`
- `operator= ( const sc_signal<T,WRITER_POLICY>& )`

The following resolved signal types are ***Not Supported***:

- `sc_signal_resolved`
- `sc_signal_rv`.

Use of `sc_dt::sc_logic` as the template parameter `T` for `sc_signal<T>` is ***Not Supported***.

Inheriting from a basic signal type is ***Not Supported***.

7.2 Basic ports

The pre-defined specialized port classes are ***Supported*** so blocks in a SystemC hierarchy can communicate through convenient access to member functions of the pre-defined SystemC primitive channels.

Use of `sc_in<T>` and `sc_out<T>` for `T` being any synthesizable type (see Section 8) is ***Supported***.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

Use of `sc_in_clk` is ***Supported***.

Furthermore, for `sc_in<T>`, only the following member functions are ***Supported***.

- `sc_in()`
- `sc_in(const char*)`
- `const T& read( ) const`
- `operator const T& () const`
- `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> & )`

For `sc_in<bool>`, in addition to the above, the following further member functions are ***Supported***.

- `sc_event_finder& pos() const`
- `sc_event_finder& neg() const`

Furthermore, for `sc_out<T>`, only the following member functions are ***Supported***.

- `sc_out()`
- `sc_out(const char*)`
- `const T& read( ) const`
- `operator const T& () const`
- `void write( const T& )`
- `operator= ( const T& )`
- `operator= ( const sc_signal_in_if<T>& )`
- `operator= ( const sc_port< sc_signal_in_if<T>, 1> & )`
- `operator= ( const sc_port< sc_signal_inout_if<T>, 1> & )`
- `operator= ( const sc_out<T> & )`
- `void bind( sc_signal_inout_if<T>& )`
- `void operator() ( sc_signal_inout_if<T>& )`
- `void bind( sc_port_b< sc_signal_inout_if<T> >& )`
- `void operator() ( sc_port_b< sc_signal_inout_if<T> >& )`

For `sc_out<bool>`, in addition to the above, the following further member functions are ***Supported***.

- `sc_event_finder& pos() const`
- `sc_event_finder& neg() const`

Use of `sc_dt::sc_logic` as the template parameter `T` for `sc_in<T>` and `sc_out<T>` is ***Not Supported***.

Use of bidirectional port `sc_inout<T>` is ***Not Supported***.

The following resolved port types are ***Not Supported***:

- `sc_in_resolved`
- `sc_inout_resolved`
- `sc_out_resolved`
- `sc_in_rv`

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

— `sc_inout_rv`

— `sc_out_rv`

Inheriting from a basic port type is ***Not Supported***.

Calling the binding function of a basic port — when that port is a data member of a module — from the constructor of the module's parent is ***Supported***.

7.3 Interface, export and port

The SystemC core language classes — `sc_interface`, `sc_export`, and `sc_port` — are designed to support interface method call channel semantics. This allows ports to invoke member functions of the channel they are bound to.

However, such interface method calls can cross module boundaries without well-defined rules for translation to pin-level interfaces, which complicates synthesis.

Within a design, for any `sc_module`, the following rules apply:

— Direct or indirect inheritance from `sc_interface` is ***Ignored***.

— Direct or indirect inheritance from `sc_port` or `sc_export` is ***Not Supported***.

— Direct or indirect instantiation of data members of type `sc_port` or `sc_export` — except for the types explicitly allowed in Section 7.2 — is ***Not Supported***.

7.4 Modular ports

The purpose of a modular port is to encapsulate related ports and signals into a single structure, and to optionally provide a convenient API that encapsulates a particular I/O protocol, and hierarchical bind functions.

Modular ports are ***Supported with Restrictions***.

7.4.1 Syntactic requirements

The syntactic requirements for a modular port are:

— A modular port *shall* be a struct or a class.

— A modular port *may* be derived from `sc_module`.

Example: output port, without base class:

```
template<typename T>
struct msg_out   // no base class
{
    sc_in<bool>     rdy;
    sc_out<bool>    vld;
    sc_out<T>       dat;

    // Default constructor, required for C arrays of modular ports.
    // Delegating constructor, to avoid a stale name pointer.
    msg_out() : msg_out(std::string(sc_gen_unique_name("msg_out")).c_str()) {}

    msg_out(const char* n)
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

```
        : rdy((std::string(n)+"_rdy").c_str())
        , vld((std::string(n)+"_vld").c_str())
        , dat((std::string(n)+"_dat").c_str())
    {}

    // Example of I/O member function:
    void reset() {
        vld.write(0);
    }

    // Example of I/O member function:
    void put(const T& value) {
        // I/O code with protocol details goes here
        ...
    }

    // Hierarchical bind function:
    template<typename CHAN>
    void operator()(CHAN& c) { bind(c); }

    // Hierarchical bind function:
    template <typename CHAN>
    void bind(CHAN& c) {
        rdy(c.rdy);
        vld(c.vld);
        dat(c.dat);
    }
};
```

Example: input port, deriving from `sc_module`

```
template<typename T>
SC_MODULE(msg_in)   // Deriving from SystemC module
{
    sc_out<bool>    SC_NAMED(rdy);
    sc_in<bool>     SC_NAMED(vld);
    sc_in<T>        SC_NAMED(dat);

    SC_CTOR(msg_in) {};

    // Default constructor, required for C arrays of modular ports:
    msg_in() : msg_in(sc_gen_unique_name("msg_in")) {}

    // Example of I/O member function:
    void reset() {
        // protocol reset code here
        rdy.write(0);
    }

    // Example of I/O member function:
    T get() {
        // I/O code with protocol details goes here
        ...
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

```
    }

    // Hierarchical bind function:
    template<typename CHAN>
    void operator()(CHAN& c) { bind(c); }

    // Hierarchical bind function:
    template <typename CHAN>
    void bind(CHAN& c) {
        rdy(c.rdy);
        vld(c.vld);
        dat(c.dat);
    }
};
```

Modular ports are instantiated in parent modules, using the same syntax as for a basic port.

Example: a modular port instantiation:

```
SC_MODULE(ParentModule) {
    sc_in<bool>             SC_NAMED(clk);
    sc_in<bool>             SC_NAMED(rst);
    msg_in<sc_uint<32>>     SC_NAMED(bus_in);
    msg_out<sc_uint<32>>    SC_NAMED(bus_out);
    ...

    SC_CTOR(ParentModule) {
        ...
    }
    ...
};
```

7.4.2 Synthesis rules

Basic ports in the modular ports can be treated in two ways, depending on where they are connected. If they are connected to signals outside the parent module, a corresponding port will appear in the interface of the generated RTL module. Otherwise, if connected to ports of the parent module, e.g. reset or clock ports, then there will be no additional port created.

The following rules describe the expected interface in the generated RTL for modular ports:

- If a basic port in a modular port is not connected to a basic port in the parent module, the *synthesis tool* *shall* inline it in the parent module, and
- for such inlined port, the *synthesis tool* *shall* name it in this form: `"<modular port name>_<basic port name>"`, where:
  - `<modular port name>` is the C++ identifier of the data member that declares the modular port in the parent module, and
  - `<basic port name>` is the C++ identifier of the data member that declares the basic port in the modular port.

The *synthesis tool* *shall* derive these names from the C++ declarations, independently of the names passed to the constructors of the modular port or of its basic ports.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

The example below demonstrates the above port instantiation and naming rules:

```
module ParentModule (
    input wire clk,
    input wire rst,
    output wire bus_in_rdy,
    input wire bus_in_vld,
    input wire [31:0] bus_in_dat,
    input wire bus_out_rdy,
    output wire bus_out_vld,
    output wire [31:0] bus_out_dat
    ...
);
```

7.5 Memories

To implement a memory in a design, the standard approach is to declare an array and apply a synthesis directive to map that array to a memory resource.

Modeling a memory using an array of type `std::array` or a C-style array is ***Supported with Restrictions***.

Such an array *shall not* be allocated dynamically (e.g., using `new`) within the *design*, and the synthesis tool *shall* infer the number and type of read and write ports by analyzing the bodies of the processes that access the array.

The mechanism for designating an array to be implemented as a memory is **tool-specific** and provided through synthesis directives.

7.5.1 Array size and value type

Arrays used to model memories are ***Supported*** for any number of dimensions. The size of each dimension *shall* be a constant expression greater than zero.

For the array element (value) type, *Basic array value types* are defined as any of the following:

- Fundamental integer and Boolean types: `bool`, `char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`
- SystemC bit-accurate and fixed-point types (with valid template parameters): `sc_bv<N>`, `sc_uint<W>`, `sc_biguint<W>`, `sc_int<W>`, `sc_bigint<W>`, `sc_fixed<W, I>`, `sc_ufixed<W, I>`

Memory arrays whose element type is a *Basic array value type* are ***Supported***.

For the purposes of this standard, a **Basic-layout class** is a class/struct whose non-static data members are limited to:

- Scalars of *Basic array value type*,
- Arrays of *Basic array value type* whose extents are compile-time constants,
- Other **Basic-layout classes** (recursively defined as above).

Memory arrays whose element type is a **Basic-layout class** are ***Supported***.

All other element (value) types are ***Not Supported***.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

7.5.2 Memories local to a module

Mapping arrays declared as module data members, or as automatic objects created once per process activation (e.g., before the main `while` loop of an `SC_THREAD`), to a memory is ***Supported***.

Mapping arrays whose lifetime is limited to a single iteration (e.g., declared inside a `while(1)` loop) to a memory is ***Not Supported***.

If multiple processes share a read or write port, the synthesis tool *shall* insert the required memory port multiplexing automatically.

Because the order in which process instances are run within the evaluation phase is implementation-defined, access to a memory shared by different processes, also across modules (see Section 7.5.3), *should* be explicitly synchronized to avoid non-deterministic simulation behavior (see SystemC LRM 4.3.2.3).

Example — Memory used within a module

```
struct MyClass {
    std::array<sc_uint<32>, 64> array_a;
    // ...
};

SC_MODULE(MyBlock) {
    // ...
    std::array<sc_uint<32>, 128> array_b; // map to memory: SUPPORTED
    // ...

    SC_CTOR(MyBlock) {
        SC_THREAD(proc);
        // ...
    }

    void proc() {
        sc_uint<32> array_c[128]; // map to memory: SUPPORTED (allocated once per thread start)
        MyClass     myobj_1;      // map to memory: SUPPORTED for array_a (constructed once here)
        // ...
        wait();
        while (1) {
            // map to memory: NOT SUPPORTED for array_a (constructed each iteration)
            MyClass myobj_2;
            // ...
        }
    }
};
```

7.5.3 Memory shared between modules

Sharing a memory between modules is ***Supported with Restrictions*** and is allowed only when using `std::array`.

The array *shall* be specified as follows:

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

— Modules (or classes instantiated within modules) *shall* maintain references to the array that models the memory.
— The array instance *shall* be passed to each submodule through its constructor, and the constructor *shall* either bind the parameter to a reference data member or forward it to a subobject that maintains the reference.

If multiple modules share a read or write port, the synthesis tool *shall* automatically insert the required memory port multiplexing.

Example: memory shared between modules

```
// Memory: 32-bit words, 128 entries
using mem_t = std::array<sc_dt::sc_uint<32>, 128>;

// -----------------------
// Reader (read-only user)
// -----------------------
SC_MODULE(LeafReader) {
    const mem_t& mem;   // read-only reference to shared memory

    LeafReader(const sc_module_name& n, const mem_t& mem_in)
        : sc_module(n)
        , mem(mem_in) // bind shared memory reference
    { /* ... */
    }

    void proc() {
        // ... read from mem ...
        // auto v = mem[0];
    }
};

// -----------------------
// Writer (write-capable user)
// -----------------------
SC_MODULE(LeafWriter) {
    mem_t& mem; // non-const reference for write access

    LeafWriter(const sc_module_name& n, mem_t& mem_in)
        : sc_module(n)
        , mem(mem_in) // bind shared memory reference
    { /* ... */ }

    void proc() {
        // ... write to mem ...
        // mem[0] = 0x12345678;
    }
};

// ------------------------------------------------------
// Wrapper that forwards the memory reference to a subobject
// (illustrates: "constructor passes memory to a subobject that maintains the reference")
// ------------------------------------------------------
struct Worker {
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

```
    mem_t& mem; // holds the reference
    explicit Worker(mem_t& m) : mem(m) {}

    void do_something() {
        // ... read/write mem ...
        // mem[1] = 0xDEADBEEF;
    }
};

SC_MODULE(LeafWrapper) {
    Worker worker1; // subobject that retains the shared memory reference
    Worker worker2; // subobject that retains the shared memory reference

    LeafWrapper(const sc_module_name& n, mem_t& mem_in)
        : sc_module(n)
        , worker1(mem_in) // forward reference to subobject, single memory port to be inferred
        , worker2(mem_in) // forward reference to subobject, single memory port to be inferred
    { /* ... */ }

    void proc() {
        worker1.do_something();
        worker2.do_something();
    }
};

// -----------------------
// Parent that owns memory
// -----------------------
SC_MODULE(ParentMod) {
    mem_t       mem;       // SINGLE shared memory instance (statically allocated in the design)
    LeafWriter  writer;    // write-capable user
    LeafReader  reader1;   // read-only user
    LeafReader  reader2;   // another read-only user (demonstrates shared read port)
    LeafWrapper wrapper;   // demonstrates forwarding to a subobject

