SystemC 2.0.1 Language Reference Manual

Revision 1.0

1177 Braham Lane #302
San Jose, CA 95118-3799

Acknowledgements

The SystemC 2.0.1 Language Reference Manual (LRM) was developed by representatives from different fields including system architects, design and verification engineers, Electronic Design Automation (EDA) companies and universities. The primary contributors include:

El Mustapha Aboulhamid
Mike Baird
Bishnupriya Bhattacharya
David Black
Dundar Dumlogal
Abhijit Ghosh
Andy Goodrich
Robert Graulich
Thorsten Groetker
Martin Jannsen
Evan Lavelle
Kevin Kranen
Wolfgang Mueller
Kurt Schwartz
Adam Rose
Ray Ryan
Minoru Shoji
Stuart Swan

Mike Baird served as the Language Reference Manual editor.

1 Introduction

1.1 Intent and scope

SystemC is a set of C++ class definitions and a methodology for using these classes. The primary intent of this document is to define the constructs and semantics of SystemC that all compliant implementation must provide. The secondary intent is to provide detailed reference information for the standard SystemC classes and global functions.

This document is not intended as a user's guide or to provide an introduction to SystemC. Readers desiring user-oriented information should consult the Open SystemC Initiative website for such information. For example such users should consult www.systemc.org → Products & Solutions → Books.

The scope of this document encompasses the entire language definition, but does not cover implementation issues. Neither does this document cover methodology issues related to the use of SystemC.

This document is written under the assumption that the reader is familiar with C++.

1.2 Overview of SystemC

This section is informative and describes in general terms a SystemC “system and how it simulates.

The SystemC library of classes and simulation kernel extend C++ to enable the modeling of systems. The extensions include providing for concurrent behavior, a notion of time sequenced operations, data types for describing hardware, structure hierarchy and simulation support.

Figure 1 – SystemC Language Architecture

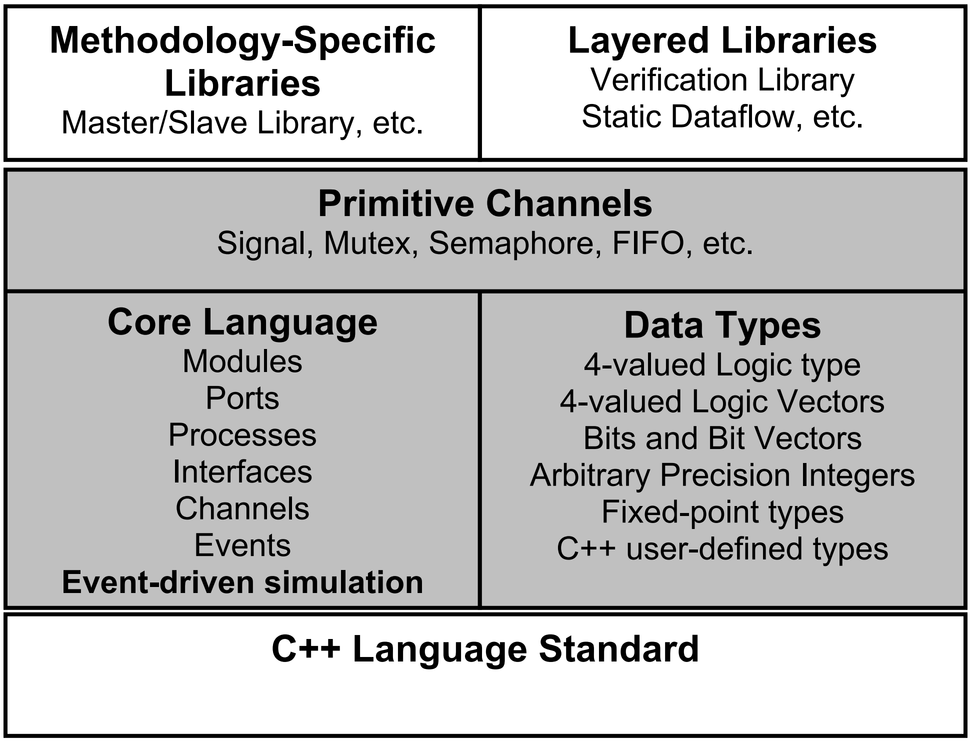


Figure 1 shows the SystemC language architecture. The blocks shaded with gray are part of the SystemC core language standard. SystemC is built on standard C++. The layers above or on top of the SystemC standard consist of design libraries and standards considered to be separate from the SystemC core language. The user may choose to use them or not. Over time other standard or methodology specific libraries may be added and conceivably be incorporated into the core language.

The core language consists of an event-driven simulator as the base. It works with events and processes. The other core language elements consist of modules and ports for representing structure, while interfaces and channels are used to describe communication.

The data types are useful for hardware modeling and certain types of software programming.

The primitive channels are built-in channels that have wide use such as signals and FIFOs.

A SystemC system consists of a set of one or more modules. Modules provide the ability to describe structure. Modules typically contain processes, ports, internal data, channels and possibly instances of other modules. All processes are conceptually concurrent and can be used to model the functionality of the module. Ports are objects through which the module communicates with other modules. The internal data and channels provide for communication between processes and maintaining module state. Module instances provide for hierarchical structures.

Communication between processes inside different modules is accomplished using ports, interfaces and channels. The port of a module is the object through which the process accesses a channels interface. The interface defines the set of access functions for a channel while the channel itself provides the implementation of these functions. At elaboration time the ports of a module are connected (bound) to designated channels. The interface, port, channel structure provides for great flexibility in modeling communication and in model refinement.

Events are the basic synchronization objects. They are used to synchronize between processes and implement blocking behavior in channels. Processes are triggered or caused to run based on sensitivity to events. Both dynamic and static sensitivity are supported. Static sensitivity provides for processes sensitivity that is defined before simulation starts. Dynamic sensitivity provides for process sensitivity that is defined after simulation starts and can be altered during simulation. Processes may wait for a particular event or set of events. Dynamic sensitivity coupled with the ability of processes to wait on one or more events provide for simple modeling at higher levels of abstraction and for efficient simulation.

1.3 Using the SystemC library

Access to all SystemC classes and functions is provided in a single header file named “systemc.h”. This file may include other files, but the end user is only required to include systemc.h.

2 Execution Semantics

This section describes elaboration, initialization and the simulation semantics. SystemC is an event based simulator. Events occur at a given simulation time. Time starts at time = 0 and moves forward only. Time increments are based on the default time unit and the time resolution.

2.1 main() & sc_main()

The function `main()` is part of the SystemC library. It calls the function `sc_main()`,(see Chapter 5)which is the entry point from the library to the user's code.

If the `main()` function provided by the SystemC library does not meet the user's needs, the user will have to mimic SystemC's `main()`. In this case the user will have to make sure the object file containing the new main() function is linked in before the SystemC library.

2.2 Elaboration

Elaboration is defined as the execution of the `sc_main()` function from the start of `sc_main()` to the first invocation of `sc_start()`.

Elaboration may include the construction of instances (instantiation) of modules, and channels to connect them, `sc_clock` objects and `sc_time` variables.

The functions for changing the default time unit (`sc_set_default_time_unit()`, Chapter 12.14) and the time resolution (`sc_set_time_resolution()` , Chapter 12.15) if called, must be called during elaboration. They must also be called before any `sc_time` objects are constructed.

During elaboration, the structural elements of the system are created and connected throughout the system hierarchy. This is facilitated by the C++ class object construction behavior. When a module (or hierarchical channel) comes into existence, it constructs any sub-modules it contains, which in turn initialize their sub-modules, and so forth. As elaboration proceeds port to channel binding occurs. Importantly, there are no constraints on the order in which port to channel binding occurs during elaboration. All that is required is that if a port must be bound to some channel, then the port must be bound by the time elaboration completes.

Finally, the top level modules are connected via channels in the `sc_main()` function.

SystemC does not support the dynamic creation of modules. The structure of the system is created during elaboration time and does not change during simulation.

2.3 Initialization

Initialization is the first step in the SystemC scheduler. Each method process is executed once during initialization and each thread process is executed until a wait statement is encountered.

To turn off initialization for a particular process the `dont_initialize()` function can be called after the SC_METHOD or SC_THREAD process declaration inside a module constructor. A process that is not initialized is not ready to run. That means that the process starts executing with its first statement as soon as it is triggered by the first event.

The order of execution of processes is unspecified. The order of execution between processes is deterministic. This means that two simulation runs using the same version of the same simulator must yield identical results. However, different versions or a different simulator may yield a different result if care is not taken when writing models

2.4 Simulation semantics

The SystemC scheduler controls the timing and order of process execution, handles event notifications and manages updates to channels. It supports the notion of delta-cycles. A delta-cycle consists of the execution of an evaluate and update phase. There may be a variable number of delta-cycles for every simulation time.

SystemC processes are non-preemptive. This means that for thread processes, code delimited by two wait statements will execute without any other process interruption and a method process completes its execution without interruption by another process.

The scheduler is invoked by the execution of the `sc_start()` function. It may be invoked with an explicit amount of time to simulate. Once the scheduler returns, simulation may continue from the time the scheduler last stopped by invoking the `sc_start()` function.

The scheduler may be invoked such that it will run indefinitely. Once started the scheduler continues until either there are no more events, or a process explicitly stops it (by calling the `sc_stop()` function), or an exception condition occurs.

2.4.1 Scheduler Steps

The semantics of the SystemC simulation scheduler is defined by the following eight steps. A delta-cycle consists of steps 2 through 4.

1) *Initialization Phase*. This step is described in Chapter 2.3.

2) *Evaluate Phase*. From the set of processes that are ready to run, select a process and resume its execution. The order in which processes are selected for execution from the set of processes that are ready to run is unspecified.

The execution of a process may cause immediate event notifications to occur, possibly resulting in additional processes becoming ready to run in the same evaluate phase.

The execution of a process may include calls to the `request_update()` function which schedules pending calls to `update()`function in the update phase. The `request_update()` function may only be called inside member functions of a primitive channel.

3) Repeat Step 2 for any other processes that are ready to run.

4) *Update Phase*. Execute any pending calls to `update()` from calls to the `request_update()` function executed in the evaluate phase.

5) If there are pending delta-delay notifications, determine which processes are ready to run and go to step 2.

6) If there are no more timed event notifications, the simulation is finished.

7) Else, advance the current simulation time to the time of the earliest (next) pending timed event notification.

8) Determine which processes become ready to run due to the events that have pending notifications at the current time. Go to step 2.

2.5 Simulation functions

A number of functions are provided for setting up and reporting the timing and controlling the simulation execution.

2.5.1 Starting the simulation

The `sc_start()` function (Chapter 12.17) is called in `sc_main()` to start the scheduler.

Once the `sc_start()` function returns, signifying that the scheduler is done, the user may call `sc_start()` again. The simulation will continue at the time where the scheduler last stopped.

2.5.2 Stopping the simulation

The `sc_stop()` function (Chapter 12.18) is called to stop the scheduler and return control back to the `sc_main()` function. In this case the simulation can not be continued anymore.

2.5.3 Obtaining Current Simulation time

Two functions are provided for the user to obtain the current simulation time, `sc_time_stamp()` (Chapter 12.20) and `sc_simulation_time()` (Chapter 12.16).

3 Time

SystemC uses an integer-valued absolute time model. Time is internally represented by an unsigned integer of at least 64-bits. Time starts at 0, and moves forward only.

3.1 sc_time

The `sc_time` type (Chapter 11.68) is used to represent time or time intervals in SystemC. A `sc_time` object is constructed from a numeric value (of type `double`) and a time unit (of type `sc_time_unit`, Chapter 13.1).

3.2 Time Resolution

The time resolution is the smallest amount of time that can be represented by all sc_time objects in a SystemC simulation. The default value for the time resolution is 1 picosecond (10^{-12} seconds).

A user may set the time resolution to some other value by calling the `sc_set_time_resolution()` function (Chapter 12.15). This function, if called, must be called before any sc_time objects are constructed.

A user may ascertain the current time resolution by calling the `sc_get_time_resolution()` function (Chapter 12.11).

Any time smaller than the time resolution will be rounded off, using round-to-nearest.

3.3 Default Time Unit

Time values may sometimes be specified with a numeric value without time unit. The default time unit is used to specify the unit of time for the values in these cases.

The default value for the default time unit is 1 nanosecond(10^{-9} seconds).

An example use of these types to represent a time value would be in specifying the amount of time in the `sc_start()` function.

Example:

```
// run simulation for 1000 time units
// default time unit = 1ns
sc_start(1000);
```

A user may set the default time unit to some other value by calling the `sc_set_default_time_unit()` function (Chapter 12.14).

A user may ascertain the current default time unit by calling the `sc_get_default_time_unit()` function (Chapter 12.10).

4 Events

An event is an object, represented by class `sc_event` (Chapter 11.11)that determines whether and when a process execution should be triggered or resumed.

In more concrete terms, an event is used to represent a condition that may occur during the course of simulation and to control the triggering of processes.

The `sc_event` class provides basic synchronization for processes. Event notification causes the kernel to call a method process, or to resume a thread process that is sensitive to the event.

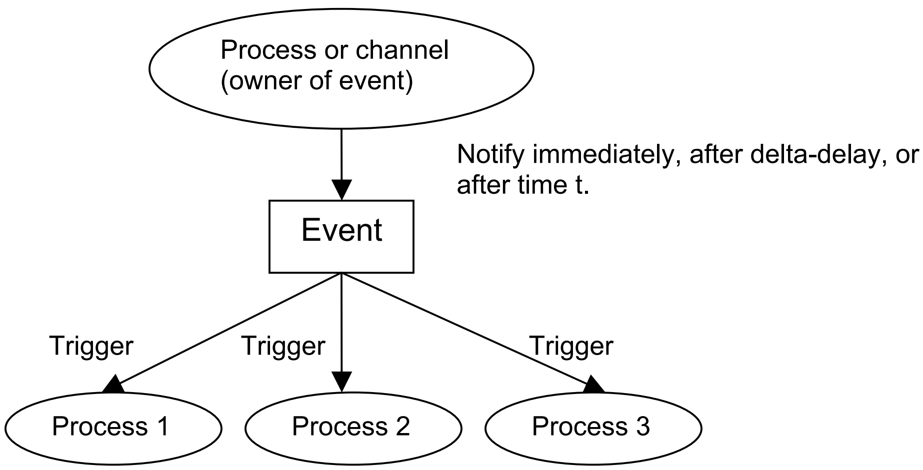
Example:

```
sc_event my_event ;     // event declaration
```


4.1 Event Occurrence

We need to distinguish an event from the actual occurrence of an event. There may be multiple occurrences of an event, and each occurrence is unique though reported through the same event object. An event is usually, though not necessarily, associated with some change of state in a process or of a channel. The owner of the event is responsible for reporting the change to the event. The event object, in turn, is responsible for keeping a list of processes that are sensitive to it. Thus, when notified, the event object will inform the scheduler of which processes to trigger.

Figure 2 Event Occurrence



4.2 Event Notification

Events can be notified in three ways – immediate, delta-cycle delayed and timed. The timing of the notification is specified at invocation of the `notify()` method

Immediate notification means that the event is triggered in the current evaluation phase of the current delta-cycle. The `notify` method with no arguments (`notify()`) indicates immediate notification.

Delta-cycle delayed notification means that the event will be triggered during the evaluate phase of the next delta-cycle. The `notify` method with a time argument specified as 0 (`notify(0, SC_NS)`) or SC_ZERO_TIME (`notify(SC_ZERO_TIME)`) indicates a delta-cycle delayed notification - the event is scheduled for the next delta-cycle.

Timed notification means that the event will be triggered at the specified time in the future. The `notify` method with a non-zero time argument (`notify(10, SC_NS)`) indicates a timed notification. The time of notification is relative to the time of execution of the `notify` method as opposed to an absolute time.

Examples:

```
sc_event my_event ;        // event declaration
sc_time  t (10, SC_NS)     // declaration of a 10 ns time
                           // interval
. . .
my_event.notify();         // immediate notification
...
my_event.notify (SC_ZERO_TIME); // delta-delay
                                // notification
...
my_event.notify (t);            // notification in 10 ns
```

4.3 Multiple event notifications

Events can have only one pending notification, and retain no "memory" of past notifications. Multiple notifications to the same event, without an intermediate trigger are resolved according to the following rule:

An earlier notification will always override one scheduled to occur later, and an immediate notification is always earlier than any delta-cycle delayed or timed notification.

Note that according to this rules, a potential non-determinism exists. Assume that processes A and B are ready to run in the same delta-cycle. Process A issues an immediate notification on an event, and process B issues a delta-cycle delayed notification on the same event. Also, let process C be sensitive to the event. According to the scheduler semantics, processes A and B execute in an unspecified order.

Example

`Process_A {`	`Process_B {`	`Process_C {`
…	…	
`my_event.notify();` …	`my_event.notify(SC_ZERO_TIME);`	`wait(my_event)`
`}`		`}`

If process A executes first, then the event is triggered immediately, causing process C to be executed in the same delta-cycle. Then, process B is executed, and since the event was triggered immediately, there is no conflict and the second notification is accepted, causing process C to be executed again in the next delta-cycle.

If, however, process B executes first, then the delta-cycle delayed notification is scheduled first. Then, process A executes and the immediate notification overrides the delta-cycle delayed notification, causing process C to be executed only once, in the current delta-cycle.

4.4 Canceling event notifications

A pending delayed event notification may be canceled using the `cancel()` method . Immediate event notifications cannot be canceled, since their effect occurs immediately.

Example

```
sc_event a, b, c;
sc_time t(10, SC_MS);

a.notify();              // current delta-cycle
notify(SC_ZERO_TIME, b); // next delta-cycle
notify(t, c);            // 10 ms delay

//Cancel an event notification
a.cancel();  // Error! Can't cancel immediate notification
b.cancel();  // cancel notification on event b
c.cancel();  // cancel notification on event c
```


5 sc_main() Function

The `sc_main()`, function is the entry point from the SystemC library to the user's code. It is called by the function `main()` which is part of the SystemC library. Its prototype is:

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

The arguments `argc` and `argv[]` are the standard command-line arguments. They are passed to `sc_main()` from `main()` in the library.

The body of sc_main() typically consists of configuring simulation variables (default time unit, time resolution, etc.), Instantiation of the module hierarchy and channels, simulation, clean-up and returning a status code.

Elaboration is defined as the execution of the `sc_main()` function from the start of `sc_main()` to the first invocation of `sc_start()`.

The user defines the `sc_main()` function.

Example:

```
int sc_main(int argc, char* argv[ ])
  // Create FIFO channels with a depth of 10
  sc_fifo<int> s1(10);
  sc_fifo<int> s2(10);
  sc_fifo<int> s3(10);

  // Module instantiations
  // Stimulus Generator
  stimgen stim("stim");
  stim(s1, s2);

  // Adder
  adder add("add");
  add(s1, s2, s3);

  // Response Monitor
  monitor mon("mon");
  mon.re(s3);

  // Start simulation
  sc_start();  // run indefinitely

  return 0;
}  // end sc_main()
```


5.1 Module instantiation

The construction of instance(s) (instantiation) of the top level module(s) is done in `sc_main()` before the `sc_start()` function is called for the first time.

Instantiation syntax:

```
module_type module_instance_name(“string_name”);
```

Where:
`module_type` is the module type (a class derived from sc_module).
`module_instance_name` is the module instance name (object name).
`string_name` is the string the module instance is initialized with.

5.2 Port binding

After a module is instantiated in `sc_main()`, binding of its ports to channels may occur. There are two different ways to bind ports.

5.2.1 Named Port Binding

Named port binding explicitly binds a port to a channel.

Named port binding syntax:

```
module_type module_instance_name(“string_name”);
module_instance_name.port_name(channel_name) ;
```

Where:
`module_instance_name` is the instance name of the module.
`port_name` is the instance name of the port being bound.
`channel_name` is the instance name of the channel to which the port is bound.

Example:

```
sc_fifo<int> s3(10);  // channel instantiation
monitor mon("mon");  // module instantiation
mon.re(s3);  // named port binding
```

5.2.2 Positional Port Binding

Positional port binding implicitly binds a port to a channel by mapping the order listing of channel instances to the order of the declaration of the ports within a module.

Positional port binding is limited to modules with 64 or fewer ports.

Positional port binding syntax:

```
module_type module_instance_name("string_name");
module_instance_name(channel_name1, channel_name2, ... ) ;
```

Where:
module_instance_name is the instance name of the module.
channel_nameX is the instance name of the channel to which the port is bound to.
The first channel listed is bound to the first port declared in *module_instance_name*, the second channel listed is bound to the second port declared in *module_instance_name* and so forth.

Example:

```
sc_fifo<int> s1(10);  // channel instantiation
sc_fifo<int> s2(10);  // channel instantiation
sc_fifo<int> s3(10);  // channel instantiation
adder add("add");  // module instantiation
add(s1, s2, s3);  // positional port binding
// s1 bound to first port
// s2 bound to second port
// s3 bound to third port
```

5.3 Simulation function usage

The function `sc_start()` (see Chapter 12.17 for the details of `sc_start()`) is called after configuration of simulation variables (default time unit, time resolution etc.), and elaborations which creates the design structure (instantiation of the module hierarchy and channels, and port binding etc.). This function starts or resumes the SystemC scheduler. On return control is returned to the sc_main() function.

5.4 Function Return

A return of 0 from `sc_main()` indicates a normal return.

Example:

```
int sc_main(int argc, char *argv[ ])
  // Rest of function not shown

  // Start simulation
  sc_start();  // run indefinitely

  return 0;
}  // end sc_main()
```


6 Data types

All C++ data types are supported. In addition SystemC provides types for describing hardware where C++ data types are insufficient.

The copy constructor always creates a copy of the specified object, which has the same value and the same word length.

All SystemC data types T support the streaming operator to print it onto a stream.

```
ostream& operator << ( ostream&, T );
```

6.1 Operators

For SystemC data types the operator symbols always have the same meaning as they have for the native C++ types.

- Arithmetic

+ Add the two operands.
- Subtract the second operand from the first operand.
* Multiply the two operands.
/ Divide the first operand by the second operand.
% Calculate rest of the division of the first operand by the second operand. (modulo operation)

- Bitwise

& Calculate the bitwise AND of the two operands.
| Calculate the bitwise OR of the two operands.
^ Calculate the bitwise XOR of the two operands.

- Arithmetic and bitwise assignment

+= -= *= /= %= &= |= ^=

These operators perform the same calculation as the operators above, but they also assign the result to their first operand.

- Increment and decrement

++ Increment the operand by one and store the result in the operand.
-- Decrement the operand by one and store the result in the operand.

Both operators are available in a prefix and a postfix variant. While they perform the same operation, they differ in what is returned. The prefix version performs the operation first and returns the new value. The postfix version returns the old value while the new value of the operation is stored in the operand.

- Equality and relation

== Return true if the operands are equal.
!= Return true if the operands are not equal.
< Return true if the first operand is less than the second operand.
<= Return true if the first operand is less than or equal to the second operand.
> Return true if the first operand is greater than the second operand.
>= Return true if the first operand is greater than or equal to the second operand.

6.2 Unified String Representation

All data types support a unified string representation. Instances can be converted to that string representation and read from it. This string starts with a prefix that describes the format of what follows:

Table 1 – Unified String Representation

sc_numrep	Prefix	Meaning
SC_DEC	0d	decimal
SC_BIN	0	binary
SC_BIN_US	0bus	binary unsigned
SC_BIN_SM	0bsm	binary sign & magnitude
SC_OCT	0o	octal
SC_OCT_US	0ous	octal unsigned
SC_OCT_SM	0osm	octal sign & magnitude
SC_HEX	0x	hexadecimal
SC_HEX_US	0xus	hexadecimal unsigned
SC_HEX_SM	0xsm	hexadecimal sign & magnitude
SC_CSD	0csd	canonical signed digit

This is followed by some signs and digits, compatible with the format specified by the prefix.

There might be a suffix, denoting the exponent of the number. The exponent starts with an 'E' or 'e', immediately followed by '+' or '-'. Then some decimal digits follow, denoting the exponent. The suffix is only valid for the fixed point data types.

All data types can be converted to an `sc_string` with the member function:

```
sc_string to_string(sc_numrep numrep, bool with_prefix)
```

Where numrep is described in Table 1 above. If with_prefix is false, the resulting string does not contain a prefix, if it is true, the prefix is created.

6.3 Fixed-Precision Integer Types

The following fixed-precision integer types are provided:
  `sc_int<W>` (Chapter 11.38)
  `sc_uint<W>` (Chapter 11.73)

These types are considered a fixed-precision type because the maximum precision is limited to 64 bits. The width of the integer type can be explicitly specified. `sc_int` is a signed integer type in which the value is represented by a 2's complement form and all arithmetic is done in 2's complement. `sc_uint` is unsigned. The underlying operations use 64 bits, but the result size is determined by the type declaration.

Bit select, part select, concatenation and reduction operators are supported. The rightmost bit is the LSB(0), and the leftmost bit is the MSB(width-1).

6.4 Arbitrary Precision Integer Types

The following arbitrary precision integer types are provided:

`sc_bigint<W>` (Chapter 11.3)
`sc_biguint<W>` (Chapter 11.5)

`sc_bigint` is a signed integer type of any size in which the value is represented by a 2’s complement form and all arithmetic is done in 2’s complement. `sc_biguint` is an unsigned integer of any size.

Bit select, part select, concatenation and reduction operators are supported. The rightmost bit is the LSB(0), and the leftmost bit is the MSB(width-1).

6.5 Arbitrary Width Bit Vectors

The arbitrary width bit-vector type is `sc_bv<W>` (Chapter 11.8). This type has two values:

‘0’, sc_logic_0, Log_0:	Interpreted as false
`‘1’, sc_logic_1, Log_1:`	Interpreted as true

Single bit values are represented using type `bool`. The type `sc_bv_base` defines a bit vector of any size. More than one bit is represented with the characters within double quotes (“0011”).

Bit select, part select, concatenation and reduction operators are supported. The rightmost bit is the LSB(0), and the leftmost bit is the MSB(width-1).

6.6 Logic Type

The logic type is `sc_logic` (Chapter 11.43). This type has four values:

‘0’, sc_logic_0, Log_0:	Interpreted as false
`‘1’, sc_logic_1, Log_1:`	Interpreted as true
‘X’, ‘x’, sc_logic_X, Log_X:	Interpreted as unknown
`‘Z’, ‘z’, sc_logic_Z, Log_Z:`	Interpreted as high_impedence

6.7 Arbitrary Width Logic Vectors

The arbitrary width logic vector type is `sc_lv<W>` (Chapter 11.44). This type has four values:

'0', sc_logic_0, Log_0:	Interpreted as false
'1', sc_logic_1, Log_1:	Interpreted as true
'X', 'x', sc_logic_X, Log_X:	Interpreted as unknown
'Z', 'z', sc_logic_Z, Log_Z:	Interpreted as high_impedence

Bit select, part select, concatenation and reduction operators are supported. The rightmost bit is the LSB(0), and the leftmost bit is the MSB(width-1).

6.8 Fixed-point Types

A fixed-point variable that is declared without an initial value is uninitialized. Uninitialized variables can be used anywhere initialized variables can be used. An operation on an uninitialized variable does not produce an error or warning. The result of such an operation is undefined.

6.8.1 Fixed-Point Format

The fixed-point format used by the fixed-point data types consists of three parameters: wl, iwl, and enc.

`wl`:
Total word length, i.e., the total number of bits
`iwl`:
Integer word length, i.e., the number of bits left from the binary point
`enc`:
Sign encoding, i.e., signed (two's complement) and unsigned

The total word length and integer word length parameters are parameters for the fixed-point types. For the two sign encodings, i.e., signed and unsigned, separate fixed-point types will be provided.
The binary point (indicated by `iwl`) can be located outside the `wl` bits. This is explained below.
The fixed-point format can be interpreted according to the following three cases:
`iwl` > `wl`
The number of zeros between the binary point and the LSB of the fixed-point number is `iwl-wl`. See index 1 in Table 2 for an example of this case.
0 <= `iwl` <= `wl`
For examples of this case, see index 2, 3, 4, and 5 in Table 2 .
`iwl` < 0
There are -*iwl* sign extended bits between the binary point and the MSB of the fixed- point number. Since these are sign extended bits, they are not part of the actual fixed-point number. For the unsigned types, the sign extended bits are always zero.
For examples of this case, see index 6 and 7 in Table 2 .

In all three cases, the MSB in the fixed-point representation of the signed types is the sign bit. The sign bit can be behind the binary point.

The range of values for a given signed fixed-point format is as follows:

EQ 1 $[-2^{(iwl-1)}, 2^{(iwl-1)}-2^{-fwl}]$

The range of values for a given unsigned fixed-point format is as follows:

EQ 2 $[0, 2^{iwl}-2^{-fwl}]$

In both equations, `fwl` denotes the fractional word length, i.e., the number of bits right from the binary point.

Table 2. Examples of Fixed-Point Formats

Index	wl `wl`	iwl `iwl`	**Internal representation (*)**	**Range signed**	**Range unsigned**
1	5	7	*xxxxx*00.	[-64,60]	[0,124]
2	5	5	*xxxxx*.	[-16,15]	[0,31]
3	5	3	*xxx*.*xx*	[-4,3.75]	[0,7.75]
4	5	1	*x*.*xxxx*	[-1,0.9375]	[0,1.9375]
5	5	0	.*xxxxx*	[-0.5,0.46875]	[0,0.96875]
6	5	-2	.*ssxxxxx*	[-0.125,0.109375]	[0,0.234375]
7	1	-1	.*sx*	[-0.25,0]	[0,0.25]

(*) *x* is an arbitrary binary digit, 0 or 1. *s* is a sign extended digit, 0 or 1.

6.8.2 Fixed-Point Type Casting

Type casting is essential for fixed-point types. Fixed-point type casting, from now on referred to as type casting in this chapter, is performed by the fixed-point types during initialization (declaration) and assignment. Type casting is performed in two steps:
First, quantization is performed to reduce the number of bits at the LSB (least significant bit) side, if needed.
Next, overflow handling reduces the number of bits at the MSB (most significant bit) side, if needed
If the number of bits at the LSB side does not have to be reduced but has to be extended, then zero extension is used. If the number of bits at the MSB side does not have to be reduced but has to be extended, then sign extension is used. For unsigned fixed-point types, sign extension always means zero extension. One can choose from seven distinct quantization characteristics (from now on referred to as quantization modes) and five distinct overflow characteristics (from now on referred to as overflow modes).

6.8.2.1 Overflow Modes

During overflow handling, bits at the MSB side of a fixed-point number are deleted if the fixed-point number uses more integer bits than specified by a given fixed-point format. The result of overflow handling is a function of both the remaining bits and the deleted bits of the original fixed-point number. The supported and distinct overflow modes are listed in Table 3.

Table 3. Overflow Modes

Overflow Mode	Name
Saturation	SC_SAT
Saturation to zero	SC_SAT_ZERO
Symmetrical saturation	SC_SAT_SYM
Wrap-around (*)	SC_WRAP
Sign magnitude wrap-around (*)	SC_WRAP_SM

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

For a detailed description of each of the overflow modes, refer to Chapter 6.8.12.1.

6.8.2.2 Quantization Modes

During quantization, bits at the LSB side of a fixed-point number are deleted if the fixed-point number uses more fractional bits than specified by a given fixed-point format. The result of quantization is a function of both the remaining bits and the deleted bits of the original fixed-point number.

The supported and distinct quantization modes are listed in Table 4.

Table 4. Quantization Modes

Quantization Mode	Name
Rounding to plus infinity	SC_RND
Rounding to zero	SC_RND_ZERO
Rounding to minus infinity	SC_RND_MIN_INF
Rounding to infinity	SC_RND_INF
Convergent rounding	SC_RND_CONV
Truncation	SC_TRN
Truncation to zero	SC_TRN_ZERO

6.8.3 Fixed-Point Data Types

The following fixed-point data types are provided:

```
sc_fixed<wl,iwl,q_mode,o_mode,n_bits>
sc_ufixed<wl,iwl,q_mode,o_mode,n_bits>
sc_fix
sc_ufix
```

Templatized type sc_fixed and unconstrained type sc_fix are signed (two's complement) types. These types behave the same. The difference between the two types is that the fixed-point type parameters wl, iwl, q_mode, o_mode, and n_bits are part of the type in sc_fixed. Unconstrained type sc_fix allows specifying these parameters as variables, while templatized type sc_fixed requires that these parameters are constant expressions.

Templatized type sc_ufixed and unconstrained type sc_ufix are unsigned types. These types behave the same. The difference between the two types is that the fixed- point type parameters wl, iwl, q_mode, o_mode, and n_bits are part of the type in sc_ufixed. Unconstrained type sc_ufix allows specifying these parameters as variables, while templatized type sc_ufixed requires that these parameters are constant expressions.

For a description of the initialization, operators, functions, bit and part selection, querying the parameters, determining the state, and conversion to primitive, character and SystemC integer types for fixed-point data types see the reference for each in the class reference section.

sc_fixed (Chapter 11.20)
sc_fix (Chapter 11.18)
sc_fixed_fast (Chapter 11.21)
sc_fix_fast (Chapter 11.19)
sc_ufixed (Chapter 11.71)
sc_ufix (Chapter 11.69)
sc_ufixed_fast (Chapter 11.72)
sc_ufix_fast (Chapter 11.20)

6.8.3.1 Limited Precision Fixed-Point Types

All four fixed-point types are arbitrary precision types. To speed up simulations, limited precision versions of the four fixed-point types can be used. These limited precision fixed-point types are:

```
sc_fixed_fast<wl,iwl,q_mode,o_mode,n_bits>
sc_ufixed_fast<wl,iwl,q_mode,o_mode,n_bits>
sc_fix_fast
sc_ufix_fast
```

The limited precision types provide the same API as the corresponding arbitrary precision types. This allows an easy exchange between arbitrary precision types and limited precision types by changing just the types of fixed-point variables. Furthermore, arbitrary precision types and limited precision types can be mixed freely. Because the API is the same, the limited precision types are not described separately.

Limited precision fixed-point types use double precision (floating-point) values instead of arbitrary precision (floating-point) values. The mantissa of a double precision value is limited to 53 bits, whereas the mantissa of an arbitrary precision value is virtually unlimited. This means that bit-true behavior cannot be guaranteed with the limited precision types.
For bit-true behavior with the limited precision types, the following guidelines should be followed:

Make sure that the word length of the result of any operation or expression does not exceed 53 bits.
The result of an addition or subtraction requires a word length that is one bit more than the maximum *aligned* word length of the two operands.
The result of a multiplication requires a word length that is the sum of the word lengths of the two operands.

6.8.4 Fixed-Point Value Type

Arithmetic and bitwise fixed-point operations are performed according to the following paradigm:
First, the operations are performed in arbitrary precision.
Next, the necessary type casting is performed.
Type `sc_fxval` is the arbitrary precision value type. It can hold the value of any of the fixed-point types, and it performs the arbitrary precision fixed-point arithmetic operations. Type casting is performed by the fixed-point types themselves. In cases where arbitrary precision is not needed or too slow, one can use a limited precision type. Type `sc_fxval_fast` is the corresponding limited precision value type, which is limited to a mantissa of 53 bits. See Chapter 6.8.3.1. This type has the same API as type `sc_fxval`. Limited precision type `sc_fxval_fast` and arbitrary precision type `sc_fxval` can be mixed freely.

In some cases, such as division, using arbitrary precision would lead to infinite word lengths. This does not apply to the limited precision type `sc_fxval_fast`, because its precision is already limited, it only applies to `sc_fxval.`
To limit the resulting word lengths in these cases, three parameters are provided. See Chapter 11.28 for a complete description of these parameters. Their built-in default values are given in Chapter 6.8.8.
div_wl - the maximum word length for the result of a division operation.
cte_wl - the maximum word length for the result of converting a decimal character string constant into a sc_fxval variable.
max_wl - the maximum word length for the mantissa used in a sc_fxval variable.
Caution! Be careful with changing the default values of the div_wl, cte_wl, and max_wl parameters, as they affect both bit-true behavior and simulation performance.

Type `sc_fxval` is used to hold fixed-point values for the arbitrary precision fixed-point types. The div_wl, cte_wl, and max_wl parameters should be set higher than the word lengths used by the fixed-point types in the user code, otherwise bit-true behavior cannot be guaranteed. On the other hand, these parameters should not be set too high, because that would degrade simulation performance. Typically, the max_wl parameter should be set (much) higher than the div_wl and cte_wl parameters.

The div_wl, cte_wl, and max_wl parameters will be used by the fixed-point value type sc_fxval, whether used directly or as part of a fixed-point type. By default, the built-in default values given in Chapter 6.8.8 are used. These default values can be overruled per translation unit by specifying the compiler flags SC_FXDIV_WL, SC_FXCTE_WL, and SC_FXMAX_WL with the appropriate values. For example:

```
CC -DSC_FXDIV_WL=128 -c my_file.cpp
```

This compiles my_file.cpp with the div_wl parameter set to 128 bits i.s.o. 64 bits.

For a description of the initialization, operators, functions, determining the state, and conversion to primitive, character and SystemC integer types for fixed-point value types see the reference for each in the class reference section.
sc_fxval (Chapter 11.28)
sc_fxval_fast (Chapter 11.29)

6.8.5 Parameter Types

6.8.5.1 Parameter Type sc_fxtype_param

To configure the type parameters of a variable of fixed-point type `sc_fix`, or `sc_ufix`, (and the corresponding limited precision types), a variable of type `sc_fxtype_params` (Chapter 11.27) can be used. This variable can be passed as an argument when initializing a fixed-point variable. See Chapters 11.18 and 11.69.

6.8.5.2 Parameter Type sc_fxcast_switch

To configure the cast switch parameter of a fixed-point variable, a variable of type `sc_fxcast_switch` (Chapter 11.23) can be used. This variable can be passed as an argument when initializing a fixed-point variable See Chapters 11.18 and 11.69.

6.8.6 Contexts (informative)

This section is for informative purposes only.
This discussion focuses on the fixed-point types, but the same applies to any type that requires additional parameters.

During declaration, the fixed-point types need a number of parameters. Most notably the wl, iwl, o_mode, n_bits, q_mode, and cast_switch parameters. These parameters have to be set during declaration, and they cannot change anymore after declaration.

In some cases, it is not possible to specify these parameters. This is the case when a fixed-point array is declared. In other cases, it becomes cumbersome to have to specify all parameters with each fixed-point variable declaration.

Let's assume that we allow declarations of fixed-point variables where not always all parameters are specified. These variables are therefore incompletely specified. The first problem we face is how to make these variables completely specified. In essence, there are two solutions:

The parameters that are not specified are set to built-in default values. An example is a built-in default value of 32 for the wl parameter.

The parameters that are not specified are fetched from global default values. The most important property of these global default values is that these values can be changed during the execution of the program.

The advantage of the first solution is that all fixed-point variable declarations are actually completely specified, because the unspecified parameters are always the same.

The disadvantage of the first solution is that fixed-point variable declarations are not very flexible. If the built-in default values are unsuitable for a particular use, then the only solution is to specify all parameters with each fixed-point variable declaration. For arrays, this is not possible.

The disadvantage of the second solution is that fixed-point variable declarations can indeed be incompletely specified. Exchanging functions with incompletely specified fixed-point variable declarations has to follow clear rules, such as indicating what the global default values are that are assumed for the function.

The advantage of the second solution is its flexibility. With global default values that can be changed, no particular target (e.g. ASIC or DSP) is assumed. Arrays can be declared with the proper parameters. Furthermore, it is possible to configure (through the global default values) a particular function without affecting other functions in the program. Certain behavior for an entire function can be changed with a single line of code. An example is fixed-point casting. Within a function, fixed-point casting for all fixed-point variables can be switched on or off with a single line of code.

With respect to how the global default values can be changed, the second solution can be refined in two ways:

The user is completely responsible for changing the global default values. It is possible to set new global default values, with the risk that the behavior of other functions changes. This means that in almost all cases the old global default values have to be stored by the user when setting new global default values. The old global default values have to be restored to make sure that other functions are not affected.

The user is responsible only for changing the global default values within a certain part of the program, such as in a certain function and the functions that are directly and indirectly called from this function. Storing the old global default values when setting new global default values and restoring the old global default values is done automatically. This effectively prevents the user from changing the behavior of functions that are not called directly or indirectly from the actual function.

The advantage of the first way is that it is easier to understand and more appealing to C programmers. The disadvantage of the first way is that the behavior of other functions can be changed. Clear rules are needed on how to change the global default values. Enforcement of these rules may be difficult.

The disadvantage of the second way is that it is less easy to understand, because things that are happening, such as restoring the old global default values, are not directly visible from the code. The advantage of the second way is that changing the behavior of other functions, which are not directly or indirectly called from the actual function, is not possible. An exception is when new global default values are set outside of the main function.

Contexts currently implement the second way of the second solution. It is however possible to provide only some of the current functionality. If the first way of the second solution is more desirable, contexts could provide storage for the old global default values. The user would still be responsible for restoring the old global default values

6.8.7 Fixed-Point Context Types

To configure the default behavior of the fixed-point types, a fixed-point context type can be used. A variable of a fixed-point context type is not passed as an argument to the fixed-point types.
During declaration of a variable of a fixed-point context type, the values specified become the new default values. The old default values are stored. When the variable goes out of scope, the old default values are restored. It is possible to set the new default values after declaring the context variable. It is also possible to restore the old default values before the context variable goes out of scope.
Two fixed-point context types are provided: `sc_fxtype_context` (Chapter 11.26) and `sc_fxcast_context` (Chapter 11.22).

6.8.8 Built-in Default Values

The set of built-in default values for the parameters of the fixed-point types and the fixed-point value type are listed in Table 5.

Table 5 – Built-in Default Values

Parameter	**Value**
wl	32
iwl	32
q_mode	SC_TRN
o_mode	SC_WRAP
n_bits	0
cast_switch	SC_ON
div_wl	64
cte_wl	64
max_wl	1024

6.8.9 Conversion to/from Character String

For the fixed-point types and the value types, conversion to and from character string is supported. Conversion to character string is supported with the `to_string()` method. Conversion from character string is supported with constructors, assignment operators, and binary operators.

6.8.9.1 Conversions to Character String

Conversion to character string of the fixed-point types and the value types is supported by the `to_string()` method. The syntax of this method is:

var_name`.to_string([`*numrep*`][,`*fmt*`])`

`var_name`
The name of the variable, whose value is to be converted to character string.

`numrep`
The number representation to be used in the character string. The numrep argument is of type `sc_numrep`. Valid values for `numrep` are given in Table 6. The default value for `numrep` is `SC_DEC`.

Table 6 – Number Representations

Value	Description	Prefix
SC_DEC	decimal, sign mangnitude	
SC_BIN	binary, two's complement	0b
SC_BIN_US	binary, unsigned	0bus
SC_BIN_SM	binary, sign magnitude	0bsm
SC_OCT	octal, two's complement	0o
SC_OCT_US	octal, unsigned	0ous
SC_OCT_SM	octal, sign magnitude	0osm
SC_HEX	hexadecimal, two's complement	0x
SC_HEX_US	US hexadecimal, unsigned	0xus
SC_HEX_SM	hexadecimal, sign magnitude	0xsm
SC_CSD	canonical signed digit	0csd

`fmt`

Format to use for the resulting character string. The `fmt` argument is of type `sc_fmt`. Valid values for `sc_fmt` are:

```
SC_F fixed
SC_E scientific
```

The default value for `fmt` is `SC_F` for the fixed-point types. For type `sc_fxval`, the default value for `fmt` is `SC_E`.

The selected format gives different character strings only when the binary point is not located within the *wl* bits. In that case, either sign extension (MSB side) or zero extension (LSB side) has to be done (`SC_F` format), or exponents are used (`SC_E` format).

As an example, consider a fixed-point type variable with *wl*=4 and *iwl*=6. Converting the value 20 to a two's complement binary character string without prefix results in:

```
010100 (SC_F format)
0101e+2 (SC_E format)
```

In the scientific format, the `+` (or `-`) after the `'e'` is mandatory.

The `to_string()` method returns a value of type `const char*`. If this return value is to be stored for later usage, it must be copied. For short lifetime usage, such as printing, copying is not needed.

The difference between converting fixed-point variables and value variables to character string is the number of bits printed. For fixed-point variables, at least the *wl* bits are printed. For value variables, only those bits are printed that are necessary to uniquely represent the value.

EXAMPLE:
```
sc_fixed<4,2> a = -1;
printf(a.to_string()); // writes “-1”
printf(a.to_string(SC_BIN)); // writes “0b11.00”
```

6.8.9.2 Shortcut Methods

For debugging and/or convenience reasons, several shortcut methods to the `to_string` method are provided for frequently used combinations of arguments. The shortcut methods are listed in Table 7.

Table 7 – Shortcut Methods

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

The shortcut methods use the default format as defined above.

EXAMPLE:
```
sc_fixed<4,2> a = -1;
printf(a.to_dec()); // writes “-1”
printf(a.to_bin()); // writes “0b11.00”
```

6.8.9.3 Conversion from Character String

A character string can be used during initialization (declaration), assignment, and in expressions with fixed-point variables and value variables. The character string is converted into a value object.

Note:

A character string is seen as value, i.e., the size of the character string is not used in any way to determine the size of a fixed-point variable.

6.8.9.4 Conversion to/from bit vector Character String

Conversion to and from bit vector character strings is done through part selection. Conversion to a bit vector character string can be done as follows:
```
sc_fixed<8,8> a = -1;
printf(a.range(7,0).to_string());
  // prints “11111111”
cout << a.range(7,0); // ditto
```

Conversion from a bit vector character string can be done as follows:
```
sc_fixed<8,8> a;
a.range(7,0) = “11111111”; // a gets -1
```

Instead of specifying the full range as arguments to the `range()` method, the shortcut without any arguments can be used as well.

6.8.10 Fixed-Point Array Declaration

When one declares a fixed-point variable, one can specify the appropriate parameters as constructor arguments. When declaring an array of fixed-point variables, however, one cannot use this method. C++ does not allow one to declare an array of a certain type and specify constructor arguments. In this case, the default constructor is called for each element in the array.
For the fixed-point types `sc_fix` and `sc_ufix`, this restriction can be circumvented by specifying the appropriate type parameters up front as default values with the fixed- point context type `sc_fxtype_context`. For example:

```
sc_fxtype_context c1(16,1,SC_RND_CONV,SC_SAT_SYM);
sc_fix a[10];
```

For the fixed-point types `sc_fixed` and `sc_ufixed`, the type parameters are part of the type. Hence, an array of these types can be declared in a straightforward manner. For example:

```
sc_fixed<32,32> a[10];
sc_ufixed<16,1,SC_RND_CONV,SC_SAT_SYM> b[4];
```

Only the cast switch parameter is an optional argument to the constructors of the fixed- point types. To declare a fixed-point array with casting switched off or with casting switched with a variable, this requires that the appropriate cast switch value is specified up front as default value with the fixed-point context type `sc_fxcast_context`. For example:

```
sc_fxcast_context no_casting(SC_OFF);
sc_fixed<8,8> a[10];
```

6.8.11 Observation

For observing fixed-point variables and fixed-point value variables, two mechanisms are provided. First of all, the SystemC trace functions can be used with fixed-point variables and fixed-point value variables. Second, observer abstract base classes are provided as hooks to define one's own observer functionality.

The following observer abstract base classes are provided:

```
sc_fxnum_observer
sc_fxnum_fast_observer
sc_fxval_observer
sc_fxval_fast_observer
```

6.8.12 Finite Word length Effects

SystemC implements fixed-point arithmetic, i.e., computations are performed with a finite number of bits. Because of this, quantization and/or overflow occurs. In addition to the fixed-point arithmetic, SystemC also provides a number of modes to deal with these effects.
When applying these quantization and overflow modes, keep in mind that fixed-point numbers in SystemC can be signed or unsigned. Some overflow and

quantization modes favor a 2’s complement representation, while others favor a 1’s complement representation.
The quantization and overflow handling process works along the following steps:
An operation is performed with a temporary result type that does not generate any overflow or quantization effect, i.e., the operation is performed with full precision.
During fixed-point type casting, the temporary result is quantized as specified. Note here that overflow may occur.
The appropriate overflow behavior is then applied to the result of the process up until now, which gives the final value.

6.8.12.1 Overflow Modes

Overflow occurs when a result of an arithmetic operation needs more bits than can be represented. Specific overflow modes can then be used.
The supported overflow modes are listed in Table 8. They are mutually exclusive. The default overflow mode is `SC_WRAP`. When using a wrap-around overflow mode, the number of saturated bits (*n_bits*) is by default set to 0, but can be modified.

Table 8 – Overflow Modes

Overflow Mode	Name
Saturation	SC_SAT
Saturation to zero	SC_SAT_ZERO
Symmetrical saturation	SC_SAT_SYM
Wrap-around (*)	SC_WRAP
Sign magnitude wrap-around (*)	SC_WRAP_SM

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

In what follows, each of the overflow modes will be explained in more detail. A figure will be given to explain the behavior graphically. The x-axis shows the input values and the y-axis represents the output values. Together they determine what is called the overflow mode.

In order to facilitate the explanation of each overflow mode, the concepts *MIN* and *MAX* are used:
In case of signed numbers, MIN is the lowest (negative) number that can be represented; MAX is the highest (positive) number that can be represented with a certain number of bits. A **value** *x* lies then in the range:

-2^{n-1} (= MIN) • x • 2n-1 - 1 (= MAX). n indicates the number of bits.

In case of unsigned numbers, MIN equals 0 and MAX equals 2*n* - 1.
n indicates the number of bits.

6.8.12.1.1 Overflow for Signed Fixed-Point Numbers

The following template contains a signed fixed-point number before and after an overflow mode has been applied and a number of flags which are explained below. The flags between parentheses indicate additional optional properties of a bit.

Before:	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*
After:						*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*
Flags:	*sD*	*D*	*D*	*D*	*lD*	*sR*	*R(N)*	*R(lN)*	*R*	*R*	*R*	*R*	*R*	*R*	*R*	*R*	*lR*

The following flags and symbols are used in the template above and in Table :
x. A binary digit (0 or 1).
sD. Sign bit before overflow handling.
Deleted bits.
lD. Least significant deleted bit.
sR. Bit on the MSB position of the result number. For the `SC_WRAP_SM`, 0 and `SC_WRAP_SM`, 1 modes a distinction is made between the original value (*sRo*) and the new value (*sRn*) of this bit.
N. Saturated bits. Their number is equal to the *n_bits* argument minus 1. They are always taken after the sign bit of the result number. The *n_bits* argument is only taken into account for the `SC_WRAP` and `SC_WRAP_SM` overflow modes.
lN. Least significant saturated bit. This flag is only relevant for the `SC_WRAP` and `SC_WRAP_SM` overflow modes. For the other overflow modes these bits are treated as R-bits. For the `SC_WRAP_SM`, *n_bits* > 1 mode, *lNo* represents the original value of this bit.
R. Remaining bits.
lR. Least significant remaining bit.
There is always overflow when the value of at least one of the deleted bits (*sD*, *D*, *lD*) is not equal to the original value of the bit on the MSB position of the result (*sRo*). For example, a number of type `sc_fixed<31,11>` is cast to a `sc_fixed<28,8>` number. Overflow for Unsigned Fixed-Point Numbers

Bit 27, when we start counting from 0 at the LSB side of the number, equals 1. If any of the bits 28, 29 or 30 of the initial number equals 0, there is an overflow. In the other case, all bits except for the deleted bits are copied to the result number.

Table 9 shows how a signed fixed-point number is cast (in case there is an overflow) for each of the possible overflow modes. The operators used in the table are “!” for a bitwise negation and “^” for a bitwise exclusive-or.

Table 9 – Overflow Handling for Signed Fixed-Point Numbers

Overflow Mode	**Result**		
	Sign Bit *(sR)*	**Saturated Bits** *(N, lN)*	**Remaining Bits** *(R, lR)*
SC_SAT	*sD*		! *sD*
	The result number gets the sign bit of the original number. The remaining bits get the inverse value of the sign bit.		
SC_SAT_ZERO	0		0
	All bits are set to zero.		
SC_SAT_SYM	*sD*		! *sD*,
	The result number gets the sign bit of the original number. The remaining bits get the inverse value of the sign bit, except the least significant remaining bit, which is set to one.		
SC_WRAP, (*n_bits* =) 0	*sR*		*x*
	All bits except for the deleted bits are copied to the result number.		
SC_WRAP, (*n_bits* =) 1	*sD*		*x*
	The result number gets the sign bit of the original number. The remaining bits are simply copied from the original number.		
SC_WRAP, *n_bits* > 1	*sD*	! *sD*	*x*
	The result number gets the sign bit of the original number. The saturated bits get the inverse value of the sign bit of the original number. The remaining bits are sim ply copied.		
SC_WRAP_SM, (*n_bits* =) 0	*lD*		*x* ^ *sRo* ^ *sRn*
	The sign bit of the result number gets the value of the least significant deleted bit. The remaining bits are exor-ed with the original and the new value of the sign bit of the result number.		
SC_WRAP_SM, (*n_bits* =) 1	*sD*		*x* ^ *sRo* ^ *sRn*
	The result number gets the sign bit of the original number. The remaining bits are exor-ed with the original and the new value of the sign bit of the result number.		
SC_WRAP_SM, *n_bits* > 1	*sD*	! *sD*	*x* ^ *lNo* ^ ! *sD*
	The result number gets the sign bit of the original number. The saturated bits get the inverse value of the sign bit of the original number. The remaining bits are exor-ed with the original value of the least significant saturated bit and the inverse value of the		

	original sign bit.

6.8.12.1.2 Overflow for Unsigned Fixed-Point Numbers

The following template contains an unsigned fixed-point number before and after an overflow mode has been applied and a number of flags, which are explained below.

Before:	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*
After:						*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*
Flags:	*D*	*D*	*D*	*D*	*lD*	*R(N)*	*R(N)*	*R(N)*	*R*	*R*	*R*	*R*	*R*	*R*	*R*	*R*

The following flags and symbols are used in the template above and in Table 10:

x. A binary digit (0 or 1).

Deleted bits.

lD. Least significant deleted bit.

N. Saturated bits. Their number is equal to the *n_bits* argument. The *n_bits* argument is only taken into account for the `SC_WRAP` and `SC_WRAP_SM` overflow modes.

R. Remaining bits.

Table 10 shows how an unsigned fixed-point number is cast in case there is an overflow for each of the possible overflow modes.

Table 10 – Overflow Handling for Unsigned Fixed-Point Numbers

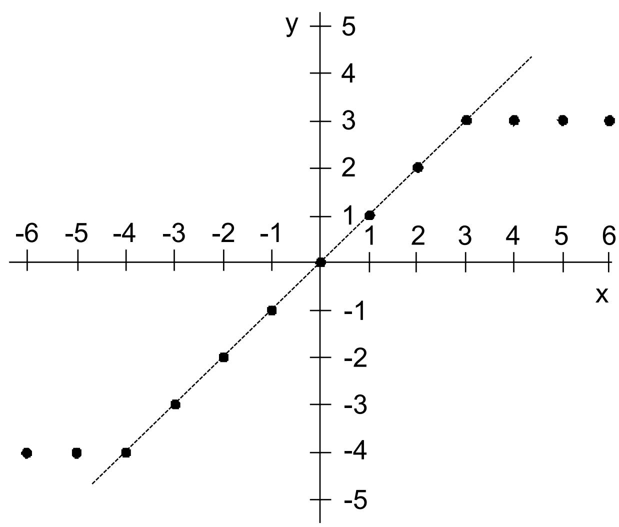
Overflow Mode	Result	
	Saturated Bits (*N*)	**Remaining Bits (*R*)**
SC_SAT		1 (overflow) 0 (underflow)
	The remaining bits are set to 1 (overflow) or 0 (underflow).	
SC_SAT_ZERO		0
	The remaining bits are set to 0.	
SC_SAT_SYM		1 (overflow) 0 (underflow)
	The remaining bits are set to 1 (overflow) or 0 (underflow).	
SC_WRAP, (*n_bits* =) 0		*x*
	All bits except for the deleted bits are copied to the result number.	
SC_WRAP, *n_bits* > 0	1	*x*
	The saturated bits of the result number are set to 1. The remaining bits are copied to the result number.	
SC_WRAP_SM	Not defined for unsigned numbers.	

During the conversion from signed to unsigned, sign extension occurs before overflow handling, while in the unsigned to signed conversion, zero extension occurs first.

6.8.12.2 SC_SAT

Use the SC_SAT overflow mode to indicate that the output is saturated to MAX in case of overflow or to MIN in the case of negative overflow. The ideal situation is represented by the diagonal dashed line, as illustrated in Figure 3.

Figure 3 - Saturation



EXAMPLE (signed):
You specify a word length of three bits. Figure 3 - Saturation
illustrates the possible values when the SC_SAT overflow mode for signed numbers is taken into account.

```
0110 (6)
after saturation: 011 (3)
```

There is an overflow because the decimal number 6 is outside the range of values that can be represented exactly by means of three bits. The result is then rounded to the highest positive representable number, which is 3.

```
1011 (-5)
after saturation: 100 (-4)
```

There is an overflow because the decimal number -5 is outside the range of values that can be represented exactly by means of three bits. The result is then rounded to the lowest negative representable number, which is -4.

EXAMPLE (unsigned):
The result number is three bits wide.

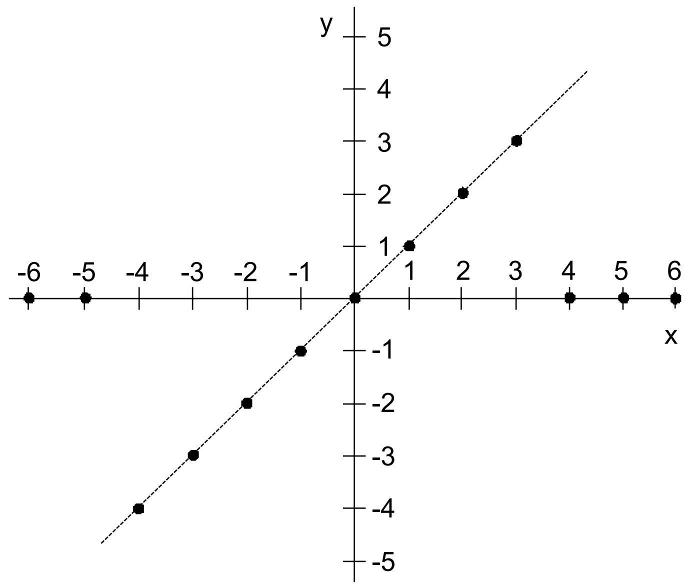
```
01110 (14)
after saturation: 111 (7)
```

The SC_SAT mode corresponds to the SC_WRAP and SC_WRAP_SM modes with the number of bits to be saturated equal to the number of kept bits.

6.8.12.3 SC_SAT_ZERO

Use the SC_SAT_ZERO overflow mode to indicate that the output is forced to zero in case of an overflow, that is, if MAX or MIN is exceeded.

Figure 4 – Saturation to Zero



EXAMPLE (signed):
You specify a word length of three bits. Figure 4 – Saturation to Zero illustrates the possible values for this word length when SC_SAT_ZERO is taken into account as overflow mode.

```
0110 (6)
after saturation to zero: 000 (0)
```

There is an overflow because the decimal number 6 is outside the range of values that can be represented exactly by means of three bits. The result is saturated to zero.

```
1011 (-5)
after saturation to zero: 000 (0)
```

There is an overflow because the decimal number -5 is outside the range of values that can be represented exactly by means of three bits. The result is saturated to zero.

EXAMPLE (unsigned):
The result number is three bits wide.

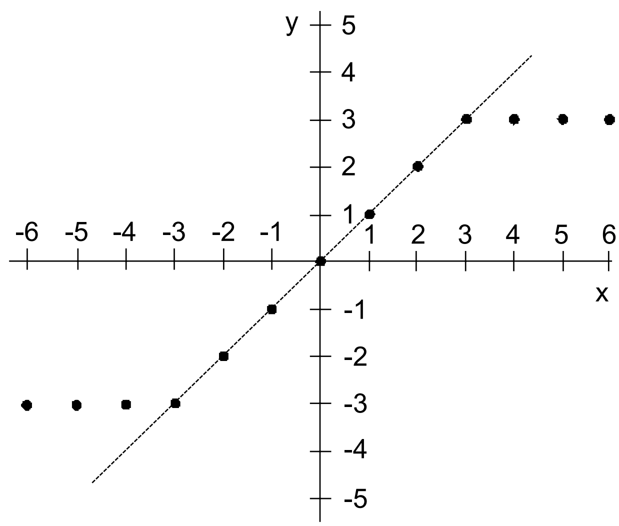
```
01110 (14)
after saturation to zero: 000 (0)
```


6.8.12.4 SC_SAT_SYM

Use the `SC_SAT_SYM` overflow mode to indicate that the output is saturated to MAX in case of overflow or to -MAX (signed) or MIN (unsigned) in the case of negative overflow. The ideal situation is represented by the diagonal dashed line, as illustrated in Figure 5 – Symmetrical Saturation

.

Figure 5 – Symmetrical Saturation



EXAMPLE (signed):
You specify a word length of three bits. Figure 5 illustrates the possible values when the `SC_SAT_SYM` overflow mode for signed numbers is taken into account.

```
0110 (6)
after symmetrical saturation: 011 (3)
```

There is an overflow because the decimal number 6 is outside the range of values that can be represented exactly by means of three bits. The result is then rounded to the highest positive representable number, which is 3.

```
1011 (-5)
after symmetrical saturation: 101 (-3)
```

There is an overflow because the decimal number -5 is outside the range of values that can be represented exactly by means of three bits. The result is then rounded to minus the highest positive representable number, which is -3.

EXAMPLE (unsigned):
The result number is three bits wide.

```
01110 (14)
after symmetrical saturation: 111 (7)
```

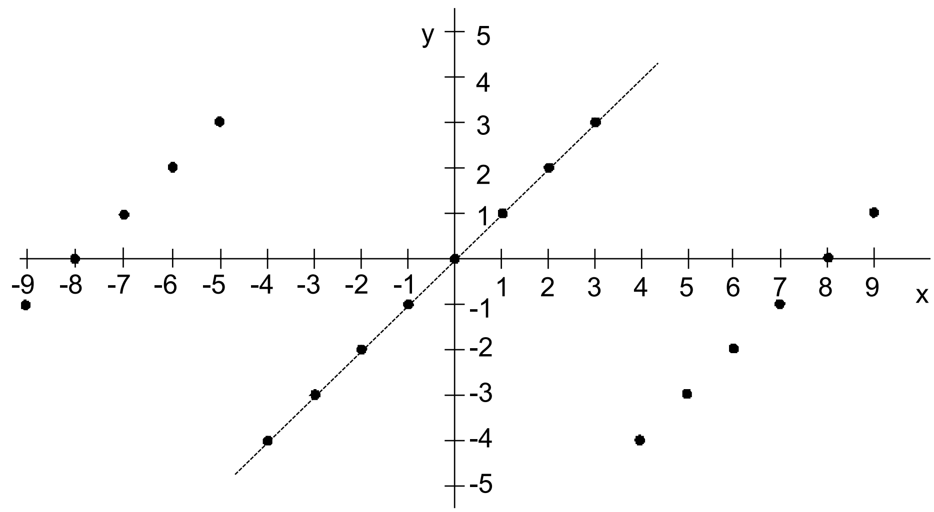

6.8.12.5 SC_WRAP

Use the `SC_WRAP` overflow mode to indicate that the output is wrapped around in the case of overflow. Two different cases are discussed: one with the *n_bits* parameter set to 0, and one with the *n_bits* parameter greater than 0.

```
SC_WRAP, 0
```

This is the default overflow mode. All bits except for the deleted bits are copied to the result number.

Figure 6 – Wrap-Around with n_bits = 0



EXAMPLE (signed):
You specify a word length of three bits. Figure 6 illustrates the possible values for this word length when wrapping around with zero bits is taken into account as overflow mode and when you use signed numbers.

```
0100 (4)
 after wrapping around with 0 bits: 100 (-4)
```

There is an overflow because the decimal number 4 is outside the range of values that can be represented exactly by means of three bits. The MSB is truncated and the result becomes negative: -4.

```
1011 (-5)
after wrapping around with 0 bits: 011 (3)
```

There is an overflow because the decimal number -5 is outside the range of values that can be represented exactly by means of three bits. The MSB is truncated and the result becomes positive: 3

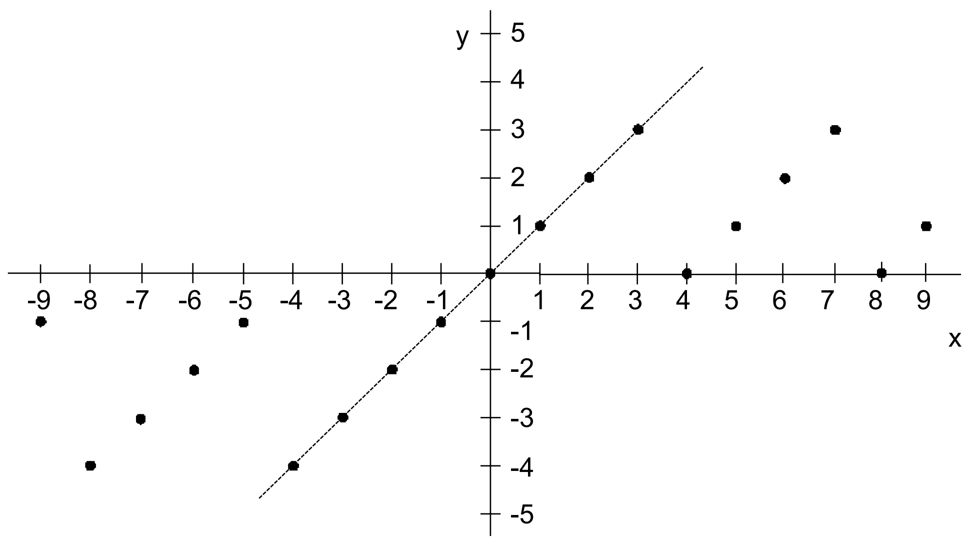
EXAMPLE (unsigned):
The result number is three bits wide.

```
11011 (27)
after wrapping around with 0 bits: 011 (3)
```

SC_WRAP, *n_bits* > 0: SC_WRAP, 1

Whenever *n_bits* is greater than 0, the specified number of bits on the MSB side of the result number are saturated with preservation of the original sign; the other bits are simply copied. Positive numbers remain positive; negative numbers remain negative.

Figure 7 – Wrap-Around with n_bits = 1



EXAMPLE (signed):
You specify a word length of three bits for the result. Figure 7 – Wrap-Around with n_bits = 1
illustrates the possible values for this word length when wrapping around with one bit is taken into account for the overflow mode.

```
0101 (5)
after wrapping around with 1 bit: 001 (1)
```

There is an overflow because the decimal number 5 is outside the range of values that can be represented exactly by means of three bits. The sign bit is kept, so that positive numbers remain positive.

```
1011 (-5)
after wrapping around with 1 bit: 111 (-1)
```


There is an overflow because the decimal number -5 is outside the range of values that can be represented exactly by means of three bits. The MSB is truncated, but the sign bit is kept, so that negative numbers remain negative.

EXAMPLE (unsigned):
For this example the SC_WRAP, 3 mode is applied. The result number is five bits wide. The 3 bits at the MSB side are set to 1; the remaining bits are copied.

```
0110010 (50)
after wrapping around with 3 bits: 11110 (30)
```

6.8.12.6 SC_WRAP_SM

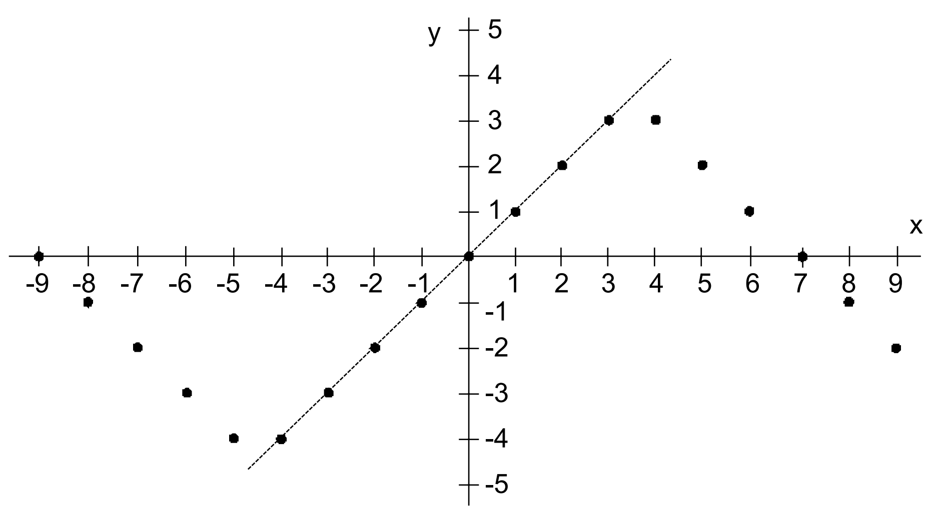
Use the SC_WRAP_SM overflow mode to indicate that the output is sign magnitude wrapped around in the case of overflow. The *n_bits* parameter again indicates the number of bits (for example, 1) on the MSB side of the cast number that are saturated with preservation of the original sign.
Below, you get two different cases of SC_WRAP_SM:

C_WRAP_SM with parameter *n_bits* = 0
SC_WRAP_SM with parameter *n_bits* > 0

```
SC_WRAP_SM, 0
```

The MSBs outside the required word length are deleted. The sign bit of the result number gets the value of the least significant of the deleted bits. The other bits are inverted in case the original and the new values of the most significant of the kept bits differ. Otherwise, the other bits are simply copied from the original to the result number.

Figure 8 – Sign Magnitude Wrap-Around with n_bits = 0



EXAMPLE:

If you want to cast a decimal number 4 into three bits and you use the overflow mode `SC_WRAP_SM`, 0, this is what happens:

```
0100 (4)
```

The original representation is truncated in order to be put in a three bit number:

```
100 (-4)
```

The new sign bit is 0. This is the value of least significant deleted bit. Because the original and the new value of the new sign bit differ, the values of the remaining bits are inverted:

```
011 (3)
```

This principle can be applied to all numbers that cannot be represented exactly by means of three bits.

Table 11 - Sign Magnitude Wrap-Around with n_bits = 0 for a Three Bit Number

Decimal	Binary
8	111
7	000
6	001
5	010
4	011
3	011
2	010
1	001
0	000
-1	111
-2	110
-3	101
-4	100
-5	100
-6	101
-7	110

`SC_WRAP_SM, n_bits > 0`

The first *n_bits* bits on the MSB side of the result number are:
Saturated to MAX in case of a positive number
Saturated to MIN in case of a negative number

Positive numbers remain positive and negative numbers remain negative.
In case *n_bits* equals 1 the other bits are copied and exor-ed with the original and the new value of the sign bit of the result number. In case *n_bits* is greater than 1, the remaining bits are exor-ed with the original value of the least significant saturated bit and the inverse value of the original sign bit.

`SC_WRAP_SM, n_bits > 0: SC_WRAP_SM, 3`

The first three bits on the MSB side of the cast number are saturated to MAX or MIN.
If you want to cast the decimal number 234 into five bits and you use the overflow mode `SC_WRAP_SM`, 3, this is what happens:


```
011101010 (234)
```

The original representation is truncated to five bits:

```
01010
```

The original sign bit is copied to the new MSB (bit position 4, starting from bit position 0):

```
01010
```

The bits at position 2, 3 and 4 are saturated; they are converted to the maximum value you can express with three bits without changing the sign bit:

```
01110
```

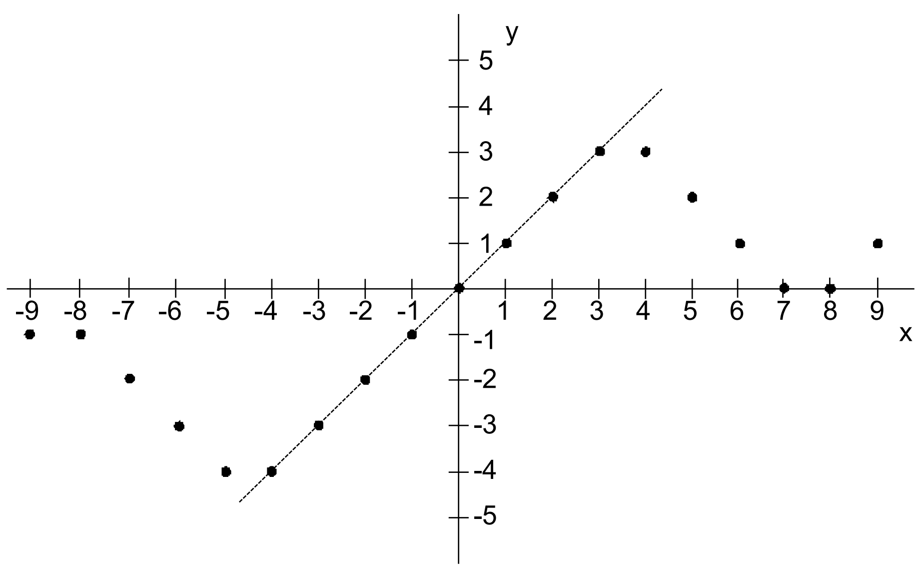
The original value of the bit on position 2 was 0. The remaining bits at the LSB side (10) are exor-ed with this value and with the inverse value of the original sign bit, that is, with 0 and 1 respectively.

```
01101 (13)
```

SC_WRAP_SM, *n_bits* > 0: SC_WRAP_SM, 1

The first bit on the MSB side of the cast number gets the value of the original sign bit. The other bits are copied and exor-ed with the original and the new value of the sign bit of the result number.

Figure 9 – Sign Magnitude Wrap-Around with n_bits = 1



If you want to cast the decimal number 12 into three bits and you use the overflow mode SC_WRAP_SM, 1, this is what happens.

```
01100 (12)
```

The original representation is truncated to three bits.

```
100
```

The original sign bit is copied to the new MSB (bit position 2, starting from bit position 0).

```
000
```

The two remaining bits at the LSB side are exor-ed with the original (1) and the new value (0) of the new sign bit.

```
011
```

This principle can be applied to all numbers that cannot be represented exactly by means of three bits.

Table 12 – Sign Magnitude Wrap-around with n_bits = 1 for a Three Bit Number

Decimal	Binary
9	001
8	000
7	000
6	001
5	010
4	011
3	011
2	010
1	001
0	000
-1	111
-2	110
-3	101
-4	100
-5	100
-6	101
-7	110
-8	111
-9	111

6.8.12.7 Quantization Modes

Aside from overflow modes, also quantization modes can be used to approximate a higher precision.
The supported quantization modes are listed in Table . They are mutually exclusive. The default quantization mode is SC_TRN.

Table 13 – Quantization Modes

Quantization Mode	Name
Rounding to plus infinity	SC_RND
Rounding to zero	SC_RND_ZERO
Rounding to minus infinity	SC_RND_MIN_INF
Rounding to infinity	SC_RND_INF
Convergent rounding	SC_RND_CONV
Truncation	SC_TRN
Truncation to zero	SC_TRN_ZERO

Each of the following quantization modes is followed by a figure. On the x-axis you find the input values, on the y-axis the output values. Together they determine what is called the quantization mode. In each figure, the quantization mode specified by the respective keyword is combined with the ideal characteristic. This ideal characteristic is represented by the diagonal dashed line. Before each quantization mode is discussed in detail, an overview is given of how the different quantization modes deal with quantization for signed and unsigned fixed-point numbers.

6.8.12.7.1 Quantization for Signed Fixed-Point Numbers

The following template contains a signed fixed-point number in 2's complement representation before and after a quantization mode has been applied and a number of flags. These are explained below.

Before:	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*
After:	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*						
Flags:	*sR*	*R*	*R*	*R*	*R*	*R*	*R*	*R*	*lR*	*mD*	*D*	*D*	*D*	*D*	*D*

The following flags and symbols are used in the template above and in Table :

x. A binary digit (0 or 1).
sR. Sign bit.
R. Remaining bits.
lR. Least significant remaining bit.
mD. Most significant deleted bit.
Deleted bits.
r. Logical or of the deleted bits except for the *mD* bit in the template above. When there are no remaining bits, *r* is false. This means that *r* is false when the two nearest numbers are at equal distance.

Table 14 shows how a signed fixed-point number is cast for each of the possible quantization modes in case there is quantization. If the two nearest representable numbers are not at equal distance, the result is, of course, the nearest representable number. This can be found by applying the `SC_RND` mode, that is, by adding the most significant of the deleted bits to the remaining bits.
The right hand column in Table contains the expression that has to be added to the remaining bits. It always evaluates to a one or a zero. The operators used in the table are "!" for a bitwise negation, "|" for a bitwise or, and "&" for a bitwise and.

Table 14 – Quantization Handling for Signed Fixed-Point Numbers

Quantization Mode	Expression to Be Added
SC_RND	*mD*
	Add the most significant deleted bit to the remaining bits.
SC_RND_ZERO	*mD* & (*sR* | *r*)
	If the most significant deleted bit is 1, and either the sign bit or at least one other deleted bit is 1, add 1 to the remaining bits.
SC_RND_MIN_INF	*mD* & *r*
	If the most significant deleted bit is 1 and at least one other deleted bit is 1, add 1 to the remaining bits.
SC_RND_INF	*mD* & (! *sR* | *r*)
	If the most significant deleted bit is 1, and either the inverted value of the sign bit or at least one other deleted bit is 1, add 1 to the remaining bits.
SC_RND_CONV	*mD* & (*lR* | *r*)
	If the most significant deleted bit is 1, and either the least significant of the remaining bits or at least one other deleted bit is 1, add 1 to the remaining bits.
SC_TRN	0
	Just copy the remaining bits.
SC_TRN_ZERO	*sR* & (*mD* | *r*)
	If the sign bit is 1, and either the most significant deleted bit or at least one other deleted bit is 1, add 1 to the remaining bits.

6.8.12.7.2 Quantization for Unsigned Fixed-Point Numbers

The following template contains an unsigned fixed-point number before and after a quantization mode has been applied, and a number of flags. These are explained below.

Before:	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*
After:	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*						
Flags:	*R*	*R*	*R*	*R*	*R*	*R*	*R*	*R*	*lR*	*mD*	*D*	*D*	*D*	*D*	*D*

The following flags and symbols are used in the template above and in Table :

x. A binary digit (0 or 1).
R. Remaining bits.
lR. Least significant remaining bit.
mD. Most significant deleted bit.
Deleted bits.
r. Logical or of the deleted bits except for the *mD* bit in the template above. When there are no remaining bits, *r* is false. This means that *r* is false when the two nearest numbers are at equal distance.

Table shows how an unsigned fixed-point number is cast for each of the possible quantization modes in case there is quantization. If the two nearest representable numbers are not at equal distance, the result is, of course, the nearest representable number. This can be found for all the rounding modes by applying the SC_RND mode, that is, by adding the most significant of the deleted bits to the remaining bits.
The right hand column in Table contains the expression that has to be added to the remaining bits. It always evaluates to a one or a zero. The “&” operator used in the table stands for a bitwise and, and the “|” for a bitwise or.

Table 15 – Quantization Handling for Unsigned Fixed-Point Numbers

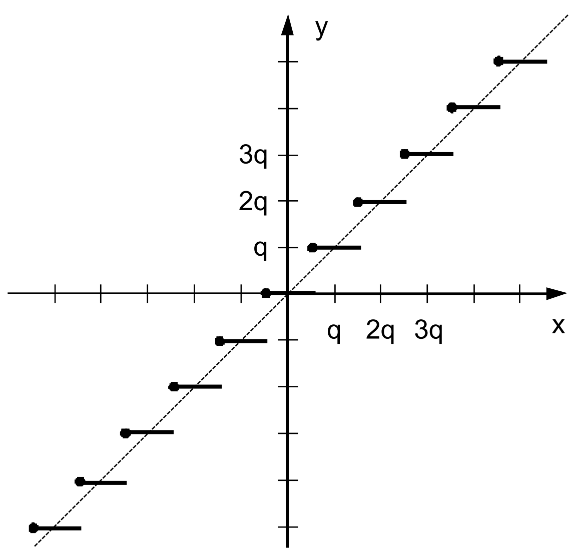
Quantization Mode	Expression to Be Added
SC_RND	*mD*
	Add the most significant deleted bit to the left bits.
SC_RND_ZERO	0
	Just copy the remaining bits.
SC_RND_MIN_INF	0
	Just copy the remaining bits.
SC_RND_INF	*mD*
	Add the most significant deleted bit to the left bits.
SC_RND_CONV	*mD* & (*lR* | *r*)
	If the most significant deleted bit is 1, and either the least significant of the remaining bits or at least one other deleted bit is 1, add 1 to the remaining bits.
SC_TRN	0
	Just copy the remaining bits.
SC_TRN_ZERO	0
	Just copy the remaining bits.

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

6.8.12.7.3 SC_RND

The result is rounded to the nearest representable number by adding the most significant of the deleted LSBs to the remaining bits. This rule is used for all rounding modes when the two nearest representable numbers are not at equal distance. When the two nearest representable numbers are at equal distance, this rule implies that there is rounding towards $+\infty$.

Figure 10 – Rounding to Plus Infinity



In Figure 10, the symbol “q” refers to the quantization step, i.e., the resolution of the data type.

EXAMPLE (signed):

Numbers of type `sc_fixed<4,2>` are assigned to numbers of type `sc_fixed<3,2,SC_RND>`.


```
(1.25)
after rounding to plus infinity: 01.1  (1.5)
```

There is quantization because the decimal number 1.25 is outside the range of values that can be represented exactly by means of a `sc_fixed<3,2,SC_RND>` number. The most significant of the deleted LSBs (1) is added to the new LSB.

```
10.11 (-1.25)
after rounding to plus infinity: 11.0  (-1)
```

There is quantization because the decimal number -1.25 is outside the range of values that can be represented exactly by means of a `sc_fixed<3,2,SC_RND>` number. The most significant of the deleted LSBs (1) is added to the new LSB.

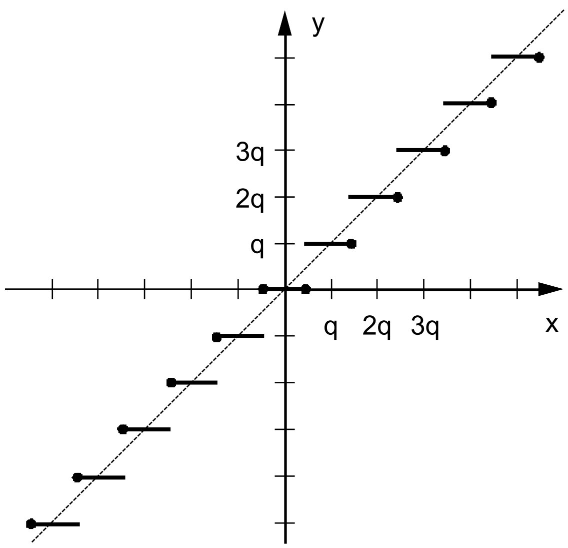
EXAMPLE (unsigned):

```
00100110.01001111 (38.30859375)
after rounding to plus infinity: 00100110.0101 (38.3125)
```


6.8.12.7.4 SC_RND_ZERO

In case the two nearest representable numbers are not at equal distance, the SC_RND mode is applied.
In case the two nearest representable numbers are at equal distance, the output is rounded towards 0. For positive numbers the redundant bits on the LSB side are deleted. For negative numbers the most significant of the deleted LSBs is added to the remaining bits.

Figure 11 – Rounding to Zero



EXAMPLE (signed):
Numbers of type `sc_fixed<4,2>` are assigned to numbers of type `sc_fixed<3,2,SC_RND_ZERO>`.

```
(1.25)
after rounding to zero: 01.0      (1)
```

There is quantization because the decimal number 1.25 is outside the range of values that can be represented exactly by means of a `sc_fixed<3,2,SC_RND_ZERO>` number. The redundant bits are omitted.

```
10.11 (-1.25)
after rounding to zero: 11.0      (-1)
```

There is quantization because the decimal number -1.25 is outside the range of values that can be represented exactly by means of a `sc_fixed<3,2,SC_RND_ZERO>` number. The most significant of the omitted LSBs (1) is added to the new LSB.

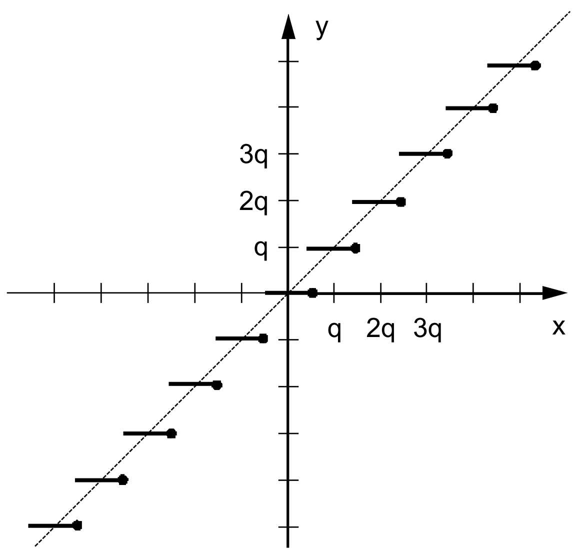
EXAMPLE (unsigned):

```
000100110.01001 (38.28125)
after rounding to zero:     000100110.0100 (38.25)
```

6.8.12.7.5 SC_RND_MIN_INF

In case the two nearest representable numbers are not at equal distance, the SC_RND mode is applied.
In case the two nearest representable numbers are at equal distance, there is rounding towards –□ by omitting the redundant bits on the LSB side.

Figure 12 – Rounding to Minus Infinity



EXAMPLE (signed):
Numbers of type sc_fixed<4,2> are assigned to numbers of type sc_fixed<3,2,SC_RND_MIN_INF>.

```
01.01 (1.25)
after rounding to minus infinity: 01.0     (1)
```

There is quantization because the decimal number 1.25 is outside the range of values that can be represented exactly by means of a sc_fixed<3,2,SC_RND_MIN_INF> number. The surplus bits are truncated.

```
10.11 (-1.25)
after rounding to minus infinity: 10.1     (-1.5)
```

There is quantization because the decimal number -1.25 is outside the range of values that can be represented exactly by means of a sc_fixed<3,2,SC_RND_MIN_INF> number. The surplus bits are truncated.

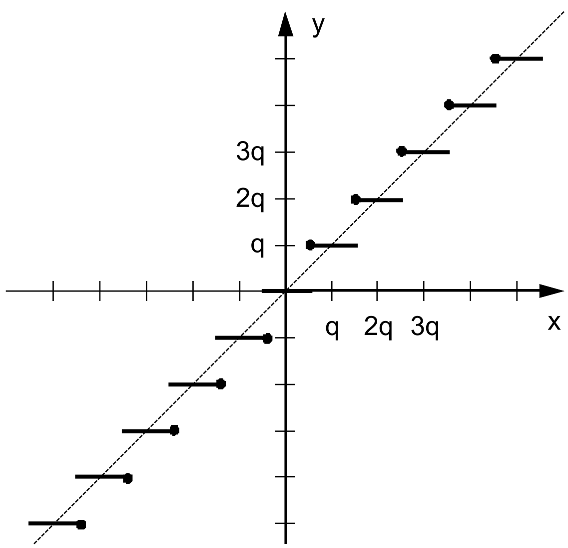
EXAMPLE (unsigned):

```
000100110.01001 (38.28125)
after rounding to minus infinity: 000100110.0100
      (38.25)
```

6.8.12.7.6 SC_RND_INF

In case the two nearest representable numbers are not at equal distance, the SC_RND mode is applied.
In case the two nearest representable numbers are at equal distance, the output is rounded to +□ or –□, depending on whether the number is positive or negative, respectively. For positive numbers the most significant of the deleted LSBs is added to the remaining bits. For negative numbers the surplus bits on the LSB side are omitted.

Figure 13 – Rounding to Infinity



EXAMPLE (signed):

Numbers of type `sc_fixed<4,2>` are assigned to numbers of type `sc_fixed<3,2,SC_RND_INF>`.

```
01.01 (1.25)
after rounding to infinity: 01.1  (1.5)
```

There is quantization because the decimal number 1.25 is outside the range of values that can be represented exactly by means of a `sc_fixed<3,2,SC_RND_INF>` number. The most significant of the deleted LSBs (1) is added to the new LSB.

```
10.11 (-1.25)
after rounding to infinity: 10.1  (-1.5)
```


There is quantization because the decimal number -1.25 is outside the range of values that can be represented exactly by means of a `sc_fixed<3,2,SC_RND_INF>` number. The surplus bits are truncated.

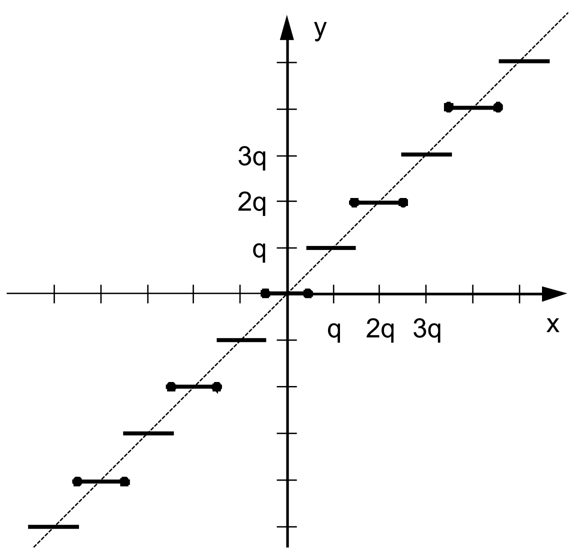
EXAMPLE (unsigned):

```
000100110.01001 (38.28125)
after rounding to infinity:  000100110.0101 (38.3125)
```

6.8.12.7.7 SC_RND_CONV

In case the two nearest representable numbers are not at equal distance, the `SC_RND` mode is applied.
In case the two nearest representable numbers are at equal distance, there is rounding towards +∞ if the LSB of the remaining bits is 1. There is rounding towards -∞, if the LSB of the remaining bits is 0.

Figure 14 – Convergent Rounding



EXAMPLE (signed):
Numbers of type `sc_fixed<4,2>` are assigned to numbers of type `sc_fixed<3,2,SC_RND_CONV>`.

```
00.11 (0.75)
after convergent rounding: 01.0   (1)
```

There is quantization because the decimal number 0.75 is outside the range of values that can be represented exactly by means of a sc_fixed<3,2,SC_RND_CONV> number. The surplus bits are truncated and the result is rounded towards +□.

```
10.11 (-1.25)
after convergent rounding: 11.0   (-1)
```

There is quantization because the decimal number -1.25 is outside the range of values that can be represented exactly by means of a sc_fixed<3,2,SC_RND_CONV> number. The surplus bits are truncated and the result is rounded towards +□.

EXAMPLE (unsigned):

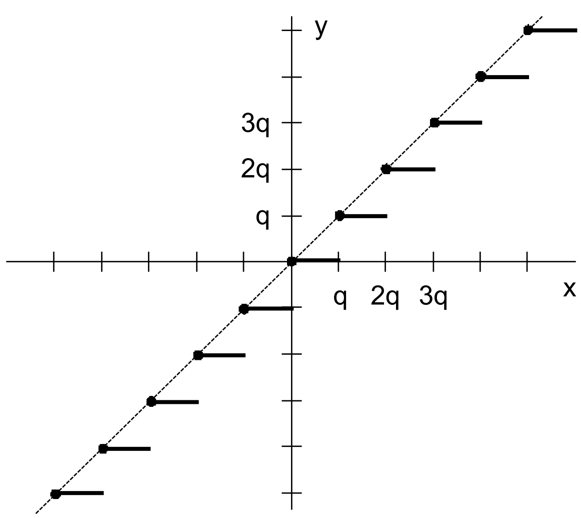
```
000100110.01001 (38.28125)
after convergent rounding: 000100110.0100   (38.25)

000100110.01011 (38.34375)
after convergent rounding: 000100110.0110   (38.375)
```

6.8.12.7.8 SC_TRN

SC_TRN is the default quantization mode. The result is rounded towards -□, that is, the superfluous bits on the LSB side are deleted. A number is then represented by the first representable number that is lower within the required bit range. In scientific literature it is usually called “value truncation.”

Figure 15 - Truncation



EXAMPLE (signed):
Numbers of type sc_fixed<4,2> are assigned to numbers of type sc_fixed<3,2,SC_TRN>.


```
01.01 (1.25)
after truncation: 01.0  (1)
```

There is quantization because the decimal number 1.25 is outside the range of values that can be represented exactly by means of a `sc_fixed<3,2,SC_TRN>` number. The LSB is truncated.

```
10.11 (-1.25)
after truncation: 10.1  (-1.5)
```

There is quantization because the decimal number -1.25 is outside the range of values that can be represented exactly by means of a `sc_fixed<3,2,SC_TRN>` number. The LSB is truncated.