    SC_CTOR(ParentMod)
        : mem{}                      // default-initialize memory array
        , writer("Writer", mem)      // pass memory by reference to submodules
        , reader1("Reader1", mem)    // read-only reference
        , reader2("Reader2", mem)    // another read-only reference (tools may share a read port)
        , wrapper("Wrapper", mem)    // constructor forwards to subobject that holds the reference
    {
        // Synthesis tools:
        //  - infer read/write ports from accesses in submodule processes,
        //  - insert automatic sharing mux if multiple modules share a read or write port.
    }
};
```

7.6 Arrays of signals and ports

Arrays of signals and ports are ***Supported with Restrictions***, either through the use of C arrays or through the use of the `sc_vector` class.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

7.6.1 Syntactic requirements

The following constructs can be used in arrays:

— Arrays of `sc_signal` are ***Supported***.
— Arrays of basic ports are ***Supported***.
— Arrays of modular ports are ***Supported***.

Note: A C array of modular ports requires the modular port to have a default constructor (ISOC++ 11.6). An `sc_vector` of modular ports requires the modular port to be derived from `sc_object`, e.g. from `sc_module`, and to be constructible from a `const char*` name (SystemC LRM 8.5.3). A modular port without a base class can therefore only be used in a C array.

Features of `sc_vector` are outlined below:

— Specification of size through constructor argument is ***Supported***.
— Specification of size through `init()` function in the parent module constructor is ***Supported***.
— The `size()` member function is ***Supported***.
— `operator[]` and `at()` methods are ***Supported***.
— `sc_vector` of `sc_vector` is ***Supported***.

Examples of supported port arrays:

```
SC_MODULE(MyModule) {
    sc_in<int>      in[10];
    sc_out<int>     out[10];
    sc_signal<int>  sig[10];
    sc_vector<sc_in<int>>           SC_NAMED(vec_in);
    sc_vector<sc_out<int>>          vec_out{"vec_out", 10};
    sc_vector<sc_signal<int>>       vec_sig{"vec_sig", 10};
    sc_vector<msg_in<sc_uint<32>>>  bus_in{"bus_in",10};
    msg_out<sc_uint<32>>            bus_out[10];

    SC_CTOR(MyModule) {
        vec_in.init(10);
    }
    ...
};
```

7.6.2 Synthesis rules

For an array of modular ports, either a C array or an `sc_vector`, the *synthesis tool* *shall* apply the rules of Section 7.4.2 to each element, with `<modular port name>` replaced by `<array name>_<index>`, where:

— `<array name>` is the C++ identifier of the data member that declares the array, and
— `<index>` is the index of the element, in decimal.

For a multi-dimensional C array, or an `sc_vector` of `sc_vector`, one `_<index>` *shall* be appended per dimension, outermost dimension first.

The *synthesis tool* *shall* derive these names from the C++ declarations, independently of the names of the elements in simulation.

Note: In simulation, elements default-constructed with `sc_gen_unique_name` get names such as `msg_out_2_rdy`,

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

which reflect neither the array name nor the element index.

The example below shows the RTL interface generated for the modular port arrays `bus_in` and `bus_out` of the example above:

```
module MyModule (
    ...
    output wire bus_in_0_rdy,
    input wire bus_in_0_vld,
    input wire [31:0] bus_in_0_dat,
    ...
    input wire bus_out_9_rdy,
    output wire bus_out_9_vld,
    output wire [31:0] bus_out_9_dat
    ...
);
```

7.7 Unsupported pre-defined SystemC channels

The following pre-defined SystemC channels are ***Not Supported***:

- `sc_buffer`
- `sc_clock`
- `sc_mutex`
- `sc_semaphore`
- `sc_fifo`
- `sc_event_queue`

The core language class `sc_event` is also ***Not Supported***.

Note: While `sc_clock` channel is not supported, clock ports are supported as specified in Section 7.2. An `sc_clock` can be used in the testbench to drive these clock ports (see 18.10).

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

8. Types

SystemC types are comprised of both the native C++ types and the additional SystemC types.

There are two kinds of native C++ types: fundamental types and compound types. Types describe objects, references, or functions.

Note:

Alignment requirements mentioned in ISOC++ 6.11 are not relevant for synthesis. Synthesis may choose alternative data representations for internal objects (not part of the interface of the *design*) provided the I/O behavior of the *design* is unchanged.

For example, the bit-width of an integer variable could be reduced based on the range of the variable or its representation could be changed from two's complement to sign-magnitude.

8.1 Fundamental types

Fundamental types are comprised of integer types, floating-point types, and *void*.

8.1.1 Integer types

The following integer types, as specified in ISOC++ 6.9.1, are ***Supported***:

- *bool*
- *unsigned char*, *signed char*, *char*
- *unsigned short*, *signed short*
- *unsigned int*, *signed int*
- *unsigned long*, *signed long*
- *unsigned long long*, *signed long long*

The integer types *wchar_t, char16_t* and *char32_t* are ***Not Supported***.

Note:

ISOC++ 6.9.1 specifies that it is implementation defined whether a *char* object can hold negative values; GNU G++ can support either mode through use of the "*-fsigned-char*/*-funsigned-char*" switch. The *synthesis tool* vendor is likely to choose the mode that best meets the architecture for which their simulation model was built in order to achieve consistent results between simulation and synthesis models.

ISOC++ 6.9.1 also specifies that plain *char*, *signed char*, and *unsigned char* are three distinct types. Even if a particular implementation allows a *char* to hold negative values, it is not the same type as a *signed char*.

8.1.1.1 Literals

Integer and Boolean literals, as specified in ISOC++ 5.13.2 and 5.13.6, are ***Supported***.

Floating literals, as defined in ISOC++ 5.13.4, have ***Restricted Support***, see Section 8.1.4.

Pointer literals, *nullptr*, as defined in ISOC++ 5.13.7 are ***Supported***.

Character literals as described in ISOC++ 5.13.3 have ***Restricted Support***. The *L* prefix denoting wide character support is ***Not Supported***.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

For synthesis, if the numerical value of the *char* literal has an effect on functionality (the exception being comparing *char*s for equality), characters *shall* be assumed to be encoded in the ASCII character set. This is a *synthesis refinement* over ISOC++ 5.13.3 and 5.13.5, which allows alternative execution character sets (ISOC++ Section 5.3, Paragraph 3).

8.1.1.2 Representation and bit sizes

Two's complement integer representations are ***Supported***. One's complement and sign magnitude integer representations are ***Not Supported***.

A *synthesis tool* *shall* have mechanisms to support the I/O observable behavior implied by bit-widths and representation for the computer platforms as indicated in the last column of the table below. It *shall* warn in case the choice of bit-width or the representation is not consistent with the definition for the computer platform.

Table 1—Bit sizes for integer types (for current compilers)

Integer Type	Relative Requirement	Representation	Bit Width
(un)signed char, char		two's complement	8
(un)signed short	bits(short) ≥ bits(char)	two's complement	16
(un)signed int	bits(int) ≥ bits(short)	two's complement	32
(un)signed long	bits(long) ≥ bits(int)	two's complement	32/64
(un)signed long long	bits(long long) ≥ bits(long)	two's complement	64

Note:

a) The representation and the bit-width of an integer type determines its numerical range and its overflow behavior.

b) The ISOC++ standards set minimum requirements for the bit widths of integer types, but leave bit widths and the representation implementation-dependent.

c) ISOC++ 6.9.1 specifies unsigned integers *shall* obey the laws of arithmetic modulo 2^n, where *n* is the number of bits. Signed integers are of a pure binary numeration system and the representations allowed are two's complement, one's complement, and sign magnitude.

d) Table 1 provides an overview of the ISOC++ 6.9.1 requirements and bits sizes for integer types used on most compilers for popular computer platforms. It constrains the relative sizes of the different integer types, and also requires the signed and unsigned (and plain in the case of characters) versions of integer types to have the same storage. The last column in the table shows the bit widths for current platforms. As the table indicates, there is only a difference for the (un)signed long types in current platforms.

e) ISOC++ provides optional typedefs (ISOC++ 21.4.1) for "exact-width" integer types. These are *intN_t* and *uintN_t*, where *N* can be 8, 16, 32, or 64. The typedefs are defined in the *std* namespace in the include <*cstdint*> header. For example, int64_t is defined as *long* on platforms where *long* is 64-bits wide and as *long long* on platforms where *long* is 32-bits wide.

8.1.2 Type conversions

Type conversions are ***Supported*** as specified in the sections below.

Note:

ISOC++ defines two kinds of conversions between integer types that are applied in the evaluation of

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

expressions: integer promotions and usual arithmetic conversions.

An example of an integer promotion is when a *short* is promoted to an *int* in the unary minus expression "-*a*" (variable "*a*" is of type *short*).

The usual arithmetic conversions are defined by the C++ language to yield a common type for many binary operators that expect operands of arithmetic or enumeration type.

An example of a usual arithmetic conversion is when an operand of type *short* is converted to *long long* in the expression "*a*+*b*" where "*a*" is of type *short* and "*b*" is of type *long long*. In that case, "*a*" is first promoted to type *int* (integer promotion that is performed as part of the usual arithmetic conversion) and then converted to *long long*.

8.1.2.1 Integer promotions

Integer (Integral) Promotions (as defined in ISOC++ 7.6) are ***Supported***.

8.1.2.2 Usual arithmetic conversions

Usual Arithmetic Conversions (as defined in ISOC++ Section 8) are ***Supported***.

8.1.3 Operators

The following operators are ***Supported*** for the integer types:

a) Unary operators (+, -, ~, and !).
b) Arithmetic binary operators (+, - , *, /, and %) and the corresponding assign operators (+=, -=, *=, /=, and %=).
c) Relational and Equality operators (>, >=, <, <=, ==, and !=).
d) Bitwise binary operators (&, |, and ^) and the corresponding assign operators (&=, |=, and ^=).
e) The Conditional Operator (?:).

The following operators have ***Restricted Support***.

a) Shift and shift assign operators (<<, >>, <<=, and >>=). Considering *E1* and *E2* as the two operands (as in "*E1 shift_op E2*"), the support is as outlined below. The support of the assign shift operators (<<=, >>=) follow from the support of the shift operators (<<, >>).
    1) Shifts are ***Supported*** for valid ranges as defined by ISOC++ and ***Not Supported*** otherwise. ISOC++ 8.8 specifies that valid ranges for *E2* are 0 to *length*(*promoted_type*(*E1*))-1 and that otherwise the behavior is undefined (Section 2.2.2).
    2) When *E1* has an unsigned type: both right and left shifts are ***Supported***.
    3) When *E1* has a signed type the shift operators are ***Supported*** with the following *synthesis refinements*:
    — For right shifts, the sign bit *shall* be shifted in. This is a *synthesis refinement* since ISOC++ leaves the behavior implementation-defined when *E1* is negative.
    — For left shifts, ISOC++ specifies cases where the shift is undefined. A *synthesis refinement* on ISOC++ is that *E1 << E2*, for *E1* signed, is defined in all cases to be equivalent to *signed(unsigned(promoted_type(E1)) << E2)*.
b) The prefix and postfix increment and decrement operators (`++x`, `--x`, `x++`, and `x--`) have ***Restricted Support***. The restriction is that the prefix and postfix increment and decrement operators on a *bool* operand are ***Not Supported*** as they are not allowed by ISOC++ (8.2.6 and 8.3.2).

Note:

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

The ~ operator on a *bool* argument x first promotes it to an *int* and then computes the one's complement (value is -x-1) of the promoted value (ISOC++ 8.3.1).