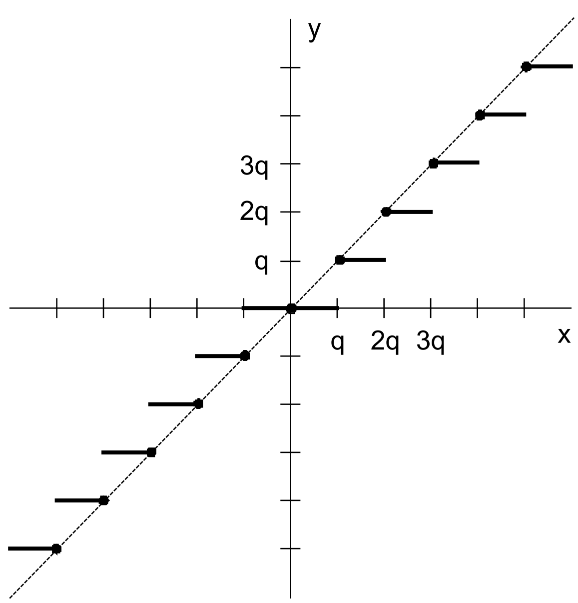
EXAMPLE (unsigned):

```
00100110.01001111 (38.30859375)
after truncation: 00100110.0100   (38.25)
```

6.8.12.7.9 SC_TRN_ZERO

For positive numbers this quantization mode corresponds to `SC_TRN`. For negative numbers the result is rounded towards zero (`SC_RND_ZERO`), that is, the superfluous bits on the right hand side are deleted and the sign bit is added to the left LSBs, but only in case at least one of the deleted bits differs from zero. A number is then approximated by the first representable number that is lower in absolute value. In scientific literature this is usually called "magnitude truncation."

Figure 16 – Truncation to Zero



EXAMPLE (signed):
A number of type `sc_fixed<4,2>` is assigned to a number of type `sc_fixed<3,2,SC_TRN_ZERO>`.

```
10.11 (-1.25)
after truncation to zero: 11.0   (-1)
```

There is quantization because the decimal number -1.25 is outside the range of values that can be represented exactly by means of a `sc_fixed<3,2,SC_TRN_ZERO>` number. The LSB is truncated and then the sign bit (1) is added at the LSB side.

EXAMPLE (unsigned):

```
00100110.01001111 (38.30859375)
after truncation to zero: 00100110.0100    (38.25)
```

6.9 User-defined types

New data types may be created by using the enum types and struct or class types. Channels of type `sc_fifo`, `sc_signal` and so forth may be declared to be of such a type. However in such cases certain functions may be required to be overloaded for the user-defined type if those functions are used.

For example a channel of type `sc_signal` (Chapter 11.60)requires the following to be overloaded:

```
operator =  (assignment)
operator == (equality)
operator <<  (stream output)
sc_trace()
```


7 Modules

A Module is the basic structural building block in SystemC. It is a container class in which processes and other modules are instantiated. Modules may contain:

- Ports (Chapter 8.3) for communication
- Data members
- Channel (Chapter 8.2)members
- Processes (Chapter 9)
- Member functions not registered as processes
- Instances of other modules

7.1 Module structure

A new type of module is created by publicly deriving from class sc_module. Example:

```
class my_module : public sc_module { . . . };
```

Alternatively, a module may be created with use of the SC_MODULE macro as follows:

```
SC_MODULE(module_name) {
    // ports, data members, member functions
    // processes etc.
    SC_CTOR(module_name) {  // Constructor
    // body of constructor
    // process registration, sensitivity lists
    // module instantiations, port binding etc.
    }
};
```

7.1.1 SC_MODULE

The SC_MODULE macro provides a simple form of module definition. Use of the SC_MODULE macro is not required. It is defined as follows:

```
#define SC_MODULE(user_module_name)   \
    struct user_module_name : sc_module
```

It simply derives the class `user_module_name` from the base class `sc_module` (Chapter 11.45).

7.1.2 Module Constructors

Modules (classes derived from sc_module) require a constructor. The macro SC_CTOR declares a constructor and is provided for convenience.

If the SC_CTOR macro does not meet the needs of the user, for example if a second constructor argument is required, then the constructor must be explicitly declared by the user.

If the user explicitly creates the constructor then one argument must be type `sc_module_name`. The `sc_module_name` class is used to manage the string names for (hierarchical) objects.

Example:
```
SC_MODULE(my_module) {
    // ports, channels, data members
    int some_parameter;
    // processes etc.
    my_module (sc_module_name name, int some_value):
       sc_module(name),
       some_parameter(some_value){
       // constructor body
    }
};
```

If a module has processes and the SC_CTOR macro is not used then the module must contain the SC_HAS_PROCESS macro.

7.1.2.1 SC_CTOR

The SC_CTOR macro has one argument which is the name of the module.

SC_CTOR provides for the management of the module name.

SC_CTOR declares a special symbol for use with the SC_METHOD (Chapter 9.4) and SC_THREAD (Chapter 9.5) macros.

7.1.3 SC_HAS_PROCESS

SC_HAS_PROCESS is required in the module when the user does not include the SC_CTOR macro and the module has processes.

SC_HAS_PROCESS declares a special symbol for use with the SC_METHOD (Chapter 9.4) and SC_THREAD (Chapter 9.5) macros.

7.1.4 Module instantiation

Modules may be instantiated inside of other modules to create hierarchy. To create a module instance two steps are required, the declaration of the module and the initialization of the module. A third step, port binding is required if the module has any ports. It is possible to instantiate a module which has no ports, which would not require port binding.

There are two valid approaches for module instantiation inside of another module. One approach uses pointers and the other does not. In the two approaches the declaration and initialization steps are different but the syntax for port binding is the same.

A module requires that a string name be provided as part of instantiation. The string name is not required to match the instance name. The string name is used by SystemC to assign a hierarchical name to the instance automatically. This hierarchical name is formed by the concatenation of the parent's hierarchical name and the string name of the child.

7.1.4.1 Module Instantiation *Not* Using Pointers

7.1.4.1.1 Declaration

The module instance is declared as a data member of the parent module. Example:

```
SC_MODULE(ex3) {
    // Ports
    sc_fifo_in<int> a;
    sc_fifo_out<int> b;
    // Internal channel
    sc_fifo<int> ch1;
    // Instances of module types ex1 and ex2
    ex1 ex1_instance;
    ex2 ex2_instance;
    // Module Constructor
    SC_CTOR(ex3){
       // Constructor body not shown
    }
    // Rest of the module body not shown
};
```


7.1.4.1.2 Initialization

The module instance is initialized in the initialization list of the constructor.
Example:

```
SC_MODULE(ex3){
    // Ports
    sc_fifo_in<int> a;
    sc_fifo_out<int> b;
    // Internal channel
    sc_fifo<int> ch1;
    // Instances of module type ex1 and ex2
    ex1 ex1_instance;
    ex2 ex2_instance;
    // Module Constructor
    SC_CTOR(ex3):
       ex1_instance("ex1_instance"),
       ex2_instance("ex2_instance")
    {
    // Rest of constructor body not shown
    }
    // Rest of the module body not shown
};
```

7.1.4.2 Module Instantiation Using Pointers

7.1.4.2.1 Declaration

The module instance is declared as a pointer to the module type in the parent module.
Example:

```
SC_MODULE(ex3) {
    // Ports
    sc_fifo_in<int> a;
    sc_fifo_out<int> b;
    // Internal channel
    sc_fifo<int> ch1;
    // Pointers to instances of module type ex1
    // and ex2
    ex1 *ex1_instance;
    ex2 *ex2_instance;
    // Module Constructor
    SC_CTOR(ex3){
       // Constructor body not shown
    }
    // Rest of the module body not shown
};
```


7.1.4.2.2 Allocation and Initialization

The module instance is allocated using the new command and initialized inside the body of the constructor.
Example

```
SC_MODULE(ex3){
    // Ports
    sc_fifo_in<int> a;
    sc_fifo_out<int> b;
    // Internal channel
    sc_fifo<int> ch1;
    // Pointers to instances of module type ex1
    // and ex2
    ex1 *ex1_instance;
    ex2 *ex2_instance;
    // Module Constructor
    SC_CTOR(ex3){
      // allocate and initialize both instances
      ex1_instance = new ex1(”ex1_in_ex3");
      ex2_instance = new ex2("ex2_in_ex3");
      // Rest of constructor body not shown
    }
    // Rest of the module body not shown
};
```

Objects allocated with new should later be deleted again. This can be done in the module destructor.
Example:

```
SC_MODULE(ex3) {
    // Rest of the module not shown
    ~ex3() {
      delete ex1_instance;
      delete ex2_instance;
    }
};
```


7.1.4.3 Port Binding

Port binding occurs in the body of the constructor. The port binding syntax is the same for either instantiation approaches (with or without pointers). There are two different ways of port binding provided: named and positional.

The ports of a child module instance may be bound to a channel instance local to the parent module or to a port of the parent module.

7.1.4.3.1 Named Port Binding

Named binding explicitly binds a named port to a channel.

Named binding syntax:

```
module_instance_name.port_name(channel_or_port_name) ;
```

Where:
`module_instance_name` is the instance name of the module.
`port_name` is the name of the port being bound
`channel_or_port_name` is either the instance name of the channel or the name of the parent port the port is being bound to.

Example:

```
SC_MODULE(ex3){
    sc_fifo_in<int> a;
    sc_fifo_out<int> b;
    sc_fifo<int> ch1;
    // Instances of module type ex1 and ex2
    ex1 ex1_instance;
    ex2 ex2_instance;
    // Module Constructor
    SC_CTOR(ex3):
      ex1_instance("ex1_instance"),
      ex2_instance("ex2_instance")
    {
      // Named connection for ex1
      ex1_instance.m(a);  // bind to parent port
      ex1_instance.n(ch1);  // bind to channel
      // Positional binding for ex2
      ex2_instance(ch1, b);
      // Rest of constructor body not shown
    }
};
```


7.1.4.3.2 Positional Port Binding

Positional binding connection implicitly binds a port to a channel by mapping the ordered list of channels and ports to corresponding ports within the module. The module ports are selected according to their declaration order within the module.

Named connection syntax:

```
module_instance_name(channel_or_port_name1,
   channel_or_port_name2, ... ) ;
```

Where:

module_instance_name is the instance name of the module.
channel_or_port_nameX is either the instance name of the channel or the name of the parent port the port is being bound to The first channel or port listed is bound to the first port declared in *module_instance_name*, the second channel or port listed is bound to the second port declared in *module_instance_name* and so forth.

Example:

```
SC_MODULE(ex3){
   sc_fifo_in<int> a;
   sc_fifo_out<int> b;
   sc_fifo<int> ch1;
   // Instances of module type ex1 and ex2
   ex1 ex1_instance;
   ex2 ex2_instance;
   // Module Constructor
   SC_CTOR(ex3):
     ex1_instance("ex1_instance"),
ex2_instance("ex2_instance")
   {
     // Named connection for ex1
     ex1_instance.m(a);
     ex1_instance.n(ch1);
     // Positional binding for ex2
     ex2_instance(ch1, b);
     // Rest of constructor body not shown
   }
};
```


8 Interfaces, Ports & Channels

The basic modeling elements for communication for inter-module communication consists of interfaces, ports and channels. An interface defines the set of access functions (methods) for a channel. A channel implements the interface methods. A port is a proxy object through which access to a channel is facilitated.

8.1 Interfaces

An interface defines a set of (member) functions. It is purely functional, that is it does not provide the implementation of the functions, but only specifies the signature of each function. It specifies the name, parameters and return type of the function but does not specify how the operations are implemented.

There are a number of interfaces provided by SystemC. Future revisions may provide additional interfaces:

`sc_fifo_in_if` (Chapter 11.15)
`sc_fifo_out_if` (Chapter 11.17)
`sc_mutex_if` (Chapter 11.49)
`sc_semaphore_if` (Chapter 11.49)
`sc_signal_in_if` (Chapter 11.61)
`sc_signal_inout_if` (Chapter 11.62)

8.2 Channels

Channels define how the functions (methods) of an interface are implemented. Channels provide the communication between modules or within a module provide the communication between processes.

Channels may implement one or more interfaces.

Different channels may implement the same interface in different ways.

There are two general classes of channels: primitive and hierarchical.

There are a number of primitive channels provided by SystemC. Future revisions may provide additional channels.

`sc_buffer` (Chapter 11.6)
`sc_fifo` (Chapter 11.12)
`sc_mutex` (Chapter 11.47)
`sc_sempahore` (Chapter 11.56)
`sc_signal` (Chapter 11.60)
`sc_signal_resolved` (Chapter 11.63)
`sc_signal_rv` (Chapter 11.64)

8.2.1 Primitive Channels

A base class, `sc_prim_channel()` (Chapter 11.55) is provided from which primitive channels are derived. A primitive channel is one that supports the request-update method of access and has no SystemC structures.

`sc_prim_channel()` provides two methods for implementation of the request-update scheme. `request_update()` is a non-virtual function which can be called during the evaluate phase of a delta-cycle. This instructs the scheduler (Chapter 2.4.1) to place the channel in an update queue. `update()` is a virtual function that must be specified by the derived channel as its behavior is dependent upon the derived channel's functionality. During the update phase of the delta-cycle, the scheduler takes the channels from the update queue and calls `update()` on each of them.

8.2.2 Hierarchical Channels

A channel that has SystemC structures is defined as a hierarchical channel. Structures may include ports, instances of modules, other channels, and processes. The channel itself may appear to be a module. This structure provides for greater flexibility in the definition of a channel in comparison to a primitive channel.

8.3 Ports

A port is an object that provides a module with a means for connection and communication with its surroundings. Through a port a module can communicate with one or more channels.

A port requires an interface. All ports are directly or indirectly derived from the template class `sc_port` (Chapter 11.54). An example port declaration is:

```
SC_MODULE(my_module){
    sc_port<IF, N > port_name ;

// rest of module not shown
};
```

`sc_port` takes two template parameters: an interface (Chapter 8.1) `IF` to which the port may be connected, and an optional integer `N` that specifies the maximum number of interfaces that may be attached to the port.

If `N = 0` then an arbitrary number of interfaces may be connected to the port. The default value of `N` is one. A port of value one is referred to as a simple port. A port of value greater than one is referred to as a multiport.

A function (interface method) of an interface connected to a port is invoked using the operator **->** which returns a pointer to the interface the port is bound to. Example:

```
  // Given:
sc_port<sc_signal_in_if<int> > a;  //port declaration
  // then:
a->read();//calls the read() interface method
           // of the channel connected to port a
```

To access individual interfaces on a multiport the **[]** operator is used. Example:

```
  // Given:
// port declaration, a is bound to two channels
sc_port<sc_signal_in_if<int>, 2 > a;
  // then:
// calls the read() interface method of the
// 2nd channel connected to port a
a[1]->read();
// calls the read() interface method of the
// 1st channel connected to port a
a[0]->read();  // or a->read();
```


8.3.1 Specialized ports

Specialized ports are ports derived from the base class `sc_port` which are customized for use with a particular (set of) interface(s). These ports typically provide additional support for use with a channel or for ease of use. SystemC provides several specialized ports. Future revisions may provide additional specialized ports. The specialized ports include:

For `sc_buffer` (Chapter 11.6) and `sc_signal` (Chapter 11.60) channels

`sc_in` (Chapter 11.32)
`sc_inout` (Chapter 11.35)
`sc_out` (Chapter 11.51)

For `sc_fifo` (Chapter 11.12) channel

`sc_fifo_in` (Chapter 11.14)
`sc_fifo_out` (Chapter 11.16)

For `sc_signal_rv` (Chapter 11.64) channel

`sc_in_rv` (Chapter 11.34)
`sc_inout_rv` (Chapter 11.37)
`sc_out_rv` (Chapter 11.53)

For `sc_signal_resolved` (Chapter 11.63) channel

`sc_in_resolved` (Chapter 11.33)
`sc_inout_resolved` (Chapter 11.36)
`sc_out_resolved` (Chapter 11.52)

9 Processes

Functionality is described in processes. Processes must be contained in a module.

A process is a member function of a module. It is registered as a process with the SystemC kernel using a process declaration in the module constructor.

Processes are not called directly from user code. A process is invoked based on its sensitivity list, which consists of zero, one, or more events, which can change during simulation run time .

Processes are not hierarchical.

SystemC has two kinds of processes: method processes and thread processes. Two macros are provided to register a member function as a process: SC_METHOD and SC_THREAD. Although not strictly required, the use of these macros is strongly recommended.

During the initialization phase (Chapter 2.3) all processes are executed. To avoid execution of a process during initialization, the dont_initialize() function (Chapter 11.45) is invoked in the module constructor following the corresponding process declaration.

9.1 Member Function Declaration

A process is declared as a member function of a module. It has a return type of `void` and has no arguments.

Example:

```
SC_MODULE(my_module){
    //ports, channels etc. not shown
    // Process function declaration
    void my_proc();
    // rest of module not shown
};
```


9.2 Process Declaration and Registration

A member function of a module is declared and registered as a process with the SystemC kernel using either the SC_METHOD or the SC_THREAD macro. The declaration occurs in the body of the module constructor. Both macros take one argument which is the name of the function which is to be declared as a process. The syntax for the declaration is shown below.

Declaration syntax:

```
SC_MODULE(my_module) {
    void my_thread_proc(); //member function declaration
    void my_method_proc(); //member function declaration
    SC_CTOR(my_module) {
       // thread process declaration and registration
       SC_THREAD(my_thread_proc);
       // method process declaration and registration
       SC_METHOD(my_method_proc);
       // rest of constructor not shown
    }
    // rest of module not shown
};
```

9.3 Process Static Sensitivity

A process is declared as statically sensitive to an event using `sensitive` in the module constructor after the process declaration and before the next process declaration. That is after a SC_METHOD or SC_THREAD statement and before the next one.

The static sensitivity list for a particular process is the collection of events declared in the module constructor for that process.

In the sc_module base class (Chapter 11.45) an object named `sensitive` of type `sc_sensitive` (Chapter 11.59) is defined for use in creating static sensitivity lists for processes. Both the ( ) and the << operators are overloaded for objects of the `sc_sensitive` class. These operators provide for both a functional notation and a streaming style notation for defining static sensitivity lists. These styles are described below.

9.3.1 Functional Notation Syntax

The functional notation takes a *single* argument (`event`), which is the event the process is sensitive too.

Syntax:

```
sensitive(event);
```

Example:

```
SC_MODULE(my_module) {
    sc_event c;
    void my_thread_proc();
    SC_CTOR(my_module) {
       SC_THREAD(my_thread_proc);
          // declare static sensitivity list
       sensitive(c);  // sensitive to event c
    }
    // rest of module not shown
};
```

If the process is sensitive to more than one event, then multiple `sensitive()` statements are required.

Example:

```
SC_MODULE(my_module) {
    sc_event c;
    sc_event d;
    void my_thread_proc();
    SC_CTOR(my_module) {
       SC_THREAD(my_thread_proc);
          // declare static sensitivity list
       sensitive(c); // sensitive to event c
       sensitive(d); //sensitive to event d
    }
    // rest of module not shown
};
```


9.3.2 Streaming Style Notation Syntax

The streaming style notation supports multiple events.

```
sensitive << event_1 << event_2 …. ;
```

Example:

```
SC_MODULE(my_module) {
    sc_event c;
    sc_event d;
    void my_thread_proc();
    SC_CTOR(my_module) {
       SC_THREAD(my_thread_proc);
       // declare static sensitivity list
       sensitive << c << d; // sensitive to events c & d
    }
    // rest of module not shown
};
```

9.3.3 Multiple Processes in a Module

When multiple processes are declared, the pattern is declaration followed by sensitivity list followed by declaration followed by sensitivity list and so on.

Example:

```
SC_MODULE(my_module) {
    sc_event c, d;
    void proc_1();
    void proc_2();
    void proc_3();
    SC_CTOR(my_module) {
       SC_THREAD(proc_1);
        sensitive << c << d; //proc_1 sensitive to c & d
       SC_THREAD(proc_2); // no static sensitivity
       SC_THREAD(proc_3);
        sensitive << d ; // proc_3 sensitive to d
    }
    // rest of module not shown
};
```


9.4 Method Process

When made to run, the entire body of the method process is executed. Upon completion it returns control to the SystemC kernel. The process does not maintain its state implicitly, meaning that all local variables are automatic and lose their value when the function returns. The user must manage process state explicitly by using state variables that are data members of the module in which the process resides.

A method process may not be explicitly suspended (may not have calls to `wait()`).

A method process may use static sensitivity, dynamic sensitivity or both. Dynamic sensitivity is created using the `next_trigger()` function (Chapter 11.45) with one or more arguments. The `next_trigger()` function may be called in the body of the method process code, or it may be called in a function called by the method process that is either a member function of the module or a method of a channel.

A member function of a module is registered with the SystemC kernel as a method process using the SC_METHOD macro in the module constructor.

The SC_METHOD macro has one argument. The argument is the name of the member function to be declared as a method process and registered with the SystemC kernel.

Example:

```
SC_MODULE(my_module) {
     sc_event c;
        // process member function declaration
     void my_method_proc();

     SC_CTOR(my_module) {
        // method process declaration & registration
        SC_METHOD(my_method_proc);
        // declare static sensitivity list
           sensitive(c);  // sensitive to event c
        dont_initialize();  // don't run at initialization
     }
     // Rest of module not shown
};
```


9.4.1 Method Process Dynamic Sensitivity

When triggered, the entire body of the method process is executed. Execution of a `next_trigger()` statement sets the sensitivity for the next trigger for the method process. It does *not* cause the method process to end prematurely. The function `next_trigger()` specifies the event, event list or time delay that is the next trigger condition for the method process.

If multiple `next_trigger(arg)` statements are executed, the last one executed before the method process is finished executing determines the next trigger condition (i.e. last one wins).

After completion the process is invoked again when the event(s) specified by the sensitivity list are notified.

9.4.1.1 Trigger on Static Sensitivity List

If the `next_trigger()` function is called without an argument, then the next trigger is the static sensitivity list of the method process. In this case, if there is no static sensitivity list specified then the method process will not be triggered again during the simulation. Syntax for triggering on the static sensitivity list:

```
next_trigger();
```

9.4.1.2 Trigger On A Single Event

If the `next_trigger()` function is called with a single event argument then the process will be triggered when that event is triggered. Syntax for triggering on a single event:

```
sc_event e1;    // event
next_trigger(e1);
```


9.4.1.3 Trigger After A Specific Amount Of Time

If the `next_trigger()` function is called with a time value argument then the process will be triggered after a delay of the specified time. Syntax for triggering after a specific amount of time:

```
sc_time t(200, SC_NS); // variable t of type sc_time
next_trigger(t); // trigger 200 ns later
next_trigger(200, SC_NS); // trigger 200 ns later
```

If the time value argument is zero then the process will be triggered after one delta-cycle (Chapter 2.4.1). Syntax for triggering after one delta-cycle delay:

```
next_trigger( 0, SC_NS );
next_trigger( SC_ZERO_TIME );
sc_time t(0, SC_NS); // variable t of type sc_time
next_trigger( t ); // trigger the next delta-cycle
next_trigger( 0, SC_NS ); // ditto
next_trigger( SC_ZERO_TIME ); // ditto
```

9.4.1.4 Trigger On One Event In A List Of Events

If the `next_trigger()` function is called with an OR-list of events then the process will be triggered when one event in the list of events has been triggered. Syntax for triggering on one event in a list of events:

```
sc_event e1,e2,e3;    // events
next_trigger(e1 | e2 | e3); //trigger on e1, e2 or e3
```

9.4.1.5 Trigger On All Events In A List Of Events

If the `next_trigger()` function is called with an AND-list of events, then the process will be triggered when all events in the list of events have been triggered. The events do not have to be triggered in the same delta-cycle or at the same time. Syntax for triggering on all events in a list of events:

```
sc_event e1,e2,e3;    // events
next_trigger(e1 & e2 & e3);//trigger on e1, e2 and e3
```


9.4.1.6 Trigger On An Event In A List Of Events With Timeout

If the `next_trigger()` function is called with a combination of a specific amount of time and an OR-list of events, then the process will be triggered when one event in the list of events has been triggered or after the specified amount of time which ever occurs first. Syntax for triggering on one event in a list of events with timeout:

```
sc_time t(200, SC_NS); // variable t of type sc_time
// trigger on e1, e2, or e3, timeout after 200 ns
next_trigger(t, e1 | e2 | e3);
// trigger on e1, e2, or e3, timeout after 200 ns
next_trigger(200, SC_NS, e1 | e2 | e3);
```

9.4.1.7 Trigger On All Events In A List Of Events With Timeout

If the `next_trigger()` function is called with a combination of a specific amount of time and an AND-list of events then the process will be triggered either when all events in the list of events have been triggered or after the specified amount of time which ever occurs first. Syntax for triggering on all events in a list of events with timeout:

```
sc_time t(200, SC_NS); // variable t of type sc_time
// trigger on e1, e2, and e3, timeout after 200ns
next_trigger(t, e1 & e2 & e3);
// trigger on e1, e2, and e3, timeout after 200ns
next_trigger(200, SC_NS, e1 & e2 & e3);
```

9.5 Thread Process

A thread process is invoked only once (during simulation initialization). The process executes until a `wait()`is executed where upon the process is suspended. Upon suspension the state of the process is implicitly saved. The process is resumed based upon its sensitivity list. Its State is then restored and execution of the process resumes from the point of suspension (statement following `wait()`).

If the body or parts of the body of the thread process are required to be executed more than once then it must be implemented with a loop, typically an infinite loop. This ensures that the process can be repeatedly reactivated.

If a thread process does not have an infinite loop and does not call wait() in any way then the process will execute entirely and exit within the same delta-cycle.

If a thread process does have an infinite loop but does not call wait() in any way then the process will continuously execute during the same delta-cycle. No other process will execute.

A thread process may use static sensitivity, dynamic sensitivity or both. Dynamic sensitivity is created using the `wait()` function (Chapter 11.45). with one ore more arguments. The `wait()`function can be called in the body of the thread process code, or can be called in a function called by the method process that is either of a member function of the module or a method of a channel.

A member function of a module is registered with the SystemC kernel as a thread process using the SC_THREAD macro declaration in the module constructor.

The SC_THREAD macro has one argument. The argument is the name of the member function that is to be declared as a thread process and registered with the SystemC kernel.

Example:
```
SC_MODULE(my_module) {
    sc_event c;
      // process member function declaration
    void my_thread_proc();

    SC_CTOR(my_module) {
      // thread process declaration & registration
      SC_METHOD(my_thread_proc);
      // declare static sensitivity list
        sensitive(c);  // sensitive to event c
      dont_initialize();  // don't run at initialization
    }
    // Rest of module not shown
};
```

9.5.1 Thread Process Dynamic Sensitivity

When triggered, a thread process is executed until a `wait()` statement is executed where upon the process is suspended. Execution of a `wait()` statement specifies the sensitivity of a thread process, that is, it specifies the condition for resuming the thread process.

The `wait()`function can be called with different arguments as described in the following sections.

9.5.1.1 Resume On Static Sensitivity List

If the wait() function is called without any argument then a thread process is resumed depending on the static sensitivity list of the thread process. In this case, if there is no static sensitivity list specified then the thread process will not be resumed again during the simulation. Syntax for resuming on the static sensitivity list:

```
wait();
```

9.5.1.2 Resume On A Single Event

If the wait() function is called with a single event argument then the process will be resumed when that event is triggered. Syntax for resuming on a single event:

```
sc_event e1;     // event
wait(e1);
```

9.5.1.3 Resume After A Specific Amount Of Time

If the wait() function is called with a time value argument then the process will be resumed after a delay of the specified time. Syntax for resuming after a specific amount of time:

```
sc_time t(200, SC_NS); // variable t of type sc_time
wait(t); // trigger 200 ns later
wait(200, SC_NS); // trigger 200 ns later
```

If the time value argument is zero then the process will be resumed after one delta-cycle (Chapter 2.4.1). Syntax for resuming after a delta-cycle delay:

```
sc_time t(0, SC_NS); // variable t of type sc_time
wait( t );  // resume after a delta-cycle delay
wait( 0, SC_NS ); // ditto
wait( SC_ZERO_TIME ); // ditto
```

9.5.1.4 Resume On An Event In A List Of Events

If the wait() function is called with an OR-list of events then the process will be resumed when one event in the list of events has been triggered. Syntax for resuming on one event in a list of events:

```
sc_event e1,e2,e3;    // events
wait(e1 | e2 | e3); //resume on e1, e2 or e3
```


9.5.1.5 Resume On All Events In A List Of Events

If the wait() function is called with an AND-list of events then the process will be resumed when all events in the list of events has been triggered. The events do not have to be triggered in the same delta-cycle or at the same time. Syntax for resuming on all events in a list of events:

```
sc_event e1,e2,e3;     // events
wait(e1 & e2 & e3);//trigger on e1, e2 and e3
```

9.5.1.6 Resume On An Event In A List Of Events With Timeout

If the wait() functions is called with a combination of a specific amount of time and an OR-list of events then the process will be resumed either when one event in the list of events has been triggered or after the specified amount of time which ever occurs first. Syntax for resuming on one event in a list of events with timeout:

```
sc_time t(200, SC_NS); // variable t of type sc_time
// resume on e1, e2, or e3, timeout after 200 ns
wait(t, e1 | e2 | e3);
// resume on e1, e2, or e3, timeout after 200 ns
wait(200, SC_NS, e1 | e2 | e3);
```

9.5.1.7 Resume On All Events In A List Of Events With Timeout

If the wait() function is called with a combination of a specific amount of time and an AND-list of events then the process will be resumed either when all events in the list of events have been triggered or after the specified amount of time which ever occurs first.. Syntax for resuming on all events in a list of events with timeout:

```
sc_time t(200, SC_NS); // variable t of type sc_time
// trigger on e1, e2, and e3, timeout after 200ns
wait(t, e1 & e2 & e3);
// trigger on e1, e2, and e3, timeout after 200ns
wait(200, SC_NS, e1 & e2 & e3);
```


10 Utilities

10.1 Mathematical functions

The global functions `sc_abs()` (Chapter 12.1), `sc_min()`(Chapter 12.13) and `sc_max()`(Chapter 12.12) are provided.

10.2 Utility functions

The following global functions provide information about or the status of the simulator

`sc_copyright()` (Chapter 12.7)
`sc_version()` (Chapter 12.22)

10.3 Debugging support

10.3.1 Tracing

Tracing data in a channel or the data member of a module consists of three steps:
1) Create a trace file
2) Register the variables to be traced
3) Close the trace file before returning from `sc_main()`.

To create a trace file the global function `sc_create_vcd_trace_file()` (Chapter 12.4) is provided. This function creates a file and returns a pointer to it. The trace files may be created in the `sc_main()` function or the constructor of a module. The requirement is that the trace file must be created before the registration of the variables to be traced.

Registration of the variable to be traced is done using the `sc_trace()` function (Chapter 12.21). Only variables with a lifetime of the complete simulation may be traced. This means local variables within a function may not be traced. SystemC provides built-in support for tracing variables, ports and certain channels.

To close a trace file the function `sc_close_vcd_trace_file()` (Chapter 12.3) is provided.

11 Class reference

The **class reference** is an alphabetical listing of classes. The entry for each class contains:

- Synopsis
    - Pseudo-class declaration

- Description
    - Description of the class
    - Sample use

- Functions and operators
    - Description of functionality
- Depending upon the class other information may be provided
    - Interfaces implemented by a channel
    - Specialized ports associated with the channel
    - Disabled member functions

Class Hierarchy. The classes are documented with the inheritance hierarchy from the reference implementation intact. Unless explicitly noted this inheritance hierarchy is not required for other implementations.

Base classes. In some cases base classes are referred to but are not documented. The purpose of these base classes in the reference implementation is to provide a single point for polymorphic access to derived template classes. For example, when one of these base classes is specified as an argument type, it means that any instantiated template class derived from this base class can be used for that argument. In these cases, the public base class methods are documented as if they belong to the derived class. These base classes are shown in an *italic* font with a superscript dagger (†). They are not required for other implementations.

Member functions are organized in categories according to general use, such as public methods, public constructors and so forth. The categories are not part of the C++ language but are used as a way to organize the functions.

Within the general categories member functions are listed alphabetically. Functions for each class fall into these general types:

- Functions unique to a class. Complete documentation for these functions are in the class where they occur
- Functions inherited from a documented base class without being redefined. These functions are not listed in the derived class. Complete documentation for these functions is in the defining base class.
- Functions inherited from an undocumented base class. Complete documentation for these functions will be in the derived class.

- Functions that are redefined in a derived class. Documentation contains relevant information in the derived class, but may also direct to the base class.

11.1 sc_attr_base

Synopsis

```
class sc_attr_base
{
public:
    // constructors & destructor
    sc_attr_base( const sc_string& name__);
    sc_attr_base( const sc_attr_base& );
    virtual ~sc_attr_base();

    // other methods
    const sc_string& name() const;
private:
    // disabled
    sc_attr_base();
    sc_attr_base& operator = ( const sc_attr_base& );
};
```

Description

sc_attr_base is the attribute base class, which provides the key of a (key,value) attribute. The key (name) is of type sc_string. Classes derived from sc_attr_base should provide the value of a (key,value) attribute.

Public Constructors & Destructor

sc_attr_base(const sc_string& name_);

Sets the attribute name to name_.

sc_attr_base(const sc_attr_base&);

Copy constructor.

virtual **~sc_attr_base**();

Does nothing but enabling derived classes to define their own virtual destructors.

Public Member Functions

const sc_string& **name**() const;

Returns a reference to the attribute name.

Disabled Member Functions

sc_attr_base();

Default constructor.

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

Default assignment operator.

11.2 sc_attribute

Synopsis

```
template <class T>
class sc_attribute
: public sc_attr_base
{
public:
        // constructors & destructor
        sc_attribute( const sc_string& name_ );
        sc_attribute( const sc_string& name_,
                   const T& value_ );
        sc_attribute( const sc_attribute<T>& a );
        virtual ~sc_attribute();
public:
        T value;
private:
        // disabled
        sc_attribute();
        sc_attribute<T>& operator = ( const
    sc_attribute<T>& );
};
};
```

Description

sc_attribute is a template class that describes an attribute. An attribute has a name and a value. Attributes can be attached to any sc_object.

Example

```
sc_attribute<int> a( “answer”, 42 );
cout << a.name() << “,” << a.value; // prints ‘answer,42’
```

Public Constructors & Destructor

sc_attribute(const sc_string& name_);
Sets the attribute name to `name_`, default construction for value.

sc_attribute(const sc_string& name_, const T& value_);
Sets the attribute name to `name_` and `value` to `value_`.

sc_attribute(const sc_attribute<T>&);
Copy constructor.

virtual
~sc_attribute();
Virtual destructor. Does nothing by default.

Public Data Members

T value;

Provides direct access to the attribute value.

Disabled Member Functions

```
sc_attribute();
```

Default constructor.

```
sc_attribute& operator = ( const sc_attribute<T>& );
```

Default assignment operator.

11.3 sc_attr_cltn

Synopsis

```
class sc_attr_cltn
{
public:
    // typedefs
    typedef sc_attr_base*         elem_type;
    typedef sc_attr_base*         iterator;
    typedef const sc_attr_base*   const_iterator;

    // constructors & destructor
    sc_attr_cltn();
    sc_attr_cltn( const sc_attr_cltn& );
    ~sc_attr_cltn();

    // other methods
    bool push_back( sc_attr_base* );
    sc_attr_base* operator [] ( const sc_string& name_ );
    const sc_attr_base* operator [] ( const sc_string&
    name_ ) const;
    sc_attr_base* remove( const sc_string& name_ );
    void remove_all();
    int size() const ;
    iterator begin();
    const_iterator begin() const ;
    iterator end();
    const_iterator end() const ;
private:
    // disabled
    sc_attr_cltn& operator = ( const sc_attr_cltn& );
};
```

Description

sc_attr_cltn is a collection of (pointers to) attributes. All SystemC objects that inherit from sc_object have an attribute collection available. This allows users to attach attributes to any such object.

Type Definitions

```
typedef sc_attr_base*       elem_type;
typedef sc_attr_base*       iterator;
typedef const sc_attr_base* const_iterator;
```

Public Constructors & Destructor

sc_attr_cltn();

Default constructor.

sc_attr_cltn(const sc_attr_cltn&);

Copy constructor.

~sc_attr_cltn();
Destructor.

Public Member Functions

```
iterator
begin();
```
Returns an iterator pointing at the beginning of the collection.

```
const_iterator
begin() const;
```
Returns a const-iterator pointing at the beginning of the collection.

```
iterator
end();
```
Returns an iterator pointing at the end of the collection.

```
const_iterator
end() const;
```
Returns a const-iterator pointing at the end of the collection.

```
iterator
operator []( const sc_string& name ) ;
```
Allows random access to attributes indexed by name. If the name does not exist, returns 0.

```
const sc_attr_base† *
operator []( const sc_string& name ) const ;
```
Allows constant random access to attributes indexed by name. If the name does not exist, returns 0.

```
bool
push_back( sc_attr_base† * new_attr ) ;
```
Appends new_attr to the end of the collection and returns true if the name is unique. If the name already exists in the collection, the attribute is not added and the function returns false.

```
sc_attr_base† *
remove( const sc_string& name ) ;
```
Removes the specified attribute from the collection. Returns a pointer to the removed attribute, or 0 if an attribute with the specified name does not exist.

```
void
remove_all() ;
```
Removes all attributes from the collection.

```
int
size() const ;
```
Returns the number of attributes stored in the collection.

Disabled Member Functions

```
sc_attr_cltn&
operator = ( const sc_attr_cltn& );
```

Default assignment operator.

11.4 sc_bigint

Synopsis

```
class sc_bigint
    : public sc_signed
{
public:
    // constructors & destructors
    sc_bigint();
    sc_bigint( const sc_bigint<W>& v );
    sc_bigint( const sc_signed& v );
    sc_bigint( const sc_signed_subref& v );
    template <class T1, class T2>;
    sc_bigint( const sc_signed_concref<T1,T2>& a );
    sc_bigint( const sc_unsigned& v );
    sc_bigint( const sc_unsigned_subref& v );
    template <class T1, class T2>;
    sc_bigint( const sc_unsigned_concref<T1,T2>& a );
    sc_bigint( const char* v );
    sc_bigint( int64 v );
    sc_bigint( uint64 v );
    sc_bigint( long v );
    sc_bigint( unsigned long v );
    sc_bigint( int v );
    sc_bigint( unsigned int v );
    sc_bigint( double v );
    sc_bigint( const sc_bv_base& v );
    sc_bigint( const sc_lv_base& v );
    explicit sc_bigint( const sc_fxval& v );
    explicit sc_bigint( const sc_fxval_fast& v );
    explicit sc_bigint( const sc_fxnum& v );
    explicit sc_bigint( const sc_fxnum_fast& v );
    ~sc_bigint();

    // assignment operators
    sc_bigint<W>& operator = ( const sc_bigint<W>& v );
    sc_bigint<W>& operator = ( const sc_signed& v );
    sc_bigint<W>& operator = (const
                    sc_signed_subref& v );
    template <class T1, class T2>
    sc_bigint<W>& operator = ( const
                    sc_signed_concref<T1,T2>& a );
    sc_bigint<W>& operator = ( const sc_unsigned& v );
    sc_bigint<W>& operator = ( const
                    sc_unsigned_subref& v );
    template <class T1, class T2>
    sc_bigint<W>& operator = ( const
                    sc_unsigned_concref<T1,T2>& a );
    sc_bigint<W>& operator = ( const char* v );
    sc_bigint<W>& operator = ( int64 v );
    sc_bigint<W>& operator = ( uint64 v );
    sc_bigint<W>& operator = ( long v );
    sc_bigint<W>& operator = ( unsigned long v );
```



```
    sc_bigint<W>& operator = ( int v );
    sc_bigint<W>& operator = ( unsigned int v );
    sc_bigint<W>& operator = ( double v );
    sc_bigint<W>& operator = ( const sc_bv_base& v );
    sc_bigint<W>& operator = ( const sc_lv_base& v );
    sc_bigint<W>& operator = ( const sc_int_base& v );
    sc_bigint<W>& operator = ( const sc_uint_base& v );
    sc_bigint<W>& operator = ( const sc_fxval& v );
    sc_bigint<W>& operator = ( const sc_fxval_fast& v );
    sc_bigint<W>& operator = ( const sc_fxnum& v );
    sc_bigint<W>& operator = ( const sc_fxnum_fast& v );
};
```

Description

`sc_bigint<W>` is an arbitrary sized signed integer. The word length is built into the type and can never change. Methods allow for addressing an individual bit or a sub range of bits.

Example

```
SC_MODULE(my_module) {
    // data types
  sc_uint<3> a;
  sc_uint<44> b;
  sc_bigint<88> c;
  sc_bigint<123> d;
    // process
  void my_proc();

  SC_CTOR(my_module) :
    a(0), // init
    c(7654321)  // init
  {
    b = 33;  // set value
    d = 2300; // set value
    SC_THREAD(my_proc);
  }
};

void my_module::my_proc() {
  a = 1;
  b[30] = a[0];
  cout << b.range(7,0) << endl;

  cout << c << endl;
  d[122] = b;


  wait(300, SC_NS);
  sc_stop();
}
```


Public Constructors

sc_bigint();

Create an sc_bigint instance with an initial value of 0.

sc_bigint(T a) ;

T in { sc_bigint<W>, *sc_[un]signed_subref*†, *sc_[un]signed_concref*†, const char*, [u]int64, [unsigned] long, [unsigned] int, double, sc_bv_base, sc_lv_base, sc_fxval, sc_fxval_fast, sc_fix[ed][_fast]}

Create an sc_bigint with value a. If the word length of a is greater then W, a gets truncated to W bits.

Copy Constructor

sc_bigint(const sc_bigint&) ;

Methods

bool
iszero() const ;

Return true if the value of the sc_bigint instance is zero.

int
length() const ;

Return the word length.

void
print(ostream& os = cout) const ;

Print the sc_bigint instance to an output stream.

void
reverse() ;

Reverse the contents of the sc_bigint instance. I.e. LSB becomes MSB and vice versa.

void
scan(istream& is = cin) ;

Read an sc_bigint value from an input stream.

bool
sign() const ;

Return false.

Assignment Operators

sc_bigint<W>& **operator =** (T) ;

T in { sc_bigint<W>, *sc_[un]signed_subref*†, *sc_[un]signed_concref*†, const char*, [u]int64, [unsigned] long, [unsigned] int, double, sc_bv_base, sc_lv_base, sc_fxval, sc_fxval_fast, sc_fix[ed][_fast]}

Assign the value of the right-hand side to the left-hand side. The value is truncated, if its word length is greater than W.

Increment and Decrement Operators

```
sc_bigint<W>& operator  ++ () ;
const sc_bigint<W> operator ++ ( int ) ;
```

The operation is performed as done for type unsigned int.

```
sc_bigint<W>& operator  -- () ;
const sc_bigint<W> operator  -- ( int ) ;
```

The operation is performed as done for type unsigned int.

Bit Selection

```
sc_signed_bitref†   operator [] ( int i ) ;
sc_signed_bitref_r† operator [] ( int i ) const ;
sc_signed_bitref†   bit( int i) ;
sc_signed_bitref_r† bit( int i) const ;
```

Return a reference to a single bit at index i.

Part Selection

```
sc_signed_subref†   range( int high, int low )
sc_signed_subref_r† range( int high, int low ) const
sc_signed_subref†   operator () ( int high, int low )
sc_signed_subref_r† operator () ( int high, int low )
    const
```

Return a reference to a range of bits. The MSB is set to the bit at position high, the LSB is set to the bit at position low.

Explicit Conversion

```
double   to_double() const ;
int      to_int() const ;
int64    to_int64() const ;
long     to_long() const ;
uint64   to_uint64() const ;
unsigned int    to_uint() const ;
unsigned long   to_ulong() const ;
```

Converts the value of sc_bigint instance into the corresponding data type. If the requested type has less word length than the sc_bigint instance, the value gets truncated accordingly. If the requested type has greater word length than the sc_bigint instance, the value gets sign extended, if necessary.

```
to_string( sc_numrep = SC_DEC ) const
to_string( sc_numrep, bool ) const
```

Convert the sc_bigint instance into its string representation.

Arithmetic Operators

```
friend sc_bigint operator OP ( sc_biguint , sc_bigint )
friend sc_bigint operator OP ( sc_bigint ,
sc_biguint )friend sc_bigint operator OP ( sc_bigint,
    sc_bigint ) ;
friend sc_bigint operator OP ( sc_bigint , T ) ;
friend sc_bigint operator OP ( T , sc_bigint ) ;
OP in { + - * / % & | ^ == != < <= > >= }
T in { sc_[u]int_base, [u]int64, [unsigned] long,
    [unsigned] int}
```

The operation OP is performed and the result is returned.

```
sc_bigint& operator OP (T)
OP in { += -= *= /= %= &= |= ^= } ;
T in { sc_[un]signed, sc_[u]int_base, [u]int64, [unsigned]
    long, unsigned] int }
```

The operation OP is performed and the result is assigned to the left hand side.

Shift Operators

```
friend sc_biguint operator OP ( sc_biguint a , sc_bigint
    b );
friend sc_bigint operator OP ( sc_bigint a, sc_bigint b );
friend sc_bigint operator OP ( sc_bigint a, T b );
OP in { << >> }
T in { sc_[u]int_base, [u]int64, [unsigned] long,
    [unsigned] int }
```

Shift a to the left/right by b bits and return the result.

```
sc_bigint& operator OP ( T i );
OP in { <<= >>= }
T in { sc_[un]signed, sc_[u]int_base, [u]int64, [unsigned]
    long, [unsigned] int } ;
```

Shift the sc_bigint instance to the left/right by i bits and assign the result to the sc_bigint instance.

Bitwise not

```
friend sc_bigint
operator ~ ( sc_bigint a );
```

Return the bitwise not of a;

11.5 sc_biguint

Synopsis

```
class sc_biguint
    : public sc_unsigned
{
public:
    // constructors
    sc_biguint();
    sc_biguint( const sc_biguint<W>& v );
    sc_biguint( const sc_unsigned& v );
    sc_biguint( const sc_unsigned_subref& v );
    template <class T1, class T2>;
    sc_biguint( const sc_unsigned_concref<T1,T2>& a );
    sc_biguint( const sc_signed& v );
    sc_biguint( const sc_signed_subref& v );
    template <class T1, class T2>;
    sc_biguint( const sc_signed_concref<T1,T2>& a );
    sc_biguint( const char* v );
    sc_biguint( int64 v );
    sc_biguint( uint64 v );
    sc_biguint( long v );
    sc_biguint( unsigned long v );
    sc_biguint( int v );
    sc_biguint( unsigned int v );
    sc_biguint( double v );
    sc_biguint( const sc_bv_base& v );
    sc_biguint( const sc_lv_base& v );
    explicit sc_biguint( const sc_fxval& v );
    explicit sc_biguint( const sc_fxval_fast& v );
    explicit sc_biguint( const sc_fxnum& v );
    explicit sc_biguint( const sc_fxnum_fast& v );
    ~sc_biguint();

    // assignment operators
    sc_biguint<W>& operator = ( const sc_biguint<W>& v);
    sc_biguint<W>& operator = ( const sc_unsigned& v );
    sc_biguint<W>& operator = ( const
                     sc_unsigned_subref& v );
    template <class T1, class T2>
    sc_biguint<W>& operator = ( const
                     sc_unsigned_concref<T1,T2>& a );
    sc_biguint<W>& operator = ( const sc_signed& v );
    sc_biguint<W>& operator = ( const
                     sc_signed_subref& v );
    template <class T1, class T2>
    sc_biguint<W>& operator = ( const
                     sc_signed_concref<T1,T2>& a );
    sc_biguint<W>& operator = ( const char* v ) ;
    sc_biguint<W>& operator = ( int64 v );
    sc_biguint<W>& operator = ( uint64 v );
    sc_biguint<W>& operator = ( long v );
    sc_biguint<W>& operator = ( unsigned long v );
```



```
    sc_biguint<W>& operator = ( int v ) ;
    sc_biguint<W>& operator = ( unsigned int v ) ;
    sc_biguint<W>& operator = ( double v );
    sc_biguint<W>& operator = ( const sc_bv_base& v );
    sc_biguint<W>& operator = ( const sc_lv_base& v );
    sc_biguint<W>& operator = ( const sc_int_base& v );
    sc_biguint<W>& operator = ( const sc_uint_base& v );
    sc_biguint<W>& operator = ( const sc_fxval& v );
    sc_biguint<W>& operator = ( const sc_fxval_fast& v);
    sc_biguint<W>& operator = ( const sc_fxnum& v );
    sc_biguint<W>& operator = ( const sc_fxnum_fast& v);
};
```

Description

`sc_biguint<W>` is an arbitrary sized unsigned integer. The word length is built into the type and can never change. Methods allow for addressing an individual bit or a sub range of bits.

Example

```
SC_MODULE(my_module) {
    // data types
  sc_uint<3> a;
  sc_uint<44> b;
  sc_biguint<88> c;
  sc_biguint<123> d;
    // process
  void my_proc();

  SC_CTOR(my_module) :
    a(0), // init
    c(7654321)  // init
  {
    b = 33;  // set value
    d = 2300; // set value
    SC_THREAD(my_proc);
  }
};

void my_module::my_proc() {
  a = 1;
  b[30] = a[0];
  cout << b.range(7,0) << endl;

  cout << c << endl;
  d[122] = b;


  wait(300, SC_NS);
  sc_stop();
}
```


Public Constructors

sc_biguint();

Create an sc_biguint instance with an initial value of 0.

sc_biguint(T a) ;

T in { sc_biguint<W>, *sc_[un]signed_subref*†, *sc_[un]signed_concref*†, const char*, [u]int64, [unsigned] long, [unsigned] int, double, sc_bv_base, sc_lv_base, sc_fxval, sc_fxval_fast, sc_fix[ed][_fast]}

Create an sc_biguint with value a. If the word length of a is greater then W, a gets truncated to W bits.

Copy Constructor

sc_biguint(const sc_biguint&) ;

Methods

bool
iszero() const ;

Return true if the value of the sc_biguint instance is zero.

int
length() const ;

Return the word length.

void
print(ostream& os = cout) const ;

Print the sc_biguint instance to an output stream.

void
reverse() ;

Reverse the contents of the sc_biguint instance. I.e. LSB becomes MSB and vice versa.

void
scan(istream& is = cin) ;

Read an sc_biguint value from an input stream.

bool
sign() const ;

Return false.

Assignment Operators

sc_biguint<W>&
operator = (T) ;

T in { sc_biguint<W>, *sc_[un]signed_subref*†, *sc_[un]signed_concref*†, const char*, [u]int64, [unsigned] long, [unsigned] int, double, sc_bv_base, sc_lv_base, sc_fxval, sc_fxval_fast, sc_fix[ed][_fast]}

Assign the value of the right-hand side to the left-hand side. The value is truncated, if its word length is greater than W.

Arithmetic Assignment Operators

```
sc_biguint<W>&
operator OP ( uint64 ) ;
OP in { += -= *= /= %= }
```

The operation of OP is performed and the result is assigned to the lefthand side. If necessary, the result gets truncated.

Bitwise Assignment Operators

```
sc_biguint<W>&
operator  OP ( uint64 ) ;
OP in { &= |= ^= <<= >>= }
```

The operation of OP is performed and the result is assigned to the left hand side. The result gets truncated.

Prefix and Postfix Increment and Decrement Operators

```
sc_biguint<W>& operator  ++ () ;
const sc_biguint<W> operator ++ ( int ) ;
```

The operation of OP is performed as done for type unsigned int.

```
sc_biguint<W>& operator  -- () ;
const sc_biguint<W> operator  -- ( int ) ;
```

The operation is performed as done for type unsigned int.

Relational Operators

```
friend bool operator OP (sc_biguint, sc_biguint ) ;
OP in { == != < <= > >= }
```

These functions return the boolean result of the corresponding equality/inequality check.

Arithmetic Operators

```
friend sc_biguint operator OP ( sc_biguint, sc_biguint ) ;
friend sc_biguint operator OP ( sc_biguint , T ) ;
friend sc_biguint operator OP ( T , sc_biguint ) ;
OP in { + - * / % & | ^ == !=  }
T in { sc_[u]int_base, [u]int64, [unsigned] long,
   [unsigned] int }
```

The operation OP is performed and the result is returned.

```
sc_biguint& operator OP (T)
OP in { += -= *= /= %= &= |= ^= } ;
T in { sc_[un]signed, sc_[u]int_base, [u]int64, [unsigned]
   long, [unsigned] int }
```

The operation OP is performed and the result is assigned to the left hand side.

Shift Operators

```
friend sc_biguint operator OP ( sc_biguint a, sc_biguint
    b );
friend sc_biguint operator OP ( sc_biguint a, T b );
OP in { << >> }
T in { sc_[u]int_base, [u]int64, [unsigned] long,
    [unsigned] int }
```

Shift a to the left/right by b bits and return the result.

```
sc_biguint& operator OP ( T i );
OP in { <<= >>= }
T in { sc_[un]signed, sc_[u]int_base, [u]int64, [unsigned]
    long, [unsigned] int } ;
```

Shift the `sc_biguint` instance to the left/right by i bits and assign the result to the `sc_biguint` instance.

Bitwise not

```
friend sc_biguint
operator ~ ( sc_biguint a );
```

Return the bitwise not of a;

Bit Selection

```
sc_unsigned_bitref†   operator [] ( int i ) ;
sc_unsigned_bitref_r† operator [] ( int i ) const ;
sc_unsigned_bitref†   bit( int i) ;
sc_unsigned_bitref_r† bit( int i) const ;
```

Return a reference to a single bit at index i.

Part Selection

```
sc_unsigned_subref†   range( int high, int low )
sc_unsigned_subref_r† range( int high, int low ) const
sc_unsigned_subref†   operator () ( int high, int low )
sc_unsigned_subref_r† operator () ( int high, int low )
    const
```

Return a reference to a range of bits. The MSB is set to the bit at position high, the LSB is set to the bit at position low.

Explicit Conversion

```
double  to_double() const ;
int     to_int() const ;
int64   to_int64() const ;
long    to_long() const ;
uint64  to_uint64() const ;
unsigned int    to_uint() const ;
unsigned long   to_ulong() const ;
```


Converts the value of sc_biguint instance into the corresponding data type. If the requested type has less word length than the sc_biguint instance, the value gets truncated accordingly. If the requested type has greater word length than the `sc_biguint` instance, the value gets sign extended, if necessary.

to_string`( sc_numrep = SC_DEC ) const`
to_string`( sc_numrep, bool ) const`
Convert the `sc_biguint` instance into its string representation.

11.6 sc_bit

Synopsis

```
class sc_bit
{
public:
    // constructors  & destructor
    sc_bit();
    explicit sc_bit( bool a );
    explicit sc_bit( int a );
    explicit sc_bit( char a );
    explicit sc_bit( const sc_logic& a );
    ~sc_bit();

    // copy constructor
    sc_bit( const sc_bit& a );

    // assignment operators
    sc_bit& operator = ( const sc_bit& b );
    sc_bit& operator = ( int b );
    sc_bit& operator = ( bool b );
    sc_bit& operator = ( char b );
    sc_bit& operator = ( const sc_logic& b );

    // bitwise assignment operators
    sc_bit& operator &= ( const sc_bit& b );
    sc_bit& operator &= ( int b );
    sc_bit& operator &= ( bool b );
    sc_bit& operator &= ( char b );
    sc_bit& operator |= ( const sc_bit& b );
    sc_bit& operator |= ( int b );
    sc_bit& operator |= ( bool b );
    sc_bit& operator |= ( char b );
    sc_bit& operator ^= ( const sc_bit& b );
    sc_bit& operator ^= ( int b );
    sc_bit& operator ^= ( bool b );
    sc_bit& operator ^= ( char b );

    // implicit conversion to bool
    operator bool () const ;
    bool operator ! () const ;

    // explicit conversions
    bool to_bool() const ;
    char to_char() const ;

    // relational operators and functions
    friend bool operator == ( const sc_bit& a, const
    sc_bit& b );
    friend bool operator == ( const sc_bit& a, int b );
    friend bool operator == ( const sc_bit& a, bool b );
    friend bool operator == ( const sc_bit& a, char b );
    friend bool operator == ( int a, const sc_bit& b );
```



```
friend bool operator == ( bool a, const sc_bit& b );
friend bool operator == ( char a, const sc_bit& b );
friend bool equal( const sc_bit& a, const sc_bit& b );
friend bool equal( const sc_bit& a, int b );
friend bool equal( const sc_bit& a, bool b );
friend bool equal( const sc_bit& a, char b );
friend bool equal( int a, const sc_bit& b );
friend bool equal( bool a, const sc_bit& b );
friend bool equal( char a, const sc_bit& b );
friend bool operator != ( const sc_bit& a, const
sc_bit& b );
friend bool operator != ( const sc_bit& a, int b );
friend bool operator != ( const sc_bit& a, bool b );
friend bool operator != ( const sc_bit& a, char b );
friend bool operator != ( int a, const sc_bit& b );
friend bool operator != ( bool a, const sc_bit& b );
friend bool operator != ( char a, const sc_bit& b );
friend bool not_equal( const sc_bit& a, const sc_bit&
b );
friend bool not_equal( const sc_bit& a, int b );
friend bool not_equal( const sc_bit& a, bool b );
friend bool not_equal( const sc_bit& a, char b );
friend bool not_equal( int a, const sc_bit& b );
friend bool not_equal( bool a, const sc_bit& b );
friend bool not_equal( char a, const sc_bit& b );

// bitwise complement
friend const sc_bit operator ~ ( const sc_bit& a );
sc_bit& b_not();
friend void b_not( sc_bit& r, const sc_bit& a );
friend const sc_bit b_not( const sc_bit& a );

// bitwise or
friend const sc_bit operator | ( const sc_bit& a, const
sc_bit& b );
friend const sc_bit operator | ( const sc_bit& a, int
b );
friend const sc_bit operator | ( const sc_bit& a, bool
b );
friend const sc_bit operator | ( const sc_bit& a, char
b );
friend const sc_bit operator | ( int a, const sc_bit&
b );
friend const sc_bit operator | ( bool a, const sc_bit&
b );
friend const sc_bit operator | ( char a, const sc_bit&
b );
friend const sc_bit b_or( const sc_bit& a, const
sc_bit& b );
friend const sc_bit b_or( const sc_bit& a, int b );
friend const sc_bit b_or( const sc_bit& a, bool b );
friend const sc_bit b_or( const sc_bit& a, char b );
friend const sc_bit b_or( int a, const sc_bit& b );
friend const sc_bit b_or( bool a, const sc_bit& b );
```



```
friend const sc_bit b_or( char a, const sc_bit& b );
friend void b_or( sc_bit& r, const sc_bit& a, const
sc_bit& b );
friend void b_or( sc_bit& r, const sc_bit& a, int b );
friend void b_or(sc_bit& r, const sc_bit& a, bool b );
friend void b_or( sc_bit& r, const sc_bit& a, char b );
friend void b_or( sc_bit& r, int a, const sc_bit& b );
friend void b_or( sc_bit& r, bool a, const sc_bit& b );
friend void b_or( sc_bit& r, char a, const sc_bit& b );

// bitwise and
friend const sc_bit operator & ( const sc_bit& a, const
sc_bit& b );
friend const sc_bit operator & ( const sc_bit& a, int
b );
friend const sc_bit operator & ( const sc_bit& a, bool
b );
friend const sc_bit operator & ( const sc_bit& a, char
b );
friend const sc_bit operator & ( int a, const sc_bit&
b );
friend const sc_bit operator & ( bool a, const sc_bit&
b );
friend const sc_bit operator & ( char a, const sc_bit&
b );
friend const sc_bit b_and( const sc_bit& a, const
sc_bit& b );
friend const sc_bit b_and( const sc_bit& a, int b );
friend const sc_bit b_and( const sc_bit& a, bool b );
friend const sc_bit b_and( const sc_bit& a, char b );
friend const sc_bit b_and( int a, const sc_bit& b );
friend const sc_bit b_and( bool a, const sc_bit& b );
friend const sc_bit b_and( char a, const sc_bit& b );
friend void b_and( sc_bit& r, const sc_bit& a, const
sc_bit& b );
friend void b_and( sc_bit& r, const sc_bit& a, int b );
friend void b_and( sc_bit& r, const sc_bit& a, bool b);
friend void b_and( sc_bit& r, const sc_bit& a, char b);
friend void b_and( sc_bit& r, int a, const sc_bit& b );
friend void b_and( sc_bit& r, bool a, const sc_bit& b);
friend void b_and( sc_bit& r, char a, const sc_bit& b);

// bitwise exor
friend const sc_bit operator ^ ( const sc_bit& a, const
sc_bit& b );
friend const sc_bit operator ^ ( const sc_bit& a, int
b );
friend const sc_bit operator ^ ( const sc_bit& a, bool
b );
friend const sc_bit operator ^ ( const sc_bit& a, char
b );
friend const sc_bit operator ^ ( int a, const sc_bit&
b );
```



```
    friend const sc_bit operator ^ ( bool a, const sc_bit&
    b );
    friend const sc_bit operator ^ ( char a, const sc_bit&
    b );
    friend const sc_bit b_xor( const sc_bit& a, const
    sc_bit& b );
    friend const sc_bit b_xor( const sc_bit& a, int b );
    friend const sc_bit b_xor( const sc_bit& a, bool b );
    friend const sc_bit b_xor( const sc_bit& a, char b );
    friend const sc_bit b_xor( int a, const sc_bit& b );
    friend const sc_bit b_xor( bool a, const sc_bit& b );
    friend const sc_bit b_xor( char a, const sc_bit& b );
    friend void b_xor( sc_bit& r, const sc_bit& a, const
    sc_bit& b );
    friend void b_xor( sc_bit& r, const sc_bit& a, int b);
    friend void b_xor( sc_bit& r, const sc_bit& a, bool b);
    friend void b_xor( sc_bit& r, const sc_bit& a, char b);
    friend void b_xor( sc_bit& r, int a, const sc_bit& b );
    friend void b_xor( sc_bit& r, bool a, const sc_bit& b);
    friend void b_xor( sc_bit& r, char a, const sc_bit& b);

    // other functions
    void print( ostream& os = cout ) const ;
    void scan( istream& = cin );
};
```

Description

Instances of **sc_bit** can have the values 0 and 1. This maps to other types as follows:

Type	Values	
sc_bit	0	1
bool	false	true
int	0	1
char	'0'	'1'

For T in { sc_bit bool int char }. Values of type T not found in the table produce undefined behavior.

Public Constructors

sc_bit() ;

Create an sc_bit with the value set to zero.

explicit
sc_bit(T a) ;
T in { sc_bit bool int char }

Create an sc_bit with the converted contents of a. If a is not specified the value is zero.

explicit


```
sc_bit( sc_logic ) ;
```
If initilized with an sc_logic instance, which is neither Log_0 nor Log_1, a warning is printed at runtime.

Copy Constructor
```
sc_bit( const sc_bit& ) ;
```

Public Member Functions & Operators
```
ostream&
operator << ( ostream&, sc_bit );
```
Print the `sc_bit` value to an output stream.

```
istream&
operator >> ( istream&, sc_bit& );
```
Read an `sc_bit` value from an input stream.

```
void
print( ostream& os = cout ) const ;
```
Print the `sc_bit` value to an output stream.

```
void
scan( istream& is = cin ) ;
```
Read an `sc_bit` value from an input stream.

Assignment Operators
```
sc_bit& operator = ( T ) ;
T in { sc_bit bool int char }
```
If assigned with an sc_logic instance, which is neither Log_0 nor Log_1, a warning is printed at runtime.

Bitwise Assignment Operators
```
sc_bit& operator &= ( T ) ;
sc_bit& operator |= ( T ) ;
sc_bit& operator ^= ( T ) ;
```

These operators calculate the boolean value of the AND, OR and XOR function and assign the result to the left-hand side.

Conversions
```
operator bool () const ;
```
Convert an sc_bit implicitly to type bool.

```
bool
operator ! () const ;
```
The NOT operator returns a value of type bool. This is the negated value of the sc_bit instance.

```
bool
to_bool() const ;
```


Convert an sc_bit explicitly to type bool.

```
char
to_char() const ;
```

Convert an sc_bit explicitly to type char.

Test for Equality

```
friend bool operator == ( sc_bit, T );
friend bool operator == ( T, sc_bit );
friend bool equal( sc_bit, T );
friend bool equal( T, sc_bit );
```

Test for Inequality

```
friend bool operator != ( sc_bit, T );
friend bool operator != ( T, sc_bit );
friend bool not_equal( sc_bit, T );
friend bool not_equal( T, sc_bit );
```

Bitwise Complement

```
friend const sc_bit operator ~ ( sc_bit );
sc_bit& b_not();
friend const sc_bit b_not( sc_bit );
```

This functions return their result in the first argument:

```
friend        void b_not( sc_bit&, sc_bit ) ;
```

Bitwise Or

```
friend const sc_bit operator | ( sc_bit, T );
friend const sc_bit operator | ( T, sc_bit );
friend const sc_bit b_or( sc_bit, T );
friend const sc_bit b_or( T, sc_bit );
```

These functions return their result in the first argument:

```
friend void b_or( sc_bit&, sc_bit, T );
friend void b_or( sc_bit&, T, sc_bit );
```

Bitwise And

```
friend const sc_bit operator | ( sc_bit, T );
friend const sc_bit operator | ( T, sc_bit );
friend const sc_bit b_or( sc_bit, T );
friend const sc_bit b_or( T, sc_bit );
```

These functions return their result in the first argument:

```
friend void b_and( sc_bit&, sc_bit, T );
friend void b_and( sc_bit&, T, sc_bit );
```

Bitwise Xor

```
friend const sc_bit operator ^ ( sc_bit, T );
friend const sc_bit operator ^ ( T, sc_bit );
friend const sc_bit b_xor( sc_bit, T );
```



```
friend const sc_bit b_xor( T, sc_bit );
```

These functions return their result in the first argument:

```
friend void b_xor( sc_bit&, sc_bit, T );
friend void b_xor( sc_bit&, T, sc_bit );
```


11.7 sc_buffer

Synopsis

```
template <class T>
class sc_buffer
: public sc_signal<T>
{
public:
    // constructors
    sc_buffer();
    explicit sc_buffer( const char* name_ );

    // interface methods
    virtual void write( const T& );

    // other methods
    sc_buffer<T>& operator = ( const T& a );
    sc_buffer<T>& operator = ( const base_type& a );
    sc_buffer<T>& operator = ( const this_type& a );
    static const char* const kind_string;
    virtual const char* kind() const;

protected:
    virtual void update();

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

Description

sc_buffer is a primitive channel that implements the sc_signal_inout_if. Its behavior is the same as the sc_signal channel with the exception of its write behavior and related events.

sc_buffer is a primitive channel that implements the sc_signal_inout_if interface.

In the description of sc_buffer, *current_value* refers to the value of the sc_buffer instance, *new_value* is the value to be written and *old_value* is the previous value. Chapter 2.4.1 describes the scheduler steps referred to in the description of sc_buffer.

Initialization

The initial current_value of a sc_buffer instance is dependent upon type T and is undefined. The current_value may be explicitly initialized in the sc_main function or in the constructor of the module where it is created.

A sc_buffer may be written by only one process, but may be read by multiple processes.

sc_buffer writes and reads follows evaluate-update semantics suitable for describing hardware.

Write
The write method is executed during the evaluate phase of a delta-cycle during which an update is requested. During the update phase the current_value is assigned the new_value and an event occurs.
The evaluate-update is accomplished using the request_update() and update() methods. request_update() is called during the execution of the write method (in the evaluate phase) indicating to the kernel that an update is required. During the update phase the kernel calls the update method provided by the sc_buffer channel.

Multiple writes in same delta-cycle
If multiple writes by a process to the same sc_buffer occur during a particular evaluate phase of a delta-cycle, the last write executed determines the new_value the sc_buffer will receive in the update phase of the same delta-cycle.

Read
A read is executed during the evaluate phase of a delta-cycle and returns the current_value. It does not consume the data.

Simultaneous reads and writes
If during the evaluate phase of a delta-cycle a read and write occur to the same sc_buffer, the read will return the current_value. The new_value from the write will not be available to read until the next delta-cycle as described above.

Example

```
// GIVEN
sc_buffer<int> m;  // channel of type int
        // channel of type sc_uint<12>
sc_buffer<sc_uint<12> > n;
sc_buffer<bool> clk; // channel of type bool
int i;

//THEN
m.write(i);   //write m with value of i
n.write(8);   //write n with value of 8
if(clk.posedge() ) // was there a posedge?
i = m.read();   // assign value of m to i
        // wait for posedge of clk
wait(clk.posedge_event() ) ;
```

Public Constructors


```
sc_buffer() ;
```
Create a `sc_buffer` instance.

```
explicit
sc_buffer( const char* name_ ) ;
```
Create a `sc_buffer` instance with the string name initialized to `name_`.

Public Member Functions

```
virtual const char*
kind() const ;
```
Returns “sc_buffer”.

```
virtual void
write( const T& val);
```
Schedules an update with `val` as new_value.

Public Operators

```
sc_buffer<T>&
operator = ( const T& val ) ;
```
Schedules an update with `val` as the new_value of the left hand side. Returns a reference to the instance.

```
sc_buffer<T>&
operator = ( const sc_buffer<T>& val ) ;
```
Schedules an update with the current_value of `val` as the new_value of the left hand side. Returns a reference to the instance.

```
sc_buffer<T>&
operator = ( const sc_signal<T>& ) ;
```
Schedules an update with the current_value of `val` as the new_value of the left hand side. Returns a reference to the instance.

Protected Member Functions

```
virtual void
update();
```
Assigns new_value to current_value and causes an event to occur. Called by the kernel during the update phase in response to the execution of a request_update method.

Disabled Member Function

```
sc_buffer( const sc_buffer<T>& );
```


11.8 sc_bv

Synopsis

```
template <int W>
class sc_bv
    : public sc_bv_base
{
public:
    // constructors
    sc_bv();
    explicit sc_bv( bool init_value );
    explicit sc_bv( char init_value );
    sc_bv( const char* a );
    sc_bv( const bool* a );
    sc_bv( const sc_logic* a );
    sc_bv( const sc_unsigned& a );
    sc_bv( const sc_signed& a );
    sc_bv( const sc_uint_base& a );
    sc_bv( const sc_int_base& a );
    sc_bv( unsigned long a );
    sc_bv( long a );
    sc_bv( unsigned int a );
    sc_bv( int a );
    sc_bv( uint64 a );
    sc_bv( int64 a );
    sc_bv( const sc_bv_base& a );
    sc_bv( const sc_bv<W>& a );

    // assignment operators
    template <class X>
    sc_bv<W>& operator = ( const sc_bv_base& a );
    sc_bv<W>& operator = ( const sc_bv<W>& a );
    sc_bv<W>& operator = ( const char* a );
    sc_bv<W>& operator = ( const bool* a );
    sc_bv<W>& operator = ( const sc_logic* a );
    sc_bv<W>& operator = ( const sc_unsigned& a );
    sc_bv<W>& operator = ( const sc_signed& a );
    sc_bv<W>& operator = ( const sc_uint_base& a );
    sc_bv<W>& operator = ( const sc_int_base& a );
    sc_bv<W>& operator = ( unsigned long a );
    sc_bv<W>& operator = ( long a );
    sc_bv<W>& operator = ( unsigned int a );
    sc_bv<W>& operator = ( int a );
    sc_bv<W>& operator = ( uint64 a );
    sc_bv<W>& operator = ( int64 a );
};
```

Description

`sc_bv< W >` is a bit vector of arbitary length. Its word length is set at construction time and can not change later.

Public Constructors

sc_bv() ;
Create an sc_bv with all bits set to zero.

explicit
sc_bv(bool a) ;
Create an sc_bv with all bits set to a.

explicit
sc_bv(char a) ;
Create an sc_bv with all bits set to a, while a can be '0' or '1'.

sc_bv(T a) ;
T in { const char*, const bool*, const sc_logic*, const sc_unsigned&, const sc_signed&, const sc_[u]int_base&, unsigned long, long, unsigned int, int, [u]int64 }
Create an sc_bv with the converted contents of a. If the length of a is greater than the length of sc_bv, a gets truncated. If the length of a is less than the length of sc_bv, the MSBs get padded with Log_0.

Copy Constructor

sc_bv(const sc_bv<W>&) ;

Assignment Operators

sc_bv<W>& **operator =** (const sc_bv<W>& a) ;
sc_bv<W>& **operator =** (T a) ;
T in { const char*, const bool*, const sc_logic*, const sc_unsigned&, const sc_signed&, const sc_[u]int_base&, unsigned long, long, unsigned int, int, [u]int64 }
The value of the righthand side is assigned to the sc_bv. If the length of a is greater than the length of sc_bv, a gets truncated. If the length of a is less than the length of sc_bv, the MSBs get padded with Log_0.

11.9 sc_bv_base

Synopsis

```
class sc_bv_base
{
public:
    // constructors
    explicit sc_bv_base( int length_ =
      sc_length_param().len() );
    explicit sc_bv_base( bool a,
      int length_ = sc_length_param().len() );
    sc_bv_base( const char* a );
    sc_bv_base( const char* a, int length_ );
    template <class X>
    sc_bv_base( const sc_bv_base& a );
    virtual ~sc_bv_base();

    // assignment operators
    template <class X>
    sc_bv_base& operator = ( const sc_bv_base& a );
    sc_bv_base& operator = ( const char* a );
    sc_bv_base& operator = ( const bool* a );
    sc_bv_base& operator = ( const sc_logic* a );
    sc_bv_base& operator = ( const sc_unsigned& a );
    sc_bv_base& operator = ( const sc_signed& a );
    sc_bv_base& operator = ( const sc_uint_base& a );
    sc_bv_base& operator = ( const sc_int_base& a );
    sc_bv_base& operator = ( unsigned long a );
    sc_bv_base& operator = ( long a );
    sc_bv_base& operator = ( unsigned int a );
    sc_bv_base& operator = ( int a );
    sc_bv_base& operator = ( uint64 a );
    sc_bv_base& operator = ( int64 a );

    // methods
    int length() const;
    bool is_01() const;
};
```

Description

sc_bv_base is a bit vector of arbitrary length. Its word length is set at construction time and can not change later.

For sc_bv_base description:

```
T in { const char*, const bool*, const sc_logic*,
    sc_[un]signed, sc_[u]int_base [unsigned] long,
    [unsigned] int, [u]int64 }
```

Pointer arguments are arrays. In the case of 'const bool*' and 'const sc_logic*' the size has to be at least as large as the length of the bitvector.

Public Constructors

```
explicit
sc_bv_base( int = sc_length_param().len() ) ;
```
Create an sc_bv_base of specified length. All bits are set to zero.

```
explicit
sc_bv_base( bool a, int = sc_length_param().len() ) ;
```
Create an sc_bv_base of specified length. All bits are set to a.

```
sc_bv_base( const char* a ) ;
```
Create an sc_bv_base with the contents of a. The character string a must be convertible into a binary string. The length of the newly created sc_bv_base is identical to the length of the binary representation of a.

```
sc_bv_base( const char* a, int b ) ;
```
Create an sc_bv_base with the contents of a. The character string a must be convertible into a binary string. The length of the newly created sc_bv_base is set to b. Sign extension takes place, if b is greater than the bit length of a. If b is less then the length of a, a gets truncated.

Copy Constructor

```
sc_bv_base( sc_bv_base ) ;
```

Methods

```
int
length() const ;
```
Return the length of the bit vector.

```
void
print( ostream& os = cout ) const ;
```
Print the sc_bv_base instance to an output stream.

```
void
scan( istream& is = cin ) ;
```
Read an sc_bv_base value from an input stream.

Assignment Operators

```
sc_bv_base& operator = ( const sc_bv_base& ) ;
sc_bv_base& operator = ( T ) ;
```

The value of the right-hand side is assigned to the left-hand side. The length of the left-hand side does not change. This means that the right-hand side gets either truncated or sign extended.

Bitwise Operators

```
sc_bv_base& operator &= ( T ) ;
```


Calculate the bitwise AND operation and assign the result to the left-hand side. Both operands have to be of equal length.

`const sc_bv_base` **`operator &`** `( T ) const ;`
Return the result of the bitwise AND operation. Both operands have to be of equal length.

`sc_bv_base&` **`operator |=`** `( T ) ;`
Calculate the bitwise OR operation and assign the result to the left-hand side. Both operands have to be of equal length.

`const sc_bv_base` **`operator |`** `( T ) const ;`
Return the result of the bitwise OR operation. Both operands have to be of equal length.

`sc_bv_base&` **`operator ^=`** `( T ) ;`
Calculate the bitwise XOR operation and assign the result to the left-hand side. Both operands have to be of equal length.

`const sc_bv_base` **`operator ^`** `( T ) const ;`
Return the result of the bitwise XOR operation. Both operands have to be of equal length.

`sc_bv_base&` **`operator`** `<<= ( int i ) ;`
Shift the contents of the left-hand side operand `i` bits to the left and assign the result to the left-hand side operand. Zero bits are inserted at the LSB side.

`const sc_bv_base` **`operator <<`** `( int i ) const ;`
Shift the contents of the left-hand side operand `i` bits to the left and return the result. Zero bits are inserted at the LSB side.

`sc_bv_base&` **`operator >>`**`= ( int i ) ;`
Shift the contents of the left-hand side operand `i` bits to the right and assign the result to the left-hand side operand. Zero bits are inserted at the MSB side.

`const sc_bv_base` **`operator >>`** `( int i ) const ;`
Shift the contents of the left hand side operand `i` bits to the right and return the result. Zero bits are inserted at the MSB side.

Bitwise Rotation & Reverse Methods

`sc_bv_base&`
`lrotate``( int i ) ;`
Rotate the contents of the bit vector `i` bits to the left.

`sc_bv_base&`


```
rrotate( int i ) ;
```

Rotate the contents of the bit vector `i` bits to the right.

```
sc_bv_base&
reverse() ;
```

Reverse the contents of the bit vector. LSB becomes MSB and vice versa.

Bit Selection

```
sc_bitref†<sc_bv_base>   operator [] ( int i ) ;
sc_bitref_r†<sc_bv_base> operator [] ( int i ) const ;
sc_bitref†<sc_bv_base>   bit( int i ) ;
sc_bitref_r†<sc_bv_base> bit( int i ) const ;
```

Return a reference to the i-th bit. Return an r-value if the bit vector is constant.

Part Selection

```
sc_subref†<sc_bv_base>   operator () ( int, int ) ;
sc_subref_r†<sc_bv_base> operator () ( int, int ) const ;
sc_subref†<sc_bv_base>   range( int, int ) ;
sc_subref_r†<sc_bv_base> range( int, int ) const ;
```

Return a reference to a range of bits. Return an r-value if the bit vector is constant.

Reduction Methods

```
sc_logic_value_t and_reduce() const ;
sc_logic_value_t nand_reduce() const ;
sc_logic_value_t or_reduce() const ;
sc_logic_value_t nor_reduce() const ;
sc_logic_value_t xor_reduce() const ;
sc_logic_value_t xnor_reduce() const ;
```

Return the result of function F with all bits of the bit vector as input arguments.

```
F in { and nand or nor xor xnor }
```

Relational Operators

```
bool operator == ( T ) const ;
```

Return true if the two bit vectors are equal.

Explicit Conversion

```
int       to_int() const ;
long      to_long() const ;
unsigned int    to_uint() const ;
unsigned long   to_ulong() const ;
```

Convert the bit vector into an int, unsigned int, long or unsigned long respectively. The LSB of the bit vector is put into the LSB of the returned value, etc.

Explicit Conversion to Character String

```
const sc_string to_string() const ;
```

Convert the bit vector into a string representing its contents. Every character represents a bit. MSBs are on the left.

```
const sc_string to_string( sc_numrep nr ) const ;
```

Convert the bit vector into a string representing its contents. The nr argument specifies the base of the number string. A prefix ensures that the string can be read back without changing the value.

```
const sc_string to_string( sc_numrep, bool prefix ) const ;
```

Convert the bit vector into a string representing its contents. The nr argument specifies the base of the number string. A prefix ensures that the string can be read back without changing the value. If prefix is false, no prefix is pre-pended to the value string.

11.10 sc_clock

Synopsis

```
class sc_clock
: public sc_signal_in_if<bool>,
  public sc_module
{
public:
    // constructors & destructor
    sc_clock();
    explicit sc_clock( sc_module_name name_ );
    sc_clock( sc_module_name name_,
         const sc_time& period_,
         double duty_cycle_ = 0.5,
         const sc_time& start_time_ = SC_ZERO_TIME,
         bool posedge_first_ = true );
    sc_clock( sc_module_name name_,
         double period_v_,
         sc_time_unit period_tu_,
         double duty_cycle_ = 0.5 );
    sc_clock( sc_module_name name_,
         double period_v_,
```



```
          sc_time_unit period_tu_,
          double duty_cycle_,
          double start_time_v_,
          sc_time_unit start_time_tu_,
          bool posedge_first_ = true );
         sc_clock( sc_module_name name_,
               double        period_,
               double        duty_cycle_ = 0.5,
               double        start_time_ = 0.0,
               bool          posedge_first_ = true
     );virtual ~sc_clock();

     // interface methods
     virtual const sc_event& default_event() const;
     virtual const sc_event& value_changed_event() const;
     virtual const sc_event& posedge_event() const;
     virtual const sc_event& negedge_event() const;
     virtual const bool& read() const;
     virtual const bool& get_data_ref() const;
     virtual bool event() const;
     virtual bool posedge() const;
     virtual bool negedge() const;

     // other methods
     operator const bool& () const;
     const sc_time& period() const;
     double duty_cycle() const;
     virtual void print( ostream& ) const;
     virtual void dump( ostream& ) const;
     virtual const char* kind() const;

 private:
     // disabled
     sc_clock( const sc_clock& );
     sc_clock& operator = ( const sc_clock& );
 };
```

Description

The **sc_clock** hierarchical channel implements the `sc_signal_in_if<bool>` interface.

An `sc_clock` instance (clock) has the same semantics used in describing hardware clocks.

In the description of `sc_clock`, *string_name* refers to the string name of the instance, *period* refers to amount of time between two edges of the same polarity, *duty_cycle* is the percentage of the period the clock is true expressed as a number of type double (0.5 = 50%), *start_time* is the simulation time when the first edge of the clock occurs, *posedge_first* refers to if the first edge of the clock is a positive edge or not, *current_value* refers to the value of the clock. Chapter 2.4.1 describes the scheduler steps referred to in the description of `sc_clock`.

The period must have a value greater than zero. The duty_cycle must have a value between 0 and 1.0.

Clock objects may be created only in the sc_main function (Chapter 5).

Examples

```
// GIVEN
  // variables of type sc_time
sc_time t (10, SC_NS), t2 (5, SC_NS);

// THEN
  // period of 10ns, 50% duty cycle, start at time = 5ns,
  //first edge positive
sc_clock clk1("clk1", t, 0.5, t2);
  // period of 1, 50% duty cycle, start at time = 0,
  // first edge positive
sc_clock clk2("clk2") ;
  // period = 20ns,50% duty cycle, start at time = 0,
  // first edge positive
sc_clock clk3("clk3", 20, SC_NS);
```

Public Constructors & Destructor

`sc_clock``();`

Create an `sc_clock` instance with an initialization of:
string_name = auto-generated unique string
period = 1 default time unit
duty_cycle = 0.5
start_time = `SC_ZERO_TIME`
posedge_first = true
current_value = `false`

```
explicit
sc_clock( sc_module_name n );
```

Create an `sc_clock` instance with an initialization of:
string_name = n
period = 1 default time unit
duty_cycle = 0.5
start_time = `SC_ZERO_TIME`
posedge_first = true
current_value = `false`

```
sc_clock( sc_module_name n,
          const sc_time& p ,
          double         dc = 0.5,
          const sc_time& st = SC_ZERO_TIME,
          bool           pf = true );
```


Create an `sc_clock` instance with a initialization of:
string_name = n
period = p
duty_cycle = dc
start_time = `st`
posedge_first = `pf`

```
sc_clock( sc_module_name n,
        double         p_val,
        sc_time_unit   p_tu,
        double         dc = 0.5 );
```

Create an `sc_clock` instance with an initialization of:
string_name = n
period = `sc_time(p_val, p_tu)`
duty_cycle = dc0.5
start_time = `SC_ZERO_TIME0`,
posedge_first = `true`
current_value = `false`

```
sc_clock( sc_module_name n ,
        double          p_val,
        sc_time_unit    p_tu,
        double          dc,
        double          st_val,
        sc_time_unit    st_tu,
        bool            pf = true );
```

Create an `sc_clock` instance with a initialization of:
string_name = n
period = `sc_time(p_val, p_tu)`
duty_cycle = dc
start_time = `sc_time(st_val, st_tu)`
posedge_first = `pf`
current_value = `!pf`

```
sc_clock( sc_module_name n,
        double          p_val,
        double          dc = 0.5,
        double          st = 0.0,
        bool            pf = true );
```

Create an `sc_clock` instance with a initialization of:
string_name = n
period = `p_val` default time units
duty_cycle = dc
start_time = `st`
posedge_first = `pf`
current_value = `!pf`

~sc_clock();
Destructor (does nothing).

Public Member Functions

virtual const sc_event&
default_event() const;
Returns a reference to an event that occurs when the value of the clock changes.

double
duty_cycle() const ;
Returns duty_cycle of the clock.

virtual void
dump(ostream&) const;
Prints the name and value of the clock to an output stream.

virtual bool
event() const;
Returns true if an event occurred in the previous delta-cycle.

virtual const bool&
get_data_ref() const ;
Returns a reference to current_value.

virtual const char*
kind() const ;
Returns the character string “sc_clock”.

virtual bool
negedge() const;
Returns true if an event occurred in the previous delta-cycle and current_value is false.

virtual const sc_event&
negedge_event() const ;
Returns a reference to an event, if an event occurred in the previous delta-cycle and current_value is false.

operator const bool& **()** const ;
Returns a reference to the current_value.

const sc_time&
period() const ;
Returns period.

virtual bool
posedge() const;
Returns true if an event occurred in the previous delta-cycle and current_value is true.


```
virtual const sc_event&
posedge_event() const;
```
Returns a reference to an event if an event occurred in the previous delta-cycle and current_value is true.

```
virtual void
print( ostream& ) const;
```
Prints current_value to an output stream.

```
virtual const bool&
read() const;
```
Returns a reference to the current_value.

```
static const sc_time&
time_stamp();
```
Returns the current simulation time.

```
void
trace( sc_trace_file* tf ) const;
```
Adds a trace of current_value to the trace file `tf`.

```
virtual const sc_event&
value_changed_event() const;
```
Returns a reference to an event that occurs when the current_value of the clock changes.

Disabled Member Functions

```
sc_clock( const sc_clock& );
```
Copy constructor.

```
sc_clock&
operator = ( const sc_clock& );
```
Default assignment operator.

11.11 sc_event

Synopsis

```
class sc_event
{
public:
    // constructors & destructor
    sc_event();
    ~sc_event();

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

    // operators
    sc_event_or_list&  operator | ( const
                                     sc_event& ) const;
    sc_event_and_list& operator & ( const
                                     sc_event& ) const;

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

Description

An `sc_event` instance (event) determines when and whether a process execution is triggered.

In the description of `sc_event,` *event* refers to the sc_event object, *delta-delay* refers to a delay of one delta-cycle, *notify_method* refers to the methods that causes event notification and *pending_notification_time* refers to the simulation time the notification or occurrence of the event is scheduled for. Chapter 2.4.1 describes the scheduler steps referred to in the description of `sc_event.`

The event keeps a list of processes that are sensitive to occurrences of the event. Execution of the notify_method schedules or causes the occurrence of an event. Upon occurrence of the event, the event causes processes sensitive to the event to trigger. When the event occurrence happens relative to the execution of the notify_method is dependent upon the type of notification. There are three types of notification:

Immediate notification.

Event occurs in the same evaluate phase within a delta-cycle as the notify_method execution causing processes sensitive to the event to be triggered in the same evaluate phase within the delta-cycle.

Delta-delay notification.
Event occurs in the evaluate phase within the next delta-cycle as the notify_method execution causing processes sensitive to the event to be triggered in the evaluate phase in the next delta-cycle.

Non-zero delay notification (timed notification).
Event occurs delayed by the time value supplied by the notify_method causing processes sensitive to the event to be triggered after the designated amount of time.

A given sc_event object can have at most one pending notification at any point. If multiple notifications are made to an event that would violate this rule, the "earliest notification wins" rule is applied to determine which notification is discarded.

Public Constructors

`sc_event``();`
Create an `sc_event` instance.

Public Member Functions

`void`
`cancel``();`
Removes pending notification of the event.

`void`
`notify``();`
Causes notification of the event in the current delta-cycle.

`void`
`notify``( const sc_time& t_var );`
If `t_var` = 0 then causes notification in the next delta-cycle else schedules notification at current time + `t_var`.

`void`
`notify``( double t_val , sc_time_unit tu);`
If `t_val` = 0 then causes notification in the next delta-cycle else schedules notification at current time + (`t_val, tu`).

Public Operators

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

Adds ev to the *sc_event_or_list*† referenced on the left hand side.

sc_event_and_list† &
operator & (const sc_event& ev) const ;

Adds ev to the *sc_event_and_list*† referenced on the left hand side.

Disabled Member Functions

sc_event(const sc_event&) ;

Copy constructor.

sc_event&
operator = (const sc_event&) ;

Default assignment operator.

11.12 sc_event_finder_t

Synopsis

```
template <class IF>
class sc_event_finder_t
: public sc_event_finder†
{
public:
    // constructors and destructor
    sc_event_finder_t( const sc_port_base†& port_,
         const sc_event& (IF::*event_method_) () const )
    virtual ~sc_event_finder_t()

    // methods
    const sc_port_base†& port() const;
    virtual const sc_event& find_event() const;

private:
    // disabled
    sc_event_finder_t();
    sc_event_finder_t( const sc_event_finder_t<IF>& );
    sc_event_finder_t<IF>& operator = ( const
    sc_event_finder_t<IF>& );
};
```

Description

`sc_event_finder_t` is a class that is used to allow a port or port method to be used in a static sensitivity list. It provides deferred access to channel events through an interface function that returns a sc_event.

Example

```
// A special port method that can be used in
// static sensitivity
sc_event_finder& data_written( ) const
{
  return *new sc_event_finder_t<in_if_type>( *this,
               &in_if_type::data_written_event_func );
}
```

Public Constructor and Destructor

`sc_event_finder_t``( const` *`sc_port_base`*`†&, const sc_event& (IF::*event_method_) () const );`

Creates an event finder object and registers the port and event method in question.

`virtual` **`~sc_event_finder_t`**`();`

Virtual destructor. Does nothing by default.

Public Member Functions


```
const sc_port_base†&
port() const;
```

Returns the port that was registered with this event finder.

```
virtual const sc_event&
find_event() const;
```

Returns a reference to the event returned by the registered event method. Can only be called when the associated port is bound.

Disabled Member Functions

```
sc_event_finder_t();
```

Default constructor.

```
sc_event_finder_t( const sc_event_finder_t<IF>& );
```

Copy constructor.

```
sc_event_finder_t<IF>& operator = ( const
    sc_event_finder_t<IF>& );
```

Default assignment operator.

11.13 sc_fifo

Synopsis

```
template <class T>
class sc_fifo
: public sc_fifo_in_if<T>,
  public sc_fifo_out_if<T>,
  public sc_prim_channel
{
public:
    // constructors and destructor
    explicit sc_fifo( int size_ = 16 );
    explicit sc_fifo( const char* name_, int size_=16);
    virtual ~sc_fifo();

    // interface methods
    virtual void read( T& );
    virtual T read();
    virtual bool nb_read( T& );
    virtual int num_available() const;
    virtual const sc_event& data_written_event() const;
    virtual void write( const T& );
    virtual bool nb_write( const T& );
    virtual int num_free() const;
    virtual const sc_event& data_read_event() const;

    // other methods
    operator T ();
    sc_fifo<T>& operator = ( const T& a );
    void trace( sc_trace_file* tf ) const;
    virtual void print( ostream& ) const;
    virtual void dump( ostream& ) const;
    static const char* const kind_string;
    virtual const char* kind() const;

protected:
    virtual void update();

private:
    // disabled
    sc_fifo( const sc_fifo<T>& );
    sc_fifo& operator = ( const sc_fifo<T>& );
};
```

Description

`sc_fifo` is a primitive channel that implements the `sc_fifo_in_if` and `sc_fifo_out_if` interfaces. It implements the behavior of a FIFO having a fixed maximum size which is set at the point of construction.

.

In the description of `sc_fifo`, *element* refers to an entry in the FIFO, *size* refers to the maximum number of entries the FIFO may have. Chapter 2.4.1 describes the scheduler steps referred to in the description of `sc_fifo`.