```
bool x = true;
int  y = ~x; // y is -2
bool z = ~x; // z is true
```

If the intended behavior is to get the logical complement, the logical negation operator *!* (ISOC++ 8.3.1) should be used.

The result of division is truncated towards zero.

8.1.4 Floating point types

Floating literals (as specified in ISOC++ 5.13.4) are ***Supported*** for initializing synthesizable datatypes. If the floating literal is used to initialize an integer type (a floating-integral conversion is involved as specified in ISOC++ 7.10) and the truncated value is not representable in the integer type, then the behavior is undefined (see Section 2.2.2).

Otherwise, floating-point types (specified in ISOC++ 6.9.1 as *float*, *double*, and *long double*) are ***Not Supported***.

Note:

One of the challenges for providing general synthesis support for floating-point datatypes is the fact that the bit accurate behavior of floating point arithmetic is dependent on implementations and compiler options used.

8.1.5 The void type

The *void* type (ISOC++ 6.9.1) is ***Supported***.

8.2 Compound types

Compound types in C++, as described in ISOC++ 6.9.2, have ***Restricted Support*** determined by the support restrictions of the constituent types.

8.3 SystemC datatypes

SystemC provides a number of datatypes that are useful for hardware design. These datatypes are implemented as C++ classes.

The following SystemC types have ***Restricted Support*** as described in the sections below:

- Limited Precision Integer Types: *sc_int* and *sc_uint* (Section 8.3.1)
- Finite Precision Integer Types: *sc_bigint* and *sc_biguint* (Section 8.3.2)
- Finite Precision Fixed-point Types: *sc_fixed* and *sc_ufixed* (Section 8.3.3)
- Finite Word-Length Bit Vector Type: *sc_bv* (Section 8.3.4.1)
- Finite Word-Length Logic Vector Types (4-valued): *sc_lv* (Section 8.3.4.2)
- Single Bit Logic (4-valued): *sc_logic* (Section 8.3.4.2)

The explicit use of SystemC datatypes that are not in the list above is ***Not Supported***. The implicit use of

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

other related types that arise as return types from operators on the types listed above has ***Restricted Support***. The restriction is that the bitwidth of a return type needs to be statically determinable.

Note:

a) All datatypes supported for synthesis have vector length/precision that is specified by template parameters. Thus, their vector length/precision is statically determinable during compilation.

b) Underlying classes, such as *sc_signed* and *sc_unsigned*, can appear as the result of expressions on supported types, but are not directly synthesizable. Their length/precision is dynamic. Some base and helper classes are specifically denoted in SystemC LRM 3.2.4 as classes that should not be used explicitly. The SystemC LRM annotates those classes with a superscript dagger (†).

8.3.1 Limited precision integer types

SystemC LRM 7.5.2 and 7.5.3 state that the limited precision *sc_int/sc_uint* *shall* be held in an implementation-dependent native C++ integer which *shall* have a minimum representation size of 64 bits. For synthesis, representation sizes greater than 64-bits are ***Not Supported***.

In summary, the following types have ***Restricted Support***.

— *sc_int*<*W*>: limited precision signed integer ($W \leq 64$).

— *sc_uint*<*W*>: limited precision unsigned integer ($W \leq 64$).

Support restrictions on operators and functions are covered in Section 8.3.5. Many operators are available through implicit conversions to *int_type* and *uint_type* and their support is determined by the support of those native C++ types.

8.3.2 Finite precision integer types

The following Finite Precision Integer types have ***Restricted Support***.

— *sc_bigint*<*W*>: finite precision signed integer.

— *sc_biguint*<*W*>: finite precision unsigned integer.

The support restrictions of common operators and functions are covered in Section 8.3.5.

8.3.3 Finite precision fixed-point types

The fixed-point types *sc_fixed* and *sc_ufixed* have ***Restricted Support*** as listed below.

a) Overflow modes have ***Restricted Support*** as specified in Table 2 below as a function of the two template parameters that determine the overflow mode: *o_mode* and *n_bits*.

b) All Quantization modes are ***Supported*** as specified in Table 3 below as a function of the template parameter that determines the quantization mode: *q_mode*.

c) Common operators/functions: see Section 8.3.5.

d) Specific member functions:

   1) Query of parameters: *wl*, *iwl*, *q_mode*, *o_mode*, and *n_bits* are ***Supported***.

   2) Query of value: *is_neg*, *is_zero*, and *value* are ***Not Supported***.

   3) Other member functions such as *overflow_flag*, *quantization_flag*, *type_params*, and *cast_switch* are ***Not Supported***.

e) Simulation specific functionality such as *sc_fxtype_param*, *sc_fxcast_switch*, and *SC_OFF* are ***Not***

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

Supported.

Table 2—Overflow Modes

Overflow Mode	Parameter (o_mode)	n_bits	Support
Wrap-around Basic (*default*)	*SC_WRAP*	0	***Supported***
Saturation	*SC_SAT*	-	***Supported***
Symmetrical Saturation	*SC_SAT_SYM*	-	***Supported***
Saturation to Zero	*SC_SAT_ZERO*	-	***Supported***
Wrap-around Advanced	*SC_WRAP*	> 0	***Not Supported***
Sign Magnitude Wrap-Around	*SC_WRAP_SM*	≥ 0	***Not Supported***

Table 3—Quantization Modes

Quantization Mode	Parameter (q_mode)	Support
Truncation (*default*)	*SC_TRN*	***Supported***
Rounding to plus Infinity	*SC_RND*	***Supported***
Truncation to zero	*SC_TRN_ZERO*	***Supported***
Rounding to zero	*SC_RND_ZERO*	***Supported***
Rounding to minus infinity	*SC_RND_MIN_INF*	***Supported***
Rounding to infinity	*SC_RND_INF*	***Supported***
Convergent rounding	*SC_RND_CONV*	***Supported***

8.3.4 Logic and vector types

8.3.4.1 Finite word-length bit vectors (sc_bv)

The finite word-length bit vector *sc_bv* has ***Restricted Support***. The restrictions on common operators/functions are outlined in Section 8.3.5.

8.3.4.2 Single-bit logic (sc_logic) and finite word-length logic vectors (sc_lv)

The single-bit logic *sc_logic* and the finite word-length logic vector *sc_lv* have ***Restricted Support***. The restrictions on common operators/functions are outlined in Section 8.3.5.

In addition, the following restrictions apply.

- The unknown logic constant (`sc_logic('X')`, `SC_LOGIC_X`) is ***Not Supported***.
- The high-impedance logic constant (`sc_logic('Z')`, `SC_LOGIC_Z`) is ***Not Supported***.

8.3.5 Common operators and functions

8.3.5.1 Bit select operator

The bit select operators as specified by SystemC LRM 7.2.6 have ***Restricted Support***. The restriction is that the index is within the bounds of the object being accessed.

Note:

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

An out-of-bound access will be treated as an error by synthesis as defined in Section 2.3.3.

The bit select *operator[i]* allows the selection of a bit of a variable either as an *rvalue* or an *lvalue*.

As defined by SystemC LRM 7.5.4.6, *sc_int*/*sc_uint* temporary values cannot have bit-select applied.

As defined by SystemC LRM 7.5.7, 7.6.8, and 7.7, the bit select on concatenations or subreferences also cannot be performed.

Example:

```
sc_int<8>    x;
sc_bigint<8> y;

x[3] = y[2]; // Legal

(x+x)[3] = 0 ; // Illegal, as x+x is promoted to a native C++ type

(y+y)[3] = 0 ; // Legal as y+y is still a sc_bigint

(y,y)[3] = 0 ; // Illegal as concatenation doesn't support bitref
```

8.3.5.2 Part select operator

The part select operators (*operator(l,r)* and *range(l,r)*), as specified by SystemC LRM 7.2.7, have ***Restricted Support***. The first restriction is both the left and the right index positions lie within the bounds of the object. The second restriction is the range of the part select has a statically determinable length. The third restriction is the left hand index is greater or equal to the right hand index.

Out-of-bound access *shall* be treated as an error (Section 2.3.3), as specified in the SystemC LRM.

Note:

The third restriction implies bit-reversal using part selects is not supported. Given the second restriction, the third restriction becomes trivial to check.

The bit-reversal behavior is specified for fixed-point types, as specified in SystemC LRM 7.10.6. It is also implied for vector types in the specification of *reversed* in SystemC LRM 7.9.8.7. Part select is not allowed to reverse bit-order for limited-precision integers, as stated in NOTE1 in SystemC LRM 7.2.7, and for finite-precision integers, as stated in SystemC LRM 7.6.8.3.

Example of statically determinable range length:

```
x(i+5, i+3) = y(k+4, k+2); // range length = 3
```

Note:

As defined by SystemC LRM 7.5.4.6, *sc_int*/*sc_uint* temporary values cannot have part-select applied.

Part select is not available for concatenations and subrefs of integer types. Part select is available for concatenations and subrefs of vector types (*sc_bv* and *sc_lv*).

Example:

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

```
sc_int<8>    x;
sc_bigint<8> y;

x(5,3) = y(4,2); // Legal

(x+x)(5,3) = 0 ; // Illegal: x+x is promoted to native C++ type

(y+y)(5,3) = 0 ; // Legal as y+y is still a sc_bigint

(y,y)(5,3) = 0 ; // Illegal: part select of a concatenation of integers is not allowed
```

The result of a part select cannot be directly assigned to a fixed point variable, but it can be assigned to a range.

8.3.5.3 Function concat(C1,C2) and operator,(C1,C2)

The concatenation function and operator are ***Supported***.

Note:

The SystemC datatype package defines concatenation functionality via a template specialized function *concat(C1,C2)* and *operator,(C1,C2)*.

The concatenation operation (*op1,op2*) may be used as an *rvalue* or an *lvalue*.

Example:

```
(x, y) = (z, w);
```

Because of the difference in return types for operators for *sc_bigint*/*sc_biguint* and *sc_int*/*sc_uint*, using expressions (unless they are cast) may give different results for finite precision integers than for limited precision integers. Using uncast expressions other than concatenation, bit select, and part select as arguments of the concatenate operation is not recommended.

For example, using the Accellera Open Source simulator:

```
sc_int<4>    t  = 1;
sc_bigint<4> tb = 1;
sc_int<4>    x  = 2;
sc_bigint<4> xb = 2;

cout << (t,  t*x)    << endl; // result = 3
cout << (t, tb*xb)   << endl; // result = 258
cout << (t,  x >> 1) << endl; // result = 3
cout << (t, xb >> 1) << endl; // result = 17
```

8.3.5.4 Reduction operators

The reduction operators specified in SystemC LRM 7.2.9 are ***Supported***.

Note:

The reduce operators are: *and_reduce*, *or_reduce*, *xor_reduce*, *nand_reduce*, *nor_reduce*, and *xnor_reduce*.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

The reduction operators are not available for the fixed-point datatypes.

8.3.5.5 Arithmetic operators

The arithmetic operators for limited and finite precision integers, as defined in the SystemC LRM, are ***Supported***.

The arithmetic operators for the finite precision fixed-point types, as defined in the SystemC LRM, have ***Restricted Support***. The implementation-dependent behavior described in SystemC LRM 7.10.7 (variable-precision fixed-point value limits) is ***Not Supported***. A consequence is that the division operator is ***Not Supported***.

Note:

Examples of implementation-dependent behavior described in the SystemC LRM 7.10.7 can be found in the Accellera open source simulator in the form of the following compiler flags:

- *SC_FIXDIV_WL:* bounds the number of bits that are computed for division. Division can require an infinite number of bits to represent.
- *SC_FXMAX_WL*: is the maximum width of a fixed-point type. It is presumed precision is set high enough so it does not change the behavior of the *design*. Synthesis presumes these limits are not present.

The behavior of operators `++` and `--` for fixed-point datatypes, while using the standard increment/decrement by 1, is in effect a NOP whenever *iwl* > *wl* and in some cases of *iwl* < 1.

The arithmetic operators are defined for the numeric types (integer and fixed-point types).

The arithmetic operators include:

- Unary operators + and -
- Binary operators +, -, *, /, and %; assign operators +=, -=, *=, /=, and %=. The operators % and %= are not available for fixed-point types.
- Prefix and Postfix increment and decrement operators `++` and `--`.

8.3.5.6 Bitwise complement operator

The unary bitwise complement operator ~ specified in the SystemC LRM is ***Supported***.

Note:

The unary ~ operator complements its argument bitwise. For vector types, the width of the return type is identical to the width of the operand. This is consistent with the vector types not having an arithmetic view (no signed/unsigned treatment). The integer and fixed-point types (these are referred to in SystemC LRM 7.1 as *numeric types*) are intended to have an arithmetic treatment. In an arithmetic context, ~x is equal to (-x-1). However, the unary operators ~ for the SystemC integer and fixed-point types are not consistent with an arithmetic treatment and are in fact inconsistent amongst themselves:

```
cout << ~((sc_uint<8>)     128) << endl; //  18446744073709551487
cout << ~((sc_biguint<8>)  128) << endl; //  383
cout << ~((sc_ufixed<8,8>) 128) << endl; //  127
cout << ~((unsigned char)  128) << endl; // -129, correct ~x = (-x-1)
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

8.3.5.7 Bitwise logical operators

The binary bitwise logical operators (&, |, and ^) and bitwise assignment operators (&=, |=, and ^=) specified in the SystemC LRM are ***Supported***.

Note:

The binary operations &, |, ^ compute the bitwise and, or and xor operation respectively.

The mixing of signed and unsigned operands is not allowed for fixed-point types (a difference compared to SystemC integer types).

8.3.5.8 Logical negation

The logical negation operator ! specified in the SystemC LRM is ***Supported***.

Note:

The operator ! is available for *sc_int_bitref*, *sc_uint_bitref*, *sc_signed_bitref_r*, and *sc_unsigned_bitref_r*. Using the operator ! on fixed-point types leads to implicit conversions, so there *may* be unexpected truncation involved.

The operator ! is available on limited precision types through implicit conversions to *int_type* (C++ native type).

8.3.5.9 Relational operators

Relational Operators, as defined in the SystemC LRM, are ***Supported***.

Note:

The relational operators compare the two operands as in C++ and return a value of type *bool*. The comparison is done arithmetically for integer and fixed-point types. The relational operators == and != operators are available for vector types.

Mixing signed and unsigned limited precision integers can lead to unexpected results.

Example:

```
(sc_uint<8>) 1 > (sc_int<8>) -1 // incorrectly returns false
```

8.3.5.10 Shift operators

8.3.5.10.1 Limited precision integers

The shift operators (<< and >>) and shift assign operators (<<= and >>=) have ***Restricted Support***. The restriction is the second operand needs to be in the range 0 to 63.

Note:

The restriction is inherited from the implicit conversion to the native C++ 64-bit integers in its implementation (see Section 8.1.3 and SystemC LRM 7.5.4.6). This restriction is also tied to the restriction in Section 8.3.1 on the maximum bitwidth for limited precision integers.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

Outside the range 0 – 63, the behavior is undefined (see Section 2.2.2).

8.3.5.10.2 Finite precision integers

The shift operators (<< and >>) and shift assign operators (<<= and >>=) have ***Restricted Support***. The first restriction is the second operand has to have a non-negative value. The support for the left shift operator (<<) is also restricted to have a statically determinable return bit-width, which implies the value of second operand is statically determinable.

The restrictions above also apply to the said operators applied on the base types (*sc_value_base*, *sc_signed*, and *sc_unsigned*), and the types resulting from part selects (*sc_signed_subref_r*, *sc_signed_subref*, *sc_unsigned_subref_r*, and *sc_unsigned_subref*) and concatenations (*sc_concatref*).

Note: SystemC LRM 7.6.3.7 specifies the behavior is undefined (see Section 2.3.2) if the second operand is negative.

8.3.5.10.3 Finite precision fixed-point types

The shift operators (<< and >>) have ***Restricted Support***. The restriction is the length and integer length of the return type have to be statically determinable, which implies the value of the second operand is statically determinable.

The shift assign operators (<<= and >>=) have ***Restricted Support***. The restriction is the value of the second operand is statically determinable for all cases other than when the first operand has *q_mode=SC_TRN*, *o_mode=SC_WRAP*, and *n_bits=0*. This restriction guarantees no hardware is required for handling rounding and/or saturation.

The restrictions above also apply to the said operators applied on the base types (*sc_fxval*, *sc_fxnum*, *sc_fix*, and *sc_ufix*) and types resulting from part selects (*sc_fxnum_subref*).

Note:

Fixed-point shifts are bidirectional (though this is not explicitly stated in the SystemC LRM) as the second argument is a C++ *int*. The hardware cost can be minimized if the range of the second argument can be reduced. For example, if the range analysis of the *synthesis tool* can determine that the second argument is non-negative, then a unidirectional shift suffices.

8.3.5.10.4 Finite word-length vector types

The shift operators (<< and >>) and shift assign operators (<<= and >>=) have ***Restricted Support***. The first restriction is the second operand has to have a non-negative value. The support for the left shift operator (<<) is also restricted to have a statically determinable return bit-width, which implies the value of second operand is statically determinable.