Initialization.
The size of the FIFO may be explicitly set to any value. If no size is specified the value defaults to 16.

A `sc_fifo` channel may be connected to only one output (write) and one input (read) port. Multiple different processes may write and read a `sc_fifo` channel.

`sc_fifo` writes and reads follow the evaluate-update semantics. Both blocking and non-blocking reads and writes are provided.

Blocking write.
The write method is executed during the evaluate phase of a delta-cycle. If the FIFO is full then the write method suspends until space is available. If space is available an update is requested. During the update phase the value is inserted into the FIFO.
The evaluate-update is accomplished using the `request_update()` and `update()` methods. `request_update()` is called during the execution of the write method indicating to the kernel that an update is required. During the update phase the kernel calls the update method provided by the `sc_fifo` channel.

Non-blocking write.
If the FIFO is full then the non-blocking write method does nothing. If there is space available then it behaves the same as a blocking write.

Multiple writes in same delta-cycle.
If multiple writes to the same `sc_fifo` occur during a particular evaluate phase of a delta-cycle, all values will be inserted during update phase of the same delta-cycle in the order they were written. No data is lost.

Blocking read.
A read is executed during the evaluate phase of a delta-cycle. If the FIFO is not empty, the read returns the value of the element and requests an update. During the update phase the element is deleted from the FIFO. The evaluate-update is accomplished using the `request_update()` and `update()` methods.

Non-blocking read.
If the FIFO is empty then the non-blocking read method does nothing. If there is data available then it behaves the same as a blocking read.

Multiple reads in same delta-cycle.
If multiple reads to the same `sc_fifo` occur during a particular evaluate phase of a delta-cycle, all values will be returned during the evaluate phase, in the order they were written to the FIFO. The elements are deleted during update phase of the same delta-cycle. Every element that is read is thus deleted from the FIFO.

Simultaneous reads and writes.
Assume a `sc_fifo` channel of depth 1. If during the evaluate phase of a delta-cycle a write followed by a read occur to the same `sc_fifo`, the write will complete, scheduling a value to be inserted on the FIFO. The read will suspend as the FIFO is empty. During the update phase the write value will be inserted and the FIFO status updated. The read will resume in the next delta-cycle where it will return the value written the previous delta-cycle.

Example

```
// GIVEN
sc_fifo<int> m;  // channel of type int
        // channel of type sc_uint<12>
sc_fifo<sc_uint<12> > n;
int i;

//THEN
m.write(i);   //write value of i into the FIFO m
        // wait for data written to n
wait(n.data_written_event() ) ;
i = n.read();   // read a value from and assign to i
if (m.num_free() > 0) // check for room in the FIFO
   m.write(8);  // write the value 8 to into the FIFO
```

Public Constructors

```
explicit
sc_fifo( int size_ = 16 ) ;
```

Create a sc_fifo instance with size initialized to 16.

```
explicit
sc_fifo( const char* name_, int size_ = 16 ) ;
```

Create a sc_fifo instance with size initialized to 16 and the string name initialized to name_.

Public Member Functions

```
virtual const sc_event&
data_read_event() const ;
```

Returns a reference to an event that occurs when an element is read.

```
virtual const sc_event&
data_written_event() const ;
```

Returns a reference to an event that occurs when an element is written.


```
virtual void
dump( ostream& ) const;
```
Prints the string name and all the element values of the `sc_fifo` instance to an output stream.

```
virtual const char*
kind() const ;
```
Returns “sc_fifo”.

```
virtual bool
nb_read( T& val );
```
Returns `false` if the FIFO is empty. Returns `true`, places the element value in `val` and schedules the elements deletion if the FIFO is not empty.

```
virtual bool
nb_write( const T& val ) ;
```
Returns `false` if the FIFO is full. Returns `true` and schedules an insertion of `val` as an element if the FIFO is not full.

```
virtual int
num_available() const ;
```
Returns the number of elements that are currently in the FIFO. However elements written in the current evaluate phase will not affect the value returned by num_available() until the next evaluate phase.

```
virtual int
num_free() const ;
```
Returns the number of free spaces currently in the FIFO. However elements read in the current evaluate phase will not affect the value returned by num_free() until the next evaluate phase.

```
virtual void
print( ostream& ) const;
```
Prints all the element values of the `sc_fifo` instance to an output stream.

```
virtual T
read();
```
Returns an element value from the FIFO and schedules the elements deletion. If the FIFO is empty it suspends until an element is written on the FIFO.

```
virtual void
read( T& val );
```
Places an element value from the FIFO in val and schedules the elements deletion

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


Checks to ensure at most only one input and one output port is connected to the `sc_fifo` instance.

```
void
trace( sc_trace_file* tf ) const;
```

Adds a trace for each element to the trace file `tf`.

```
virtual void
write( const T& val ) ;
```

Schedules an insertion of `val` as an element on the FIFO. If the FIFO is full it suspends until an element is read from the FIFO.

Public Operators

```
operator T () ;
```

Returns an element value from the FIFO and schedules the elements deletion. If the FIFO is empty it suspends until an element is written on the FIFO.

```
sc_fifo<T>&
operator = ( const T& val ) ;
```

Schedules an insertion of `val` into the sc_fifo instance on the left hand side. If the FIFO is full it suspends until an element is read from the FIFO. Returns a reference to the instance.

Protected Member Functions

```
virtual void
update();
```

Disabled Member Functions

```
sc_fifo( const sc_fifo<T>& );

sc_fifo&
operator = ( const sc_fifo<T>& );
```


11.14 sc_fifo_in

Synopsis

```
template <class T>
class sc_fifo_in
: public sc_port<sc_fifo_in_if<T>,0>
{
public:
    // constructors and destructor
    sc_fifo_in();
    sc_fifo_in( const char* name_ );
    sc_fifo_in(sc_fifo_in_if<T>& interface_ );
    sc_fifo_in( const char* name_,
                sc_fifo_in_if<T>& interface_ );
    sc_fifo_in(sc_port_b†<sc_fifo_in_if<T> >& parent_ );
    sc_fifo_in( const char* name_,
                sc_port_b†<sc_fifo_in_if<T> >& parent_ );
    sc_fifo_in( sc_fifo_in<T>& parent_ );
    sc_fifo_in( const char* name_,
                sc_fifo_in<T>& parent_ );
    virtual ~sc_fifo_in();

    // methods
    void read( T& value_ );
    T read();
    bool nb_read( T& value_ );
    int num_available() const ;
    const sc_event& data_written_event() const ;
    sc_event_finder& data_written() const ;
    static const char* const kind_string;
    virtual const char* kind() const

private:
    // disabled
    sc_fifo_in( const sc_fifo_in<T>& );
    sc_fifo_in<T>& operator = ( const sc_fifo_in<T>& );
};
```

Description

sc_fifo_in is a specialized port for use with sc_fifo channels (Chapter 11.13). Its behavior is that of a sc_port which has only one interface that is of type sc_fifo_in_if<T>. It has additional methods for convenience in accessing the FIFO channel connected to the port.

In the description of sc_fifo_in, *port* refers to the sc_fifo_in instance and *FIFO* refers to the fifo channel connected to the port.

Example

```
SC_MODULE(my_module) {
    // output port
```



```
    sc_fifo_out<int> output;
    sc_fifo_in<int>  input;
    int a;
      // process
    void my_proc();

    SC_CTOR(my_module) {
      SC_THREAD(my_proc);
      sensitive << input.data_written();
    }
  };

  void my_module::my_proc() {
    output->write(5);
    output.write(6);
    wait(input->data_written_event() );
    input->nb_read(a);
    a = input->read();
    a = input.read();
    sc_stop();
  }
```

Protected Constructor

`sc_fifo_in``() ;`

Default constructor

`explicit`
`sc_fifo_in``( const char* name_ ) ;`

Create a `sc_fifo_in` instance with the string name initialized to `name_`.

Public Member functions

`const sc_event&`
`data_written_event``() const ;`

Returns a reference to an event that occurs when an element is written to the FIFO.

`sc_event_finder`†`&`
`data_written``() const ;`

Returns a reference to an *sc_event_finder*† that finds the event that occurs when an element is written to FIFO. For use with static sensitivity list of a process.

`virtual const char*`
`kind``() const ;`

Returns “sc_fifo_in”.

`bool`
`nb_read``( T& value_ ) ;`

Returns `false` if the FIFO is full. Returns `true` and schedules an insertion of `value_` as an element if the FIFO is not full.


```
int
num_available() const ;
```
Returns the number of elements that are in the FIFO.

```
void
read( T& value_) ;
```
Places an element value from the FIFO in `value_` and schedules the elements deletion

```
T
read() ;
```
Returns an element value from the FIFO and schedules the elements deletion. If the FIFO is empty it suspends until an element is written on the FIFO.

Disabled Member Functions

```
sc_fifo_in( const sc_fifo_in<T>& ) ;

sc_fifo_in<T>&
operator = ( const sc_fifo_in<T>& ) ;
```


11.15 sc_fifo_in_if

Synopsis

```
template <class T>
class sc_fifo_in_if
: virtual public sc_interface
{
public:
    virtual void read( T& ) = 0;
    virtual T read() = 0;
    virtual bool nb_read( T& ) = 0;
    virtual int num_available() const = 0;
    virtual const sc_event&
                data_written_event() const = 0;

private:
    // disabled
    sc_fifo_in_if( const sc_fifo_in_if<T>& );
    sc_fifo_in_if<T>&
      operator = ( const sc_fifo_in_if<T>& );
};
```

Description

The **sc_fifo_in_if** class provides the signatures of the functions for the sc_fifo_in_if interface. See Chapter 8.1 and sc_fifo for a description of interfaces. Implemented by the sc_fifo channel (Chapter 11.12)

Example

```
SC_MODULE(my_module) {
sc_port<sc_fifo_in_if<int> > p1; // “read” FIFO port

template <class T>
class sc_fifo
: public sc_fifo_in_if<T>,
  public sc_fifo_out_if<T>,
  public sc_prim_channel
{  . . . . };
```

Protected Constructor

sc_fifo_in_if();

Create a sc_fifo_in_if instance.

Public Member functions

```
virtual const sc_event&
data_written_event() const = 0;

virtual bool
nb_read( T& ) = 0;

virtual int
```



```
num_available() const = 0;

virtual T
read() = 0;

virtual void
read( T& ) = 0;
```

Disabled Member Functions

```
sc_fifo_in_if( const sc_fifo_in_if<T>& );

sc_fifo_in_if<T>&
operator = ( const sc_fifo_in_if<T>& );
```


11.16 sc_fifo_out

Synopsis

```
class sc_fifo_out
: public sc_port<sc_fifo_out_if<T>,0>
{
public:
    // constructors and destructor
    sc_fifo_out();
    sc_fifo_out( const char* name_ );
    sc_fifo_out(sc_fifo_out_if<T>& interface_ );
    sc_fifo_out( const char* name_,
                 sc_fifo_out_if<T>& interface_ );
    sc_fifo_out(sc_port_b†<sc_fifo_out_if<T> >& parent_);
    sc_fifo_out( const char* name_,
                 sc_port_b†<sc_fifo_out_if<T> >& parent_);
    sc_fifo_out( sc_fifo_out<T>& parent_ );
    sc_fifo_out( const char* name_,
                 sc_fifo_out<T>& parent_ );
    virtual ~sc_fifo_out();

    // methods
    void write( const T& value_ );
    bool nb_write( const T& value_ );
    int num_free() const;
    const sc_event& data_read_event() const;
    sc_event_finder& data_read() const;
    static const char* const kind_string;
    virtual const char* kind() const;

private:
    // disabled
    sc_fifo_out( const sc_fifo_out<T>& );
    sc_fifo_out<T>& operator = ( const sc_fifo_out<T>&);
};
```

Description

sc_fifo_out is a specialized port for use with sc_fifo channels (Chapter 11.13). Its behavior is that of a sc_port which has only one interface that is of type sc_fifo_out_if<T>. It has additional methods for convenience in accessing the channel connected to the port.

In the description of sc_fifo_out, *port* refers to the sc_fifo_out instance and *FIFO* refers to the fifo channel connected to the port.

Example

```
SC_MODULE(my_module) {
    // output port
  sc_fifo_out<int> output;
  sc_fifo_in<int>  input;
```



```
    int a;
      // process
    void my_proc();

    SC_CTOR(my_module) {
      SC_THREAD(my_proc);
      sensitive << input.data_written();
    }
  };

  void my_module::my_proc() {
    output->write(5);
    output.write(6);
    wait(input->data_written_event() );
    input->nb_read(a);
    a = input->read();
    a = input.read();
    sc_stop();
  }
```

Protected Constructor

sc_fifo_out() ;

Default constructor.

explicit
sc_fifo_out(const char* name_) ;

Create a `sc_fifo_out` instance with the string name initialized to `name_`.

Public Member Functions

const sc_event&
data_read_event() const ;

Returns a reference to an event that occurs when an element is read from FIFO.

sc_event_finder†&
data_read() const ;

Returns a reference to an *sc_event_finder*† that finds the event that occurs when an element is read from FIFO. For use with static sensitivity list of a process.

virtual const char*
kind() const ;

Returns "sc_fifo_out".

bool
nb_write(const T& value_) ;

Returns `false` if the FIFO is full. Returns `true` and schedules an insertion of `val` as an element if the FIFO is not full.

int

num_free`() const ;`
Returns the number of elements that can be written to the FIFO.

```
void
write( const T& value_ ) ;
```
Schedules an insertion of `value_` as an element on the FIFO. If the FIFO is full it suspends until an element is read from the FIFO.

Disabled Member Functions

```
sc_fifo_out( const sc_fifo_out<T>& ) ;

sc_fifo_out<T>&
operator = ( const sc_fifo_out<T>&) ;
```


11.17 sc_fifo_out_if

Synopsis

```
template <class T>
class sc_fifo_out_if
: virtual public sc_interface
{
public:
    virtual void write( const T& ) = 0;
    virtual bool nb_write( const T& ) = 0;
    virtual int num_free() const = 0;
    virtual const sc_event& data_read_event() const = 0;

private:
    // disabled
    sc_fifo_out_if( const sc_fifo_out_if<T>& );
    sc_fifo_out_if<T>& operator =
                      ( const sc_fifo_out_if<T>& );
};
```

Description

The **sc_fifo_out_if** class provides the signatures of the functions for the sc_fifo_out_if interface. See Chapter 8.1 and sc_fifo for a description of interfaces. Implemented by the sc_fifo channel (Chapter 11.12)

Example

```
SC_MODULE(my_module) {
sc_port<sc_fifo_out_if<int> > p1; // “write” FIFO port

template <class T>
class sc_fifo
: public sc_fifo_in_if<T>,
  public sc_fifo_out_if<T>,
  public sc_prim_channel
{  . . . . };
```

Protected Constructor

sc_fifo_out_if();

Create a sc_fifo_in_if instance.

Public Member Functions

virtual const sc_event&
data_read_event() const = 0;

virtual bool
nb_write(const T&) = 0;

virtual int
num_free() const = 0;


```
virtual void
write( const T& ) = 0;
```

Disabled Member Functions

```
sc_fifo_out_if( const sc_fifo_out_if<T>& );

sc_fifo_out_if<T>&
operator = ( const sc_fifo_out_if<T>& );
```


11.18 sc_fix

Synopsis

```
class sc_fix : public sc_fxnum†
{
public:
    // constructors and destructor
    sc_fix( sc_fxnum_observer* = 0 );
    sc_fix( int, int,
          sc_fxnum_observer* = 0 );
    sc_fix( sc_q_mode, sc_o_mode,
          sc_fxnum_observer* = 0 );
    sc_fix( sc_q_mode, sc_o_mode, int,
          sc_fxnum_observer* = 0 );
    sc_fix( int, int, sc_q_mode, sc_o_mode,
          sc_fxnum_observer* = 0 );
    sc_fix( int, int, sc_q_mode, sc_o_mode, int,
          sc_fxnum_observer* = 0 );
    sc_fix( const sc_fxcast_switch&,
          sc_fxnum_observer* = 0 );
    sc_fix( int, int,
          const sc_fxcast_switch&,
          sc_fxnum_observer* = 0 );
    sc_fix( sc_q_mode, sc_o_mode,
          const sc_fxcast_switch&,
          sc_fxnum_observer* = 0 );
    sc_fix( sc_q_mode, sc_o_mode, int,
          const sc_fxcast_switch&,
          sc_fxnum_observer* = 0 );
    sc_fix( int, int, sc_q_mode, sc_o_mode,
          const sc_fxcast_switch&,
          sc_fxnum_observer* = 0 );
    sc_fix( int, int, sc_q_mode, sc_o_mode, int,
          const sc_fxcast_switch&,
          sc_fxnum_observer* = 0 );
    sc_fix( const sc_fxtype_params&,
          sc_fxnum_observer* = 0 );
    sc_fix( const sc_fxtype_params&,
          const sc_fxcast_switch&,
          sc_fxnum_observer* = 0 );

#define DECL_CTORS_T(tp) \
    sc_fix( tp, int, int, \
      sc_fxnum_observer* = 0 ); \
    sc_fix( tp, sc_q_mode, sc_o_mode, \
      sc_fxnum_observer* = 0 ); \
    sc_fix( tp, sc_q_mode, sc_o_mode, int, \
      sc_fxnum_observer* = 0 ); \
    sc_fix( tp, int, int, sc_q_mode, sc_o_mode, \
      sc_fxnum_observer* = 0 ); \
    sc_fix( tp,  int, int, sc_q_mode, sc_o_mode, int, \
      sc_fxnum_observer* = 0 ); \
```



```
    sc_fix( tp, const sc_fxcast_switch&, \
      sc_fxnum_observer* = 0 ); \
    sc_fix( tp, int, int, const sc_fxcast_switch&, \
      sc_fxnum_observer* = 0 ); \
    sc_fix( tp, sc_q_mode, sc_o_mode, \
      const sc_fxcast_switch&, \
      sc_fxnum_observer* = 0 ); \
    sc_fix( tp, sc_q_mode, sc_o_mode, int, \
      const sc_fxcast_switch&, \
      sc_fxnum_observer* = 0 ); \
    sc_fix( tp, int, int, sc_q_mode, sc_o_mode, \
      const sc_fxcast_switch&, \
      sc_fxnum_observer* = 0 ); \
    sc_fix( tp, int, int, sc_q_mode, sc_o_mode, int, \
      const sc_fxcast_switch&, \
      sc_fxnum_observer* = 0 ); \
    sc_fix( tp, const sc_fxtype_params&, \
      sc_fxnum_observer* = 0 ); \
    sc_fix( tp, const sc_fxtype_params&, \
      const sc_fxcast_switch&, \
      sc_fxnum_observer* = 0 );

#define DECL_CTORS_T_A(tp) \
    sc_fix( tp, sc_fxnum_observer* = 0 ); \
    DECL_CTORS_T(tp)

#define DECL_CTORS_T_B(tp) \
    explicit sc_fix( tp, sc_fxnum_observer* = 0 ); \
    DECL_CTORS_T(tp)

    DECL_CTORS_T_A(int)
    DECL_CTORS_T_A(unsigned int)
    DECL_CTORS_T_A(long)
    DECL_CTORS_T_A(unsigned long)
    DECL_CTORS_T_A(double)
    DECL_CTORS_T_A(const char*)
    DECL_CTORS_T_A(const sc_fxval&)
    DECL_CTORS_T_A(const sc_fxval_fast&)
    DECL_CTORS_T_A(const sc_fxnum&)
    DECL_CTORS_T_A(const sc_fxnum_fast&)
    DECL_CTORS_T_B(int64)
    DECL_CTORS_T_B(uint64)
    DECL_CTORS_T_B(const sc_int_base&)
    DECL_CTORS_T_B(const sc_uint_base&)
    DECL_CTORS_T_B(const sc_signed&)
    DECL_CTORS_T_B(const sc_unsigned&)
    sc_fix( const sc_fix& );

    // unary bitwise operators
    const sc_fix operator ~ () const;

    // unary bitwise functions
    friend void b_not( sc_fix&, const sc_fix& );
```



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

    // binary bitwise functions
    friend void b_and( sc_fix&, const sc_fix&,
                      const sc_fix& );
    friend void b_and( sc_fix&, const sc_fix&,
                      const sc_fix_fast& );
    friend void b_and( sc_fix&, const sc_fix_fast&,
                      const sc_fix& );
    friend void b_or ( sc_fix&, const sc_fix&,
                      const sc_fix& );
    friend void b_or ( sc_fix&, const sc_fix&,
                      const sc_fix_fast& );
    friend void b_or ( sc_fix&, const sc_fix_fast&,
                      const sc_fix& );
    friend void b_xor( sc_fix&, const sc_fix&,
                      const sc_fix& );
    friend void b_xor( sc_fix&, const sc_fix&,
                      const sc_fix_fast& );
    friend void b_xor( sc_fix&, const sc_fix_fast&,
                      const sc_fix& );

    sc_fix& operator = ( const sc_fix& );

#define DECL_ASN_OP_T(op,tp) \
    sc_fix& operator op ( tp );

#define DECL_ASN_OP_OTHER(op) \
    DECL_ASN_OP_T(op,int64) \
    DECL_ASN_OP_T(op,uint64) \
    DECL_ASN_OP_T(op,const sc_int_base&) \
    DECL_ASN_OP_T(op,const sc_uint_base&) \
    DECL_ASN_OP_T(op,const sc_signed&) \
    DECL_ASN_OP_T(op,const sc_unsigned&)

#define DECL_ASN_OP(op) \
```



```
    DECL_ASN_OP_T(op,int) \
    DECL_ASN_OP_T(op,unsigned int) \
    DECL_ASN_OP_T(op,long) \
    DECL_ASN_OP_T(op,unsigned long) \
    DECL_ASN_OP_T(op,double) \
    DECL_ASN_OP_T(op,const char*)\
    DECL_ASN_OP_T(op,const sc_fxval&)\
    DECL_ASN_OP_T(op,const sc_fxval_fast&)\
    DECL_ASN_OP_T(op,const sc_fxnum&) \
    DECL_ASN_OP_T(op,const sc_fxnum_fast&) \
    DECL_ASN_OP_OTHER(op)

    DECL_ASN_OP(=)
    DECL_ASN_OP(*=)
    DECL_ASN_OP(/=)
    DECL_ASN_OP(+=)
    DECL_ASN_OP(-=)
    DECL_ASN_OP_T(<<=,int)
    DECL_ASN_OP_T(>>=,int)
    DECL_ASN_OP_T(&=,const sc_fix&)
    DECL_ASN_OP_T(&=,const sc_fix_fast&)
    DECL_ASN_OP_T(|=,const sc_fix&)
    DECL_ASN_OP_T(|=,const sc_fix_fast&)
    DECL_ASN_OP_T(^=,const sc_fix&)
    DECL_ASN_OP_T(^=,const sc_fix_fast&)

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

Description

Unconstrained type **`sc_fix`** is a signed (two's complement) type. `sc_fix` allows specifying the fixed-point type parameters wl, iwl, q_mode, o_mode, and n_bits as variables. See Chapter 6.8.12.1.

Declaration Syntax

```
sc_fix  var_name([init_val]
                 [,wl,iwl]
                 [,q_mode,o_mode[,n_bits]]
                 [,cast_switch]
                 [,observer]);

sc_fix  var_name([init_val]
                 ,type_params
                 [,cast_switch]
                 [,observer]);
```

Examples

```
sc_fix a(1.5);
sc_fix c(16,1,SC_RND_CONV,SC_SAT_SYM);
```



```
sc_fix b = -1;
```

Public Constructors

```
sc_fix (
    [type_ init_val]
    [,int wl,int iwl]
    [,sc_q_mode q_mode,sc_o_mode o_mode[,int n_bits]]
    [,const sc_fxcast_switch& cast_switch]
    , sc_fxnum_observer* observer) ;
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }

sc_fix (
    [type_ init_val]
    ,const sc_fxtype_param& type_params
    [,sc_fxcast_switch cast_switch]
    , sc_fxnum_observer* observer) ;
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

Notes on `type_`
For all types in `type_` , except `sc_[u]fix` and `sc_[u]fix_fast`, only the value of the argument is taken, that is, any type information is discarded. This ensures that initialization during declaration and initialization after declaration behave identical.
A fixed-point variable can be initialized with a C/C++ character string (type const char*) either when the number will be expressed in binary form or when the number is too large to be written as a C/C++ built-in type literal

init_val
The initial value of the variable. If the initial value is not specified, the instance is uninitialized.
wl
The total number of bits in the fixed-point format. `wl` must be greater than zero, otherwise, a runtime error is produced. The default value for `wl` is obtained from the fixed-point context type `sc_fxtype_context`. See Chapter 11.26. The total word length parameter cannot change after declaration.
iwl
The number of integer bits in the fixed-point format. `iwl` can be positive or negative. The default value for `iwl` is obtained from the fixed-point context type

sc_fxtype_context. See See Chapter 11.26. The number of integer bits parameter cannot change after declaration.
q_mode
The quantization mode to use. Valid values for q_mode are given in Section 0 . The default value for q_mode is obtained from the fixed-point context type sc_fxtype_context. See See Chapter 11.26. The quantization mode parameter cannot change after declaration.
o_mode
The overflow mode to use. Valid values for o_mode are given in Section 0. The default value for o_mode is obtained from the fixed-point context type sc_fxtype_context. See Chapter 11.26. The overflow mode parameter cannot change after declaration.
n_bits
The number of saturated bits parameter for the selected overflow mode. n_bits must be greater than or equal to zero, otherwise a runtime error is produced. If the overflow mode is specified, the default value is zero. If the overflow mode is not specified, the default value is obtained from the fixed-point context type sc_fxtype_context. See Chapter 11.26. The number of saturated bits parameter cannot change after declaration.
type_params
A fixed-point type parameters object.
cast_switch
The cast switch, which allows to switch fixed-point type casting on or off. Valid values for cast_switch are:
SC_OFF for casting off
SC_ON for casting on
The default value for cast_switch is obtained from the fixed-point context type sc_fxcast_context. The cast_switch parameter cannot change after declaration.
observer
A pointer to an observer object. The observer argument is of type sc_fxnum_observer*. See Chapter 11.25. The default value for observer is 0 (null pointer). The observer parameter cannot change after declaration.

Copy Constructor

sc_fix(const sc_fix&);

Operators

The operators defined for the sc_fix are given in Table 16.

Table 16. Operators for sc_fix

Operator class	Operators in class
Bitwise	~ & ^ |
Arithmetic	* / + - << >> ++ --

Equality	== !=
Relational	<<= >>=
Assignment	= *= /= += -= <<= >>= &= ^= |=

Note:
Operators << and operator >> define arithmetic shifts, not bitwise shifts. The difference is that no bits are lost and proper sign extension is done.

In expressions with the non-bitwise operators from Table 16, fixed-point types can be mixed with all types given:

```
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

The return type of any arithmetic operation is the fixed-point value type, which guarantees that the operation is performed without overflow or quantization.

A floating-point variable or a fixed-point value variable can contain one of the special values +Inf (plus infinity), -Inf (minus infinity), or Nan (not a number). Assignment of one of these special values to a fixed-point variable will produce a runtime error.

For the fixed-point types, a minimal set of bitwise operators is defined. These bitwise operators are only defined on either the signed fixed-point types or the unsigned fixed- point types. Mixing between signed and unsigned fixed-point types is not allowed. Mixing with any other type is also not allowed.

The semantics of the bitwise operators is as follows. For the unary ~ operator, the type of the result is the type of the operand. The bits in the two's complement mantissa of the operand are inverted to get the mantissa of the result. For the binary operators, the type of the result is the maximum aligned type of the two operands, that is, the two operands are aligned by the binary point and the maximum integer word length and the maximum fractional word length is taken. The operands are temporarily extended to this type before performing a bitwise and, bitwise exclusive-or, or bitwise or.

Member Functions

The functions defined for `sc_fix` are given in Table 17.

Table 17. Functions for sc_fix

Function class	**Functions in class**
Bitwise	b_not, b_and, b_xor, b_or

Arithmetic	neg, mult, div, add, sub, lshift, rshift

The functions in Table 17 have return type void. The first argument of these functions is a reference to the result object. The remaining arguments of these functions are the operands.

For the bitwise functions, the result object and the operands are of the same type, which is either `sc_fix` or `sc_ufix`.

The `neg` arithmetic function takes one operand, the other arithmetic functions take two operands. At least one of the operands of the arithmetic functions should have a fixed- point type, the other operand can have any of the types given:

```
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

The arithmetic functions are defined twice: once with the result object of type sc_fxval, and once with the result object of type sc_fix or sc_ufix.

Bit Selection

```
const sc_fxnum_bitref† operator [] ( int i) const;
sc_fxnum_bitref†       operator [] ( int i);

const sc_fxnum_bitref† bit( int i) const;
sc_fxnum_bitref†       bit( int i);
```

These functions take one argument of type int, which is the index into the fixed-point mantissa. The index argument must be between `wl-1` (MSB) and 0 (LSB). Otherwise, a runtime error is produced. The return type of the bit selection functions is (const or non- const) *sc_fxnum_bitref*†, which is a proxy class. The proxy class allows bit selection to be used both as `rvalue` (for reading) and `lvalue` (for writing). For bit selection, the fixed-point binary point is ignored.

Part Selection

```
const sc_fxnum_subref† operator () ( int, int ) const;
sc_fxnum_subref†       operator () ( int, int );

const sc_fxnum_subref† range( int, int ) const;
sc_fxnum_subref†       range( int, int );
```


These functions take two arguments of type `int`, which are the begin and end indices into the fixed-point mantissa. The index arguments must be between `wl-1` (MSB) and 0 (LSB). Otherwise, a runtime error is produced. The return type of the part selection functions is (const or non-const) *sc_fxnum_subref*†, which is a proxy class that behaves like type `sc_bv_base`. The proxy class allows part selection to be used both as `rvalue` (for reading) and `lvalue` (for writing). All operators and methods that are available for type `sc_bv_base` are also available for part selection. For part selection, the fixed-point binary point is ignored.

```
const sc_fxnum_subref† operator () () const;
sc_fxnum_subref†       operator () ();

const sc_fxnum_subref† range() const;
sc_fxnum_subref†       range();
```

As a shortcut for part selection of the complete mantissa, operator () and the range() method can be called without any arguments.

Query Parameters

```
const sc_fxcast_switch&
cast_switch() const;
```
Returns the cast switch parameter.

```
int
iwl() const;
```
Returns the integer word length parameter.

```
int
n_bits() const;
```
Returns the number of saturated bits parameter.

```
sc_o_mode
o_mode() const;
```
Returns the overflow mode parameter.

```
sc_q_mode
q_mode() const;
```
Return the quantization mode parameter.

```
const sc_fxtype_params&
type_params() const;
```
Returns the type parameters.

```
int
wl() const;
```


Returns the total word length parameter.

Query Value

```
bool
is_neg() const;
```

Returns true if the variable holds a negative value. Returns false otherwise.

```
bool
is_zero() const;
```

Returns true if the variable holds a zero value. Returns false otherwise.

```
bool
overflow_flag() const;
```

Returns true if the last write action on this variable caused overflow. Returns false otherwise.

```
bool
quantization_flag() const;
```

Returns true if the last write action on this variable caused quantization. Returns false otherwise.

```
const sc_fxval
value() const;
```

Returns the value.

Implicit Conversion

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

Implicit conversion to the implementation type `double`. The value does not change.

Explicit Conversion

```
short           to_short() const;
unsigned short  to_ushort() const;
int             to_int() const;
unsigned int    to_uint() const;
long            to_long() const;
unsigned long   to_ulong() const;
float           to_float() const;
double          to_double() const
```

```
const sc_string to_string() const;
const sc_string to_string( sc_numrep ) const;
const sc_string to_string( sc_numrep, bool ) const;
const sc_string to_string( sc_fmt ) const;
const sc_string to_string( sc_numrep, sc_fmt ) const;
const sc_string to_string( sc_numrep, bool, sc_fmt ) const;
```

The value of a fixed-point variable can be converted to a character string with the to_string() method. This method takes different arguments for formatting purposes. See Chapter 6.8.8 for more information on converting fixed-point variables to/from character strings. Furthermore, writing to C++ output streams with operator << is supported, e.g. cout << a;, where a is a fixed-point variable. The decimal number representation is used in this case.

```
const sc_string to_dec() const;
const sc_string to_bin() const;
const sc_string to_oct() const;
const sc_string to_hex() const;
```

Shortcut methods for conversion to a character string. See Chapter 6.8.9.2.

Print or dump content

```
void
print( ostream& = cout ) const;
```

Print the sc_fix instance value to an output stream.

```
void
scan( istream& = cin );
```

Read an sc_fix value from an input stream.

```
void
dump( ostream& = cout )
const;
```

Prints the sc_fix instance value, parameters and flags to an output stream.

```
ostream&
operator << ( ostream& os, const sc_fix& a )
```

Print the instance value of a to an output stream os.

11.19 sc_fix_fast

Synopsis

```
class sc_fix_fast : public sc_fxnum_fast†
{
public:
    // constructors
    sc_fix_fast( sc_fxnum_fast_observer* = 0 );
    sc_fix_fast( int, int,
      sc_fxnum_fast_observer* = 0 );
    sc_fix_fast( sc_q_mode, sc_o_mode,
      sc_fxnum_fast_observer* = 0 );
    sc_fix_fast( sc_q_mode, sc_o_mode, int,
      sc_fxnum_fast_observer* = 0 );
    sc_fix_fast( int, int, sc_q_mode, sc_o_mode,
      sc_fxnum_fast_observer* = 0 );
    sc_fix_fast( int, int, sc_q_mode, sc_o_mode, int,
      sc_fxnum_fast_observer* = 0 );
    sc_fix_fast( const sc_fxcast_switch&,
      sc_fxnum_fast_observer* = 0 );
    sc_fix_fast( int, int,
      const sc_fxcast_switch&,
      sc_fxnum_fast_observer* = 0 );
    sc_fix_fast( sc_q_mode, sc_o_mode,
      const sc_fxcast_switch&,
      sc_fxnum_fast_observer* = 0 );
    sc_fix_fast( sc_q_mode, sc_o_mode, int,
      const sc_fxcast_switch&,
      sc_fxnum_fast_observer* = 0 );
    sc_fix_fast( int, int, sc_q_mode, sc_o_mode,
      const sc_fxcast_switch&,
      sc_fxnum_fast_observer* = 0 );
    sc_fix_fast( int, int, sc_q_mode, sc_o_mode, int,
      const sc_fxcast_switch&,
      sc_fxnum_fast_observer* = 0 );
    sc_fix_fast( const sc_fxtype_params&,
      sc_fxnum_fast_observer* = 0 );
    sc_fix_fast( const sc_fxtype_params&,
      const sc_fxcast_switch&,
      sc_fxnum_fast_observer* = 0 );

#define DECL_CTORS_T(tp) \
    sc_fix_fast( tp, int, int, \
      sc_fxnum_fast_observer* = 0 ); \
    sc_fix_fast( tp, sc_q_mode, sc_o_mode, \
      sc_fxnum_fast_observer* = 0 ); \
    sc_fix_fast( tp,  sc_q_mode, sc_o_mode, int, \
      sc_fxnum_fast_observer* = 0 ); \
    sc_fix_fast( tp, int, int, sc_q_mode, sc_o_mode, \
      sc_fxnum_fast_observer* = 0 ); \
    sc_fix_fast( tp, \
      int, int, sc_q_mode, sc_o_mode, int, \
```



```
        sc_fxnum_fast_observer* = 0 ); \
    sc_fix_fast( tp, const sc_fxcast_switch&,\
        sc_fxnum_fast_observer* = 0 ); \
    sc_fix_fast( tp, int, int, \
        const sc_fxcast_switch&,\
        sc_fxnum_fast_observer* = 0 ); \
    sc_fix_fast( tp, sc_q_mode, sc_o_mode, \
        const sc_fxcast_switch&,\
        sc_fxnum_fast_observer* = 0 ); \
    sc_fix_fast( tp, sc_q_mode, sc_o_mode, int, \
        const sc_fxcast_switch&,\
        sc_fxnum_fast_observer* = 0 ); \
    sc_fix_fast( tp, int, int, sc_q_mode, sc_o_mode, \
        const sc_fxcast_switch&,\
        sc_fxnum_fast_observer* = 0 ); \
    sc_fix_fast( tp, int, int, sc_q_mode, sc_o_mode, int, \
        const sc_fxcast_switch&,\
        sc_fxnum_fast_observer* = 0 ); \
    sc_fix_fast( tp, const sc_fxtype_params&,\
        sc_fxnum_fast_observer* = 0 ); \
    sc_fix_fast( tp, const sc_fxtype_params&,\
        const sc_fxcast_switch&,\
        c_fxnum_fast_observer* = 0 );

#define DECL_CTORS_T_A(tp) \
    sc_fix_fast( tp, sc_fxnum_fast_observer* = 0 ); \
     DECL_CTORS_T(tp)

#define DECL_CTORS_T_B(tp) \
     explicit sc_fix_fast( tp, \
        sc_fxnum_fast_observer* = 0 ); \
     DECL_CTORS_T(tp)

     DECL_CTORS_T_A(int)
     DECL_CTORS_T_A(unsigned int)
     DECL_CTORS_T_A(long)
     DECL_CTORS_T_A(unsigned long)
     DECL_CTORS_T_A(double)
     DECL_CTORS_T_A(const char*)
     DECL_CTORS_T_A(const sc_fxval&)
     DECL_CTORS_T_A(const sc_fxval_fast&)
     DECL_CTORS_T_A(const sc_fxnum&)
     DECL_CTORS_T_A(const sc_fxnum_fast&)
     DECL_CTORS_T_B(int64)
     DECL_CTORS_T_B(uint64)
     DECL_CTORS_T_B(const sc_int_base&)
     DECL_CTORS_T_B(const sc_uint_base&)
     DECL_CTORS_T_B(const sc_signed&)
     DECL_CTORS_T_B(const sc_unsigned&)

    // copy constructor
    sc_fix_fast( const sc_fix_fast& );

    // operators
```



```
    const sc_fix_fast operator ~ () const;
    friend void b_not( sc_fix_fast&, const
      sc_fix_fast& );
    friend const sc_fix_fast operator & ( const
      sc_fix_fast&,
      const sc_fix_fast& );
    friend const sc_fix_fast operator ^ ( const
      sc_fix_fast&,
      const sc_fix_fast& );
    friend const sc_fix_fast operator | ( const
      sc_fix_fast&,
      const sc_fix_fast& );
    friend void b_and( sc_fix_fast&, const sc_fix_fast&,
      const sc_fix_fast& );
    friend void b_or ( sc_fix_fast&, const sc_fix_fast&,
      const sc_fix_fast& );
    friend void b_xor( sc_fix_fast&, const sc_fix_fast&,
      const sc_fix_fast& );
    sc_fix_fast& operator = ( const sc_fix_fast& );

#define DECL_ASN_OP_T(op,tp) \
     sc_fix_fast& operator op ( tp );

#define DECL_ASN_OP_OTHER(op) \
     DECL_ASN_OP_T(op,int64) \
     DECL_ASN_OP_T(op,uint64) \
     DECL_ASN_OP_T(op,const sc_int_base&)\
     DECL_ASN_OP_T(op,const sc_uint_base&)\
     DECL_ASN_OP_T(op,const sc_signed&)\
     DECL_ASN_OP_T(op,const sc_unsigned&)

#define DECL_ASN_OP(op) \
     DECL_ASN_OP_T(op,int) \
     DECL_ASN_OP_T(op,unsigned int) \
     DECL_ASN_OP_T(op,long) \
     DECL_ASN_OP_T(op,unsigned long) \
     DECL_ASN_OP_T(op,double) \
     DECL_ASN_OP_T(op,const char*)\
     DECL_ASN_OP_T(op,const sc_fxval&)\
     DECL_ASN_OP_T(op,const sc_fxval_fast&)\
     DECL_ASN_OP_T(op,const sc_fxnum&)\
     DECL_ASN_OP_T(op,const sc_fxnum_fast&)\
     DECL_ASN_OP_OTHER(op)

     DECL_ASN_OP(=)
     DECL_ASN_OP(*=)
     DECL_ASN_OP(/=)
     DECL_ASN_OP(+=)
     DECL_ASN_OP(-=)
     DECL_ASN_OP_T(<<=,int)
     DECL_ASN_OP_T(>>=,int)
     DECL_ASN_OP_T(&=,const sc_fix&)
     DECL_ASN_OP_T(&=,const sc_fix_fast&)
     DECL_ASN_OP_T(|=,const sc_fix&)
```



```
      DECL_ASN_OP_T(|=,const sc_fix_fast&)
      DECL_ASN_OP_T(^=,const sc_fix&)
      DECL_ASN_OP_T(^=,const sc_fix_fast&)

     const sc_fxval_fast operator ++ ( int );
     const sc_fxval_fast operator -- ( int );
     sc_fix_fast& operator ++ ();
     sc_fix_fast& operator -- ();
};
```

Description

sc_fix_fast is a signed (two's complement) limited precision type.
sc_fix_fast allows specifying the fixed-point type parameters wl, iwl, q_mode, o_mode, and n_bits as variables. See Chapter 6.8.1.

sc_fix_fast provides the same API as sc_fix.

sc_fix_fast uses double precision (floating-point) values. The mantissa of a double precision value is limited to 53 bits. This means that bit-true behavior cannot be guaranteed with the limited precision types. For bit-true behavior with the limited precision types, the following guidelines should be followed:
Make sure that the word length of the result of any operation or expression does not exceed 53 bits.
The result of an addition or subtraction requires a word length that is one bit more than the maximum *aligned* word length of the two operands.
The result of a multiplication requires a word length that is the sum of the word lengths of the two operands.

Declaration Syntax

```
sc_fix_fast  var_name([init_val]
                 [,wl,iwl]
                 [,q_mode,o_mode[,n_bits]]
                 [,cast_switch]
                 [,observer]);

sc_fix_fast  var_name([init_val]
                 ,type_params
                 [,cast_switch]
                 [,observer]);
```

Examples

```
sc_fix_fast a(1.5);
sc_fix_fast c(16,1,SC_RND_CONV,SC_SAT_SYM);
sc_fix_fast b = -1;
```

Public Constructors

```
sc_fix_fast (
    [type_ init_val]
    [,int wl,int iwl]
```



```
    [,sc_q_mode q_mode,sc_o_mode o_mode[,int n_bits]]
    [,const sc_fxcast_switch& cast_switch]
    , sc_fxnum_fast_observer* observer) ;
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }

sc_fix_fast (
    [type_ init_val]
    ,const sc_fxtype_param& type_params
    [,sc_fxcast_switch cast_switch]
    , sc_fxnum_fast_observer* observer) ;
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

Notes on `type_`
For all types in `type_` , except `sc_[u]fix` and `sc_[u]fix_fast`, only the value of the argument is taken, that is, any type information is discarded. This ensures that initialization during declaration and initialization after declaration behave identical.
A fixed-point variable can be initialized with a C/C++ character string (type const char*) either when the number will be expressed in binary form or when the number is too large to be written as a C/C++ built-in type literal

init_val
The initial value of the variable. If the initial value is not specified, the instance is uninitialized.
wl
The total number of bits in the fixed-point format. `wl` must be greater than zero, otherwise, a runtime error is produced. The default value for `wl` is obtained from the fixed-point context type `sc_fxtype_context`. See Chapter 11.26. The total word length parameter cannot change after declaration.
iwl
The number of integer bits in the fixed-point format. `iwl` can be positive or negative. The default value for `iwl` is obtained from the fixed-point context type `sc_fxtype_context`. See See Chapter 11.26. The number of integer bits parameter cannot change after declaration.
q_mode
The quantization mode to use. Valid values for `q_mode` are given in Section 0 . The default value for `q_mode` is obtained from the fixed-point context type `sc_fxtype_context`. See See Chapter 11.26. The quantization mode parameter cannot change after declaration.

o_mode
The overflow mode to use. Valid values for o_mode are given in Section 0. The default value for o_mode is obtained from the fixed-point context type sc_fxtype_context. See Chapter 11.26. The overflow mode parameter cannot change after declaration.
n_bits
The number of saturated bits parameter for the selected overflow mode. n_bits must be greater than or equal to zero, otherwise a runtime error is produced. If the overflow mode is specified, the default value is zero. If the overflow mode is not specified, the default value is obtained from the fixed-point context type sc_fxtype_context. See Chapter 11.26. The number of saturated bits parameter cannot change after declaration.
type_params
A fixed-point type parameters object.
cast_switch
The cast switch, which allows to switch fixed-point type casting on or off. Valid values for cast_switch are:
SC_OFF for casting off
SC_ON for casting on
The default value for cast_switch is obtained from the fixed-point context type sc_fxcast_context. The cast_switch parameter cannot change after declaration.
observer
A pointer to an observer object. The observer argument is of type sc_fxnum_fast_observer*. See Chapter 11.24. The default value for observer is 0 (null pointer). The observer parameter cannot change after declaration.

Copy Constructor

sc_fix_fast(const sc_fix_fast&);

Operators

The operators defined for the sc_fix_fast are given in Table 18.

Table 18. Operators for sc_fix_fast

Operator class	Operators in class
Bitwise	~ & ^ |
Arithmetic	* / + - << >> ++ --
Equality	== !=
Relational	<<= >>=
Assignment	= *= /= += -= <<= >>= &= ^= |=

Note:

Operators << and operator >> define arithmetic shifts, not bitwise shifts. The difference is that no bits are lost and proper sign extension is done.

In expressions with the non-bitwise operators from Table 18, fixed-point types can be mixed with all types given:

```
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned&, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

The return type of any arithmetic operation is the fixed-point value type, which guarantees that the operation is performed without overflow or quantization.

A floating-point variable or a fixed-point value variable can contain one of the special values +Inf (plus infinity), -Inf (minus infinity), or Nan (not a number). Assignment of one of these special values to a fixed-point variable will produce a runtime error.

For the fixed-point types, a minimal set of bitwise operators is defined. These bitwise operators are only defined on either the signed fixed-point types or the unsigned fixed- point types. Mixing between signed and unsigned fixed-point types is not allowed. Mixing with any other type is also not allowed.

The semantics of the bitwise operators is as follows. For the unary ~ operator, the type of the result is the type of the operand. The bits in the two's complement mantissa of the operand are inverted to get the mantissa of the result. For the binary operators, the type of the result is the maximum aligned type of the two operands, that is, the two operands are aligned by the binary point and the maximum integer word length and the maximum fractional word length is taken. The operands are temporarily extended to this type before performing a bitwise and, bitwise exclusive-or, or bitwise or.

Member Functions

The functions defined for `sc_fix_fast` are given in Table 19.

Table 19. Functions for sc_fix_fast

Function class	**Functions in class**
`Bitwise`	`b_not, b_and, b_xor, b_or`
`Arithmetic`	`neg, mult, div, add, sub, lshift, rshift`

The functions in Table 19 have return type void. The first argument of these functions is a reference to the result object. The remaining arguments of these functions are the operands.

For the bitwise functions, the result object and the operands are of the same type, which is either `sc_fix_fast` or `sc_ufix_fast`.

The `neg` arithmetic function takes one operand, the other arithmetic functions take two operands. At least one of the operands of the arithmetic functions should have a fixed- point type, the other operand can have any of the types given:

```
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

The arithmetic functions are defined twice: once with the result object of type sc_fxval, and once with the result object of type sc_fix_fast.

Bit Selection

```
const sc_fxnum_bitref† operator [] ( int i) const;
sc_fxnum_bitref†       operator [] ( int i);

const sc_fxnum_bitref† bit( int i) const;
sc_fxnum_bitref†       bit( int i);
```

These functions take one argument of type int, which is the index into the fixed-point mantissa. The index argument must be between `wl-1` (MSB) and 0 (LSB). Otherwise, a runtime error is produced. The return type of the bit selection functions is (const or non- const) *sc_fxnum_bitref*†, which is a proxy class. The proxy class allows bit selection to be used both as `rvalue` (for reading) and `lvalue` (for writing). For bit selection, the fixed-point binary point is ignored.

Part Selection

```
const sc_fxnum_subref† operator () ( int, int ) const;
sc_fxnum_subref†       operator () ( int, int );

const sc_fxnum_subref† range( int, int ) const;
sc_fxnum_subref†       range( int, int );
```

These functions take two arguments of type `int`, which are the begin and end indices into the fixed-point mantissa. The index arguments must be between `wl-1` (MSB) and 0 (LSB). Otherwise, a runtime error is produced. The return type of the part selection functions is (const or non-const) *sc_fxnum_subref*†, which is a proxy class that behaves like type `sc_bv_base`. The proxy class allows part selection to be used both as `rvalue` (for reading) and `lvalue` (for writing). All operators and methods that are available for type `sc_bv_base` are also available for part selection. For part selection, the fixed-point binary point is ignored.

```
const sc_fxnum_subref† operator () () const;
sc_fxnum_subref†       operator () ();

const sc_fxnum_subref† range() const;
sc_fxnum_subref†       range();
```

As a shortcut for part selection of the complete mantissa, operator () and the range() method can be called without any arguments.

Query Parameters

```
const sc_fxcast_switch&
cast_switch() const;
```

Returns the cast switch parameter.

```
int
iwl() const;
```

Returns the integer word length parameter.

```
int
n_bits() const;
```

Returns the number of saturated bits parameter.

```
sc_o_mode
o_mode() const;
```

Returns the overflow mode parameter.

```
sc_q_mode
q_mode() const;
```

Return the quantization mode parameter.

```
const sc_fxtype_params&
type_params() const;
```

Returns the type parameters.

```
int
wl() const;
```

Returns the total word length parameter.

Query Value

```
bool
is_neg() const;
```

Returns true if the variable holds a negative value. Returns false otherwise.


```
bool
is_zero() const;
```
Returns true if the variable holds a zero value. Returns false otherwise.

```
bool
overflow_flag() const;
```
Returns true if the last write action on this variable caused overflow. Returns false otherwise.

```
bool
quantization_flag() const;
```
Returns true if the last write action on this variable caused quantization. Returns false otherwise.

```
const sc_fxval
value() const;
```
Returns the value.

Implicit Conversion

```
operator double() const;
```
Implicit conversion to the implementation type `double`. The value does not change.

Explicit Conversion

```
short           to_short() const;
unsigned short  to_ushort() const;
int             to_int() const;
unsigned int    to_uint() const;
long            to_long() const;
unsigned long   to_ulong() const;
float           to_float() const;
double          to_double() const
```

```
const sc_string to_string() const;
const sc_string to_string( sc_numrep ) const;
const sc_string to_string( sc_numrep, bool ) const;
const sc_string to_string( sc_fmt ) const;
const sc_string to_string( sc_numrep, sc_fmt ) const;
const sc_string to_string( sc_numrep, bool, sc_fmt ) const;
```
The value of a fixed-point variable can be converted to a character string with the to_string() method. This method takes different arguments for formatting purposes. See Chapter 6.8.8 for more information on converting fixed-point variables to/from character strings. Furthermore, writing to C++ output streams with operator << is supported, e.g. cout << a;, where a is a fixed-point variable. The decimal number representation is used in this case.

```
const sc_string to_dec() const;
```



```
const sc_string to_bin() const;
const sc_string to_oct() const;
const sc_string to_hex() const;
```

Shortcut methods for conversion to a character string. See Chapter 6.8.9.2.

Print or dump content

```
void
print( ostream& = cout ) const;
```

Print the `sc_fix_fast` instance value to an output stream.

```
void
scan( istream& = cin );
```

Read an `sc_fix_fast` value from an input stream.

```
void
dump( ostream& = cout )
const;
```

Prints the `sc_fix_fast` instance value, parameters and flags to an output stream.

```
ostream&
operator << ( ostream& os, const sc_fix_fast& a )
```

Print the instance value of `a` to an output stream `os`.

11.20 sc_fixed

Synopsis

```
template <int W, int I,
    sc_q_mode Q = SC_DEFAULT_Q_MODE_,
    sc_o_mode O = SC_DEFAULT_O_MODE_, int N =
    SC_DEFAULT_N_BITS_>
class sc_fixed : public sc_fix
{
public:
// constructors
    sc_fixed( sc_fxnum_observer* = 0 );
    sc_fixed( const sc_fxcast_switch&,
      sc_fxnum_observer* = 0 );

#define DECL_CTORS_T_A(tp) \
            sc_fixed( tp, sc_fxnum_observer* = 0 ); \
            sc_fixed( tp, const sc_fxcast_switch&,
            sc_fxnum_observer* = 0 );

#define DECL_CTORS_T_B(tp) \
            sc_fixed( tp, sc_fxnum_observer* = 0 ); \
            sc_fixed( tp, const sc_fxcast_switch&,
            sc_fxnum_observer* = 0 );

     DECL_CTORS_T_A(int)
     DECL_CTORS_T_A(unsigned int)
     DECL_CTORS_T_A(long)
     DECL_CTORS_T_A(unsigned long)
     DECL_CTORS_T_A(double)
     DECL_CTORS_T_A(const char*)
     DECL_CTORS_T_A(const sc_fxval&)
     DECL_CTORS_T_A(const sc_fxval_fast&)
     DECL_CTORS_T_A(const sc_fxnum&)
     DECL_CTORS_T_A(const sc_fxnum_fast&)
     DECL_CTORS_T_B(int64)
     DECL_CTORS_T_B(uint64)
     DECL_CTORS_T_B(const sc_int_base&)
     DECL_CTORS_T_B(const sc_uint_base&)
     DECL_CTORS_T_B(const sc_signed&)
     DECL_CTORS_T_B(const sc_unsigned&)
     sc_fixed( const sc_fixed<W,I,Q,O,N>& );

     // operators
     sc_fixed& operator = ( const sc_fixed<W,I,Q,O,N>& );

#define DECL_ASN_OP_T(op,tp) \
     sc_fixed& operator op ( tp );

#define DECL_ASN_OP_OTHER(op) \
     DECL_ASN_OP_T(op,int64) \
     DECL_ASN_OP_T(op,uint64) \
     DECL_ASN_OP_T(op,const sc_int_base&)\
```



```
    DECL_ASN_OP_T(op,const sc_uint_base&)\
    DECL_ASN_OP_T(op,const sc_signed&)\
    DECL_ASN_OP_T(op,const sc_unsigned&)

#define DECL_ASN_OP(op) \
    DECL_ASN_OP_T(op,int) \
    DECL_ASN_OP_T(op,unsigned int) \
    DECL_ASN_OP_T(op,long) \
    DECL_ASN_OP_T(op,unsigned long) \
    DECL_ASN_OP_T(op,double) \
    DECL_ASN_OP_T(op,const char*) \
    DECL_ASN_OP_T(op,const sc_fxval&) \
    DECL_ASN_OP_T(op,const sc_fxval_fast&) \
    DECL_ASN_OP_T(op,const sc_fxnum&) \
    DECL_ASN_OP_T(op,const sc_fxnum_fast&) \
    DECL_ASN_OP_OTHER(op)

    DECL_ASN_OP(=)
    DECL_ASN_OP(*=)
    DECL_ASN_OP(/=)
    DECL_ASN_OP(+=)
    DECL_ASN_OP(-=)
    DECL_ASN_OP_T(<<=,int)
    DECL_ASN_OP_T(>>=,int)
    DECL_ASN_OP_T(&=,const sc_fix&)
    DECL_ASN_OP_T(&=,const sc_fix_fast&)
    DECL_ASN_OP_T(|=,const sc_fix&)
    DECL_ASN_OP_T(|=,const sc_fix_fast&)
    DECL_ASN_OP_T(^=,const sc_fix&)
    DECL_ASN_OP_T(^=,const sc_fix_fast&)

    const sc_fxval operator ++ ( int );
    const sc_fxval operator -- ( int );
    sc_fixed& operator ++ ();
    sc_fixed& operator -- ();

};
```

Description

Templatized type **sc_fixed** is a signed (two's complement) type. The fixed-point type parameters wl, iwl, q_mode, o_mode, and n_bits are part of the type in sc_fixed. It is required that these parameters be constant expressions. See Chapter 6.8.1.

Declaration syntax

```
sc_fixed <wl,iwl[,q_mode[,o_mode[,n_bits]]]>
   var_name([init_val][,cast_switch])
   [,observer]);
```

wl

The total number of bits in the fixed-point format. The wl argument is of type int and must be greater than zero. Otherwise, a runtime error is produced. The wl argument must be a constant expression. The total word length parameter cannot change after declaration.

iwl

The number of integer bits in the fixed-point format. The iwl argument is of type int and can be positive or negative. See Chapter 6.8.1. The iwl argument must be a constant expression. The number of integer bits parameter cannot change after declaration.

q_mode

The quantization mode to use. The q_mode argument is of type sc_q_mode. Valid values for q_mode are given in Chapter 6.8.2.2 . The q_mode argument must be a constant expression. The default value for q_mode is obtained from the set of built-in default values. See Chapter 6.8.8. The quantization mode parameter cannot change after declaration.

o_mode

The overflow mode to use. The o_mode argument is of type sc_o_mode. Valid values for o_mode are given in Chapter 6.8.2.1 . The o_mode argument must be a constant expression. The default value for o_mode is obtained from the set of built-in default values. See Chapter 6.8.8. The overflow mode parameter cannot change after declaration.

n_bits

The number of saturated bits parameter for the selected overflow mode. The n_bits argument is of type int and must be greater than or equal to zero. Otherwise, a runtime error is produced. The n_bits argument must be a constant expression. If the overflow mode is specified, the default value is zero. If the overflow mode is not specified, the default value is obtained from the set of built-in default values. See Chapter 6.8.8. The number of saturated bits parameter cannot change after declaration.

Examples

```
sc_fixed<32,32> a;
sc_fixed<8,1,SC_RND> c(b);
```

Public Constructor

```
explicit sc_fixed ([type_ init_val]
    [, const sc_fxcast_switch& cast_switch]
    [, sc_fxnum_observer* observer]);
```

```
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```


Notes on type_
For all types in type_ , except sc_[u]fix and sc_[u]fix_fast, only the value of the argument is taken, that is, any type information is discarded. This ensures that initialization during declaration and initialization after declaration behave identical.
A fixed-point variable can be initialized with a C/C++ character string (type const char*) either when the number will be expressed in binary form or when the number is too large to be written as a C/C++ built-in type literal

init_val
The initial value of the variable. If the initial value is not specified, the instance is uninitialized.
cast_switch
The cast switch, which allows to switch fixed-point type casting on or off. Valid values for cast_switch are:
SC_OFF for casting off
SC_ON for casting on
The default value for cast_switch is obtained from the fixed-point context type sc_fxcast_context. The cast_switch parameter cannot change after declaration.
observer
A pointer to an observer object. The observer argument is of type sc_fxnum_observer*. See Chapter 11.25. The default value for observer is 0 (null pointer). The observer parameter cannot change after declaration.

Copy Constructor

```
sc_fixed( const sc_fixed<W,I,Q,O,N>& );
```

Operators
The operators defined for the sc_fixed are given in Table 20.

Table 20. Operators for sc_fixed

Operator class	Operators in class
Bitwise	~ & ^ |
Arithmetic	* / + - << >> ++ --
Equality	== !=
Relational	<<= >>=
Assignment	= *= /= += -= <<= >>= &= ^= |=

Note:
Operator << and operator >> define arithmetic shifts, not bitwise shifts. The difference is that no bits are lost and proper sign extension is done.

In expressions with the non-bitwise operators from Table 20, fixed-point types can be mixed with all types given:

type_ in {short, unsigned short, int, unsigned int, long, unsigned long, float, double, const char*, int64, uint64, const *sc_int_base*† &, const *sc_uint_base*† &, const sc_signed&, const sc_unsigned, const sc_fxval&, const sc_fxval_fast&, const sc_[u]fix&, const sc_[u]fix_fast& }

The return type of any arithmetic operation is the fixed-point value type, which guarantees that the operation is performed without overflow or quantization.

A floating-point variable or a fixed-point value variable can contain one of the special values +Inf (plus infinity), -Inf (minus infinity), or Nan (not a number). Assignment of one of these special values to a fixed-point variable will produce a runtime error.

For the fixed-point types, a minimal set of bitwise operators is defined. These bitwise operators are only defined on either the signed fixed-point types or the unsigned fixed- point types. Mixing between signed and unsigned fixed-point types is not allowed. Mixing with any other type is also not allowed.

The semantics of the bitwise operators is as follows. For the unary ~ operator, the type of the result is the type of the operand. The bits in the two's complement mantissa of the operand are inverted to get the mantissa of the result. For the binary operators, the type of the result is the maximum aligned type of the two operands, that is, the two operands are aligned by the binary point and the maximum integer word length and the maximum fractional word length is taken. The operands are temporarily extended to this type before performing a bitwise and, bitwise exclusive-or, or bitwise or.

Member Functions

The functions defined for sc_fixed are given in Table 21.

Table 21. Functions for sc_fixed

Function class	**Functions in class**
Bitwise	b_not, b_and, b_xor, b_or
Arithmetic	neg, mult, div, add, sub, lshift, rshift

The functions in Table 21 have return type void. The first argument of these functions is a reference to the result object. The remaining arguments of these functions are the operands.

For the bitwise functions, the type of the result is sc_fixed, and the type of the operands are either both sc_fixed or a mix of sc_fixed and sc_fixed_fast.

The neg arithmetic function takes one operand, the other arithmetic functions take two operands. At least one of the operands of the arithmetic functions should have a fixed- point type, the other operand can have any of the types given:

```
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

The arithmetic functions are defined twice: once with the result object of type sc_fxval, and once with the result object of type sc_fixed or sc_ufixed.

Bit Selection

```
const sc_fxnum_bitref† operator [] ( int i) const;
sc_fxnum_bitref†       operator [] ( int i);

const sc_fxnum_bitref† bit( int i) const;
sc_fxnum_bitref†       bit( int i);
```

These functions take one argument of type int, which is the index into the fixed-point mantissa. The index argument must be between wl-1 (MSB) and 0 (LSB). Otherwise, a runtime error is produced. The return type of the bit selection functions is (const or non- const) *sc_fxnum_bitref*†, which is a proxy class. The proxy class allows bit selection to be used both as rvalue (for reading) and lvalue (for writing). For bit selection, the fixed-point binary point is ignored.

Part Selection

```
const sc_fxnum_subref† operator () ( int, int ) const;
sc_fxnum_subref†       operator () ( int, int );

const sc_fxnum_subref† range( int, int ) const;
sc_fxnum_subref†       range( int, int );
```

These functions take two arguments of type int, which are the begin and end indices into the fixed-point mantissa. The index arguments must be between wl-1 (MSB) and 0 (LSB). Otherwise, a runtime error is produced. The return type of the part selection functions is (const or non-const) *sc_fxnum_subref*†, which is a proxy class that behaves like type sc_bv_base. The proxy class allows part selection to be used both as rvalue (for reading) and lvalue (for writing). All operators and methods that are available for type sc_bv_base are also available for part selection. For part selection, the fixed-point binary point is ignored.

```
const sc_fxnum_subref† operator () () const;
sc_fxnum_subref†       operator () ();

const sc_fxnum_subref† range() const;
sc_fxnum_subref†       range();
```

As a shortcut for part selection of the complete mantissa, operator () and the range() method can be called without any arguments.

Query Parameters

`const sc_fxcast_switch&`
`cast_switch() const;`

Returns the cast switch parameter.

`int`
`iwl``() const;`

Returns the integer word length parameter.

`int`
`n_bits``() const;`

Returns the number of saturated bits parameter.

`sc_o_mode`
`o_mode``() const;`

Returns the overflow mode parameter.

`sc_q_mode`
`q_mode``() const;`

Return the quantization mode parameter.

`const sc_fxtype_params&`
`type_params``() const;`

Returns the type parameters.

`int`
`wl``() const;`

Returns the total word length parameter.

Query Value

`bool`
`is_neg``() const;`

Returns true if the variable holds a negative value. Returns false otherwise.

`bool`
`is_zero``() const;`

Returns true if the variable holds a zero value. Returns false otherwise.

```
bool
overflow_flag() const;
```

Returns true if the last write action on this variable caused overflow. Returns false otherwise.

```
bool
quantization_flag() const;
```

Returns true if the last write action on this variable caused quantization. Returns false otherwise.

```
const sc_fxval
value() const;
```

Returns the value.

Implicit Conversion

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

Implicit conversion to the implementation type `double`. The value does not change, if the wordlength of the `sc_fixed` is less than or equal to 53 bits.

Explicit Conversion

```
short            to_short() const;
unsigned short   to_ushort() const;
int              to_int() const;
unsigned int     to_uint() const;
long             to_long() const;
unsigned long    to_ulong() const;
float            to_float() const;
double           to_double() const
```

```
const sc_string to_string() const;
const sc_string to_string( sc_numrep ) const;
const sc_string to_string( sc_numrep, bool ) const;
const sc_string to_string( sc_fmt ) const;
const sc_string to_string( sc_numrep, sc_fmt ) const;
const sc_string to_string( sc_numrep, bool, sc_fmt ) const;
```

The value of a fixed-point variable can be converted to a character string with the to_string() method. This method takes different arguments for formatting purposes. See Chapter 6.8.8 for more information on converting fixed-point variables to/from character strings. Furthermore, writing to C++ output streams with operator << is supported, e.g. cout << a;, where a is a fixed-point variable. The decimal number representation is used in this case.

```
const sc_string to_dec() const;
const sc_string to_bin() const;
const sc_string to_oct() const;
const sc_string to_hex() const;
```

Shortcut methods for conversion to a character string. See Chapter 6.8.9.2.

Print or dump content

```
void
print( ostream& = cout ) const;
```

Print the `sc_fixed` instance value to an output stream.

```
void
scan( istream& = cin );
```

Read an `sc_fixed` value from an input stream.

```
void
dump( ostream& = cout )
const;
```

Prints the `sc_fixed` instance value, parameters and flags to an output stream.

```
ostream&
operator << ( ostream& os, const sc_fixed& a )
```

Print the instance value of `a` to an output stream `os`.

11.21 sc_fixed_fast

Synopsis

```
template <int W, int I,
    sc_q_mode Q = SC_DEFAULT_Q_MODE_,
    sc_o_mode O = SC_DEFAULT_O_MODE_, int N =
      SC_DEFAULT_N_BITS_>
class sc_fixed_fast : public sc_fix_fast
{
public:
    // constructors

    sc_fixed_fast( sc_fxnum_fast_observer* = 0 );
    sc_fixed_fast( const sc_fxcast_switch&,
      sc_fxnum_fast_observer* = 0 );

#define DECL_CTORS_T_A(tp) \
    sc_fixed_fast( tp, sc_fxnum_fast_observer* = 0 ); \
    sc_fixed_fast( tp, const sc_fxcast_switch&, \
      sc_fxnum_fast_observer* = 0 );

#define DECL_CTORS_T_B(tp) \
    sc_fixed_fast( tp, sc_fxnum_fast_observer* = 0 ); \
    sc_fixed_fast( tp, const sc_fxcast_switch&, \
      sc_fxnum_fast_observer* = 0 );

     DECL_CTORS_T_A(int)
     DECL_CTORS_T_A(unsigned int)
     DECL_CTORS_T_A(long)
     DECL_CTORS_T_A(unsigned long)
     DECL_CTORS_T_A(double)
     DECL_CTORS_T_A(const char*)
     DECL_CTORS_T_A(const sc_fxval&)
     DECL_CTORS_T_A(const sc_fxval_fast&)
     DECL_CTORS_T_A(const sc_fxnum&)
     DECL_CTORS_T_A(const sc_fxnum_fast&)
     DECL_CTORS_T_B(int64)
     DECL_CTORS_T_B(uint64)
     DECL_CTORS_T_B(const sc_int_base&)
     DECL_CTORS_T_B(const sc_uint_base&)
     DECL_CTORS_T_B(const sc_signed&)
     DECL_CTORS_T_B(const sc_unsigned&)

     sc_fixed_fast( const sc_fixed_fast<W,I,Q,O,N>& );

     // operators
    sc_fixed_fast& operator = ( const
      sc_fixed_fast<W,I,Q,O,N>& );

#define DECL_ASN_OP_T(op,tp) \
    sc_fixed_fast& operator op ( tp );

#define DECL_ASN_OP_OTHER(op) \
```



```
    DECL_ASN_OP_T(op,int64) \
    DECL_ASN_OP_T(op,uint64) \
    DECL_ASN_OP_T(op,const sc_int_base&) \
    DECL_ASN_OP_T(op,const sc_uint_base&) \
    DECL_ASN_OP_T(op,const sc_signed&) \
    DECL_ASN_OP_T(op,const sc_unsigned&)

#define DECL_ASN_OP(op) \
    DECL_ASN_OP_T(op,int) \
    DECL_ASN_OP_T(op,unsigned int) \
    DECL_ASN_OP_T(op,long) \
    DECL_ASN_OP_T(op,unsigned long) \
    DECL_ASN_OP_T(op,double) \
    DECL_ASN_OP_T(op,const char*) \
    DECL_ASN_OP_T(op,const sc_fxval&) \
    DECL_ASN_OP_T(op,const sc_fxval_fast&) \
    DECL_ASN_OP_T(op,const sc_fxnum&) \
    DECL_ASN_OP_T(op,const sc_fxnum_fast&) \
    DECL_ASN_OP_OTHER(op)

    DECL_ASN_OP(=)
    DECL_ASN_OP(*=)
    DECL_ASN_OP(/=)
    DECL_ASN_OP(+=)
    DECL_ASN_OP(-=)
    DECL_ASN_OP_T(<<=,int)
    DECL_ASN_OP_T(>>=,int)
    DECL_ASN_OP_T(&=,const sc_fix&)
    DECL_ASN_OP_T(&=,const sc_fix_fast&)
    DECL_ASN_OP_T(|=,const sc_fix&)
    DECL_ASN_OP_T(|=,const sc_fix_fast&)
    DECL_ASN_OP_T(^=,const sc_fix&)
    DECL_ASN_OP_T(^=,const sc_fix_fast&)

    const sc_fxval_fast operator ++ ( int );
    const sc_fxval_fast operator -- ( int );
    sc_fixed_fast& operator ++ ();
    sc_fixed_fast& operator -- ();

};
```

Description

Templatized type **`sc_fixed_fast`** is a signed (two's complement) type. The fixed-point type parameters wl, iwl, q_mode, o_mode, and n_bits are part of the type in `sc_fixed_fast`. It is required that these parameters be constant expressions. See Chapter 6.8.1.

`sc_fixed_fast` provides the same API as `sc_fixed`.

`sc_fixed_fast` uses double precision (floating-point) values. The mantissa of a double precision value is limited to 53 bits. This means that bit-true behavior

cannot be guaranteed with the limited precision types. For bit-true behavior with the limited precision types, the following guidelines should be followed:
Make sure that the word length of the result of any operation or expression does not exceed 53 bits.
The result of an addition or subtraction requires a word length that is one bit more than the maximum *aligned* word length of the two operands.
The result of a multiplication requires a word length that is the sum of the word lengths of the two operands.

Declaration syntax

```
sc_fixed_fast <wl,iwl[,q_mode[,o_mode[,n_bits]]]>
    var_name([init_val][,cast_switch])
    [,observer]);
```

wl
The total number of bits in the fixed-point format. The wl argument is of type int and must be greater than zero. Otherwise, a runtime error is produced. The wl argument must be a constant expression. The total word length parameter cannot change after declaration.
iwl
The number of integer bits in the fixed-point format. The iwl argument is of type int and can be positive or negative. See Chapter 6.8.1. The iwl argument must be a constant expression. The number of integer bits parameter cannot change after declaration.
q_mode
The quantization mode to use. The q_mode argument is of type sc_q_mode. Valid values for q_mode are given in Chapter 6.8.2.2 . The q_mode argument must be a constant expression. The default value for q_mode is obtained from the set of built-in default values. See Chapter 6.8.8. The quantization mode parameter cannot change after declaration.
o_mode
The overflow mode to use. The o_mode argument is of type sc_o_mode. Valid values for o_mode are given in Chapter 6.8.2.1 . The o_mode argument must be a constant expression. The default value for o_mode is obtained from the set of built-in default values. See Chapter 6.8.8. The overflow mode parameter cannot change after declaration.
n_bits
The number of saturated bits parameter for the selected overflow mode. The n_bits argument is of type int and must be greater than or equal to zero. Otherwise, a runtime error is produced. The n_bits argument must be a constant expression. If the overflow mode is specified, the default value is zero. If the overflow mode is not specified, the default value is obtained from the set of built-in default values. See Chapter 6.8.8. The number of saturated bits parameter cannot change after declaration.

Examples


```
sc_fixed_fast<32,32> a;
sc_fixed_fast<8,1,SC_RND> c(b);
sc_fixed_fast<8,8> c = “0.1”;
sc_fixed_fast<8,8> d = 1;
sc_ufixed<16,8> e = 2;
sc_fixed_fast<16,16> f = d + e;
d *= 2;
```

Public Constructor

```
explicit sc_fixed_fast ([type_ init_val]
    [, const sc_fxcast_switch& cast_switch]
    [, sc_fxnum_fast_observer* observer]);
```

```
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

Notes on `type_`

For all types in `type_` , except `sc_[u]fix` and `sc_[u]fix_fast`, only the value of the argument is taken, that is, any type information is discarded. This ensures that initialization during declaration and initialization after declaration behave identical.

A fixed-point variable can be initialized with a C/C++ character string (type const char*) either when the number will be expressed in binary form or when the number is too large to be written as a C/C++ built-in type literal

init_val

The initial value of the variable. If the initial value is not specified, the instance is uninitialized.

cast_switch

The cast switch, which allows to switch fixed-point type casting on or off. Valid values for `cast_switch` are:

`SC_OFF` for casting off

`SC_ON` for casting on

The default value for `cast_switch` is obtained from the fixed-point context type `sc_fxcast_context`. The `cast_switch` parameter cannot change after declaration.

observer

A pointer to an observer object. The `observer` argument is of type `sc_fxnum_fast_observer*`. See Chapter 11.24. The default value for `observer` is 0 (null pointer). The `observer` parameter cannot change after declaration.

Copy Constructor


```
sc_fixed_fast( const sc_fixed_fast<W,I,Q,O,N>& );
```

Operators

The operators defined for the `sc_fixed_fast` are given in Table 22.

Table 22. Operators for sc_fixed_fast

Operator class	Operators in class
Bitwise	~ & ^ |
Arithmetic	* / + - << >> ++ --
Equality	== !=
Relational	<<= >>=
Assignment	= *= /= += -= <<= >>= &= ^= |=

Note:
Operator << and operator >> define arithmetic shifts, not bitwise shifts. The difference is that no bits are lost and proper sign extension is done.

In expressions with the non-bitwise operators from Table 22, fixed-point types can be mixed with all types given:

```
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

The return type of any arithmetic operation is the fixed-point value type, which guarantees that the operation is performed without overflow or quantization.

A floating-point variable or a fixed-point value variable can contain one of the special values +Inf (plus infinity), -Inf (minus infinity), or Nan (not a number). Assignment of one of these special values to a fixed-point variable will produce a runtime error.

For the fixed-point types, a minimal set of bitwise operators is defined. These bitwise operators are only defined on either the signed fixed-point types or the unsigned fixed- point types. Mixing between signed and unsigned fixed-point types is not allowed. Mixing with any other type is also not allowed.

The semantics of the bitwise operators is as follows. For the unary ~ operator, the type of the result is the type of the operand. The bits in the two's complement mantissa of the operand are inverted to get the mantissa of the result. For the binary operators, the type of the result is the maximum aligned type of the two operands, that is, the two operands are aligned by the binary point and the maximum integer word length and the maximum fractional word length is taken. The operands are temporarily extended to this type before performing a bitwise and, bitwise exclusive-or, or bitwise or.

Member Functions

The functions defined for `sc_fixed_fast` are given in Table 23.

Table 23. Functions for sc_fixed_fast

Function class	Functions in class
Bitwise	b_not, b_and, b_xor, b_or
Arithmetic	neg, mult, div, add, sub, lshift, rshift

The functions in Table 23 have return type void. The first argument of these functions is a reference to the result object. The remaining arguments of these functions are the operands.

For the bitwise functions, the type of the result is `sc_fixed_fast` and the type of the operands are either both `sc_fixed_fast` or a mix of `sc_fixed` and `sc_fixed_fast`.

The `neg` arithmetic function takes one operand, the other arithmetic functions take two operands. At least one of the operands of the arithmetic functions should have a fixed- point type, the other operand can have any of the types given:

```
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

The arithmetic functions are defined twice: once with the result object of type sc_fxval, and once with the result object of type sc_fixed_fast or sc_ufixed_fast.

Bit Selection

```
const sc_fxnum_bitref† operator [] ( int i) const;
sc_fxnum_bitref†       operator [] ( int i);

const sc_fxnum_bitref† bit( int i) const;
sc_fxnum_bitref†       bit( int i);
```

These functions take one argument of type int, which is the index into the fixed-point mantissa. The index argument must be between `wl-1` (MSB) and 0 (LSB). Otherwise, a runtime error is produced. The return type of the bit selection functions is (const or non- const) *sc_fxnum_bitref*†, which is a proxy class. The proxy class allows bit selection to be used both as `rvalue` (for reading) and `lvalue` (for writing). For bit selection, the fixed-point binary point is ignored.

Part Selection

```
const sc_fxnum_subref† operator () ( int, int ) const;
sc_fxnum_subref†       operator () ( int, int );

const sc_fxnum_subref† range( int, int ) const;
sc_fxnum_subref†       range( int, int );
```

These functions take two arguments of type `int`, which are the begin and end indices into the fixed-point mantissa. The index arguments must be between `wl-1` (MSB) and 0 (LSB). Otherwise, a runtime error is produced. The return type of the part selection functions is (const or non-const) *sc_fxnum_subref*†, which is a proxy class that behaves like type `sc_bv_base`. The proxy class allows part selection to be used both as `rvalue` (for reading) and `lvalue` (for writing). All operators and methods that are available for type `sc_bv_base` are also available for part selection. For part selection, the fixed-point binary point is ignored.

```
const sc_fxnum_subref† operator () () const;
sc_fxnum_subref†       operator () ();

const sc_fxnum_subref† range() const;
sc_fxnum_subref†       range();
```

As a shortcut for part selection of the complete mantissa, operator () and the range() method can be called without any arguments.

Query Parameters

```
const sc_fxcast_switch&
cast_switch() const;
```

Returns the cast switch parameter.

```
int
iwl() const;
```

Returns the integer word length parameter.

```
int
n_bits() const;
```

Returns the number of saturated bits parameter.

```
sc_o_mode
```



```
o_mode() const;
```
Returns the overflow mode parameter.

```
sc_q_mode
q_mode() const;
```
Return the quantization mode parameter.

```
const sc_fxtype_params&
type_params() const;
```
Returns the type parameters.

```
int
wl() const;
```
Returns the total word length parameter.

Query Value

```
bool
is_neg() const;
```
Returns true if the variable holds a negative value. Returns false otherwise.

```
bool
is_zero() const;
```
Returns true if the variable holds a zero value. Returns false otherwise.

```
bool
overflow_flag() const;
```
Returns true if the last write action on this variable caused overflow. Returns false otherwise.

```
bool
quantization_flag() const;
```
Returns true if the last write action on this variable caused quantization. Returns false otherwise.

```
const sc_fxval
value() const;
```
Returns the value.

Implicit Conversion

```
operator double() const;
```
Implicit conversion to the implementation type `double`. The value does not change, if the wordlength of the `sc_fixed_fast` is less than or equal to 53 bits.

Explicit Conversion

```
short            to_short() const;
unsigned short   to_ushort() const;
int              to_int() const;
unsigned int     to_uint() const;
```



```
long            to_long() const;
unsigned long   to_ulong() const;
float           to_float() const;
double          to_double() const

const sc_string to_string() const;
const sc_string to_string( sc_numrep ) const;
const sc_string to_string( sc_numrep, bool ) const;
const sc_string to_string( sc_fmt ) const;
const sc_string to_string( sc_numrep, sc_fmt ) const;
const sc_string to_string( sc_numrep, bool, sc_fmt ) const;
```

The value of a fixed-point variable can be converted to a character string with the to_string() method. This method takes different arguments for formatting purposes. See Chapter 6.8.8 for more information on converting fixed-point variables to/from character strings. Furthermore, writing to C++ output streams with operator << is supported, e.g. cout << a;, where a is a fixed-point variable. The decimal number representation is used in this case.

```
const sc_string to_dec() const;
const sc_string to_bin() const;
const sc_string to_oct() const;
const sc_string to_hex() const;
```

Shortcut methods for conversion to a character string. See Chapter 6.8.9.2.

Print or dump content

```
void
print( ostream& = cout ) const;
```

Print the `sc_fixed_fast` instance value to an output stream.

```
void
scan( istream& = cin );
```

Read an `sc_fixed_fast` value from an input stream.

```
void
dump( ostream& = cout )
const;
```

Prints the `sc_fixed_fast` instance value, parameters and flags to an output stream.

```
ostream&
operator << ( ostream& os, const sc_fixed_fast& a )
```

Print the instance value of a to an output stream os.

11.22 sc_fxcast_context

Synopsis

```
template <class sc_fxcast_switch>
class sc_context
{
public:
// constructors and destructor
    sc_context( const sc_fxcast_switch&,
      sc_context_begin = SC_NOW );
    ~sc_context();

// methods
    void begin();
    void end();
    static const sc_fxcast_switch& default_value();
    const sc_fxcast_switch& value() const;

// disabled
private:
    sc_context( const sc_context<sc_fxcast_switch>& );
    void* operator new( size_t );
};

typedef sc_context<sc_fxcast_switch> sc_fxcast_context;
```

Description

sc_fxcast_context instance is used to set a new default value for the fixed-point cast switch cast_switch. This new default value affects the behavior of fixed-point types sc_fixed, sc_ufixed, sc_fix, sc_ufix, sc_fixed_fast, sc_ufixed_fast, sc_fix_fast, and sc_ufix_fast. When declaring a variable of any of these types without specifying the cast_switch argument, it is obtained from the current default value.

Examples

```
sc_fxcast_context no_casting(SC_OFF,SC_LATER);
...
{
  ...
  no_casting.begin();
  sc_fix a;  // no casting
  no_casting.end();
  sc_fix b;  // casting
}
```

Public Constructor

```
sc_fxcast_context (
    sc_fxcast_switch cast_switch
    [,sc_context_begin context_begin]);
```


cast_switch
A cast switch object, which contains the new default value.
context_begin
A context begin object. Valid values for context_begin are:
SC_NOW (set new default value now)
SC_LATER (set new default value later)
The default value for context_begin is SC_NOW, which means to set the new default value during declaration of the fixed-point context variable.

Public Member Functions

```
void
begin();
```

Sets the default fixed-point cast switch value to the value specified when declaring a `sc_fxcast_context` variable *var_name*. The old default fixed-point cast switch value is stored. The `begin()` method can be called either after *var_name* has been declared with the context_begin argument set to SC_LATER, or after calling the `end()` method on *var_name*. Otherwise, a runtime error is produced.

```
static const T&
default_value();
```

Returns the default fixed-point cast switch value.

```
void
end();
```

Restores the old default fixed-point cast switch value. The end method can be called either after the `sc_fxcast_context` variable var_name has been declared with the context_begin argument set to SC_NOW (or not specified at all), or after calling the `begin()` method on var_name. Otherwise, a runtime error is produced.

```
const T&
value() const;
```

Returns the fixed-point cast switch value specified with the instance.

Disabled Member Functions

```
sc_context( const sc_context<sc_fxcast_switch>& );
void* operator new( size_t );
```


11.23 sc_fxcast_switch

Synopsis

```
class sc_fxcast_switch
{
public:
// constructors
    sc_fxcast_switch();
    sc_fxcast_switch( sc_switch );
    sc_fxcast_switch( const sc_fxcast_switch& );
    sc_fxcast_switch( sc_without_context );

// operators
    sc_fxcast_switch& operator = ( const
      sc_fxcast_switch& );
    friend bool operator == ( const sc_fxcast_switch&,
      const sc_fxcast_switch& );
    friend bool operator != ( const sc_fxcast_switch&,
      const sc_fxcast_switch& );

// methods
    const sc_string to_string() const;
    void print( ostream& = cout ) const;
    void dump( ostream& = cout ) const;
};
```

Description

sc_fxcast_switch variable is used to configure the type parameters of a variable of fixed-point type sc_fix and sc_ufix (and the corresponding limited precision types).

A sc_fxcast_switch variable can be initialized with another sc_fxcast_switch variable. Variables of this type can also be used in assignment to a sc_fxcast_switch variable.

Examples

```
sc_fxcast_switch my_casting(SC_OFF);
sc_fixed<12,4> a(my_casting);
```

Public Constructors

```
sc_fxcast_switch [(sc_switch cast_switch)];
```

cast_switch
The cast switch value. The cast_switch argument is of type sc_switch. Valid values for cast_switch are:
SC_OFF for casting off
SC_ON   for casting on

The default value for cast_switch is obtained from the fixed-point context type `sc_fxcast_context`.

Public Member Functions

```
void
print( ostream& = cout ) const;
```

Print the `sc_fxcast_switch` instance value to an output stream.

```
void
dump( ostream& = cout ) const;
```

Print the `sc_fxcast_switch` instance value to an output stream.

Explicit Conversion

```
const sc_string
to_string() const;
```

The value of the `sc_fxcast_switch` value is converted to a character string

Operators

```
sc_fxcast_switch&
operator = ( const sc_fxtype_params& cast_switch );
```

`cast_switch` is assigned to the left hand side.

```
friend bool
operator == (const sc_fxcast_switch& switch_a , const
   sc_fxcast_switch& switch_b) ;
```

Returns true if `switch_a` is equal to `switch_b` else false.

```
friend bool
operator != ( const sc_fxcast_switch& switch_a ,
   const sc_fxcast_switch& switch_b );
```

Returns true if `switch_a` is not equal to `switch_b` else false.

```
ostream&
operator << ( ostream& os, const sc_fxcast_switch& a )
```

Print the instance value of `a` to an output stream `os`.

11.24 sc_fxnum_fast_observer

Synopsis

```
class sc_fxnum_fast_observer
{
protected:
     sc_fxnum_fast_observer() {}
     virtual ~sc_fxnum_fast_observer() {}
public:
    // methods
    virtual void  construct( const sc_fxnum_fast& );
    virtual void  destruct( const sc_fxnum_fast& );
    virtual void  read( const sc_fxnum_fast& );
    virtual void  write( const sc_fxnum_fast& );
    static sc_fxnum_fast_observer* (*default_observer)();
};
```

Description

sc_fxnum_fast_observer is an abstract base class provided as a hook to define one's own observer functionality.

Public Methods

```
virtual void construct( const sc_fxnum_fast& );
virtual void  destruct( const sc_fxnum_fast& );
virtual void      read( const sc_fxnum_fast& );
virtual void     write( const sc_fxnum_fast& );
```

These methods allow to observe construction, destruction, read, and write actions on a particular variable. The destruct and read methods are called before the action takes place, while the construct and write methods are called after the action has taken place. Each of these methods can query the variable under observation, which is passed as the single argument to the methods.

The default behavior of the methods is to do nothing (and return).

11.25 sc_fxnum_observer

Synopsis

```
class sc_fxnum_observer
{
protected:
    sc_fxnum_observer() {}
    virtual ~sc_fxnum_observer() {}
public:
    // methods
    virtual void  construct( const sc_fxnum& );
    virtual void  destruct( const sc_fxnum& );
    virtual void  read( const sc_fxnum& );
    virtual void  write( const sc_fxnum& );
    static sc_fxnum_observer* (*default_observer) ();
};
```

Description

sc_fxnum_observer is an abstract base class provided as a hook to define one's own observer functionality.

Public Methods

```
virtual void construct( const sc_fxnum & );
virtual void  destruct( const sc_fxnum & );
virtual void      read( const sc_fxnum & );
virtual void     write( const sc_fxnum & );
```

These methods allow to observe construction, destruction, read, and write actions on a particular variable. The destruct and read methods are called before the action takes place, while the construct and write methods are called after the action has taken place. Each of these methods can query the variable under observation, which is passed as the single argument to the methods.

The default behavior of the methods is to do nothing (and return).

11.26 sc_fxtype_context

Synopsis

```
template <class sc_fxtype_params>
class sc_context
{
public:
// constructors and destructor
    sc_context( const sc_fxtype_params&,
      sc_context_begin = SC_NOW );
    ~sc_context();

// methods
    void begin();
    void end();
    static const sc_fxtype_params& default_value();
    const sc_fxtype_params& value() const;

// disabled
    sc_context( const sc_context< sc_fxtype_params >& );
    void* operator new( size_t );
};

typedef sc_context<sc_fxtype_params> sc_fxtype_context;
```

Description

`sc_fxtype_context` variable is used to set new default values for the fixed-point type parameters wl, iwl, q_mode, o_mode, and n_bits. These new default values affect the behavior of fixed-point types `sc_fix`, `sc_ufix`, `sc_fix_fast`, and `sc_ufix_fast`. When declaring a variable of these types, any type parameter that is missing as argument is obtained from the current default values.

Examples

```
sc_fxtype_params p1(16,16,SC_TRN,SC_WRAP);
sc_fxtype_params p2(16,1,SC_RND_CONV,SC_SAT);
...
{
  sc_fxtype_context c1(p1);
  sc_fxtype_context c2(p2,SC_LATER);
  ...
  sc_fix a;  // uses p1
  c2.begin();
  sc_fix b;  // uses p2
  c2.end();
  sc_fix c;  // uses p1
}
```

Public Constructor

```
sc_fxtype_context (
```



```
sc_fxtype_params type_params
[,sc_context_begin context_begin]);
```

type_params
A fixed-point type parameters object, which contains the new default values. The *type_params* argument is of type sc_fxtype_params.
context_begin
A context begin object. The optional *context_begin* argument is of type `sc_context_begin`. Valid values for *context_begin* are:
SC_NOW (set new default values now)
SC_LATER (set new default values later)
The default value for *context_begin* is SC_NOW, which means to set the new default values during declaration of the fixed-point context variable.

Public Member Functions

`void`
`begin``();`
Sets the default fixed-point type values to the values specified when declaring the `sc_fxtype_context` instance *var_name* . The old default fixed-point type values are stored. The `begin()` method can be called either after *var_name* has been declared with the *`context_begin`* argument set to SC_LATER, or after calling the `end()` method on *var_name*. Otherwise, a runtime error is produced.

`static const T&`
`default_value``();`
Returns the default fixed-point type values.

`void`
`end``();`
Restores the old default fixed-point type values. The `end()` method can be called either after *the* `sc_fxcast_context` *instance* var_name has been declared with the *`context_begin`* argument set to SC_NOW (or not specified at all), or after calling `begin()` method on var_name. Otherwise, a runtime error is produced.

`const T&`
`value``() const;`
Returns the fixed-point type values specified with the instance.

11.27 sc_fxtype_params

Synopsis

```
class sc_fxtype_params
{
public:
// constructors and destructor
    sc_fxtype_params();
    sc_fxtype_params( int, int );
    sc_fxtype_params( sc_q_mode, sc_o_mode, int = 0 );
    sc_fxtype_params( int, int, sc_q_mode, sc_o_mode,
      int = 0 );
    sc_fxtype_params( const sc_fxtype_params& );
    sc_fxtype_params( const sc_fxtype_params&,
      int, int );
    sc_fxtype_params( const sc_fxtype_params&,
      sc_q_mode, sc_o_mode, int = 0 );
    sc_fxtype_params( sc_without_context );

// operators
    sc_fxtype_params& operator = ( const
      sc_fxtype_params& );
    friend bool operator == ( const sc_fxtype_params&,
      const sc_fxtype_params& );
    friend bool operator != ( const sc_fxtype_params&,
      const sc_fxtype_params& );

// methods
    int wl() const;
    void wl( int );
    int iwl() const;
    void iwl( int );
    sc_q_mode q_mode() const;
    void q_mode( sc_q_mode );
    sc_o_mode o_mode() const;
    void o_mode( sc_o_mode );
    int n_bits() const;
    void n_bits( int );
    const sc_string to_string() const;
    void print( ostream& = cout ) const;
    void dump( ostream& = cout ) const;
};
```

Description

`sc_fxtype_params` variable is used to configure the type parameters of a variable of fixed-point type `sc_fix` and `sc_ufix` (and the corresponding limited precision types).

An sc_fxtype_params variable can be initialized with another sc_fxtype_params variable. Variables of this type can also be used in assignment to an sc_fxtype_params variable.

Public Constructors

```
sc_fxtype_params ([int wl,int iwl]
   [,sc_q_mode q_mode,sc_o_mode o_mode[,int n_bits]] ) ;
```

wl
The total number of bits in the fixed-point format. `wl` must be greater than zero, otherwise, a runtime error is produced. The default value for `wl` is obtained from the fixed-point context type `sc_fxtype_context`. See Chapter 11.26.
iwl
The number of integer bits in the fixed-point format. `iwl` can be positive or negative. The default value for `iwl` is obtained from the fixed-point context type `sc_fxtype_context`. See See Chapter 11.26.
q_mode
The quantization mode to use. Valid values for `q_mode` are given in Chapter 6.8.12.7. The default value for `q_mode` is obtained from the fixed-point context type `sc_fxtype_context`. See See Chapter 11.26
o_mode
The overflow mode to use. Valid values for `o_mode` are given in Chapter 6.8.12.1. The default value for `o_mode` is obtained from the fixed-point context type `sc_fxtype_context`. See Chapter 11.26.
n_bits
The number of saturated bits parameter for the selected overflow mode. `n_bits` must be greater than or equal to zero, otherwise a runtime error is produced. If the overflow mode is specified, the default value is zero. If the overflow mode is not specified, the default value is obtained from the fixed-point context type `sc_fxtype_context`. See Chapter 11.26.

Public Member Functions

```
int
iwl() const;
```
Returns the `iwl` value.