The restrictions above also apply to the said operators applied on the base types (*sc_lv_base* and *sc_bv_base*), and the types resulting from part selects (*sc_subref_r* and *sc_subref*) and concatenations (*sc_concref_r* and *sc_concref*).

Note:

SystemC LRM 7.9.3.7, 7.9.4.7, and 7.9.8.6 specify it is an error (Section 2.3.3) if the second operand is negative. The SystemC LRM is silent on what happens when the second operand is negative for the classes resulting from concatenation (*sc_concref_r* and *sc_concref*).

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

8.3.5.11 Rotate operators

The rotate operators *lrotate* and *rrotate* have ***Restricted Support***. The first restriction is the argument (the amount by which to rotate) is not negative. The second restriction is the value of the argument is statically determinable.

Note:

The rotate operators are available for the vector types (*sc_lv* and *sc_bv*), their base classes (*sc_lv_base* and *sc_bv_base*), and the types resulting from part select (*sc_subref*) and concatenations (*sc_concref*) of vector classes.

The SystemC LRM does not explicitly define the behavior for negative values.

8.3.5.12 Explicit conversions

Explicit conversions have ***Restricted Support*** as outlined below.

- Conversions *to_int*, *to_uint*, *to_long*, *to_ulong*, *to_int64*, and *to_uint64* are ***Supported***.
- The following conversions are ***Not Supported***:
  - *to_float* and *to_double*;
  - *to_char* (conversion to character for *sc_logic* and bit select for vector types);
  - *to_string* (and its shortcut variants *to_dec*, *to_hex*, *to_oct*, and *to_bin*).
- Conversion *to_bool* is
  - ***Not Supported*** for types *sc_logic* and vector types;
  - ***Supported*** for bit-select for the limited and finite precision integers.
- Member function *is_01* (the SystemC LRM labels this as an explicit conversion function) is ***Not Supported***.
- Member function *value* (for *sc_logic* and vector types) is ***Not Supported***.

8.3.5.13 Conversion operators

The conversion operators have ***Restricted Support***. The restriction is the implicit conversion operator is to a supported type.

Note:

The following conversion operators are defined in the SystemC LRM, along with the classes that define them.

conversion operator	classes where defined
`operator int_type()`	limited precision integers.
`operator uint_type()`	limited precision integers.
`operator uint64()`	*sc_int_bitref_r*, *sc_uint_bitref_r*, *sc_signed_bitref_r*, *sc_unsigned_bitref_r*, and *sc_concatref*.
`operator sc_unsigned()`	*sc_signed_subref_r*, *sc_unsigned_subref_r*, and *sc_concatref*
`operator sc_logic()`	*sc_bitref_r*.
`operator double()`	*sc_fxnum* and *sc_fxval*.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

conversion operator	classes where defined
`operator bool()`	*sc_fxnum_bitref*.
`operator sc_bv_base()`	*sc_fxnum_subref*.

8.3.5.14 Assignment operators and constructors

Assignment operators and constructors have ***Restricted Support*** as described below.

Initialization through String Literals, as specified in SystemC LRM 7.3, has ***Restricted Support***. Initialization of numeric or vector type object with a string literal as the argument of a constructor or assignment operator is ***Supported***, but an initialization through a variable of *char* or *std::string* type is ***Not Supported***. The *sc_numrep* of *SC_NOBASE* (implementation-defined prefix or missing prefix) is ***Not Supported***.

The SystemC LRM introduces the concept of case insensitivity to the prefix and magnitude representations. These are ***Supported***.

Note:

In accordance with the SystemC definition of string literals, SystemC LRM 7.3, a literal representation may be used as the value of a SystemC numeric or vector type object.

It consists of a standard case-insensitive prefix, followed by a magnitude expressed as one or more digits.

sc_numrep	Prefix	Magnitude format
SC_NOBASE	Implementation-defined	Implementation-defined
SC_DEC	0d	Decimal Number, digits 0-9
SC_BIN	0b	Binary Number, digits 0-1
SC_BIN_US	0bus	Binary Unsigned
SC_BIN_SM	0bsm	Binary Sign & Magnitude
SC_OCT	0o	Octal Number, digits 0-7
SC_OCT_US	0ous	Octal Unsigned
SC_OCT_SM	0osm	Octal Signed & Magnitude
SC_HEX	0x	Hexadecimal Number, digits 0-9, a-f, A-F
SC_HEX_US	0xus	Hexadecimal Unsigned
SC_HEX_SM	0xsm	Hexadecimal Sign & Magnitude
SC_CSD	0csd	Canonical Signed Digit

8.3.5.15 String input and output

- ***Not Supported*** for *dump( )*, *print( )*, and *scan( )*. See SystemC LRM 7.2.11.
- ***Not Supported*** for *to_string( )*. See SystemC LRM 7.3.
- ***Not Supported*** for *String Shortcut Methods* (*to_dec( )*, *to_bin()*, and so on). See SystemC LRM 7.10.9.2.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

8.3.5.16 Length

The member function *length* is ***Supported***.

8.3.5.17 Reversed

The member function *reversed* is ***Not Supported***.

Note:

This is consistent with the restriction on bit-reversal in Section 8.3.5.2.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

9. Declarations

Declarations, as defined for ISOC++ Section 10, have ***Restricted Support*** and those specific restrictions and coding guidelines are listed in the subsequent sections and chapters.

The *static_assert* and *using* declarations (ISOC++ Section 10) are ***Supported***.

Note:

ISOC++ allows *static_assert* with a single argument (no message string required).

9.1 Specifiers

Specifiers (ISOC++ 10.1) have ***Restricted Support***.

9.1.1 Storage class specifiers

Storage class specifiers (ISOC++ 10.1.1) have ***Restricted Support***.

a) The *register* storage class specifier was removed in C++17 (the keyword remains reserved) and is ***Not Supported***.
b) The *mutable* specifier is ***Supported***.
c) The *extern* specifier has ***Restricted Support***, as specified in Section 4.1.
d) The *static* specifier has ***Restricted Support***.
    1) Static variables have ***Restricted Support***. Only *const* static variables are supported.
    2) Static functions and static member functions are ***Supported***.
    3) Static data members of *class* have ***Restricted Support***. Only *const* static data members of *class* are ***Supported***.

Note:

The auto keyword used to be a storage class specifier in ISO/IEC 14882:2003 and was Ignored in the 1.4.7 version of the Accellera SystemC Synthesis Standard.

Examples:

```
static class my_class c; // Static non-const variable, Not Supported

static void my_func() { } // Static function definition, Supported

static int var; // Static non-const variable, Not Supported

static const int const_var1 = 5; // Static const variable, Supported
static constexpr int const_var2 = 5; // Static const variable, Supported

struct my_struct {
    static int member; // Non-const, Not Supported
    static const int const_member1;   // Static const variable, Supported
    static constexpr int const_member2 = 5;   // Static const variable, Supported
    static void my_member_func() { } // Static member function, Supported
};
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

9.1.2 Function specifiers

The function specifiers have ***Restricted Support***.

— The *inline* specifier is ***Supported*** as defined in ISOC++ 10.1.6.

— The *explicit* specifier is ***Supported*** as defined in ISOC++ 10.1.2.

— The *virtual* specifier has ***Restricted Support*** as defined in ISOC++ 10.1.2 and 13.3. The restrictions are described in Section 14.10.3.

9.1.3 The typedef specifier

The typedef specifier and alias declarations (`using X = T;`) are ***Supported*** as defined in ISOC++ 10.1.3.

9.1.4 The friend specifier

Supported as defined in ISOC++ 10.1.4.

9.1.5 The constexpr specifier

The *constexpr* specifier (ISOC++ 10.1.5) is ***Supported***.

9.1.6 The inline specifier

The *inline* specifier for variables (ISOC++ 10.1.6) is ***Supported***. Inline variables allow definition of constants in header files without violating the one-definition rule, which is useful for shared compile-time configuration across translation units.

9.1.7 Type specifiers

Type specifiers (ISOC++ 10.1.7) have ***Restricted Support***.

9.1.7.1 cv-qualifiers

The cv-qualifiers (ISOC++ 10.1.7.1) have ***Restricted Support***.

a) The cv-qualifier *const* is ***Supported***.

b) The cv-qualifier *volatile* is ***Not Supported***.

9.1.7.2 Simple type specifiers

Simple type specifiers (ISOC++ 10.1.7.2) have ***Restricted Support***. They are ***Supported*** for types that are ***Supported*** as specified in Sections 8, 14 and 16.

The *decltype* specifier is also ***Supported***.

Support of the *auto* specifier is covered in Section 9.1.7.4.

9.1.7.3 Elaborated type specifiers

Elaborated type specifiers (ISOC++ 10.1.7.3) are ***Supported***.

9.1.7.4 auto specifier

Auto type specifiers (ISOC++ 10.1.7.4) are ***Supported***.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

Note:

ISOC++ updated the deduction rules for *auto* from braced-init-list (ISOC++ 10.1.7.4.1): direct initialization `auto x{1}` deduces the element type, while copy initialization `auto x = {1}` deduces `std::initializer_list`.

9.2 Enumerations

Supported as defined in ISOC++ 10.2.

Direct list initialization of scoped and unscoped enumerations with a fixed underlying type (ISOC++ 11.6.4) is ***Supported***.

Note:

ISOC++ allows direct list initialization of enumerations with a fixed underlying type, e.g. `enum class byte : unsigned char { }; byte b{42};`

9.3 Namespaces

The support of namespaces (ISOC++ 10.3) is covered in Section 13.

9.4 The asm declaration

Not Supported.

9.5 Linkage specifications

Linkage specifications (ISOC++ 10.5), such as `extern "C"` and `extern "C++"`, are ***Not Supported***. See Section 4 on translation units.

9.6 Attributes

Attributes (ISOC++ 10.6) are ***Ignored*** as specified in the following sections.

9.6.1 Attribute syntax and semantics

Attribute syntax and semantics are ***Ignored***.

9.6.2 Alignment specifier

The *alignas* specifier is ***Ignored***.

9.6.3 Carries dependency attribute

The *carries_dependency* attribute is ***Ignored***.

9.6.4 Deprecated attribute

The *deprecated* attribute is ***Ignored***.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

9.6.5 Noreturn attribute

The *noreturn* attribute is ***Ignored***.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

10. Declarators

Declarators, as defined in ISOC++ Section 11, have ***Restricted Support***.

10.1 Type names

Supported as defined in ISOC++ 11.1.

10.2 Ambiguity resolution

Supported as defined in ISOC++ 11.2.

10.3 Kinds of declarators

10.3.1 Pointers

Pointers have ***Restricted Support***. Pointers that are statically determinable are ***Supported***. Otherwise, they are ***Not Supported***. Statically determinable implies the *synthesis tool* is able to determine the actual object whose address is contained by the pointer. If the pointer points to an array, the size of the array *shall* also be statically determinable.

Pointer arithmetic, and the use of pointer values as data, are ***Not Supported***.

This includes operations such as:

- Testing whether a pointer is equal to zero (`nullptr` or `0`)
- Using a pointer value in hashing or other data-dependent computations

10.3.2 References

Supported as defined in ISOC++ 11.3.2.

10.3.3 Pointers to nonstatic class members

Pointers to members using `::*` and access via `->*` and `.*` have ***Restricted Support***. They are ***Supported*** as described in ISOC++ 11.3.3 and 8.5, subject to the restrictions described in Section 10.3.1 on pointers.

10.3.4 Arrays

The element type of an array (ISOC++ 11.3.4) *shall* be any of the types which are permitted by ISOC++ as element types and which are ***Supported*** for synthesis; or any SystemC data type which is ***Supported*** for synthesis and which conforms to the requirements on element types stated in ISOC++.

Arrays of type `std::array` and C-style arrays are ***Supported***.

Any array declaration *shall* specify its dimensions either:

- **Explicitly**, when no initializer is provided, or
- **Implicitly**, as inferred from the initializer, when one is specified.

Array dimensions *shall* be compile-time constants and *shall not* be modified during the array's lifetime.

Statically allocated arrays are ***Supported***. Dynamically allocated arrays are ***Not Supported***, except as allowed in Section 11.5.4.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

See Section 17.1.4 for more information on `std::array` support.

Examples:

```
// Static allocation:
sc_uint<32>                  buf1[64];      // Supported
sc_ufixed<8, 3>              buf2[32][64];  // Supported
std::array<sc_uint<16>, 256> mem;           // Supported

int numbers[] = {1, 2, 3, 4, 5};  // Supported, size deduced as 5 from initializer

// Dynamic allocation:
int* buf1 = new int[64];        // Not Supported
delete[] buf1;                  // Not Supported
sc_uint<32>* buf2 = new sc_uint<32>[64];    // Not Supported
```

10.3.5 Function parameters

Function parameters (ISOC++ 11.3.5) have ***Restricted Support***. The restriction is that ellipsis (. . .) function parameters are ***Not Supported***.

10.3.6 Default arguments

Default arguments (ISOC++ 11.3.6) are ***Supported***.

10.4 Function definition

Function definitions (ISOC++ 11.4) are ***Supported***.

10.4.1 Explicitly-defaulted functions

Explicitly-defaulted functions (ISOC++ 11.4.2) are ***Supported***.

10.4.2 Deleted definitions

Deleted definitions (ISOC++ 11.4.3) are ***Supported***.

10.5 Structured binding declarations

Structured binding declarations (ISOC++ 11.5) are ***Supported***. Structured bindings allow decomposing an object into its constituent elements, providing a concise syntax for accessing members of structs, pairs, tuples, and arrays.

10.6 Initializers

Initializers (ISOC++ 11.6) are ***Supported***.

10.6.1 Aggregates

Initialization of aggregates (ISOC++ 11.6.1) is ***Supported***.

Note:

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

ISOC++ extends aggregate initialization to support public, non-virtual base classes, allowing initialization of derived aggregates that include base class members.

10.6.2 Character arrays

Initialization of character arrays (ISOC++ 11.6.2) is ***Supported***. Also see Section 8.1.1.1 for character literals that are supported.

10.6.3 References

Initialization of references (ISOC++ 11.6.3) is ***Supported***.

10.6.4 List-initialization

List-initialization (ISOC++ 11.6.4) is ***Supported***.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

11. Expressions

An expression is a sequence of operators and operands that can result in a value. Expressions can cause side effects (ISOC++ Section 8). Except where ISOC++ specifies otherwise, the evaluations of operands and the side effects are unsequenced, and some such expressions have undefined behavior (ISOC++ 4.6). The relative order of the evaluation of function arguments is indeterminately sequenced, that is, unspecified (ISOC++ 8.2.2). For example the statement:

```
f(i++, i++);
```

has a behavior that is not specified in ISOC++. Expressions that are legal in ISOC++, but whose behavior is unspecified (not undefined) due to order of evaluation or order of side effects, are supported for synthesis. *Synthesis tools* are permitted to interpret such expressions in any way that is compliant with the ISOC++ standard.

Note:

User code *should not* use such expressions to avoid differences in the results between simulation using a particular compiler and synthesis using a particular *synthesis tool*.

11.1 Primary expressions

11.1.1 General

General expressions (ISOC++ 8.1) are ***Supported***.

11.1.2 Lambda expressions

Lambda expressions (ISOC++ 8.1.5) are ***Supported***.

constexpr lambda expressions are ***Supported***. A lambda expression can be declared or implicitly deduced as *constexpr*, allowing its use in constant expressions for compile-time computation.

Capture of `*this` by value (ISOC++ 8.1.5.2) is ***Supported***. This allows a lambda to capture a copy of the enclosing object, ensuring the lambda does not depend on the lifetime of the original object.

11.1.3 Fold expressions

Fold expressions (ISOC++ 8.1.6) are ***Supported***.

11.2 Function call

Non-recursive function calls (ISOC++ 8.2.2) are ***Supported***.

Recursive functions — that is, functions that invoke themselves either directly or indirectly — are ***Supported with Restrictions***. Recursion is permitted only when the maximum recursion depth can be determined statically. The maximum allowable recursion depth is implementation-defined.

11.3 Explicit type conversion

An explicit type conversion (ISOC++ 8.4) can be expressed using a type conversion operator, a functional notation (ISOC++ 8.2.3), or cast notation. An explicit type conversion of non-pointer type is ***Supported***,

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

except as stated in Sections 11.3.1.1 and 11.3.1.3. On the other hand, an explicit type conversion of pointer is ***Supported with Restrictions***.

11.3.1 Type conversion operators

11.3.1.1 Dynamic cast

The type conversion operator function *dynamic_cast<T>(v)* (ISOC++ 8.2.7) is ***Not Supported***.

11.3.1.2 Static cast

The type conversion operator function *static_cast<T>(v)* (ISOC++ 8.2.9) is ***Supported***. Conversions of pointer type are subject to the restrictions on pointers in Section 10.3.1.

11.3.1.3 Reinterpret cast

The type conversion operator function *reinterpret_cast<T>(v)* (ISOC++ 8.2.10) is ***Not Supported***.

11.3.1.4 Const cast

The type conversion operator function *const_cast<T>(v)* (ISOC++ 8.2.11) is ***Supported***. Conversions of pointer type are subject to the restrictions on pointers in Section 10.3.1.

11.3.2 Functional notation

An explicit type conversion of pointer type using functional notation (ISOC++ 8.2.3) is ***Supported with Restrictions***. Each type conversion of pointer type using functional notation can be mapped to any of type conversion operators described in Section 11.3. The restriction is the same as those of the type conversion operators.

11.3.3 Cast notation

An explicit type conversion of pointer type using cast notation (ISOC++ 8.4) is ***Supported with Restrictions***. Each type conversion of pointer type using cast notation can be mapped to any of type conversion operators described in Section 11.3. The restriction is the same as those of the type conversion operators, which are equivalent to the given explicit type conversion using cast notation.

11.4 typeid

The type identification function *typeid* (ISOC++ 8.2.8) is ***Not Supported***.

11.5 Unary expressions and operators

11.5.1 Unary operators

The unary operators (ISOC++ 8.3.1) of the form * & + - ! and ~ are ***Supported***. For pointers see Section 10.3.1.

11.5.2 Increment and decrement

Supported (ISOC++ 8.2.6 and 8.3.2). See Section 8.1.3 for the restrictions on integer types.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

11.5.3 sizeof

The *sizeof* operator (ISOC++ 8.3.3) is ***Not Supported***, except for `sizeof...`, which is ***Supported***.

11.5.4 new and delete

The *new* operator (ISOC++ 8.3.4) has ***Restricted Support***. It is ***Supported*** for the instantiation of variable arrays and *SC_MODULE*. It is ***Not Supported*** for the instantiation of types derived from the *sc_object* type, other than *SC_MODULE*. The *new* operator *may* appear in the constructor or constructor initializer list within *SC_MODULE* only and that use of *new* is ***Not Supported*** anywhere within the behavioral description, such as inside *SC_METHOD*, *SC_THREAD*, or *SC_CTHREAD*. When allocating an array of objects using *new*, the number of elements *shall* be statically determinable.

The *delete* operator (ISOC++ 8.3.5) is ***Not Supported***.

The use of *set_new_handler() shall* be ***Ignored***.

Overloading the *new* or *delete* operators is ***Not Supported***. The *placement new* construct is ***Not Supported***.

Example:

```
SC_MODULE(MyModule) {
    sc_in<bool>     SC_NAMED(CLK);
    sc_in<bool>     SC_NAMED(RST);
    sc_in<int>      SC_NAMED(a);
    sc_in<int>      SC_NAMED(b);
    sc_out<int>     SC_NAMED(c);
    sc_out<bool>    SC_NAMED(RDY);
    sc_signal<int>  SC_NAMED(tmp);

    GCD *gcd;

    unsigned int *mem ;

    SC_CTOR(MyModule): mem(new unsigned[128])
    {
        gcd = new GCD("GCD");
        gcd->CLK(CLK);
        gcd->RST(RST);
        gcd->x(tmp);
        gcd->y(b);
        gcd->z(c);
        gcd->RDY(RDY);
    }

};
```

11.5.5 alignof

The *alignof* expression (ISOC++ 8.3.6) is ***Ignored***.

11.5.6 noexcept operator

The *noexcept* operator (ISOC++ 8.3.7) is ***Ignored***.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

11.6 Pointer-to-member operators

The pointer-to-member operators ->* and .* have ***Restricted Support*** (ISOC++ 8.5).

The restrictions are based on the ***Restricted Support*** of pointers as defined in Section 10.3.1.

11.7 Multiplicative, additive, shift, relational and equality operators

Supported (ISOC++ 8.6, 8.7, 8.8, 8.9, and 8.10). See Section 8.1.3 for the restrictions on integer types.

11.8 Bitwise AND/XOR/OR and logical AND/OR operators

Supported (ISOC++ 8.11 to 8.15).

11.9 Conditional operator

Supported (ISOC++ 8.16).

11.10 Throwing an exception

Not Supported (ISOC++ 8.17).

11.11 Assignment and compound assignment operators

Supported (ISOC++ 8.18).

11.12 Comma operator

Supported (ISOC++ 8.19).

11.13 Constant expressions

Supported (ISOC++ 8.20).

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

12. Statements

Statements (as defined in ISOC++ Section 9) have ***Restricted Support***.

12.1 Labeled statement

Labels are ***Supported*** as defined in ISOC++ 9.1.

12.2 Expression statement

Supported as defined in ISOC++ 9.2, including the null statement.

12.3 Compound statement

The *compound* statement (also, and equivalently, called *block*) is ***Supported***, as defined in ISOC++ 9.3, to group sets of statements together.

12.4 Selection statements

Selection statements include *if*, *if-else*, and *switch* statements.

Selection statements are ***Supported*** as defined in ISOC++ 9.4.

12.4.1 The if statement

Supported as defined in ISOC++ 9.4.1.

constexpr if statements (ISOC++ 9.4.1) are ***Supported***. A *constexpr if* evaluates its condition at compile time, discarding the branch not taken. In a template, the discarded branch is not instantiated. This enables template-based hardware generation with compile-time specialization.

Init-statements in *if* statements (ISOC++ 9.4.1) are ***Supported***. This allows declaring a variable scoped to the *if* statement, e.g. `if (auto v = expr; condition) { ... }`.

12.4.2 The switch statement

Supported as defined in ISOC++ 9.4.2.

Init-statements in *switch* statements (ISOC++ 9.4.2) are ***Supported***. This allows declaring a variable scoped to the *switch* statement.

12.5 Iteration statements

The iteration statements *while*, *do*, *for* and range-based *for* are ***Supported*** as defined in ISOC++ 9.5. The supported forms of infinite loop in a thread process are given in Section 6.2.2.

12.6 Jump statements

Jump Statements (ISOC++ 9.6) have ***Restricted Support*** as described below.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

12.6.1 The break statement

The *break* statement (as defined in ISOC++ 9.6.1) has ***Restricted Support***.

The restriction is that a *break* that exits the infinite loop of an *SC_CTHREAD* or *SC_THREAD* process (see Section 6.2.2) is ***Not Supported***.

12.6.2 The continue statement

The *continue* statement (as defined in ISOC++ 9.6.2) is ***Supported***.

12.6.3 The return statement

The *return* statement (as defined in ISOC++ 9.6.3) has ***Restricted Support***.

The restriction is that a *return* statement that occurs within the function defining an *SC_CTHREAD* or *SC_THREAD* process (see Section 6.2.2) is ***Not Supported***.

12.6.4 The goto statement

The *goto* statement (as defined in ISOC++ 9.6.4) is ***Not Supported***.

12.7 Declaration statement

Declaration statements are ***Supported*** as defined in ISOC++ 9.7.

12.8 Exception handling

The exception handling mechanisms (as defined in ISOC++ Section 18) are ***Not Supported***. These include the *try-block*, the *throw-expression*, and the function *terminate()*.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

13. Namespaces

namespaces (as defined in ISOC++ 10.3) are ***Supported***. Non-const global variables are ***Not Supported*** for synthesis. Within a function body, only variables declared previously in the function body, passed as parameters, or declared as global constants *shall* be used. Global constants are ***Supported*** for synthesis.

Examples:

```
// Example Namespace
namespace NSP {
    int var;
    constexpr int CNST = 42;
}

void foo(const int val) {
    using namespace NSP;

    int dummy = CNST; // OK: Note that the occurrence of name CNST may be
                      // replaced by the value '42' by a synthesis tool.

    dummy = var; // Error: A non-const global variable shall not be used
                 // within a function body.