```
void
iwl( int val );
```
Sets the `iwl` value to `val`.

```
int
n_bits() const;
```
Returns the `n_bits` value.

```
void
n_bits( int );
```
Sets the `n_bits` value to `val`.

```
sc_o_mode
o_mode() const;
```
Returns the `o_mode`.


```
void
o_mode( sc_o_mode mode );
```
Sets the o_mode to mode.

```
sc_q_mode
q_mode() const;
```
Returns the q_mode.

```
void
q_mode( sc_q_mode mode);
```
Sets the q_mode to mode.

```
int
wl() const;
```
Returns the wl value.

```
void
wl( int val);
```
Sets the wl value to val.

Operators

```
sc_fxtype_params&
operator = ( const sc_fxtype_params& param_ );
```
The wl, iwl, q_mode, o_mode and n_bits of param_ are assigned to the left hand side.

```
friend bool
operator == ( const sc_fxtype_params& param_a, const
    sc_fxtype_params& param_b);
```
Returns true if the wl, iwl, q_mode, o_mode and n_bits of param_a are equal to the corresponding values of param_b else false.

```
friend bool
operator != ( const sc_fxtype_params&,
              const sc_fxtype_params& )
```
Returns true if all of wl, iwl, q_mode, o_mode and n_bits of param_a are not equal to the corresponding values of param_b else false.

```
ostream&
operator << ( ostream& os, const sc_fxtype_params& a )
```
Print the instance value of a to an output stream os.

11.28 sc_fxval

Synopsis

```
class sc_fxval
{
protected:
     sc_fxval_observer* observer() const;
public:
// Constructors and destructor
    sc_fxval( sc_fxval_observer* = 0 );
    sc_fxval( int,
      sc_fxval_observer* = 0 );
    sc_fxval( unsigned int,
      sc_fxval_observer* = 0 );
    sc_fxval( long,
      sc_fxval_observer* = 0 );
    sc_fxval( unsigned long,
      sc_fxval_observer* = 0 );
    sc_fxval( double,
      sc_fxval_observer* = 0 );
    sc_fxval( const char*,
      sc_fxval_observer* = 0 );
    sc_fxval( const sc_fxval&,
      sc_fxval_observer* = 0 );
    sc_fxval( const sc_fxval_fast&,
      sc_fxval_observer* = 0 );
    sc_fxval( const sc_fxnum&,
      sc_fxval_observer* = 0 );
    sc_fxval( const sc_fxnum_fast&,
      sc_fxval_observer* = 0 );
    sc_fxval( int64,
      sc_fxval_observer* = 0 );
    sc_fxval( uint64,
      sc_fxval_observer* = 0 );
    sc_fxval( const sc_int_base&,
      sc_fxval_observer* = 0 );
    sc_fxval( const sc_uint_base&,
      sc_fxval_observer* = 0 );
    sc_fxval( const sc_signed&,
      sc_fxval_observer* = 0 );
    sc_fxval( const sc_unsigned&,
      sc_fxval_observer* = 0 );
     ~sc_fxval();

// unary operators
    const sc_fxval  operator - () const;
    const sc_fxval& operator + () const;
    friend void neg( sc_fxval&, const sc_fxval& );

// binary operators
#define DECL_BIN_OP_T(op,tp) \
    friend const sc_fxval operator op ( const \
      sc_fxval&, tp ); \
```



```
    friend const sc_fxval operator op ( tp, const \
      sc_fxval& );

#define DECL_BIN_OP_OTHER(op) \
     DECL_BIN_OP_T(op,int64) \
     DECL_BIN_OP_T(op,uint64) \
     DECL_BIN_OP_T(op,const sc_int_base&) \
     DECL_BIN_OP_T(op,const sc_uint_base&) \
     DECL_BIN_OP_T(op,const sc_signed&) \
     DECL_BIN_OP_T(op,const sc_unsigned&)

#define DECL_BIN_OP(op,dummy) \
    friend const sc_fxval operator op ( const \
      sc_fxval&, const sc_fxval& ); \
     DECL_BIN_OP_T(op,int) \
     DECL_BIN_OP_T(op,unsigned int) \
     DECL_BIN_OP_T(op,long) \
     DECL_BIN_OP_T(op,unsigned long) \
     DECL_BIN_OP_T(op,double) \
     DECL_BIN_OP_T(op,const char*) \
     DECL_BIN_OP_T(op,const sc_fxval_fast&) \
     DECL_BIN_OP_T(op,const sc_fxnum_fast&) \
     DECL_BIN_OP_OTHER(op)

     DECL_BIN_OP(*,mult)
     DECL_BIN_OP(+,add)
     DECL_BIN_OP(-,sub)
     DECL_BIN_OP(/,div)
     DECL_BIN_OP_T(/,int)
     DECL_BIN_OP_T(/,unsigned int)
     DECL_BIN_OP_T(/,long)
     DECL_BIN_OP_T(/,unsigned long)
     DECL_BIN_OP_T(/,double)
     DECL_BIN_OP_T(/,const char*)
     DECL_BIN_OP_T(/,const sc_fxval_fast&)
     DECL_BIN_OP_T(/,const sc_fxnum_fast&)

     DECL_BIN_OP_T(/,int64) \
     DECL_BIN_OP_T(/,uint64) \
     DECL_BIN_OP_T(/,const sc_int_base&) \
     DECL_BIN_OP_T(/,const sc_uint_base&) \
     DECL_BIN_OP_T(/,const sc_signed&) \
     DECL_BIN_OP_T(/,const sc_unsigned&)

    friend const sc_fxval operator << ( const sc_fxval&,
      int );
    friend const sc_fxval operator >> ( const sc_fxval&,
      int );

// binary functions
#define DECL_BIN_FNC_T(fnc,tp) \
    friend void fnc ( sc_fxval&, const sc_fxval&, tp );\
    friend void fnc ( sc_fxval&, tp, const sc_fxval& );
```



```
#define DECL_BIN_FNC_OTHER(fnc) \
     DECL_BIN_FNC_T(fnc,int64) \
     DECL_BIN_FNC_T(fnc,uint64) \
     DECL_BIN_FNC_T(fnc,const sc_int_base&) \
     DECL_BIN_FNC_T(fnc,const sc_uint_base&) \
     DECL_BIN_FNC_T(fnc,const sc_signed&) \
     DECL_BIN_FNC_T(fnc,const sc_unsigned&)

#define DECL_BIN_FNC(fnc) \
    friend void fnc ( sc_fxval&, const sc_fxval&, const\
      sc_fxval& ); \
     DECL_BIN_FNC_T(fnc,int) \
     DECL_BIN_FNC_T(fnc,unsigned int) \
     DECL_BIN_FNC_T(fnc,long) \
     DECL_BIN_FNC_T(fnc,unsigned long) \
     DECL_BIN_FNC_T(fnc,double) \
     DECL_BIN_FNC_T(fnc,const char*) \
     DECL_BIN_FNC_T(fnc,const sc_fxval_fast&) \
     DECL_BIN_FNC_T(fnc,const sc_fxnum_fast&) \
     DECL_BIN_FNC_OTHER(fnc)

     DECL_BIN_FNC(mult)
     DECL_BIN_FNC(div)
     DECL_BIN_FNC(add)
     DECL_BIN_FNC(sub)

    friend void lshift( sc_fxval&, const sc_fxval&,int );
    friend void rshift( sc_fxval&, const sc_fxval&,int );

// relational (including equality) operators
#define DECL_REL_OP_T(op,tp) \
    friend bool operator op ( const sc_fxval&, tp ); \
    friend bool operator op ( tp, const sc_fxval& );

#define DECL_REL_OP_OTHER(op) \
     DECL_REL_OP_T(op,int64) \
     DECL_REL_OP_T(op,uint64) \
     DECL_REL_OP_T(op,const sc_int_base&) \
     DECL_REL_OP_T(op,const sc_uint_base&) \
     DECL_REL_OP_T(op,const sc_signed&) \
     DECL_REL_OP_T(op,const sc_unsigned&)

#define DECL_REL_OP(op) \
     friend bool operator op ( const sc_fxval&, const \
      sc_fxval& ); \
     DECL_REL_OP_T(op,int) \
     DECL_REL_OP_T(op,unsigned int) \
     DECL_REL_OP_T(op,long) \
     DECL_REL_OP_T(op,unsigned long) \
     DECL_REL_OP_T(op,double) \
     DECL_REL_OP_T(op,const char*) \
     DECL_REL_OP_T(op,const sc_fxval_fast&) \
     DECL_REL_OP_T(op,const sc_fxnum_fast&) \
     DECL_REL_OP_OTHER(op)
```



```
     DECL_REL_OP(<)
     DECL_REL_OP(<=)
     DECL_REL_OP(>)
     DECL_REL_OP(>=)
     DECL_REL_OP(==)
     DECL_REL_OP(!=)

// assignment operators
#define DECL_ASN_OP_T(op,tp) \
     sc_fxval& operator op( tp );

#define DECL_ASN_OP_OTHER(op) \
     DECL_ASN_OP_T(op,int64) \
     DECL_ASN_OP_T(op,uint64) \
     DECL_ASN_OP_T(op,const sc_int_base&) \
     DECL_ASN_OP_T(op,const sc_uint_base&) \
     DECL_ASN_OP_T(op,const sc_signed&) \
     DECL_ASN_OP_T(op,const sc_unsigned&)

#define DECL_ASN_OP(op) \
     DECL_ASN_OP_T(op,int) \
     DECL_ASN_OP_T(op,unsigned int) \
     DECL_ASN_OP_T(op,long) \
     DECL_ASN_OP_T(op,unsigned long) \
     DECL_ASN_OP_T(op,double) \
     DECL_ASN_OP_T(op,const char*) \
     DECL_ASN_OP_T(op,const sc_fxval&) \
     DECL_ASN_OP_T(op,const sc_fxval_fast&) \
     DECL_ASN_OP_T(op,const sc_fxnum&) \
     DECL_ASN_OP_T(op,const sc_fxnum_fast&) \
     DECL_ASN_OP_OTHER(op)

     DECL_ASN_OP(=)
     DECL_ASN_OP(*=)
     DECL_ASN_OP(/=)
     DECL_ASN_OP(+=)
     DECL_ASN_OP(-=)

     DECL_ASN_OP_T(<<=,int)
     DECL_ASN_OP_T(>>=,int)

// auto-increment and auto-decrement
    const sc_fxval operator ++ ( int );
    const sc_fxval operator -- ( int );
    sc_fxval& operator ++ ();
    sc_fxval& operator -- ();

// implicit conversion
    operator double() const;

// explicit conversion to primitive types
    short          to_short() const;
    unsigned short to_ushort() const;
```



```
    int             to_int() const;
    unsigned int    to_uint() const;
    long            to_long() const;
    unsigned long   to_ulong() const;
    float           to_float() const;
    double          to_double() const;

     // explicit conversion to character string
    const sc_string to_string() const;
    const sc_string to_string( sc_numrep ) const;
    const sc_string to_string( sc_numrep, bool ) const;
    const sc_string to_string( sc_fmt ) const;
    const sc_string to_string( sc_numrep,sc_fmt ) const;
    const sc_string to_string( sc_numrep, bool,
      sc_fmt ) const;
    const sc_string to_dec() const;
    const sc_string to_bin() const;
    const sc_string to_oct() const;
    const sc_string to_hex() const;

 // methods
    bool is_neg() const;
    bool is_zero() const;
    bool is_nan() const;
    bool is_inf() const;
    bool is_normal() const;
    bool rounding_flag() const;
    void print( ostream& = cout ) const;
    void scan( istream& = cin );
    void dump( ostream& = cout ) const;

 protected:
    sc_fxval_observer* lock_observer() const;
    void unlock_observer( sc_fxval_observer* ) const;
    void get_type( int&, int&, sc_enc& ) const;
    const sc_fxval quantization( const scfx_params&,
      bool& ) const;
    const sc_fxval overflow( const scfx_params&,
      bool& ) const;
 };
```

Description

Type **sc_fxval** is the arbitrary precision value type. It can hold the value of any of the fixed-point types, and it performs the arbitrary precision fixed-point arithmetic operations. Type casting is performed by the fixed-point types themselves. Limited precision type sc_fxval_fast and arbitrary precision type sc_fxval can be mixed freely.
See Chapter 6.8.4.

In some cases, such as division, using arbitrary precision would lead to infinite word lengths. To limit the resulting word lengths in these cases, three parameters are provided:

div_wl
The maximum word length for the result of a division operation. If the result of a division exceeds div_wl, it will be convergent rounded to div_wl bits. The div_wl argument is of type int. It must be greater than zero. Otherwise, a runtime error is produced. The default value for div_wl is obtained from the set of built-in default values. See 6.8.8. This default value can be overruled with compiler flag SC_FXDIV_WL.
cte_wl
The maximum word length for the result of converting a decimal character string constant into a sc_fxval variable. If the result of such a conversion exceeds cte_wl, it will be convergent rounded to cte_wl bits. The cte_wl argument is of type int. It must be greater than zero. Otherwise, a runtime error is produced. The default value for cte_wl is obtained from the set of built-in default values. See 6.8.8. This default value can be overruled with compiler flag SC_FXCTE_WL.
max_wl
The maximum word length for the mantissa used in a sc_fxval variable. If the result of an operation exceeds max_wl, it will be convergent rounded to max_wl bits. The max_wl argument is of type int. It must be greater than zero, or minus one. Otherwise, a runtime error is produced. Minus one is used to indicate no maximum word length. The default value for max_wl is obtained from the set of built-in default values. See 6.8.8. This default value can be overruled with compiler flag SC_FXMAX_WL.
Caution!
Be careful with changing the default values of the div_wl, cte_wl, and max_wl parameters, as they affect both bit-true behavior and simulation performance.

Type sc_fxval is used to hold fixed-point values for the arbitrary precision fixed-point types. The div_wl, cte_wl, and max_wl parameters should be set higher than the word lengths used by the fixed-point types in the user code, otherwise bit-true behavior cannot be guaranteed. On the other hand, these parameters should not be set too high, because that would degrade simulation performance. Typically, the max_wl parameter should be set (much) higher than the div_wl and cte_wl parameters.

The div_wl, cte_wl, and max_wl parameters will be used by the fixed-point value type, whether used directly or as part of a fixed-point type. By default, the built-in default values given in Chapter 6.8.8 are used. These default values can be overruled per translation unit by specifying the compiler flags SC_FXDIV_WL, SC_FXCTE_WL, and SC_FXMAX_WL with the appropriate values. For example:

```
CC -DSC_FXDIV_WL=128 -c my_file.cpp
```

This compiles my_file.cpp with the div_wl parameter set to 128 bits i.s.o. 64 bits.

A sc_fxval variable that is declared without initial value is uninitialized, unless it is declared as a static variable, which is always initialized to zero. Uninitialized variables can be used anywhere initialized variables can be used. An operation on an uninitialized variable does not produce an error or warning. The result of such an operation is undefined.

Examples

```
sc_fxval a = 1;
sc_fxval b = 0.5;
sc_fixed<8,8> c = 1.25;
sc_fxval d = c;
sc_biguint<16> e = 8;
sc_fxval f = e;
sc_fxval j;
sc_fxval k(0.5);
sc_fxval l = 0;

sc_fxval m = 1;
sc_fxval n = 2;
sc_fxval p = m / n;
n *= 1.25;
```

Public Constructors

```
sc_fxval ( [type_ init_val]
           [,sc_fxnum_observer* observer] ) ;
type_ in {short, unsigned short, int, unsigned int, long,
   unsigned long, float, double, const char*, int64,
   uint64, const sc_int_base†&, const sc_uint_base†&,
   const sc_signed&, const sc_unsigned, const sc_fxval&,
   const sc_fxval_fast&, const sc_[u]fix&, const
   sc_[u]fix_fast& }
```

Notes on `type_`

For all types in `type_` only the value of the argument is taken, that is, any type information is discarded.

A variable of type sc_fxval can be initialized with a C/C++ character string (type const char*) either when the number will be expressed in binary form or when the number is too large to be written as a C/C++ built-in type literal.

init_val

The initial value of the variable. If the initial value is not specified, the variable is uninitialized.

observer

A pointer to an observer object. The observer argument is of type sc_fxval_observer*. See Chapter 11.31. The default value for observer is 0 (null pointer). The observer parameter cannot change after declaration.

Operators

The operators defined for the `sc_fxval` are given in Table 24.

Table 24. Operators for sc_fxval

Operator	Operators in class

class	
Arithmetic	* / + - << >> ++ --
Equality	== !=
Relational	<<= >>=
Assignment	= *= /= += -= <<= >>=

Note:
Operator << and operator >> define arithmetic shifts, not bitwise shifts. The difference is that no bits are lost and proper sign extension is done. Hence, these operators are well defined also for signed types, such as sc_fxval.

In expressions with the operators from Table 24, variables of type sc_fxval can be mixed with all types given:

type_ in {short, unsigned short, int, unsigned int, long, unsigned long, float, double, const char*, int64, uint64, const *sc_int_base*†&, const *sc_uint_base*†&, const sc_signed&, const sc_unsigned, const sc_fxval&, const sc_fxval_fast&, const sc_[u]fix&, const sc_[u]fix_fast& }

The return type of any arithmetic operation is `sc_fxval`.

Member Functions

The functions defined for `sc_fxval` are given in Table 25.

Table 25. Functions for sc_fxval

Function class	Functions in class
Arithmetic	neg, mult, div, add, sub, lshift, rshift

The functions in Table 25 have return type void. The first argument of these functions is a reference to the result object. The remaining arguments of these functions are the operands.

The neg arithmetic function takes one operand, the other arithmetic functions take two operands. At least one of the operands of the arithmetic functions should have a fixed- point value type, the other operand can have any of the types given:

type_ in {short, unsigned short, int, unsigned int, long, unsigned long, float, double, const char*, int64, uint64, const *sc_int_base*†&, const *sc_uint_base*†&, const sc_signed&, const sc_unsigned, const sc_fxval&, const sc_fxval_fast&, const sc_[u]fix&, const sc_[u]fix_fast& }

The arithmetic are defined with the result object of type sc_fxval.

Query Value

```
bool
is_inf() const;
```

Returns true if the variable holds an (plus or minus) infinity value. Returns false otherwise.

```
bool
is_nan() const;
```

Returns true if the variable holds not-a-number value. Returns false otherwise.

```
bool
is_neg() const;
```

Returns true if the variable holds a negative value. Returns false otherwise.

```
bool
is_zero() const;
```

Returns true if the variable holds a zero value. Returns false otherwise.

```
bool
rounding_flag() const;
```

Returns true if the last write action on this variable caused rounding to div_wl, cte_wl, or max_wl. Returns false otherwise.

```
bool
is_normal() const;
```

Returns true if both `is_nan()` and `is_inf()` return false. Returns false otherwise.

Implicit Conversion

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

Implicit conversion to the implementation type `double`. The value does not change.

Explicit Conversion

```
short           to_short() const;
unsigned short  to_ushort() const;
int             to_int() const;
unsigned int    to_uint() const;
long            to_long() const;
unsigned long   to_ulong() const;
float           to_float() const;
double          to_double() const


const sc_string to_string() const;
const sc_string to_string( sc_numrep ) const;
const sc_string to_string( sc_numrep, bool ) const;
const sc_string to_string( sc_fmt ) const;
```



```
const sc_string to_string( sc_numrep, sc_fmt ) const;
const sc_string to_string( sc_numrep, bool, sc_fmt ) const;
```

The value of a fixed-point variable can be converted to a character string with the to_string() method. This method takes different arguments for formatting purposes. See Chapter 6.8.8 for more information on converting fixed-point variables to/from character strings. Furthermore, writing to C++ output streams with operator << is supported, e.g. cout << a;, where a is a fixed-point variable. The decimal number representation is used in this case.

```
const sc_string to_dec() const;
const sc_string to_bin() const;
const sc_string to_oct() const;
const sc_string to_hex() const;
```

Shortcut methods for conversion to a character string. See Chapter 6.8.9.2.

Print or dump content

```
void
print( ostream& = cout ) const;
```

Print the `sc_fxval` instance value to an output stream.

```
void
scan( istream& = cin );
```

Read an `sc_fxval` value from an input stream.

```
void
dump( ostream& = cout )
const;
```

Prints the `sc_fxval` instance value, parameters and flags to an output stream.

```
ostream&
operator << ( ostream& os, const sc_fix& a )
```

Print the instance value of `a` to an output stream `os`.

11.29 sc_fxval_fast

Synopsis

```
class sc_fxval_fast
{
protected:
    sc_fxval_fast_observer* observer() const;

public:
    sc_fxval_fast( sc_fxval_fast_observer* = 0 );
    sc_fxval_fast( int,
      sc_fxval_fast_observer* = 0 );
    sc_fxval_fast( unsigned int,
      sc_fxval_fast_observer* = 0 );
    sc_fxval_fast( long,
      sc_fxval_fast_observer* = 0 );
    sc_fxval_fast( unsigned long,
      sc_fxval_fast_observer* = 0 );
    sc_fxval_fast( double,
      sc_fxval_fast_observer* = 0 );
    sc_fxval_fast( const char*,
      sc_fxval_fast_observer* = 0 );
    sc_fxval_fast( const sc_fxval&,
      sc_fxval_fast_observer* = 0 );
    sc_fxval_fast( const sc_fxval_fast&,
      sc_fxval_fast_observer* = 0 );
    sc_fxval_fast( const sc_fxnum&,
      sc_fxval_fast_observer* = 0 );
    sc_fxval_fast( const sc_fxnum_fast&,
      sc_fxval_fast_observer* = 0 );
    sc_fxval_fast( int64,
      sc_fxval_fast_observer* = 0 );
    sc_fxval_fast( uint64,
      sc_fxval_fast_observer* = 0 );
    sc_fxval_fast( const sc_int_base&,
      sc_fxval_fast_observer* = 0 );
    sc_fxval_fast( const sc_uint_base&,
      sc_fxval_fast_observer* = 0 );
    sc_fxval_fast( const sc_signed&,
      sc_fxval_fast_observer* = 0 );
    sc_fxval_fast( const sc_unsigned&,
      sc_fxval_fast_observer* = 0 );
    ~sc_fxval_fast();

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

    // unary functions
    friend void neg( sc_fxval_fast&, const
      sc_fxval_fast& );

     // binary operators
```



```
#define DECL_BIN_OP_T(op,tp) \
   friend const sc_fxval_fast operator op ( const \
     sc_fxval_fast&, tp ); \
   friend const sc_fxval_fast operator op ( tp, const \
     sc_fxval_fast& );

#define DECL_BIN_OP_OTHER(op) \
    DECL_BIN_OP_T(op,int64) \
    DECL_BIN_OP_T(op,uint64) \
    DECL_BIN_OP_T(op,const sc_int_base&) \
    DECL_BIN_OP_T(op,const sc_uint_base&) \
    DECL_BIN_OP_T(op,const sc_signed&) \
    DECL_BIN_OP_T(op,const sc_unsigned&)

#define DECL_BIN_OP(op,dummy) \
   friend const sc_fxval_fast operator op ( const \
     sc_fxval_fast&, const sc_fxval_fast& ); \
    DECL_BIN_OP_T(op,int) \
    DECL_BIN_OP_T(op,unsigned int) \
    DECL_BIN_OP_T(op,long) \
    DECL_BIN_OP_T(op,unsigned long) \
    DECL_BIN_OP_T(op,double) \
    DECL_BIN_OP_T(op,const char*) \
    DECL_BIN_OP_OTHER(op)

    DECL_BIN_OP(*,mult)
    DECL_BIN_OP(+,add)
    DECL_BIN_OP(-,sub)
    DECL_BIN_OP(/,div)
    DECL_BIN_OP_T(/,int)
    DECL_BIN_OP_T(/,unsigned int)
    DECL_BIN_OP_T(/,long)
    DECL_BIN_OP_T(/,unsigned long)
    DECL_BIN_OP_T(/,double)
    DECL_BIN_OP_T(/,const char*)
    DECL_BIN_OP_OTHER(/)
    DECL_BIN_OP_T(/,int64) \
    DECL_BIN_OP_T(/,uint64) \
    DECL_BIN_OP_T(/,const sc_int_base&) \
    DECL_BIN_OP_T(/,const sc_uint_base&) \
    DECL_BIN_OP_T(/,const sc_signed&) \
    DECL_BIN_OP_T(/,const sc_unsigned&)

   friend const sc_fxval_fast operator << ( const
     sc_fxval_fast&, int );
   friend const sc_fxval_fast operator >> ( const
     sc_fxval_fast&, int );

    // binary functions
#define DECL_BIN_FNC_T(fnc,tp) \
   friend void fnc ( sc_fxval_fast&, const \
     sc_fxval_fast&, tp ); \
   friend void fnc ( sc_fxval_fast&, tp, const \
     sc_fxval_fast& );
```



```
#define DECL_BIN_FNC_OTHER(fnc) \
     DECL_BIN_FNC_T(fnc,int64) \
     DECL_BIN_FNC_T(fnc,uint64) \
     DECL_BIN_FNC_T(fnc,const sc_int_base&) \
     DECL_BIN_FNC_T(fnc,const sc_uint_base&) \
     DECL_BIN_FNC_T(fnc,const sc_signed&) \
     DECL_BIN_FNC_T(fnc,const sc_unsigned&)

#define DECL_BIN_FNC(fnc) \
    friend void fnc ( sc_fxval_fast&, const \
      sc_fxval_fast&, const sc_fxval_fast& ); \
     DECL_BIN_FNC_T(fnc,int) \
     DECL_BIN_FNC_T(fnc,unsigned int) \
     DECL_BIN_FNC_T(fnc,long) \
     DECL_BIN_FNC_T(fnc,unsigned long) \
     DECL_BIN_FNC_T(fnc,double) \
     DECL_BIN_FNC_T(fnc,const char*) \
     DECL_BIN_FNC_T(fnc,const sc_fxval&) \
     DECL_BIN_FNC_T(fnc,const sc_fxnum&) \
     DECL_BIN_FNC_OTHER(fnc)

     DECL_BIN_FNC(mult)
     DECL_BIN_FNC(div)
     DECL_BIN_FNC(add)
     DECL_BIN_FNC(sub)

    friend void lshift( sc_fxval_fast&, const
      sc_fxval_fast&, int );
    friend void rshift( sc_fxval_fast&, const
      sc_fxval_fast&, int );

     // relational (including equality) operators
#define DECL_REL_OP_T(op,tp) \
    friend bool operator op ( const sc_fxval_fast&,tp);\
    friend bool operator op ( tp, const sc_fxval_fast& );

#define DECL_REL_OP_OTHER(op) \
     DECL_REL_OP_T(op,int64) \
     DECL_REL_OP_T(op,uint64) \
     DECL_REL_OP_T(op,const sc_int_base&) \
     DECL_REL_OP_T(op,const sc_uint_base&) \
     DECL_REL_OP_T(op,const sc_signed&) \
     DECL_REL_OP_T(op,const sc_unsigned&)

#define DECL_REL_OP(op) \
    friend bool operator op ( const sc_fxval_fast&, \
      const sc_fxval_fast& ); \
     DECL_REL_OP_T(op,int) \
     DECL_REL_OP_T(op,unsigned int) \
     DECL_REL_OP_T(op,long) \
     DECL_REL_OP_T(op,unsigned long) \
     DECL_REL_OP_T(op,double) \
     DECL_REL_OP_T(op,const char*) \
```



```
      DECL_REL_OP_OTHER(op)

      DECL_REL_OP(<)
      DECL_REL_OP(<=)
      DECL_REL_OP(>)
      DECL_REL_OP(>=)
      DECL_REL_OP(==)
      DECL_REL_OP(!=)

      // assignment operators
 #define DECL_ASN_OP_T(op,tp) \
      sc_fxval_fast& operator op( tp );

 #define DECL_ASN_OP_OTHER(op) \
      DECL_ASN_OP_T(op,int64) \
      DECL_ASN_OP_T(op,uint64) \
      DECL_ASN_OP_T(op,const sc_int_base&) \
      DECL_ASN_OP_T(op,const sc_uint_base&) \
      DECL_ASN_OP_T(op,const sc_signed&) \
      DECL_ASN_OP_T(op,const sc_unsigned&)

 #define DECL_ASN_OP(op) \
      DECL_ASN_OP_T(op,int) \
      DECL_ASN_OP_T(op,unsigned int) \
      DECL_ASN_OP_T(op,long) \
      DECL_ASN_OP_T(op,unsigned long) \
      DECL_ASN_OP_T(op,double) \
      DECL_ASN_OP_T(op,const char*) \
      DECL_ASN_OP_T(op,const sc_fxval&) \
      DECL_ASN_OP_T(op,const sc_fxval_fast&) \
      DECL_ASN_OP_T(op,const sc_fxnum&) \
      DECL_ASN_OP_T(op,const sc_fxnum_fast&) \
      DECL_ASN_OP_OTHER(op)

      DECL_ASN_OP(=)
      DECL_ASN_OP(*=)
      DECL_ASN_OP(/=)
      DECL_ASN_OP(+=)
      DECL_ASN_OP(-=)
      DECL_ASN_OP_T(<<=,int)
      DECL_ASN_OP_T(>>=,int)

     // auto-increment and auto-decrement
     const sc_fxval_fast operator ++ ( int );
     const sc_fxval_fast operator -- ( int );
     sc_fxval_fast& operator ++ ();
     sc_fxval_fast& operator -- ();

     // implicit conversion
     operator double() const;

     // explicit conversion to primitive types
     short        to_short() const;
     unsigned short to_ushort() const;
```



```
    int          to_int() const;
    unsigned int to_uint() const;
    long         to_long() const;
    unsigned long  to_ulong() const;
    float         to_float() const;
    double       to_double() const;

    // explicit conversion to character string
    const sc_string to_string() const;
    const sc_string to_string( sc_numrep ) const;
    const sc_string to_string( sc_numrep, bool ) const;
    const sc_string to_string( sc_fmt ) const;
    const sc_string to_string( sc_numrep,
       sc_fmt ) const;
    const sc_string to_string( sc_numrep, bool,
       sc_fmt ) const;
    const sc_string to_dec() const;
    const sc_string to_bin() const;
    const sc_string to_oct() const;
    const sc_string to_hex() const;

    // other methods
    bool is_neg() const;
    bool is_zero() const;
    bool is_nan() const;
    bool is_inf() const;
    bool is_normal() const;
    bool rounding_flag() const;
    void print( ostream& = cout ) const;
    void scan( istream& = cin );
    void dump( ostream& = cout ) const;
};
```

Description

Type **`sc_fxval_fast`** is the fixed precision value type and is limited to a mantissa of 53 bits. It can hold the value of any of the fixed-point types, and it performs the fixed precision fixed-point arithmetic operations. Type casting is performed by the fixed-point types themselves. Limited precision type sc_fxval_fast and arbitrary precision type sc_fxval can be mixed freely. See Chapter 6.8.4.

Type sc_fxval is used to hold fixed-point values for the fixed precision fixed-point types.

A sc_fxval variable that is declared without initial value is uninitialized, unless it is declared as a static variable, which is always initialized to zero. Uninitialized variables can be used anywhere initialized variables can be used. An operation on an uninitialized variable does not produce an error or warning. The result of such an operation is undefined.

Examples

```
sc_fxval_fast a = 1;
sc_fxval_fast b = 0.5;
sc_fixed<8,8> c = 1.25;
sc_fxval_fast d = c;
sc_biguint<16> e = 8;
sc_fxval_fast f = e;
sc_fxval_fast j;
sc_fxval_fast k(0.5);
sc_fxval_fast l = 0;

sc_fxval_fast m = 1;
sc_fxval_fast n = 2;
sc_fxval_fast p = m / n;
n *= 1.25;
```

Public Constructors

```
sc_fxval_fast ( [type_ init_val]
                [,sc_fxval_fast_observer* observer] ) ;
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

Notes on `type_`

For all types in `type_` only the value of the argument is taken, that is, any type information is discarded.

A variable of type `sc_fxval_fast` can be initialized with a C/C++ character string (type const char*) either when the number will be expressed in binary form or when the number is too large to be written as a C/C++ built-in type literal.

init_val

The initial value of the variable. If the initial value is not specified, the variable is uninitialized.

observer

A pointer to an observer object. The observer argument is of type sc_fxval_fast_observer*. See Chapter 11.30. The default value for observer is 0 (null pointer). The observer parameter cannot change after declaration.

Operators

The operators defined for the `sc_fxval` are given in Table 26.

Table 26. Operators for sc_fxval _fast

Operator class	Operators in class

Arithmetic	* / + - << >> ++ --
Equality	== !=
Relational	<<= >>=
Assignment	= *= /= += -= <<= >>=

Note:
Operator << and operator >> define arithmetic shifts, not bitwise shifts. The difference is that no bits are lost and proper sign extension is done. Hence, these operators are well defined also for signed types, such as sc_fxval_fast.

In expressions with the operators from Table 26, variables of type sc_fxval_fast can be mixed with all types given:

type_ in {short, unsigned short, int, unsigned int, long, unsigned long, float, double, const char*, int64, uint64, const *sc_int_base*†&, const *sc_uint_base*†&, const sc_signed&, const sc_unsigned, const sc_fxval&, const sc_fxval_fast&, const sc_[u]fix&, const sc_[u]fix_fast& }

The return type of any arithmetic operation is `sc_fxval_fast`.

Member Functions

The functions defined for `sc_fxval_fast` are given in Table 27.

Table 27. Functions for sc_fxval_fast

Function class	Functions in class
Arithmetic	neg, mult, div, add, sub, lshift, rshift

The functions in Table 27 have return type void. The first argument of these functions is a reference to the result object. The remaining arguments of these functions are the operands.

The neg arithmetic function takes one operand, the other arithmetic functions take two operands. At least one of the operands of the arithmetic functions should have a fixed- point value type, the other operand can have any of the types given:

type_ in {short, unsigned short, int, unsigned int, long, unsigned long, float, double, const char*, int64, uint64, const *sc_int_base*†&, const *sc_uint_base*†&, const sc_signed&, const sc_unsigned, const sc_fxval&, const sc_fxval_fast&, const sc_[u]fix&, const sc_[u]fix_fast& }

The arithmetic are defined with the result object of type sc_fxval_fast.

Query Value


```
bool
is_inf() const;
```
Returns true if the variable holds an (plus or minus) infinity value. Returns false otherwise.

```
bool
is_nan() const;
```
Returns true if the variable holds not-a-number value. Returns false otherwise.

```
bool
is_neg() const;
```
Returns true if the variable holds a negative value. Returns false otherwise.

```
bool
is_zero() const;
```
Returns true if the variable holds a zero value. Returns false otherwise.

```
bool
rounding_flag() const;
```
Returns true if the last write action on this variable caused rounding to div_wl, cte_wl, or max_wl. Returns false otherwise.

```
bool
is_normal() const;
```
Returns true if both `is_nan()` and `is_inf()` return false. Returns false otherwise.

Implicit Conversion

```
operator double() const;
```
Implicit conversion to the implementation type `double`. The value does not change.

Explicit Conversion

```
short           to_short() const;
unsigned short  to_ushort() const;
int             to_int() const;
unsigned int    to_uint() const;
long            to_long() const;
unsigned long   to_ulong() const;
float           to_float() const;
double          to_double() const


const sc_string to_string() const;
const sc_string to_string( sc_numrep ) const;
const sc_string to_string( sc_numrep, bool ) const;
const sc_string to_string( sc_fmt ) const;
const sc_string to_string( sc_numrep, sc_fmt ) const;
```



```
const sc_string to_string( sc_numrep, bool, sc_fmt ) const;
```

The value of a fixed-point variable can be converted to a character string with the to_string() method. This method takes different arguments for formatting purposes. See Chapter 6.8.8 for more information on converting fixed-point variables to/from character strings. Furthermore, writing to C++ output streams with operator << is supported, e.g. cout << a;, where a is a fixed-point variable. The decimal number representation is used in this case.

```
const sc_string to_dec() const;
const sc_string to_bin() const;
const sc_string to_oct() const;
const sc_string to_hex() const;
```

Shortcut methods for conversion to a character string. See Chapter 6.8.9.2.

Print or dump content

```
void
print( ostream& = cout ) const;
```

Print the `sc_fxval_fast` instance value to an output stream.

```
void
scan( istream& = cin );
```

Read an `sc_fxval_fast` value from an input stream.

```
void
dump( ostream& = cout )
const;
```

Prints the `sc_fxval_fast` instance value, parameters and flags to an output stream.

```
ostream&
operator << ( ostream& os, const sc_fix& a )
```

Print the instance value of `a` to an output stream `os`.

11.30 sc_fxval_fast_observer

Synopsis

```
class sc_fxval_fast_observer
{
protected:
    sc_fxval_fast_observer() {}
    virtual ~sc_fxval_fast_observer() {}

public:
    virtual void  construct( const sc_fxval_fast& );
    virtual void  destruct( const sc_fxval_fast& );
    virtual void  read( const sc_fxval_fast& );
    virtual void  write( const sc_fxval_fast& );
    static sc_fxval_fast_observer*
      (*default_observer) ();
};
```

Description

`sc_fxval_fast_observer` is an abstract base class provided as a hook to define one's own observer functionality.

Public Methods

```
virtual void construct( const sc_fxval_fast& );
virtual void  destruct( const sc_fxval_fast& );
virtual void      read( const sc_fxval_fast& );
virtual void     write( const sc_fxval_fast& );
```

These methods allow to observe construction, destruction, read, and write actions on a particular variable. The destruct and read methods are called before the action takes place, while the construct and write methods are called after the action has taken place. Each of these methods can query the variable under observation, which is passed as the single argument to the methods.

The default behavior of the methods is to do nothing (and return).

11.31 sc_fxval_observer

Synopsis

```
class sc_fxval_observer
{
protected:
    sc_fxval_observer() {}
    virtual ~sc_fxval_observer() {}

public:
    virtual void  construct( const sc_fxval& );
    virtual void  destruct( const sc_fxval& );
    virtual void  read( const sc_fxval& );
    virtual void  write( const sc_fxval& );
    static sc_fxval_observer* (*default_observer) ();
};
```

Description

sc_fxval_observer is an abstract base class provided as a hook to define one's own observer functionality.

Public Methods

```
virtual void construct( const sc_fxval& );
virtual void  destruct( const sc_fxval& );
virtual void      read( const sc_fxval& );
virtual void     write( const sc_fxval& );
```

These methods allow to observe construction, destruction, read, and write actions on a particular variable. The destruct and read methods are called before the action takes place, while the construct and write methods are called after the action has taken place. Each of these methods can query the variable under observation, which is passed as the single argument to the methods.

The default behavior of the methods is to do nothing (and return).

11.32 sc_in

Synopsis

```
template <class T>
class sc_in
: public sc_port<sc_signal_in_if<T>,1>
{
public:

    // constructors and destructor
    sc_in();
    sc_in( const char* name_ );
    sc_in( const sc_signal_in_if<T>& interface_ );
    sc_in( const char* name_,
      const sc_signal_in_if<T>& interface_ );
    sc_in(sc_port<sc_signal_in_if<T> >& parent_ );
    sc_in( const char* name_,
      sc_port<sc_signal_in_if<T> >& parent_ );
    sc_in(sc_port<sc_signal_inout_if<T> >& parent_ );
    sc_in( const char* name_,
      sc_port<sc_signal_inout_if<T> >& parent_ );
    sc_in( sc_in<T>& parent_ );
    sc_in( const char* name_, sc_in<T>& parent_ );
    virtual ~sc_in();

// methods
    void bind( const sc_signal_in_if<T>& interface_ );
    void operator () ( const
      sc_signal_in_if<T>& interface_ );
    void bind( sc_port< sc_signal_in_if<T> >& parent_ );
    void operator () (
      sc_port< sc_signal_in_if<T> >& parent_ );
    void bind(
      sc_port<sc_signal_inout_if<T> >& parent_ );
    void operator () (
      sc_port<sc_signal_inout_if<T> >& parent_ );
    const sc_event& default_event() const;
    const sc_event& value_changed_event() const;
    const T& read() const;
    operator const T& () const;
    bool event() const;
    sc_event_finder& value_changed() const;
    virtual void end_of_elaboration();
    static const char* const kind_string;
    virtual const char* kind() const;
    void add_trace( sc_trace_file*,
      const sc_string& ) const;
};
```

Description

`sc_in` is a specialized port for use with `sc_signal` channels (Chapter 11.59). Its behavior is that of a sc_port which has only one interface that is of type sc_signal_in_if<T>. It has additional methods for convenience in accessing the channel connected to the port.

In the description of sc_in, *port* refers to the sc_in instance, *current_value* refers to the value of the sc_signal instance connected to the port, *new_value* is the value to be written and *old_value* is the previous value. Chapter 2.4.1 describes the scheduler steps referred to in the description of sc_inout.

Public Constructors

```
sc_in();
```
Create a sc_in instance.

```
explicit
sc_in( const char* name_ ) ;
```
Create a sc_in instance with the string name initialized to name_.

Public Member Functions

```
void
add_trace( sc_trace_file†*, const sc_string& ) const;
```

```
void
bind( const sc_signal_in_if<T>& interface_ ) ;
```
Binds interface_ to the port. For port to channel binding.

```
void
bind( sc_port<sc_signal_in_if<T>,1 >& parent_ ) ;
```
Binds parent_ to the port. For port to port binding.

```
void
bind( sc_port<sc_signal_inout_if<T>,1 >& parent_ );
```
Binds parent_ to the port. For port to port binding.

```
const sc_event&
default_event() const ;
```
Returns a reference to an event that occurs when new_value on a write is different from current_value.

```
bool
event() const ;
```
Returns true if an event occurred in the previous delta-cycle.

```
virtual void
end_of_elaboration();
```
Called at the end of the elaboration phase, after ports have been bound to channels. If a trace has been requested on this port during elaboration, then end_of_elaboration adds a trace using the attached channel’s data.

```
virtual const char*
```


kind() const ;
Returns “sc_in”.

sc_event_finder[†]&
neg() const ;
Type bool and sc_logic only. Returns a reference to an *sc_event_finder*[†] that occurs when new_value on a write is false and the current_value is not false. For use with static sensitivity list of a process.

bool
negedge() const ;
Type bool and sc_logic only. Returns true if an event occurred in the previous delta-cycle and current_value is false.

const sc_event&
negedge_event() const ;
Type bool and sc_logic only. Returns a reference to an event that occurs when new_value on a write is false and the current_value is not false.

sc_event_finder[†]&
pos() const ;
Type bool and sc_logic only. Returns a reference to an *sc_event_finder*[†] that occurs when new_value on a write is true and the current_value is not true. For use with static sensitivity list of a process.

bool
posedge() const ;
Type bool and sc_logic only. Returns a reference to an event that occurs when new_value on a write is true and the current_value is not true.

const sc_event&
posedge_event() const ;
Type bool and sc_logic only. Returns a reference to an event that occurs when new_value on a write is true and the current_value is not true.

const T&
read() const ;
Returns a reference to current_value.

sc_event_finder[†]&
value_changed() const ;
Returns a reference to an *sc_event_finder*[†] that occurs when new_value on a write is different from current_value. For use with static sensitivity list of a process.

const sc_event&
value_changed_event() const ;

Returns a reference to an event that occurs when new_value on a write is different from current_value.

Public Operators

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

Binds `interface_` to the port. For port to channel binding.

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

Binds `parent_` to the port. For port to port binding.

```
void
operator () (sc_port<sc_signal_inout_if<T>,1 >& ) ;
```

Binds `parent_` to the port. For port to port binding.

```
operator const T& () const ;
```

Disabled Member Functions

```
sc_in( const sc_in<T>& );
sc_in<T>& operator = ( const sc_in<T>& );
```


11.33 sc_in_resolved

Synopsis

```
class sc_in_resolved
    : public sc_in<sc_logic>
{
public:
// constructors and destructor
    sc_in_resolved();
    sc_in_resolved( const char* name_ );
    sc_in_resolved( const
      sc_signal_in_if<sc_logic>& interface_ );
    sc_in_resolved( const char* name_,
      const sc_signal_in_if<sc_logic>& interface_ );
    sc_in_resolved(
      sc_port<sc_signal_in_if<sc_logic> >& parent_ );
    sc_in_resolved( const char* name_,
      sc_port<sc_signal_in_if<sc_logic> >& parent_ );
    sc_in_resolved(
      sc_port<sc_signal_inout_if<sc_logic> >& parent_ );
    sc_in_resolved( const char* name_,
      sc_port<sc_signal_inout_if<sc_logic> >& parent_ );
    sc_in_resolved(sc_in_resolved& parent_ );
    sc_in_resolved( const char* name_,
      sc_in_resolved& parent_ );
    virtual ~sc_in_resolved();

// methods
    virtual void end_of_elaboration();
    static const char* const kind_string;
    virtual const char* kind() const;

// disabled
    sc_in_resolved( const sc_in_resolved& );
    sc_in_resolved& operator = (const sc_in_resolved& );
};
```

Description

sc_in_resolved is a specialized port for use with sc_signal_resolved channels (Chapter 11.63). Its behavior is that of a sc_port which has only one interface that is of type sc_signal_in_if<sc_logic>. It has additional methods for convenience in accessing the channel connected to the port.

In the description of sc_in_resolved, *port* refers to the sc_in_resolved instance.

Public Constructors

sc_in_resolved() ;

Create a sc_in_resolved instance.


```
explicit
sc_in_resolved( const char* );
```

Create a `sc_in_resolved` instance with the string name initialized to `name_`.

Public Member Functions

```
virtual void
end_of_elaboration() ;
```

Checks to make sure the channel bound to the port is of type `sc_signal_resolved`.

```
virtual const char*
kind() const ;
```

Returns “sc_in_resolved”.