    var = val; // Error: A non-const global variable shall not be used
               // within a function body.
}
```

13.1 Namespace definition

Supported as defined in ISOC++ 10.3.1.

Nested namespace definitions are ***Supported***. This allows the shorthand `namespace A::B::C { }` in place of nested namespace definitions.

13.1.1 Unnamed namespaces

Supported as defined in ISOC++ 10.3.1.1.

13.1.2 Namespace member definitions

Supported as defined in ISOC++ 10.3.1.2.

13.2 Namespace alias

Supported as defined in ISOC++ 10.3.2.

13.3 The using declaration

Supported as defined in ISOC++ 10.3.3.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

13.4 Using directive

Supported as defined in ISOC++ 10.3.4.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

14. Classes

Restrictions pertaining to classes defined in the SystemC LRM are specified in the respective Section in this standard.

This chapter describes the support for user-defined classes.

Restricted Support as described below (ISOC++ Section 12).

14.1 Class names

Supported as defined in ISOC++ 12.1.

14.2 Class members

Supported as defined in ISOC++ 12.2.

14.3 Member functions

Supported as defined in ISOC++ 12.2.1.

14.3.1 Non-static member functions

Supported as defined in ISOC++ 12.2.2.

14.3.2 The this pointer

Supported as defined in ISOC++ 12.2.2.1.

14.4 Static members

Restricted Support (ISOC++ 12.2.3).

14.4.1 Static member functions

Supported (ISOC++ 12.2.3.1) as specified in Section 9.1.

14.4.2 Static data members

Static data members (ISOC++ 12.2.3.2) have ***Restricted Support*** as specified in Section 9.1.

14.5 Unions

Not Supported (ISOC++ 12.3).

14.6 Bit-fields

Bit-fields, as defined in ISOC++ 12.2.4, are ***Not Supported***.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

14.7 Nested class declarations

Supported as defined in ISOC++ 12.2.5.

14.8 Local class declarations

Supported as defined in ISOC++ 12.4.

14.9 Nested type names

Supported as defined in ISOC++ 12.2.6.

14.10 Derived classes

Supported as defined in ISOC++ Section 13.

14.10.1 Multiple base classes

Supported as defined in ISOC++ 13.1.

14.10.2 Member name lookup

Supported as defined in ISOC++ 13.2.

14.10.3 Virtual functions

Virtual functions, as defined in ISOC++ 13.3, have ***Restricted Support*** provided that when such functions are called, the type of the "*this*" object can be statically determined.

14.10.4 Abstract classes

Supported as defined in ISOC++ 13.4.

14.11 Member access control

Supported as defined in ISOC++ Section 14.

14.11.1 Access specifiers

Supported as defined in ISOC++ 14.1.

14.11.2 Accessibility of base classes and base class members

Supported as defined in ISOC++ 14.2.

14.11.3 Friends

Supported as defined in ISOC++ 14.3.

14.11.4 Protected member access

Supported as defined in ISOC++ 14.4.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

14.11.5 Access to virtual functions

Supported as defined in ISOC++ 14.5.

14.11.6 Multiple access

Supported as defined in ISOC++ 14.6.

14.11.7 Nested classes

Supported as defined in ISOC++ 14.7.

14.12 Special member functions

Restricted Support as described below (ISOC++ Section 15).

14.12.1 Constructors

Constructors of user-defined classes (as defined in ISOC++ 15.1) are ***Supported***.

14.12.2 Temporary objects

Supported as defined in ISOC++ 15.2.

14.12.3 Conversions

Supported as defined in ISOC++ 15.3.

14.12.3.1 Conversion by constructor

Supported as defined in ISOC++ 15.3.1.

14.12.3.2 Conversion functions

Supported as defined in ISOC++ 15.3.2.

14.12.4 Destructors

Supported as defined in ISOC++ 15.4.

14.12.5 Free store

Class-specific allocation and deallocation functions (ISOC++ 15.5) are ***Not Supported***. For the *new* and *delete* operators see Section 11.5.4.

14.12.6 Initialization

Supported (ISOC++ 15.6).

14.12.6.1 Explicit initialization

Explicit initialization (as defined in ISOC++ 15.6.1) is ***Supported***.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

14.12.6.2 Initializing bases and members

Supported as defined in ISOC++ 15.6.2.

14.12.7 Copying and moving class objects

Supported as defined in ISOC++ 15.8, except for `sc_module`, see Section 5.6.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

15. Overloading

Restricted Support as described below (ISOC++ Section 16).

15.1 Overloadable declarations

Supported as defined in ISOC++ 16.1.

15.2 Declaration matching

Supported as defined in ISOC++ 16.2.

15.3 Overload resolution

Supported as defined in ISOC++ 16.3.

15.3.1 Candidate functions and argument lists

Supported as defined in ISOC++ 16.3.1.

15.3.1.1 Function call syntax

Supported as defined in ISOC++ 16.3.1.1.

15.3.1.1.1 Call to named function

Supported as defined in ISOC++ 16.3.1.1.1.

15.3.1.1.2 Call to object of class type

Supported as defined in ISOC++ 16.3.1.1.2.

15.3.1.2 Operators in expressions

Supported as defined in ISOC++ 16.3.1.2.

15.3.1.3 Initialization by constructor

Supported as defined in ISOC++ 16.3.1.3.

15.3.1.4 Copy-initialization of class by user-defined conversion

Supported as defined in ISOC++ 16.3.1.4.

15.3.1.5 Initialization by conversion function

Supported as defined in ISOC++ 16.3.1.5.

15.3.1.6 Initialization by conversion function for direct reference binding

Supported as defined in ISOC++ 16.3.1.6.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

15.3.2 Viable functions

Supported as defined in ISOC++ 16.3.2.

15.3.3 Best viable function

Supported as defined in ISOC++ 16.3.3.

15.3.3.1 Implicit conversion sequences

Supported as defined in ISOC++ 16.3.3.1.

15.3.3.1.1 Standard conversion sequences

Supported as defined in ISOC++ 16.3.3.1.1.

15.3.3.1.2 User-defined conversion sequences

Supported as defined in ISOC++ 16.3.3.1.2.

15.3.3.1.3 Ellipsis conversion sequences

Not Supported (ISOC++ 16.3.3.1.3).

15.3.3.1.4 Reference binding

Supported as defined in ISOC++ 16.3.3.1.4.

15.3.3.2 Ranking implicit conversion sequences

Supported as defined in ISOC++ 16.3.3.2.

15.4 Address of overloaded function

Taking the address of an overloaded function is ***Not Supported*** (ISOC++ 16.4).

15.5 Overloaded operators

Restricted Support as defined in ISOC++ 16.5. Overloading the *new*, *delete*, *new[]* and *delete[]* operators is ***Not Supported***, see Section 11.5.4.

15.5.1 Unary operators

Supported as defined in ISOC++ 16.5.1.

15.5.2 Binary operators

Supported as defined in ISOC++ 16.5.2.

15.5.3 Assignment

Supported as defined in ISOC++ 16.5.3.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

15.5.4 Function call

Supported as defined in ISOC++ 16.5.4.

15.5.5 Subscripting

Supported as defined in ISOC++ 16.5.5.

15.5.6 Class member access

Supported as defined in ISOC++ 16.5.6.

15.5.7 Increment and decrement

Supported as defined in ISOC++ 16.5.7.

15.6 Built-in operators

Supported as defined in ISOC++ 16.6.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

16. Templates

The support for templates (ISOC++ Section 17) in this standard is explained below.

16.1 Template parameters

Supported (ISOC++ 17.1).

Declaring non-type template parameters with *auto* (ISOC++ 17.1) is ***Supported***. This allows the type of a non-type template parameter to be deduced, e.g. `template<auto N>`.

16.2 Names of template specializations

Supported as defined in ISOC++ 17.2.

16.3 Template arguments

Supported as defined in ISOC++ 17.3.

16.3.1 Template type arguments

Supported as defined in ISOC++ 17.3.1.

16.3.2 Template non-type arguments

Supported as defined in ISOC++ 17.3.2.

16.3.3 Template template arguments

Supported as defined in ISOC++ 17.3.3.

16.4 Type equivalence

Supported as defined in ISOC++ 17.4.

16.5 Template declarations

Supported as defined in ISOC++ 17.5.

16.5.1 Class templates

Supported as defined in ISOC++ 17.5.1.

16.5.1.1 Member functions of class templates

Supported as defined in ISOC++ 17.5.1.1.

16.5.1.2 Member classes of class templates

Supported as defined in ISOC++ 17.5.1.2.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

16.5.1.3 Static data members of class templates

Restricted Support (ISOC++ 17.5.1.3). See Section 14.4.2 for static data members of classes and Section 9.1.1 for the static storage class specifier.

16.5.2 Member templates

Supported as defined in ISOC++ 17.5.2.

16.5.3 Variadic templates

Supported as defined in ISOC++ 17.5.3.

16.5.4 Friends

Supported as defined in ISOC++ 17.5.4.

16.5.5 Class template partial specializations

Supported as defined in ISOC++ 17.5.5.

16.5.5.1 Matching of class template partial specializations

Supported as defined in ISOC++ 17.5.5.1.

16.5.5.2 Partial ordering of class template specializations

Supported as defined in ISOC++ 17.5.5.2.

16.5.5.3 Members of class template specializations

Supported as defined in ISOC++ 17.5.5.3.

16.5.6 Function templates

Supported as defined in ISOC++ 17.5.6.

16.5.6.1 Function template overloading

Supported as defined in ISOC++ 17.5.6.1.

16.5.6.2 Partial ordering of function templates

Supported as defined in ISOC++ 17.5.6.2.

16.5.7 Alias templates

Supported as defined in ISOC++ 17.5.7.

16.6 Name resolution

Supported as defined in ISOC++ 17.6.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

16.6.1 Locally declared names

Supported as defined in ISOC++ 17.6.1.

16.6.2 Dependent names

Supported as defined in ISOC++ 17.6.2.

16.6.2.1 Dependent types

Supported as defined in ISOC++ 17.6.2.1.

16.6.2.2 Type-dependent expressions

Supported as defined in ISOC++ 17.6.2.2.

16.6.2.3 Value-dependent expressions

Supported as defined in ISOC++ 17.6.2.3.

16.6.2.4 Dependent template arguments

Supported as defined in ISOC++ 17.6.2.4.

16.6.3 Non-dependent names

Supported as defined in ISOC++ 17.6.3.

16.6.4 Dependent name resolution

Supported as defined in ISOC++ 17.6.4.

16.6.4.1 Point of instantiation

Supported as defined in ISOC++ 17.6.4.1.

16.6.4.2 Candidate functions

Supported as defined in ISOC++ 17.6.4.2.

16.6.5 Friend names declared within a class template

Supported as defined in ISOC++ 17.6.5.

16.7 Template instantiation and specialization

Supported as defined in ISOC++ 17.7.

16.7.1 Implicit instantiation

Supported as defined in ISOC++ 17.7.1.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

16.7.2 Explicit instantiation

Supported as defined in ISOC++ 17.7.2.

16.7.3 Explicit specialization

Supported as defined in ISOC++ 17.7.3.

16.8 Function template specializations

Supported as defined in ISOC++ 17.8.

16.8.1 Explicit template argument specification

Supported as defined in ISOC++ 17.8.1.

16.8.2 Template argument deduction

Supported as defined in ISOC++ 17.8.2.

16.8.2.1 Deducing template arguments from a function call

Supported as defined in ISOC++ 17.8.2.1.

16.8.2.2 Deducing template arguments taking the address of a function template

Supported as defined in ISOC++ 17.8.2.2.

16.8.2.3 Deducing conversion function template arguments

Supported as defined in ISOC++ 17.8.2.3.

16.8.2.4 Deducing template arguments during partial ordering

Supported as defined in ISOC++ 17.8.2.4.

16.8.2.5 Deducing template arguments from a type

Supported as defined in ISOC++ 17.8.2.5.

16.8.2.6 Deducing template arguments from a function declaration

Supported as defined in ISOC++ 17.8.2.6.

16.8.3 Overload resolution

Supported as defined in ISOC++ 17.8.3.

16.9 Deduction guides

Class template argument deduction (CTAD) and deduction guides (ISOC++ 17.9) are ***Supported***. CTAD allows the template arguments of a class template to be deduced from the constructor arguments, reducing verbosity when instantiating template classes.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

17. Libraries

This section outlines support for external C++ or SystemC libraries.

17.1 Standard C and C++ libraries

Generally, library functions from C/C++ have ***Restricted Support***. Most of them are ***Not Supported***, while some are ***Ignored*** as described below.

17.1.1 Standard console I/O

Stream-based I/O (C++ Standard Library `<iostream>`):

- The `operator<<` when applied to `cout` or `cerr` is ***Ignored***.
- The `operator<<` when applied to `std::ostream` is ***Ignored***, except when applied to the file stream types listed in Section 17.1.2, where it is ***Not Supported***.

C Standard Library (`<cstdio>`):

- All `printf` functions and `puts` are ***Ignored***.

Functions or operators used to compute arguments for an Ignored function are not considered part of the Ignored construct. Specifically, if an argument to an Ignored function includes a side effect, that side effect *should* be supported, but a synthesis tool may choose to reject the input with an appropriate error message.

Examples:

```
printf("x = %d", x);  // Ignored
printf("y = %d", ++y); // Side effect ++y is Supported
cout << "z = " << z << endl; // Ignored
```

17.1.2 Standard file streaming

C Standard Library (`<cstdio>`):

- The following functions for file input/output operations are ***Not Supported***: `fopen`, `freopen`, `fclose`, `fread`, `fwrite`, `fseek`, `ftell`, `fgetc`, `fscanf`, `fputs`.

C++ Standard Library (`<fstream>`):

- The following constructs for file streaming are ***Not Supported***:
  - `ifstream` and `ofstream` constructors
  - `operator<<` applied to `fstream`
  - `operator>>` applied to `fstream`
  - `operator<<` applied to `ofstream`
  - `operator>>` applied to `ifstream`

17.1.3 Type traits

The C++ metaprogramming facilities provided in the `<type_traits>` header of the standard library are ***Supported*** for all constructs defined up to the ISO C++17 standard.

17.1.4 Standard C++ array

Usage of the standard C++ array class (`std::array`), defined in `<array>` is ***Supported with Restrictions***.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

A `std::array` *shall* have a constant size greater than zero. Multidimensional arrays of type `std::array` are ***Supported*** and *shall* be of constant size in every dimension.

For `std::array`, the following member types are ***Supported***:

- `value_type`
- `size_type`
- `difference_type`
- `reference`
- `const_reference`
- `iterator`
- `const_iterator`
- `reverse_iterator`
- `const_reverse_iterator`

For `std::array`, usage of the following member types is ***Not Supported***:

- `pointer`
- `const_pointer`

For `std::array`, the following member functions are ***Supported***:

- `at()`
- `operator[]`
- `front()`
- `back()`
- `begin()`
- `cbegin()`
- `end()`
- `cend()`
- `rbegin()`
- `rend()`
- `crbegin()`
- `crend()`
- `empty()`
- `size()`
- `fill()`

For `std::array`, usage of the following member functions is ***Not Supported***:

- `data()`
- `swap()`
- `max_size()`

For `std::array`, the following non-member functions are ***Supported***:

- `get()`
- `operator==`

For `std::array`, usage of the following non-member functions is ***Not Supported***:

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

- `operator!=`
- `operator<`
- `operator<=`
- `operator>`
- `operator>=`
- `swap()`

17.2 SystemC functions and types

17.2.1 Tracing

The following tracing constructs are ***Ignored***:

- declaration of a variable of type `sc_trace_file*`
- assignment to a variable of type `sc_trace_file*`,
- calls to `sc_create_vcd_trace_file`,
- calls to `sc_close_vcd_trace_file`,
- declaration or definition of any function named `sc_trace`, and
- calls to any function named `sc_trace`.

17.2.2 `set_stack_size`

Ignored.

17.2.3 `sc_gen_unique_name`

Ignored.

17.2.4 `before_end_of_elaboration` and `end_of_elaboration`

Not Supported.

17.2.5 `dont_initialize`

Supported.

17.2.6 `next_trigger`

Not Supported.

17.2.7 `print` and `dump`

The SystemC functions `print` and `dump` are ***Ignored***.

17.2.8 `kind`

Ignored.

17.2.9 `sc_report_handler`

The `sc_report_handler` class has ***Restricted Support***. The member function `sc_report_handler::report` is ***Ignored***. All other member functions of `sc_report_handler` are ***Not Supported***.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

Note:

This implies that the macros `sc_assert`, `SC_REPORT_FATAL`, `SC_REPORT_ERROR`, `SC_REPORT_WARNING` and `SC_REPORT_INFO` are ***Ignored***.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

18. The SystemC HLS Library

This section specifies the SystemC HLS library. The library currently includes a communication library, a set of handshake-based ports and channels for throughput-accurate, synthesizable communication between processes and modules. Its future versions may include other SystemC and C++ components.

The purpose of the communication library is to provide an API and the supporting simulation infrastructure for modeling *designs* with concurrent I/O operations, such that the same concurrent transactions observed in the SystemC model are also observed in the generated RTL. A *transaction* is an observable event at a module interface, using variations of the valid, data, and ready protocols.

The library is a defined API and behavioral contract. A *synthesis tool* *may* provide its own implementation; any implementation that conforms to the class names, member signatures, and behavior specified in this chapter is a conforming implementation. The proof-of-concept reference implementation released with this standard is one such implementation; its internal structure is not normative.

18.1 Overview

All library types are made available through a single include, and reside in the `sc_hls::msg_lib` namespace. The library-wide utility `add_clock_alias` is additionally available in the enclosing `sc_hls` namespace; it is available in the simulation view only.

```
#include <systemc-hls>
using namespace sc_hls;
using namespace sc_hls::msg_lib;
```

The library provides two message passing protocols. Each protocol defines a *producer* port, a *consumer* port, and a *portless* channel usable directly between two processes of the same module.

Protocol	Producer port	Consumer port	Channel	Signals
Message passing with flow control	`msg_out`	`msg_in`	`msg_channel`	`vld`, `rdy`, `dat`
Message passing without back-pressure	`msg_dv_out`	`msg_dv_in`	`msg_dv_channel`	`vld`, `dat`

The protocols differ in when a transfer occurs:

- **Message passing with flow control** (dat/vld/rdy): data is transferred on a clock edge when `vld` = 1 and `rdy` = 1. The producer asserts `vld` when it has data; the consumer asserts `rdy` when it can accept. Either side *may* stall the other; flow control enables the design to use back-pressure.
- **Message passing without back-pressure** (dat/vld): data is transferred on a clock edge when `vld` = 1. There is no `rdy`; the consumer *shall* accept the beat in that cycle and cannot stall the producer.

The library also provides a rendezvous synchronization between two processes, with no data transfer: `sync_out` drives `vld` and samples `rdy`, `sync_in` drives `rdy` and samples `vld`, and `sync_channel` is the portless form. Neither side has precedence; a rendezvous completes on a clock edge on which both `vld` and `rdy` are asserted (see 18.5).

The library models concurrent I/O within a process. The transactions of distinct message passing endpoints *may* overlap in time — several transfers *may* be in progress in the same cycle. In simulation, the partial order in which the endpoint calls of a single process appear in the source code *shall* be preserved: a `pop` or `push` that follows another in program order *shall* not start before it.

A *synthesis tool* *may* overlap or reorder the transactions of a process, for example by pipelining a loop

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

so that the transactions of successive iterations overlap, provided that for each endpoint the sequence of transferred values is the same as in program order and the data dependencies of the process are preserved. Such a transformation can change the timing of the transactions relative to the simulation view; latency and capacity back-annotation (18.8.3) can be used to model it in simulation.

A rendezvous *shall* not overlap with any other transaction of the same process: a `sync` (or `sync_read` / `sync_write`) *shall* act as a barrier, so that every transaction preceding it in program order completes before it, and no transaction following it in program order begins until it has completed. This rule applies in both simulation and synthesis: a *synthesis tool* *shall* not move a transaction across a rendezvous.

The message passing APIs are defined by two SystemC interfaces:

```
template <typename Message> class msg_in_if : public sc_interface {
public:
    virtual void    reset_pop() = 0;
    virtual Message pop() = 0;
    virtual bool    pop_nb(Message& m) = 0;
    virtual bool    peek_nb(Message& m) = 0;
};

template <typename Message> class msg_out_if : public sc_interface {
public:
    virtual void reset_push() = 0;
    virtual void push(const Message& m) = 0;
    virtual bool push_nb(const Message& m) = 0;
};
```

For message passing, a producer port *shall* implement the writer interface (`msg_out_if`), a consumer port *shall* implement the reader interface (`msg_in_if`), and a channel *shall* implement both.

18.2 Simulation and Synthesis Views

The library defines two views of each port and channel:

- a **simulation view**, used to simulate the SystemC model of the *design* with the concurrent transactions, and
- a **synthesis view**, used as input to an HLS *synthesis tool* for generating RTL.

Each port and channel takes a trailing view-selection template parameter for the view tag:

```
template <typename Message, typename ViewTag = default_view_tag> class msg_in;
```

The `sync_in`, `sync_out`, and `sync_channel` classes carry no `Message` and are parameterized on `ViewTag` only.

The view is selected through `default_view_tag`, which is resolved at compile time according to whether the `__SYNTHESIS__` macro is defined:

- When `__SYNTHESIS__` is not defined (simulation), `default_view_tag` *shall* resolve to the throughput-accurate simulation view `sim_view_tag`.
- When `__SYNTHESIS__` is defined (synthesis), `default_view_tag` *shall* resolve to the tag named by the macro `SC_HLS_DEFAULT_SYNTH_VIEW_TAG`. A *synthesis tool* selects its own synthesis view by defining this macro to its view tag; if `__SYNTHESIS__` is defined and `SC_HLS_DEFAULT_SYNTH_VIEW_TAG` is not, the program is ill-formed.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

A *synthesis tool* *shall* define both `__SYNTHESIS__` and `SC_SYNTHESIS` (see Section 4.2).

A *design* that names no explicit `ViewTag` therefore compiles to the simulation view for simulation and to the *synthesis tool*'s view for synthesis, from a single source.

A third view, `tlm_view_tag`, selects a fast functional simulation view backed by TLM FIFOs; it is described in 18.9.

18.3 Message Passing with Flow Control

Message passing with flow control provides reliable message passing; flow control enables the design to use back-pressure. A message passed with this protocol *shall* not be dropped: a data value is transferred on signal `dat` on a clock edge only when both `vld` and `rdy` are asserted, and either endpoint *may* stall the other until it is ready.

```
template <typename Message, typename ViewTag = default_view_tag> class msg_out;    // producer
template <typename Message, typename ViewTag = default_view_tag> class msg_in;     // consumer
template <typename Message, typename ViewTag = default_view_tag> class msg_channel; // portless
```

A `msg_in` consumer port implements `msg_in_if` (`reset_pop`, `pop`, `pop_nb`, `peek_nb`); a `msg_out` producer port implements `msg_out_if` (`reset_push`, `push`, `push_nb`); a `msg_channel` implements both.

The ports *shall* implement the following behavior:

- `reset_pop` / `reset_push` *shall* be called in the reset state before any transfer.
- `pop()` *shall* not return until a value has been received.
- `push()` *shall* not return until the value has been accepted.
- The non-blocking `pop_nb`, `peek_nb`, and `push_nb` *shall* return within the current cycle and report success through their return value.

Each port and channel *shall* provide a member type alias `message_type` equal to its `Message` parameter.

The `msg_in` and `msg_out` ports are modular ports as specified in 7.4. In the generated RTL, their `rdy`, `vld`, and `dat` ports *shall* be named as specified in 7.4.2, for example `in1_rdy`, `in1_vld`, `in1_dat` for a port `in1`.

A `msg_channel` *shall* additionally support portless access: it *shall* expose the `msg_in_if` and `msg_out_if` interface functions directly on the channel object, with the same required behavior specified above, so that two processes of the same module can communicate through it without ports.

In addition to implementing the `msg_in_if` / `msg_out_if` interfaces, a port or channel *shall* provide the binding functions `bind()` and `operator()`, used to bind it to the ports or channels it connects to; `operator()` *shall* have the same effect as `bind()`. The example below shows the reader side of a `msg_in` consumer port: it implements `msg_in_if` and provides the binding functions over its `vld`, `rdy`, and `dat` signals. The class layout is implementation-dependent; only the interface methods, the `message_type` alias, the three signal-level ports and the presence of the binding functions are normative.

Example: a consumer port implementing `msg_in_if` with binding functions

```
template <typename Message>   // ViewTag parameter omitted for brevity
class msg_in
    : public msg_in_if<Message>, public sc_module // class layout is implementation-dependent
{
public:
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

```
    using message_type = Message;

    sc_out<bool>    SC_NAMED(rdy);
    sc_in<bool>     SC_NAMED(vld);
    sc_in<Message>  SC_NAMED(dat);

    // msg_in_if implementation:
    void    reset_pop() override;
    Message pop() override;
    bool    pop_nb(Message& m) override;
    bool    peek_nb(Message& m) override;

    // Binding functions:
    template <typename C> void bind(C& rhs)       { rdy(rhs.rdy); vld(rhs.vld); dat(rhs.dat); }
    template <typename C> void operator()(C& rhs) { bind(rhs); }
};
```

Example: a message passing passthrough module using ports

```
SC_MODULE(dut) {
    sc_in<bool> SC_NAMED(clk);
    sc_in<bool> SC_NAMED(rst_bar);

    msg_in<sc_uint<32>>  SC_NAMED(in1);
    msg_out<sc_uint<32>> SC_NAMED(out1);

    SC_CTOR(dut) {
        SC_THREAD(main);
        sensitive << clk.pos();
        async_reset_signal_is(rst_bar, false);
    }

    void reset() {
        in1.reset_pop();
        out1.reset_push();
        wait();
    }

    void main() {
        reset();
        while (1) {
            auto v = in1.pop();
            out1.push(v);
        }
    }
};
```

Example: two processes of one module communicating through an internal channel

```
SC_MODULE(dut) {
    sc_in<bool> SC_NAMED(clk);
    sc_in<bool> SC_NAMED(rst_bar);
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

```
    msg_in<uint32_t>      SC_NAMED(in1);
    msg_out<uint32_t>     SC_NAMED(out1);
    msg_channel<uint32_t> SC_NAMED(chan1);   // internal, portless

    SC_CTOR(dut) {
        SC_THREAD(main1);
        sensitive << clk.pos();
        async_reset_signal_is(rst_bar, false);
        SC_THREAD(main2);
        sensitive << clk.pos();
        async_reset_signal_is(rst_bar, false);
    }

    void main1() {
        in1.reset_pop();
        chan1.reset_push();
        wait();
        while (1) { chan1.push(in1.pop()); }
    }

    void main2() {
        chan1.reset_pop();
        out1.reset_push();
        wait();
        while (1) { out1.push(chan1.pop()); }
    }
};
```

18.4 Message Passing without Back-Pressure

The dat/vld message passing protocol provides valid-only signaling with no `rdy` back-pressure. The producer drives data whenever it has data available and *shall* not be stalled by the consumer. The consumer *shall* accept a beat in the cycle in which `vld` is asserted.

```
template <typename Message, typename ViewTag = default_view_tag> class msg_dv_out;     // producer
template <typename Message, typename ViewTag = default_view_tag> class msg_dv_in;      // consumer
template <typename Message, typename ViewTag = default_view_tag> class msg_dv_channel; // portless
```

A `msg_dv_in` consumer port implements `msg_in_if` (`reset_pop`, `pop`, `pop_nb`, `peek_nb`); a `msg_dv_out` producer port implements `msg_out_if` (`reset_push`, `push`, `push_nb`); a `msg_dv_channel` implements both. Each port and channel provides the `message_type` alias. As for `msg_in` / `msg_out`, a dat/vld port or channel *shall* also provide the binding functions (`bind()` and `operator()`) over its `vld` and `dat` signals, and a `msg_dv_channel` *shall* support portless access by exposing the interface functions directly on the channel object.

The `msg_dv_in` and `msg_dv_out` ports are modular ports as specified in 7.4. In the generated RTL, their `vld` and `dat` ports *shall* be named as specified in 7.4.2, for example `in1_vld`, `in1_dat` for a port `in1`.

The consumer *shall* be able to buffer one unconsumed beat. If a new valid beat arrives while the buffer is still occupied, the buffered beat is overwritten and the previous value is lost; this is faithful to a valid-only wire.

The ports *shall* implement the following behavior:

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

- `reset_pop` / `reset_push` *shall* be called in the reset state before any transfer.
- `push()` *shall* not return until the producer's beat has been driven; the consumer cannot apply back-pressure.
- `pop()` *shall* not return until a beat is available in the consumer's buffer.
- The non-blocking `pop_nb`, `peek_nb`, and `push_nb` *shall* return within the current cycle and report success through their return value; `push_nb` reports `false` when the producer's previous beat has not yet been driven.

Example: a dat/vld producer-consumer module

```
SC_MODULE(dut) {
    using T = unsigned;

    sc_in<bool> SC_NAMED(clk);
    sc_in<bool> SC_NAMED(rst_bar);

    msg_dv_in<T>  SC_NAMED(in1);
    msg_dv_out<T> SC_NAMED(out1);

    SC_CTOR(dut) {
        SC_THREAD(main);
        sensitive << clk.pos();
        async_reset_signal_is(rst_bar, false);
    }

    void reset() {
        in1.reset_pop();
        out1.reset_push();
        wait();
    }

    void main() {
        reset();
        while (1) {
            auto x = in1.pop();
            out1.push(x);
        }
    }
};
```

18.5 Rendezvous Synchronization

The rendezvous protocol synchronizes two processes without transferring data. A rendezvous completes on a clock edge in which both `vld` and `rdy` are asserted.

```
template <typename ViewTag = default_view_tag> class sync_out;     // drives vld
template <typename ViewTag = default_view_tag> class sync_in;      // drives rdy
template <typename ViewTag = default_view_tag> class sync_channel; // portless
```

A `sync_in` port provides `reset_pop` and `sync`; a `sync_out` port provides `reset_push` and `sync`. A portless `sync_channel` provides `reset_pop` and `sync_read` for its reader process, and `reset_push` and `sync_write` for its writer process. These classes carry no `Message`. A sync port or channel *shall* also provide the binding

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

functions (`bind()` and `operator()`) over its `vld` and `rdy` signals, and a `sync_channel` *shall* support portless access by exposing the rendezvous functions directly on the channel object.

The `sync_in` and `sync_out` ports are modular ports as specified in 7.4. In the generated RTL, their `vld` and `rdy` ports *shall* be named as specified in 7.4.2, for example `sync_port_vld`, `sync_port_rdy` for a port `sync_port`.

The ports *shall* implement the following behavior:

- `reset_pop` / `reset_push` *shall* be called in the reset state before any rendezvous.
- `sync()` (and, on a channel, `sync_read()` / `sync_write()`) *shall* block until the rendezvous completes, that is, until a clock edge on which both `vld` and `rdy` are asserted, and *shall* then return on both sides in that cycle.
- No data is transferred.

Example: a synchronization barrier before processing

```
SC_MODULE(dut) {
    sc_in<bool> SC_NAMED(clk);
    sc_in<bool> SC_NAMED(rst_bar);

    sync_in<>            SC_NAMED(sync_port);
    msg_out<uint32_t>    SC_NAMED(out1);

    uint32_t compute();  // user function, not part of the library

    SC_CTOR(dut) {
        SC_THREAD(main);
        sensitive << clk.pos();
        async_reset_signal_is(rst_bar, false);
    }

    void reset() {
        sync_port.reset_pop();
        out1.reset_push();
        wait();
    }

    void main() {
        reset();
        while (1) {
            sync_port.sync();   // wait for the other side to rendezvous
            out1.push(compute());
        }
    }
};
```

18.6 FIFO

A conforming implementation *shall* provide buffered message passing FIFO channels of configurable depth in two forms.

These FIFO channels are one possible alternative to `sc_fifo`, which is ***Not Supported*** (7.7).

Both forms *shall* buffer up to `Depth` elements and preserve order, and their back-pressure *shall* prevent

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

overflow and underflow. Both provide the message passing API (through forwarding functions or ports) and *shall* conform to the same required behavior specified in 18.3.

`msg_fifo_module` presents the FIFO as a module with clock, reset, and `msg_in` / `msg_out` enqueue/dequeue ports to be bound to channels:

```
template <typename Message, unsigned int Depth, typename ViewTag = default_view_tag>
class msg_fifo_module : public sc_module {
public:
    sc_in_clk                 SC_NAMED(clk);
    sc_in<bool>               SC_NAMED(rst);
    msg_in<Message, ViewTag>  SC_NAMED(enq);   // enqueue side
    msg_out<Message, ViewTag> SC_NAMED(deq);   // dequeue side
};
```

Example: buffering two modules with a depth-8 `msg_fifo_module`

```
SC_MODULE(dut) {
    using T = sc_uint<32>;
    static constexpr int N = 8;

    sc_in<bool> SC_NAMED(clk);
    sc_in<bool> SC_NAMED(rst_bar);

    msg_fifo_module<T, N> SC_NAMED(fifo1);
    msg_channel<T>        SC_NAMED(fifo_in);
    msg_channel<T>        SC_NAMED(fifo_out);
    // ... producer and consumer submodules ...

    SC_CTOR(dut) {
        fifo1.clk(clk);
        fifo1.rst(rst_bar);
        fifo1.enq(fifo_in);    // producer pushes into fifo_in
        fifo1.deq(fifo_out);   // consumer pops from fifo_out
        // ... bind producer.to_fifo(fifo_in), consumer.from_fifo(fifo_out) ...
    }
};
```

`msg_fifo_channel` wraps a `msg_fifo_module` and presents it as two channels `in` and `out`, and forwards the message passing API:

```
template <typename Message, unsigned int Depth, typename ViewTag = default_view_tag>
class msg_fifo_channel : public sc_module {
public:
    sc_in_clk                    SC_NAMED(clk);
    sc_in<bool>                  SC_NAMED(rst);
    msg_channel<Message, ViewTag> SC_NAMED(in);    // producer pushes here
    msg_channel<Message, ViewTag> SC_NAMED(out);   // consumer pops here

    // Producer side (forwards to in)
    void reset_push() { in.reset_push(); }
    void push(const Message &m) { in.push(m); }
    bool push_nb(const Message &m) { return in.push_nb(m); }
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

```
    // Consumer side (forwards to out)
    void reset_pop() { out.reset_pop(); }
    Message pop() { return out.pop(); }
    bool pop_nb(Message &m) { return out.pop_nb(m); }
    bool peek_nb(Message &m) { return out.peek_nb(m); }

private:
    msg_fifo_module<Message, Depth, ViewTag> fifo;
};
```

Example: two processes communicating through a `msg_fifo_channel`

```
SC_MODULE(dut) {
    using T = sc_uint<32>;
    static constexpr int N = 8;

    sc_in<bool> SC_NAMED(clk);
    sc_in<bool> SC_NAMED(rst_bar);

    msg_in<T>  SC_NAMED(in1);
    msg_out<T> SC_NAMED(out1);

    msg_fifo_channel<T, N> SC_NAMED(fifo_chan);   // FIFO as in/out channels

    SC_CTOR(dut) {
        fifo_chan.clk(clk);
        fifo_chan.rst(rst_bar);
        SC_THREAD(producer);
        sensitive << clk.pos();
        async_reset_signal_is(rst_bar, false);
        SC_THREAD(consumer);
        sensitive << clk.pos();
        async_reset_signal_is(rst_bar, false);
    }

    void producer() {
        in1.reset_pop();
        fifo_chan.in.reset_push();
        wait();
        while (1) { fifo_chan.in.push(in1.pop()); }
    }

    void consumer() {
        fifo_chan.out.reset_pop();
        out1.reset_push();
        wait();
        while (1) { out1.push(fifo_chan.out.pop()); }
    }
};
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

18.7 Direct Port Access

A conforming implementation *shall* support direct port access as follows. A port or channel endpoint *may* opt out of the blocking/non-blocking API and instead drive and sample its ports (`vld`, `rdy`, `dat`) directly. This is enabled with:

```
void enable_port_access();          // on msg_in / msg_out / msg_dv_in / msg_dv_out
void enable_port_access_reader();   // on a channel, for its reader side
void enable_port_access_writer();   // on a channel, for its writer side
```

`enable_port_access` *shall* be called during elaboration (in the module constructor); calling it later *shall* be reported as an error. When enabled on an endpoint:

- `push`, `pop`, `push_nb`, `pop_nb`, and `peek_nb` on that endpoint are ***Not Supported*** and *shall* be reported as an error, and
- `reset_pop` / `reset_push` on that endpoint are ***Not Supported*** and *shall* be reported as an error; the process instead drives the idle port values (for example `rdy.write(false)` / `vld.write(false)`) in its reset state.

The `vld`, `rdy`, and `dat` members are basic ports; the member functions available on them are those specified in 7.2.

Example: driving the `msg_in` / `msg_out` ports directly

```
SC_MODULE(dut) {
    sc_in<bool> SC_NAMED(clk);
    sc_in<bool> SC_NAMED(rst_bar);

    msg_in<sc_uint<16>>  SC_NAMED(in);
    msg_out<sc_uint<16>> SC_NAMED(out);

    SC_CTOR(dut) {
        in.enable_port_access();
        out.enable_port_access();
        SC_THREAD(proc);
        sensitive << clk.pos();
        async_reset_signal_is(rst_bar, false);
    }

    void reset() {
        in.rdy.write(false);    // drive idle ports; reset_pop()/reset_push() forbidden
        out.vld.write(false);
        wait();
    }

    void proc() {
        reset();
        while (1) {
            // read in.vld / in.dat, drive in.rdy, out.vld, out.dat ...
            wait();
        }
    }
};
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

18.8 Simulation and Verification Features

This subsection specifies simulation and verification features. A conforming implementation *shall* support the syntax presented below and provide the described behavior. These features are available only in the simulation view; they are not available under synthesis.

18.8.1 Tracing

An implementation *shall* provide `trace_hierarchy`, which traces every library channel in an object hierarchy to a SystemC trace file:

```
void trace_hierarchy(sc_object* obj, sc_trace_file* tf);
void trace_hierarchy(sc_object& obj, sc_trace_file* tf);
```

When invoked after the top-level module has been constructed, it *shall* trace, for each library channel in the hierarchy, the `vld`, `rdy`, and `dat` signals of a `msg_channel`; the `vld` and `dat` signals of a `msg_dv_channel`; and the `vld` and `rdy` signals of a `sync_channel`.

```
sc_trace_file* tf = sc_create_vcd_trace_file("waves");
trace_hierarchy(&top, tf);
```

18.8.2 Transaction Recording

An implementation *shall* provide `channel_logs`, which records every message accepted by a `push` or a successful `push_nb`, and every message returned by a `pop` or a successful `pop_nb`, on every message passing port and channel in a hierarchy (`peek_nb` and rendezvous operations are not recorded):

```
struct emitter_options {
    bool log_direction = true;
    bool unbuffered    = false;
};

class channel_logs {
public:
    int  enable(const std::string& fname = "channel_logs.txt",
                const emitter_options& opts = emitter_options());
    void log_hierarchy(sc_object& obj);
};
```

Function `enable` *shall* select the output file name and recording options, and `log_hierarchy` *shall* attach every channel in the subtree to the log. A `channel_logs` object *shall* outlive `sc_start()`.

The `emitter_options` argument of `enable` is optional; when it is omitted, the default options are used.

The output is textual; each message is written using its `operator<<()`.

Example: transaction recording with the default options

```
channel_logs logs;
logs.enable("run.txt");
logs.log_hierarchy(top);
```


Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

Example: transaction recording with explicit options

```
channel_logs    logs;
emitter_options opts;
opts.unbuffered = true;     // flush the log after each write
logs.enable("run.txt", opts);
logs.log_hierarchy(top);
```

18.8.3 Latency and Capacity Back-Annotation

A writer endpoint (a `msg_out` or `msg_dv_out` port, or a channel written directly by a process) *shall* support back-annotation of its latency (delay in clock cycles) and capacity (items in flight), used to improve pre-synthesis timing accuracy. A writer endpoint *shall* provide:

```
void set_latency_capacity(int latency, int capacity);   // on a writer endpoint
```

which *shall* configure the writer endpoint to model the given latency and capacity for subsequent transfers.

An implementation *shall* also support loading annotations for a hierarchy from a file, and writing a template of the current annotations to a file:

```
namespace annotate {
    bool load_annotations(const std::string& filename);       // text
    bool load_annotations_json(const std::string& filename);  // JSON
    void write_annotations(const std::string& filename);      // emit a template
}
```

```
top.dut_.block0_inst.dout.set_latency_capacity(3, 3);         // programmatic
sc_hls::msg_lib::annotate::load_annotations("latencies.txt"); // from file
```

18.8.4 Random Stall Injection

An implementation *shall* provide `rand_stall::set`, which injects randomized back-pressure stalls into every message passing channel with flow control in a hierarchy, to stress-test synchronization:

```
struct rand_stall {
    static void set(sc_object& obj, bool verbose = false);
};
```

Random stall injection is ***Supported*** for message passing with flow control only; it is ***Not Supported*** for message passing without back-pressure, which has no back-pressure path. The stall sequence *shall* be seeded from the `SC_HLS_RAND_SEED` environment variable, so a given seed reproduces the same sequence.

```
sc_hls::msg_lib::rand_stall::set(top, /*verbose=*/true);
```

18.9 TLM Fast-Simulation View

An implementation *shall* support the `tlm_view_tag` view, which selects a fast functional simulation view in which each channel is backed by a `tlm::tlm_fifo` rather than by `vld`/`rdy`/`dat` signals. It trades throughput accuracy for simulation speed and is intended for fast functional verification and early system-level

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

exploration. This view is ***Not Supported*** under synthesis: `tlm_view_tag` is available only when `__SYNTHESIS__` is not defined.

A channel in this view *shall* expose the same `msg_in_if` / `msg_out_if` reader and writer API (`pop`/`pop_nb`/`peek_nb` and `push`/`push_nb`, with `reset_pop` / `reset_push`) as the simulation view `sim_view_tag`, so process code is unchanged between the two views. The channel constructor *shall* take the FIFO depth as an argument, which defaults to 16:

```
template <typename Message>
class msg_channel<Message, tlm_view_tag> {
public:
    msg_channel(const sc_module_name& nm, unsigned int depth = 16);
};
```

Because the view is a trailing template parameter, a *design* templated on `ViewTag` can be instantiated in either view from a single source.

Example: instantiating a design and its channels in the TLM view

```
dut<tlm_view_tag> SC_NAMED(dut_);

msg_channel<sc_uint<16>, tlm_view_tag> in1{"in1", 20};   // FIFO depth 20
msg_channel<sc_uint<32>, tlm_view_tag> out1{"out1", 20};
```

18.10 Clock Management

A conforming implementation *shall* automatically discover the `sc_clock` objects used by a design.

If a design's clock is not an `sc_clock` (for example a hand-toggled `sc_signal<bool>`), a companion `sc_clock` of matching period and phase *may* be aliased to the signal so the library can schedule against it:

```
void add_clock_alias(sc_clock& real_clk,
                     const sc_core::sc_signal_in_if<bool>& alias_sig);
```

```
sc_clock clk_alias("clk_alias", 10, SC_NS);
add_clock_alias(clk_alias, clk);   // clk is a toggled sc_signal<bool>
```

18.11 Diagnostics

A conforming implementation *shall* report at least the following conditions:

Report	Severity	Condition
`SC_HLS-101`	Fatal	An endpoint operation (`push`, `pop`, `push_nb`, `pop_nb`, `peek_nb`) was invoked without its reset call (`reset_pop` / `reset_push`) having been invoked in the reset state.
`SC_HLS-102`	Error	Two writer endpoints drive the same signal, or an endpoint operation is invoked from a process other than the one that invoked its reset call.
`SC_HLS-401`	Implementation-defined; not reported by default	A `msg_dv` consumer receives a valid beat while the previous beat has not been consumed.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

Report	Severity	Condition
`SC_HLS`	Error	An operation forbidden in direct port-access mode (see 18.7); `enable_port_access` called after elaboration; no `sc_clock` found, or the clock of a process cannot be resolved; a back-annotation on an endpoint whose reset call was never invoked.
`SC_HLS-ANNOTATE`	Warning	An annotation file cannot be opened, is malformed, or names a channel that is not in the hierarchy.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.

Annex A: References

Ref.	Document
[1]	“ISO/IEC 14882:2017, Programming Languages - C++,” 2017.
[2]	IEEE, “IEEE Standard for Standard SystemC Language Reference Manual,” 2023.
[3]	“ISO/IEC 14882:2011, Programming languages - C++,” 2011.

Copyright © 2026 Accellera. All rights reserved.
This is an unapproved Accellera Standards Draft, subject to change.