Disabled Member Functions

```
sc_in_resolved (const sc_in_resolved& );
sc_in_resolved& operator = ( const sc_in_resolved& );
```


11.34 sc_in_rv

Synopsis

```
template <int W>
class sc_in_rv
    : public sc_in<sc_lv<W> >
{
public:
// constructors and destructor
    sc_in_rv();
    sc_in_rv( const char* name_ );
    sc_in_rv( const
      sc_signal_in_if<sc_lv<W> >& interface_ );
    sc_in_rv( const char* name_,
      const sc_signal_in_if<sc_lv<W> >& interface_ );
    sc_in_rv(
      sc_port< sc_signal_in_if<sc_lv<W> > >& parent_ );
    sc_in_rv( const char* name_,
      sc_port< sc_signal_in_if<sc_lv<W> > >& parent_ );
    sc_in_rv(
      sc_port<sc_signal_inout_if<sc_lv<W> > >& parent_);
    sc_in_rv( const char* name_,
      sc_port<sc_signal_inout_if<sc_lv<W> > >& parent_);
    sc_in_rv( sc_in_rv<W>& parent_ );
    sc_in_rv( const char* name_, sc_in_rv<W>& parent_ );
    virtual ~sc_in_rv();

// methods
    virtual void end_of_elaboration();
    static const char* const kind_string;
    virtual const char* kind() const;

private:
// disabled
    sc_in_rv( const sc_in_rv<W>& );
    sc_in_rv<W>& operator = ( const sc_in_rv<W>& );
};
```

Description

sc_in_rv is a specialized port for use with sc_signal_rv channels (Chapter 11.63). Its behavior is that of a sc_port which has only one interface that is of type sc_signal_in_if<sc_lv<W> >. It has additional methods for convenience in accessing the channel connected to the port.

In the description of sc_in_rv, *port* refers to the sc_in_rv instance.

Public Constructors

sc_in_rv() ;

Create a sc_in_rv instance.


```
explicit
sc_in_rv( const char* );
```

Create a `sc_in_rv` instance with the string name initialized to `name_`.

Public Member Functions

```
virtual void
end_of_elaboration() ;
```

Checks to make sure the channel bound to the port is of type `sc_signal_rv`.

```
virtual const char*
kind() const ;
```

Returns “sc_in_rv”.

Disabled Member Functions

```
sc_in_rv( const sc_in_rv<W>& );
sc_in_rv<W>& operator = ( const sc_in_rv<W>& );
```


11.35 sc_inout

Synopsis

```
template <class T>
class sc_inout
: public sc_port<sc_signal_inout_if<T>,1>
{
public:
    // constructors and destructor

    sc_inout();
    sc_inout( const char* name_ );
    sc_inout(sc_signal_inout_if<T>& interface_ );
    sc_inout( const char* name_,
      sc_signal_inout_if<T>& interface_ );
    sc_inout(sc_port<sc_signal_inout_if<T> >& parent_ );
    sc_inout( const char* name_,
      sc_port<sc_signal_inout_if<T> >& parent_ );
    sc_inout( sc_inout<T>& parent_ );
    sc_inout( const char* name_, sc_inout<T>& parent_ );
    virtual ~sc_inout();

    // methods
    const sc_event& default_event() const;
    const sc_event& value_changed_event() const;
    const T& read() const;
    operator const T& () const;
    bool event() const;
    sc_inout<T>& write( const T& value_ );
    sc_inout<T>& operator = ( const T& value_ );
    sc_inout<T>& operator = ( const
      sc_signal_in_if<T>& interface_ );
    sc_inout<T>& operator = ( const
      sc_port< sc_signal_inout_if<T> >& port_ );
    sc_inout<T>& operator = ( const
      sc_port< sc_signal_inout_if<T> >& port_ );
    sc_inout<T>& operator = (const sc_inout<T>& port_ );
    void initialize( const T& value_ );
    void initialize( const
      sc_signal_in_if<T>& interface_ );
    virtual void end_of_elaboration();
    sc_event_finder& value_changed() const
    static const char* const kind_string;
    virtual const char* kind() const;
    void add_trace( sc_trace_file*,
      const sc_string& ) const;
};
```

Description

`sc_inout` is a specialized port for use with sc_signal channels (Chapter 11.59). Its behavior is that of a sc_port which has only one interface that is of type sc_signal_inout_if<T>. It has additional methods for convenience in accessing the channel connected to the port.

In the description of `sc_in`, *port* refers to the `sc_inout` instance, *current_value* refers to the value of the `sc_signal` instance connected to the port, *new_value* is the value to be written and *old_value* is the previous value. Chapter 2.4.1 describes the scheduler steps referred to in the description of `sc_inout`.

Public Constructors

sc_inout();

Create a `sc_inout` instance.

explicit
sc_inout(const char*) ;

Create a `sc_inout` instance with the string name initialized to `name_`.

Public Member Functions

void
add_trace(sc_trace_file*, const sc_string&) const;

const sc_event&
default_event() const ;

Returns a reference to an event that occurs when new_value on a write is different from current_value.

virtual void
end_of_elaboration();

Sets up tracing of the port.

bool
event() const ;

Returns true if an event occurred in the previous delta-cycle.

void
initialize(const T& val);

Sets current_value to `val`.

void
initialize(const sc_signal_in_if<T>& interface_) ;

Sets current_value to the current_value of the channel argument `interface_`.

virtual const char*
kind() const ;

Returns “sc_inout”.

sc_event_finder† &
neg() const ;

Type `bool` and `sc_logic` only. Returns a reference to an *sc_event_finder*† that occurs when new_value on a write is false and the current_value is not false. For use with static sensitivity list of a process.

```
bool
negedge() const ;
```

Type `bool` and `sc_logic` only. Returns true if an event occurred in the previous delta-cycle and current_value is false.

```
const sc_event&
negedge_event() const ;
```

Type `bool` and `sc_logic` only. Returns a reference to an event that occurs when new_value on a write is false and the current_value is not false.

```
sc_event_finder†&
pos() const ;
```

Type `bool` and `sc_logic` only. Returns a reference to an *sc_event_finder*† that occurs when new_value on a write is true and the current_value is not true. For use with static sensitivity list of a process.

```
bool
posedge() const ;
```

Type `bool` and `sc_logic` only. Returns a reference to an event that occurs when new_value on a write is true and the current_value is not true.

```
const T&
read() const ;
```

Returns a reference to current_value.

```
sc_event_finder†&
value_changed() const ;
```

For use with static sensitivity list for a process. Returns a reference to an *sc_event_finder*† that occurs when new_value on a write is different from current_value. For use with static sensitivity list of a process.

```
const sc_event&
value_changed_event() const ;
```

Returns a reference to an event that occurs when new_value on a write is different from current_value.

```
sc_inout<T>&
write( const T& val ) ;
```

If `val` is not equal to current_value then schedules an update with `val` as new_value.

Public Operators

```
operator const T& () const ;
```


Returns current_value.

```
sc_inout<T>&
operator = ( const Type_& val) ;
Type_ in {T, sc_signal_in_if<T>, sc_port<
    sc_signal_in_if<T> >, sc_port< sc_signal_inout_if<T> >,
    sc_inout<T> }
```

If `val` is not equal to current_value of the left hand side, then an update is scheduled with `val` as the new_value of the left hand side. Returns a reference to the instance.

Disabled Member Function

```
sc_inout( const sc_inout<T>& );
```


11.36 sc_inout_resolved

Synopsis

```
class sc_inout_resolved
    : public sc_inout<sc_logic>
{
public:
    // constructors and destructor
    sc_inout_resolved();
    sc_inout_resolved( const char* name_ );
    sc_inout_resolved(
      sc_signal_inout_if<sc_logic>& interface_ );
    sc_inout_resolved( const char* name_,
      sc_signal_inout_if<sc_logic>& interface_ );
    sc_inout_resolved(
      sc_port<sc_signal_inout_if<sc_logic> >& parent_);
    sc_inout_resolved( const char* name_,
      sc_port<sc_signal_inout_if<sc_logic> >& parent_);
    sc_inout_resolved( sc_inout_resolved& parent_ );
    sc_inout_resolved( const char* name_,
      sc_inout_resolved& parent_ );
    virtual ~sc_inout_resolved();

    // methods
    sc_inout_resolved& operator = ( const
      sc_logic& value_ );
    sc_inout_resolved& operator = ( const
      sc_signal_in_if<sc_Logic>& interface_ );
    sc_inout_resolved& operator = ( const
      sc_port<sc_signal_in_if<sc_logic> >& port_ );
    sc_inout_resolved& operator = ( const
      sc_port<sc_signal_inout_if<sc_logic> >& port_ );
    sc_inout_resolved& operator = ( const
      sc_inout_resolved& port_ );
    virtual void end_of_elaboration();
    static const char* const kind_string;
    virtual const char* kind() const;
private:
    // disabled
    sc_inout_resolved( const sc_inout_resolved& );
};
```

Description

sc_inout_resolved is a specialized port for use with sc_signal_resolved channels (Chapter 11.63). Its behavior is that of a sc_port which has only one interface that is of type sc_signal_inout_if<sc_logic>. It has additional methods for convenience in accessing the channel connected to the port.

Public Constructors

sc_inout_resolved() ;

Create a `sc_inout_resolved` instance.

```
explicit
sc_inout_resolved( const char* );
```

Create a `sc_inout_resolved` instance with the string name initialized to `name_`.

Public Member Functions

```
virtual void
end_of_elaboration() ;
```

Checks to make sure the channel bound to the port is of type `sc_signal_resolved`.

```
virtual const char*
kind() const ;
```

Returns “sc_inout_resolved”.

Public Operators

```
sc_inout_resolved&
operator = ( const Type_& val ) ;
Type_ in {sc_logic, sc_signal_inout_if<sc_logic>, sc_port<
   sc_signal_inout_if <sc_logic> >, sc_inout_resolved& }
```

If `val` is not equal to current_value of the left hand side, then an update is scheduled with `val` as the new_value of the left hand side. Returns a reference to the instance.

Disabled Member Function

```
sc_inout_resolved (const sc_inout_resolved& );
```


11.37 sc_inout_rv

Synopsis

```
template <int W>
class sc_inout_rv
    : public sc_inout<sc_lv<W> >
{
public:
    // constructors and destructor
    sc_inout_rv();
    sc_inout_rv( const char* name_ );
    sc_inout_rv(
      sc_signal_inout_if<sc_lv<W> >& interface_ );
    sc_inout_rv( const char* name_,
      sc_signal_inout_if<sc_lv<W> >& interface_ );
    sc_inout_rv(
      sc_port<sc_signal_inout_if<sc_lv<W> > >& parent_);
    sc_inout_rv( const char* name_,
      sc_port<sc_signal_inout_if<sc_lv<W> > >& parent_);
    sc_inout_rv( sc_inout_rv<W>& parent_ );
    sc_inout_rv( const char* name_,
      sc_inout_rv<W>& parent_ );
    virtual ~sc_inout_rv();

    // methods
    sc_inout_rv<W>& operator = ( const
      sc_lv<W>& value_ );
    sc_inout_rv<W>& operator = ( const
      sc_signal_in_if<sc_lv<W> >& interface_ );
    sc_inout_rv<W>& operator = ( const
      sc_port<sc_signal_in_if<sc_lv<W> > >& port_ );
    sc_inout_rv<W>& operator = ( const
      sc_port<sc_signal_inout_if<sc_lv<W> > >& port_ );
    sc_inout_rv<W>& operator = ( const
      sc_inout_rv<W>& port_ );
    virtual void end_of_elaboration();
    static const char* const kind_string;
    virtual const char* kind() const;
private:
    // disabled
    sc_inout_rv( const sc_inout_rv<W>& );
};
```

Description

sc_inout_rv is a specialized port for use with sc_signal_rv channels (Chapter 11.63). Its behavior is that of a sc_port which has only one interface that is of type sc_signal_inout_if<sc_lv<W> >. It has additional methods for convenience in accessing the channel connected to the port.

In the description of sc_inout_rv, *port* refers to the sc_inout_rv instance.

Public Constructors

sc_inout_rv() ;
Create a sc_inout_rv instance.

explicit
sc_inout_rv(const char*);
Create a sc_inout_rv instance with the string name initialized to name_.

Public Member Functions

virtual void
end_of_elaboration() ;
Checks to make sure the channel bound to the port is of type sc_signal_rv.

virtual const char*
kind() const ;
Returns "sc_inout_rv".

Public Operators

sc_inout_rv<W>&
operator = (const Type_& val) ;
Type_ in {sc_lv<W>, sc_signal_inout_if<T>, sc_port<
sc_signal_inout_if<T>,1>, sc_inout_rv<W> }
If val is not equal to current_value of the left hand side, then an update is scheduled with val as the new_value of the left hand side. Returns a reference to the instance.

Disabled Member Functions

sc_inout_rv(const sc_inout_rv<W>&);

11.38 sc_int

Synopsis

```
template <int W>
class sc_int
    : public sc_int_base
{
public:
    // constructors
    sc_int();
    sc_int( int64 v );
    sc_int( const sc_int<W>& a );
    sc_int( const sc_int_base& a );
    sc_int( const sc_int_subref_r& a );
    template <class T1, class T2>
    sc_int( const sc_int_concref_r<T1,T2>& a );
    sc_int( const sc_signed& a );
    sc_int( const sc_unsigned& a );
    explicit sc_int( const sc_fxval& a );
    explicit sc_int( const sc_fxval_fast& a );
    explicit sc_int( const sc_fxnum& a );
    explicit sc_int( const sc_fxnum_fast& a );
    sc_int( const sc_bv_base& a );
    sc_int( const sc_lv_base& a );
    sc_int( const char* a );
    sc_int( unsigned long a );
    sc_int( long a );
    sc_int( unsigned int a );
    sc_int( int a );
    sc_int( uint64 a );
    sc_int( double a );

    // assignment operators
    sc_int<W>& operator = ( int64 v );
    sc_int<W>& operator = ( const sc_int_base& a );
    sc_int<W>& operator = ( const sc_int_subref_r& a );
    sc_int<W>& operator = ( const sc_int<W>& a );
    template <class T1, class T2>
    sc_int<W>& operator = ( const sc_int_concref_r<T1,T2>&
    a );
    sc_int<W>& operator = ( const sc_signed& a );
    sc_int<W>& operator = ( const sc_unsigned& a );
    sc_int<W>& operator = ( const sc_fxval& a );
    sc_int<W>& operator = ( const sc_fxval_fast& a );
    sc_int<W>& operator = ( const sc_fxnum& a );
    sc_int<W>& operator = ( const sc_fxnum_fast& a );
    sc_int<W>& operator = ( const sc_bv_base& a );
    sc_int<W>& operator = ( const sc_lv_base& a );
    sc_int<W>& operator = ( const char* a );
    sc_int<W>& operator = ( unsigned long a );
    sc_int<W>& operator = ( long a );
    sc_int<W>& operator = ( unsigned int a );
    sc_int<W>& operator = ( int a );
```



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

    // arithmetic assignment operators
    sc_int<W>& operator += ( int64 v );
    sc_int<W>& operator -= ( int64 v );
    sc_int<W>& operator *= ( int64 v );
    sc_int<W>& operator /= ( int64 v );
    sc_int<W>& operator %= ( int64 v );

    // bitwise assignment operators
    sc_int<W>& operator &= ( int64 v );
    sc_int<W>& operator |= ( int64 v );
    sc_int<W>& operator ^= ( int64 v );
    sc_int<W>& operator <<= ( int64 v );
    sc_int<W>& operator >>= ( int64 v );

    // prefix and postfix increment and decrement operators
    sc_int<W>& operator ++ (); // prefix
    const sc_int<W> operator ++ ( int ); // postfix
    sc_int<W>& operator -- (); // prefix
    const sc_int<W> operator -- ( int ); // postfix
};
```

Description

sc_int<W> is an integer with a fixed word length W between 1 and 64 bits. The word length is built into the type and can never change. If the chosen word length exceeds 64 bits, an error is reported and simulation ends. All operations are performed with 64 bits of precision with the result converted to appropriate size through truncation.
Methods allow for addressing an individual bit or a sub range of bits.

Example

```
SC_MODULE(my_module) {
    // data types
  sc_int<3> a;
  sc_int<44> b;
  sc_biguint<88> c;
  sc_biguint<123> d;
    // process
  void my_proc();

  SC_CTOR(my_module) :
    a(0), // init
    c(7654321)  // init
  {
    b = 33;  // set value
    d = 2300; // set value
    SC_THREAD(my_proc);
  }
};
```



```
void my_module::my_proc() {
  a = 1;
  b[30] = a[0];
  cout << b.range(7,0) << endl;

  cout << c << endl;
  d[122] = b;


  wait(300, SC_NS);
  sc_stop();
}
```

Public Constructors

`sc_int();`

Create an `sc_int` instance with an initial value of 0.

`sc_int``( int64 a) ;`

Create an `sc_int` with value a. If the word length of a is greater then W, a gets truncated to W bits.

`sc_int``( T a ) ;`

`T in { sc_int, sc_int_base,` *`sc_int_subref`*†, *`sc_int_concref`*†, *`sc_[un]signed`*†`, sc_fxval, sc_fxval_fast, sc_fix[ed][_fast], sc_bv_base, sc_lv_base, const char*, [unsigned] long, [unsigned] int, int64, double }`

Create an `sc_int` with value a. If the word length of a is greater then W, a gets truncated to W bits.

Copy Constructor

`sc_int``( const sc_int& )`

Methods

`int`
`length``() const ;`

Return the word length.

`void`
`print``( ostream& os = cout ) const ;`

Print the `sc_int` instance to an output stream.

`void`
`scan``( istream& is = cin ) ;`

Read an `sc_int` value from an input stream.

Reduction Methods

`bool` **`and_reduce`**`() const;`
`bool` **`nand_reduce`**`() const ;`


```
bool or_reduce() const ;
bool nor_reduce() const ;
bool xor_reduce() const ;
bool xnor_reduce() const ;
F in { and nand or nor xor xnor }
```

Return the result of function F with all bits of the `sc_int` instance as input arguments.

Assignment Operators

```
sc_int<W>&
operator = ( int64 ) ;

sc_int<W>&
operator = ( T ) ;
T in { sc_int, sc_int_base, sc_int_subref†, sc_int_concref†,
    sc_[un]signed†, sc_fxval, sc_fxval_fast,
    sc_fix[ed][_fast],               sc_bv_base,
    sc_lv_base, const char*, [unsigned] long, [unsigned]
    int, int64, double }
```

Assign the value of the right-hand side to the left-hand side. The value is truncated, if its word length is greater than W.

Arithmetic Assignment Operators

```
sc_int<W>&
operator OP ( int64 ) ;
OP in { += -= *= /= %= }
```

The operation of OP is performed and the result is assigned to the lefthand side. If necessary, the result gets truncated.

Bitwise Assignment Operators

```
sc_int<W>&
operator  OP ( uint64 ) ;
OP in { &= |= ^= <<= >>= }
```

The operation of `OP` is performed and the result is assigned to the left hand side. The result gets truncated.

Prefix and Postfix Increment and Decrement Operators

```
sc_int<W>& operator  ++ () ;
const sc_int<W> operator ++ ( int ) ;
```

The operation of OP is performed as done for type int.

```
sc_int<W>& operator  -- () ;
const sc_int<W> operator  -- ( int ) ;
```

The operation is performed as done for type int.

Relational Operators

```
friend bool operator OP (sc_int, sc_int ) ;
```



```
OP in { == != < <= > >= }
```
These functions return the boolean result of the corresponding equality/ inequality check.

Bit Selection

```
sc_int_bitref†   operator [] ( int i ) ;
sc_int_bitref_r† operator [] ( int i ) const ;
sc_int_bitref†   bit( int i) ;
sc_int_bitref_r† bit( int i) const ;
```

Return a reference to a single bit at index i.

Implicit Conversion

```
operator int64() const
```
Implicit conversion to the implementation type uint64. The value does not change.

Explicit Conversion

```
int64
value() const ;
```
Returns the value without changing it.

```
int      to_int() const ;
double   to_double() const ;
int64    to_int64() const ;
long     to_long() const ;
uint64   to_uint64() const ;
unsigned int    to_uint() const ;
unsigned long   to_ulong() const ;
```

Converts the value of sc_int instance into the corresponding data type. If the requested type has less word length than the sc_int instance, the value gets truncated accordingly. If the requested type has greater word length than the sc_int instance, the value gets sign extended, if necessary.

11.39 sc_int_base

Synopsis

```
class sc_int_base
{
public:
    // constructors & destructors
    explicit sc_int_base( int w = sc_length_param().len() )
    sc_int_base( int64 v, int w )
    sc_int_base( const sc_int_base& a )
    explicit sc_int_base( const sc_int_subref_r& a )
    template <class T1, class T2>
   explicit sc_int_base( const sc_int_concref_r<T1,T2>& a )
    explicit sc_int_base( const sc_signed& a );
    explicit sc_int_base( const sc_unsigned& a );
    ~sc_int_base()

    // assignment operators
    sc_int_base& operator = ( int64 v )
    sc_int_base& operator = ( const sc_int_base& a )
    sc_int_base& operator = ( const sc_int_subref_r& a )
    template <class T1, class T2>
    sc_int_base& operator = ( const
    sc_int_concref_r<T1,T2>& a )
    sc_int_base& operator = ( const sc_signed& a );
    sc_int_base& operator = ( const sc_unsigned& a );
    sc_int_base& operator = ( const sc_fxval& a );
    sc_int_base& operator = ( const sc_fxval_fast& a );
    sc_int_base& operator = ( const sc_fxnum& a );
    sc_int_base& operator = ( const sc_fxnum_fast& a );
    sc_int_base& operator = ( const sc_bv_base& a );
    sc_int_base& operator = ( const sc_lv_base& a );
    sc_int_base& operator = ( const char* a );
    sc_int_base& operator = ( unsigned long a )
    sc_int_base& operator = ( long a )
    sc_int_base& operator = ( unsigned int a )
    sc_int_base& operator = ( int a )
    sc_int_base& operator = ( uint64 a )
    sc_int_base& operator = ( double a )

    // arithmetic assignment operators
    sc_int_base& operator += ( int64 v )
    sc_int_base& operator -= ( int64 v )
    sc_int_base& operator *= ( int64 v )
    sc_int_base& operator /= ( int64 v )
    sc_int_base& operator %= ( int64 v )

    // bitwise assignment operators
    sc_int_base& operator &= ( int64 v )
    sc_int_base& operator |= ( int64 v )
    sc_int_base& operator ^= ( int64 v )
    sc_int_base& operator <<= ( int64 v )
    sc_int_base& operator >>= ( int64 v )
```



```
// prefix and postfix increment and decrement operators
sc_int_base& operator ++ ()
const sc_int_base operator ++ ( int ) // postfix
sc_int_base& operator -- () // prefix
const sc_int_base operator -- ( int ) // postfix

// relational operators
friend bool operator == ( const sc_int_base& a, const
sc_int_base& b )
friend bool operator != ( const sc_int_base& a, const
sc_int_base& b )
friend bool operator <  ( const sc_int_base& a, const
sc_int_base& b )
friend bool operator <= ( const sc_int_base& a, const
sc_int_base& b )
friend bool operator >  ( const sc_int_base& a, const
sc_int_base& b )
friend bool operator >= ( const sc_int_base& a, const
sc_int_base& b )

// bit selection
sc_int_bitref   operator [] ( int i );
sc_int_bitref_r operator [] ( int i ) const;
sc_int_bitref   bit( int i );
sc_int_bitref_r bit( int i ) const;

// part selection
sc_int_subref   operator () ( int left, int right );
sc_int_subref_r operator () ( int left, int right )
const;
sc_int_subref   range( int left, int right );
sc_int_subref_r range( int left, int right ) const;

// bit access
bool test( int i ) const
void set( int i )
void set( int i, bool v )

// Methods
int length() const
bool and_reduce() const;
bool nand_reduce() const
bool or_reduce() const;
bool nor_reduce() const
bool xor_reduce() const;
bool xnor_reduce() const
operator int64() const
int64 value() const
int to_int() const
unsigned int to_uint() const
long to_long() const
unsigned long to_ulong() const
int64 to_int64() const
```



```
    uint64 to_uint64() const
    double to_double() const
    const sc_string to_string( sc_numrep numrep = SC_DEC )
    const;
    const sc_string to_string( sc_numrep numrep, bool
    w_prefix ) const;
    void print( ostream& os = cout ) const
    void scan( istream& is = cin );
};
```

Description

sc_int_base is an integer with a fixed word length between 1 and 64 bits. The word length is set when construction takes place and cannot be changed later.

Public Constructors

explicit
sc_int_base(int = sc_length_param().len());
Create an sc_int_base instance with specified word length. Its initial value is 0.

sc_int_base(int64 a, int b);
Create an sc_int_base instance with value a and word length b.

sc_int_base(T a) ;
T in { *sc_int_subref*†, *sc_int_concref*†, sc_[un]signed }
Create an sc_int_base with value a. The word length of a must not exceed 64 bits. If it does, an error is reported and simulation ends.

Copy Constructor

sc_int_base(const sc_int_base&)

Methods

int
length() const ;
Return the word length.

void
print(ostream& os = cout) const ;
Print the sc_int_base instance to an output stream.

void
scan(istream& is = cin) ;
Read a sc_int_base value from an input stream.

Reduction Methods

bool **and_reduce**() const;
bool **nand_reduce**() const ;


```
bool nor_reduce() const ;
bool or_reduce() const ;
bool xnor_reduce() const ;
bool xor_reduce() const;
F in { and nand or nor xor xnor }
```

Return the result of function F with all bits of the `sc_int_base` instance as input arguments.

Assignment Operators

```
sc_int_base& operator = ( int64 ) ;
sc_int_base& operator = ( T ) ;
T in { sc_int_base, sc_int_subref†, sc_int_concref†,
    sc_[un]signed, sc_fxval, sc_fxval_fast, sc_fxnum,
    sc_fxnum_fast, sc_bv_base, sc_lv_base, char*, [unsigned]
    long, [unsigned] int, uint64, double }
```

Assign the value of the right-hand side to the left-hand side. The value is truncated, if its word length does not fit into the sc_int_base instance on the left hand side . If not, the value is sign extended.

Arithmetic Assignment Operators

```
sc_int_base&
operator OP ( int64 ) ;
OP in { += -= *= /= %= }
```

The operation of OP is performed and the result is assigned to the lefthand side. If necessary, the result gets truncated or sign extended.

Bitwise Assignment Operators

```
sc_int_base&
operator  OP ( int64 ) ;
OP in { &= |= ^= <<= >>= }
```

The operation of OP is performed and the result is assigned to the lefthand side. The result gets truncated or sign extended.

Prefix and Postfix Increment and Decrement Operators

```
sc_int_base<W>& operator  ++ () ;
const sc_int_base<W> operator ++ ( int ) ;
```

The operation is performed as done for type unsigned int.

```
sc_int_base<W>& operator  -- () ;
const sc_int<W> operator  -- ( int ) ;
```

The operation is performed as done for type unsigned int.

Relational Operators

```
friend bool operator OP (sc_int_base, sc_int_base ) ;
OP in { == != < <= > >= }
```

These functions return the boolean result of the corresponding equality/ inequality check.

Bit Selection

```
sc_int_bitref   operator [] ( int i ) ;
sc_int_bitref_r operator [] ( int i ) const ;
sc_int_bitref   bit( int i) ;
sc_int_bitref_r bit( int i) const ;
```

Return a reference to a single bit at index i.

Implicit Conversion

```
operator int64() const ;
```

Implicit conversion to the implementation type `int64`. The value does not change.

Explicit Conversion

```
double   to_double() const ;
int      to_int() const ;
int64    to_int64() const ;
long     to_long() const ;
uint64   to_uint64() const ;
unsigned int    to_uint() const ;
unsigned long   to_ulong() const ;
```

Converts the value of `sc_int_base` instance into the corresponding data type. If the requested type has less word length than the `sc_int_base` instance, the value gets truncated accordingly.

11.40 sc_interface

Synopsis

```
class sc_interface
{
public:
    virtual void register_port( sc_port_base& port_,
            const char* if_typename_ );
    virtual const sc_event& default_event() const;
    virtual ~sc_interface();

protected:
    // constructor
    sc_interface();

private:
    // disabled
    sc_interface( const sc_interface& );
    sc_interface& operator = ( const sc_interface& );
};
```

Description

Class `sc_interface` is the abstract class for interfaces. Users inherit from this class to create their own interfaces. The methods `default_event`() and `register_port`() are “placeholders” for classes that inherit from `sc_interface`. Classes that directly derive from `sc_interface` must do this virtual.

Example

```
// define an interface
class my_if : virtual public sc_interface {
public:
     virtual int read() = 0;
};

// define a channel implementing interface my_if
class  my_ch : public my_if, public sc_channel {
public:
     ...
     virtual int read() { return m_val; }
     virtual const sc_event& default_event() const { return
    m_ev; }
    ...
};
```

Protected Constructor

`sc_interface``();`

Default constructor.

Public Member Functions


```
virtual const sc_event&
default_event() const;
```
Except produce a warning message, does nothing by default. Can be defined by channels.

```
virtual void
register_port( sc_port_base†&, const char* );
```
Does nothing by default. Can be defined by channels for registering ports.

Disabled Member Functions

```
sc_interface( const sc_interface& );
```
Copy constructor.

```
sc_interface& operator = ( const sc_interface& );
```
Default assignment operator.

11.41 sc_length_context

Synopsis

```
typedef sc_context<sc_length_param> sc_length_context;
```

Description

`sc_length_context` manage a stack of `sc_length_param` objects. When a new `sc_length_context` is created, it gets stacked together with its `sc_length_param` object. When the `sc_length_context` leaves scope, it gets destructed, and therefore removed from that global stack.

Public Constructors

```
explicit
sc_length_context( const sc_length_param& a,
    sc_context_begin b = SC_NOW );
```

Create an sc_length_context with sc_length_param a. If b equals SC_NOW, which is the default, it gets pushed onto the global sc_length_context stack. If b equals SC_LATER, it is not pushded onto that stack.

Public Methods

```
void
begin() ;
```

Push the sc_length_context object onto the stack. An sc_length_context must not be pushed more than once onto the stack.

```
void
end() ;
```

Remove the sc_length_context object from the stack. It must be the top most object on that stack.

```
static const sc_length_param&
default_value();
```

Return the default length parameter.

```
const sc_length_param&
value() const;
```

Return the sc_length_param object of the sc_length_context object.

11.42 sc_length_param

Synopsis

```
class sc_length_param
{
public:

    sc_length_param();
    sc_length_param( int );
    sc_length_param( const sc_length_param& );
    explicit sc_length_param( sc_without_context );

    sc_length_param& operator = ( const sc_length_param& );
    friend bool operator == ( const sc_length_param&,
      const sc_length_param& );
    friend bool operator != ( const sc_length_param&,
      const sc_length_param& );

    int len() const;
    void len( int );
    const sc_string to_string() const;
    void print( ostream& = cout ) const;
    void dump( ostream& = cout ) const;
};
```

Description

Instances of sc_length_param define the default word length of newly created sc_[u]int_base, sc_[un]signed, sc_bv_base and sc_lv_base objects. This is especially needed to construct arrays of those data types, because this is the only way to pass the length parameter to these objects.

With the help of sc_length_context objects, sc_length_params are put onto a stack. If, for example, an sc_bv_base is constructed by using its default constructor, which gets the word length from the top element of that stack.

Public Constructors

`sc_length_param``();`

Create an `sc_length_param` with the default word length of 32.

`sc_length_param``( int n ) ;`

Create an `sc_length_param` with n as the word length with `n > 0.`

`explicit`
`sc_length_param``( sc_without_context ) ;`

Create an `sc_length_param` with the default word length of 32.

Copy Constructor

`sc_length_param``( const sc_length_param& );`

Public Methods

```
int
len() const;
```

Get the word length stored in the `sc_length_param`.

```
void
len( int n );
```

Set the word length of the `sc_length_param` to n, with `n > 0`.

```
const sc_string
to_string() const;
```

Convert the `sc_length_param` into its string representation.

```
void
print( ostream& = cout ) const;
```

Print the contents to a stream.

Public Operators

```
sc_length_param&
operator = ( const sc_length_param& a )
```

Assign the word length value of a to the lefthand side `sc_length_param` instance.

```
friend bool
operator == ( const sc_length_param& a, sc_length_param&
    b );
```

Return true if the stored lengths of a and b are equal.

```
friend bool
operator != ( const sc_length_param& a, const
    sc_length_param& b );
```

Return true if the stored `lengths` of a and b are not equal.

11.43 sc_logic

Synopsis

```
class sc_logic
{
public:
    // constructors & destructor
    sc_logic();
    sc_logic( const sc_logic& a );
    sc_logic( sc_logic_value_t v );
    explicit sc_logic( bool a );
    explicit sc_logic( char a );
    explicit sc_logic( int a );
    explicit sc_logic( const sc_bit& a );
    ~sc_logic();

    // assignment operators
    sc_logic& operator = ( const sc_logic& a );
    sc_logic& operator = ( sc_logic_value_t v );
    sc_logic& operator = ( bool a );
    sc_logic& operator = ( char a );
    sc_logic& operator = ( int a );
    sc_logic& operator = ( const sc_bit& a );

    // bitwise assignment operators
    sc_logic& operator &= ( const sc_logic& b );
    sc_logic& operator &= ( sc_logic_value_t v );
    sc_logic& operator &= ( bool b );
    sc_logic& operator &= ( char b );
    sc_logic& operator &= ( int b );
    sc_logic& operator |= ( const sc_logic& b );
    sc_logic& operator |= ( sc_logic_value_t v );
    sc_logic& operator |= ( bool b );
    sc_logic& operator |= ( char b );
    sc_logic& operator |= ( int b );
    sc_logic& operator ^= ( const sc_logic& b );
    sc_logic& operator ^= ( sc_logic_value_t v );
    sc_logic& operator ^= ( bool b );
    sc_logic& operator ^= ( char b );
    sc_logic& operator ^= ( int b );

    // bitwise complement
    const sc_logic operator ~ () const ;
    sc_logic& b_not();

    // bitwise and
    friend const sc_logic operator & ( const sc_logic& a,
    const sc_logic& b );
    friend const sc_logic operator & ( const sc_logic& a,
    sc_logic_value_t b );
    friend const sc_logic operator & ( const sc_logic& a,
    bool b );
```



```
friend const sc_logic operator & ( const sc_logic& a,
char b );
friend const sc_logic operator & ( const sc_logic& a,
int b );
friend const sc_logic operator & ( sc_logic_value_t a,
const sc_logic& b );
friend const sc_logic operator & ( bool a, const
sc_logic& b );
friend const sc_logic operator & ( char a, const
sc_logic& b );
friend const sc_logic operator & ( int a, const
sc_logic& b );

// bitwise or
friend const sc_logic operator | ( const sc_logic& a,
const sc_logic& b );
friend const sc_logic operator | ( const sc_logic& a,
sc_logic_value_t b );
friend const sc_logic operator | ( const sc_logic& a,
bool b );
friend const sc_logic operator | ( const sc_logic& a,
char b );
friend const sc_logic operator | ( const sc_logic& a,
int b );
friend const sc_logic operator | ( sc_logic_value_t a,
const sc_logic& b );
friend const sc_logic operator | ( bool a, const
sc_logic& b );
friend const sc_logic operator | ( char a, const
sc_logic& b );
friend const sc_logic operator | ( int a, const
sc_logic& b );

// bitwise xor
friend const sc_logic operator ^ ( const sc_logic& a,
const sc_logic& b );
friend const sc_logic operator ^ ( const sc_logic& a,
sc_logic_value_t b );
friend const sc_logic operator ^ ( const sc_logic& a,
bool b );
friend const sc_logic operator ^ ( const sc_logic& a,
char b );
friend const sc_logic operator ^ ( const sc_logic& a,
int b );
friend const sc_logic operator ^ ( sc_logic_value_t a,
const sc_logic& b );
friend const sc_logic operator ^ ( bool a, const
sc_logic& b );
friend const sc_logic operator ^ ( char a, const
sc_logic& b );
friend const sc_logic operator ^ ( int a, const
sc_logic& b );

// relational operators and functions
```



```
    friend bool operator == ( const sc_logic& a, const
    sc_logic& b );
    friend bool operator == ( const sc_logic& a,
    sc_logic_value_t b );
    friend bool operator == ( const sc_logic& a, bool b );
    friend bool operator == ( const sc_logic& a, char b );
    friend bool operator == ( const sc_logic& a, int b );
    friend bool operator == ( sc_logic_value_t a, const
    sc_logic& b );
    friend bool operator == ( bool a, const sc_logic& b );
    friend bool operator == ( char a, const sc_logic& b );
    friend bool operator == ( int a, const sc_logic& b );
    friend bool operator != ( const sc_logic& a, const
    sc_logic& b );
    friend bool operator != ( const sc_logic& a,
    sc_logic_value_t b );
    friend bool operator != ( const sc_logic& a, bool b );
    friend bool operator != ( const sc_logic& a, char b );
    friend bool operator != ( const sc_logic& a, int b );
    friend bool operator != ( sc_logic_value_t a, const
    sc_logic& b );
    friend bool operator != ( bool a, const sc_logic& b );
    friend bool operator != ( char a, const sc_logic& b );
    friend bool operator != ( int a, const sc_logic& b );

    // explicit conversions
    sc_logic_value_t value() const ;
    bool is_01()const ;
    bool to_bool()const ;
    char to_char()const ;

    // other methods
    void print( ostream& os = cout ) const ;
    void scan( istream& is = cin );

    // memory (de);allocation
    static void* operator new( size_t, void* p ); //
    placement new
    static void* operator new( size_t sz );
    static void operator delete( void* p, size_t sz );
    static void* operator new [] ( size_t sz );
    static void operator delete [] ( void* p, size_t sz );

private:
    // disabled
    explicit sc_logic( const char* );
    sc_logic& operator = ( const char* );
};
```

Description

Instances of type `sc_logic` can have the values shown in Table 28.

Table 28 – sc_logic Values

Type	Values			
sc_logic_value_t	Log_0	Log_1	Log_Z	Log_X
bool	false	true	n/a	n/a
int	0	1	n/a	n/a
char	‘0’	‘1’	‘Z’	‘X’

Values of types not found in Table 28 (`sc_logic_value_t, bool, int, char`) produce undefined behavior.

Public Constructors

```
sc_logic() ;
sc_logic( sc_logic ) ;
sc_logic( sc_logic_value_t ) ;
explicit
sc_logic( T ) ;
T in { sc_logic, bool, int, char }
```

If not otherwise specified, an `sc_logic` is initialized with `Log_X`.

General functions

```
bool
is_01() const ;
```

Return true if the sc_logic instance is either Log_0 or Log_1, else return false.

```
friend ostream&
operator << ( ostream&, sc_logic ) ;
```

Print the value of the `sc_logic` object to a stream.

```
friend istream&
operator >> ( istream&, sc_logic& ) ;
```

Read the next value from a stream.

```
void
print( ostream& os = cout ) const ;
```

Print the value of the `sc_logic` object to a stream.

```
void
scan( istream& is = cin ) ;
```

Read the next value from a stream.

```
bool
to_bool() const ;
```

Explicit conversion.to type `bool`.

```
char
to_char() const ;
```

Explicit conversion.to type `char`

```
sc_logic_value_t
```


value() const ;
Explicit conversion.to type `sc_logic_value_t`. Value remains the same.

Assignment Operators

```
sc_logic& operator = ( sc_logic_value_t ) ;
sc_logic& operator = ( sc_bit ) ;
sc_logic& operator = ( T ) ;
T in { sc_logic, bool, int, char }
```

Bitwise Assignment Operators

```
sc_logic& operator &= ( sc_logic_value_t v ) ;
sc_logic& operator &= ( T ) ;

sc_logic& operator |= ( sc_logic_value_t v ) ;
sc_logic& operator |= ( T ) ;

sc_logic& operator ^= ( sc_logic_value_t v ) ;
sc_logic& operator ^= ( T ) ;

T in { sc_logic, bool, int, char }
```

These operators calculate the four logic value of the AND, OR and XOR function and assign the result to the left-hand side.

Bitwise complement

```
const sc_logic operator ~ () const ;
sc_logic& b_not() ;
```

Bitwise AND

```
friend const sc_logic operator & ( sc_logic,
    sc_logic_value_t ) ;
friend const sc_logic operator & ( sc_logic_value_t,
    sc_logic ) ;
friend const sc_logic operator & ( sc_logic, T ) ;
friend const sc_logic operator & ( T, sc_logic ) ;
T in { sc_logic, bool, int, char }
```

Bitwise OR

```
friend const sc_logic operator | ( sc_logic,
    sc_logic_value_t ) ;
friend const sc_logic operator | ( sc_logic_value_t,
    sc_logic ) ;
friend const sc_logic operator | ( sc_logic, T ) ;
friend const sc_logic operator | ( T, sc_logic ) ;
T in { sc_logic, bool, int, char }
```

Bitwise XOR

```
friend const sc_logic operator ^ ( sc_logic,
    sc_logic_value_t ) ;
friend const sc_logic operator ^ ( sc_logic_value_t,
    sc_logic ) ;
```



```
friend const sc_logic operator ^ ( sc_logic, T ) ;
friend const sc_logic operator ^ ( T, sc_logic ) ;
T in { sc_logic, bool, int, char }
```

Test for equality:

```
friend bool operator == ( sc_logic, sc_logic_value_t ) ;
friend bool operator == ( sc_logic_value_t, sc_logic ) ;
friend bool operator == ( sc_logic, T ) ;
friend bool operator == ( T, sc_logic ) ;
T in { sc_logic, bool, int, char }
```

Test for inequality:

```
friend bool operator != ( sc_logic, sc_logic_value_t ) ;
friend bool operator != ( sc_logic_value_t, sc_logic ) ;
friend bool operator != ( sc_logic, T ) ;
friend bool operator != ( T, sc_logic ) ;
T in { sc_logic, bool, int, char }
```

Disabled Member Functions

```
explicit
sc_logic( const char* );

sc_logic&
operator = ( const char* );
```


11.44 sc_lv

Synopsis

```
template <int W>
class sc_lv
    : public sc_lv_base
{
public:
    // constructors
    sc_lv();
    explicit sc_lv( const sc_logic& init_value );
    explicit sc_lv( bool init_value );
    explicit sc_lv( char init_value );
    sc_lv( const char* a );
    sc_lv( const bool* a );
    sc_lv( const sc_logic* a );
    sc_lv( const sc_unsigned& a );
    sc_lv( const sc_signed& a );
    sc_lv( const sc_uint_base& a );
    sc_lv( const sc_int_base& a );
    sc_lv( unsigned long a );
    sc_lv( long a );
    sc_lv( unsigned int a );
    sc_lv( int a );
    sc_lv( uint64 a );
    sc_lv( int64 a );
    template <class X>
    sc_lv( const sc_bv_base& a );
    sc_lv( const sc_lv<W>& a );

    // assignment operators
    template <class X>
    sc_lv<W>& operator = ( const sc_bv_base& a );
    sc_lv<W>& operator = ( const sc_lv<W>& a );
    sc_lv<W>& operator = ( const char* a );
    sc_lv<W>& operator = ( const bool* a );
    sc_lv<W>& operator = ( const sc_logic* a );
    sc_lv<W>& operator = ( const sc_unsigned& a );
    sc_lv<W>& operator = ( const sc_signed& a );
    sc_lv<W>& operator = ( const sc_uint_base& a );
    sc_lv<W>& operator = ( const sc_int_base& a );
    sc_lv<W>& operator = ( unsigned long a );
    sc_lv<W>& operator = ( long a );
    sc_lv<W>& operator = ( unsigned int a );
    sc_lv<W>& operator = ( int a );
    sc_lv<W>& operator = ( uint64 a );
    sc_lv<W>& operator = ( int64 a );
};
```

Description

sc_lv< W > is a four value logic vector of arbitrary length. Its length is built into the type and can not change later.

Pointer arguments are arrays. In the case of 'const bool*' and 'const sc_logic*' the size has to be at least as large as the length of the bit vector.

Examples

```
sc_lv<38> a;  // 38-bit bit vector
sc_lv<4> b;
b = "ZZZZ";
```

Public Constructors

```
sc_lv() ;
```

Create an sc_lv with all bits set to Log_X.

explicit
```
sc_lv( bool a ) ;
```

Create an sc_lv with all bits set to a.

explicit
```
sc_lv( char a ) ;
```

Create an sc_lv with all bits set to a. a can be '0', '1', 'Z' or 'X'.

```
sc_lv( T a ) ;
T in { const char*, const bool*, const sc_logic*, const
    sc_unsigned&, const sc_signed&, const sc_uint_base†&,
    const sc_int_base&, [unsigned] long, [unsigned] int,
    [u]int64 }
```

Create an sc_lv with the converted contents of a. If the length of a is greater than the length of sc_lv, a gets truncated. If the length of a is less than the length of sc_lv, the MSBs get padded with Log_0.

Copy Constructor

```
sc_lv( const sc_lv<W>& ) ;
```

Assignment Operators

```
sc_lv<W>& operator = ( const sc_lv<W>& a ) ;
sc_lv<W>& operator = ( T a ) ;
T in { const char*, const bool*, const sc_logic*, const
    sc_unsigned&, const sc_signed&, const sc_uint_base†&,
    const sc_int_base&, unsigned long, long, unsigned int,
    int, [u]int64 }
```

The value of the right handside is assigned to the sc_lv. If the length of a is greater than the length of sc_lv, a gets truncated. If the length of a is less than the length of sc_lv, the MSBs get padded with Log_0.

11.45 sc_lv_base

Synopsis

```
class sc_lv_base
{
public:
    // constructors & destructors
    explicit sc_lv_base( int length_ =
    sc_length_param().len() );
    explicit sc_lv_base( const sc_logic& a,
      int length_ = sc_length_param().len()  );
    sc_lv_base( const char* a );
    sc_lv_base( const char* a, int length_ );
    template <class X>
    sc_lv_base( const sc_bv_base& a );
    sc_lv_base( const sc_lv_base& a );
    virtual ~sc_lv_base();

    // assignment operators
    template <class X>
    sc_lv_base& operator = ( const sc_bv_base& a );
    sc_lv_base& operator = ( const sc_lv_base& a );
    sc_lv_base& operator = ( const char* a );
    sc_lv_base& operator = ( const bool* a );
    sc_lv_base& operator = ( const sc_logic* a );
    sc_lv_base& operator = ( const sc_unsigned& a );
    sc_lv_base& operator = ( const sc_signed& a );
    sc_lv_base& operator = ( const sc_uint_base& a );
    sc_lv_base& operator = ( const sc_int_base& a );
    sc_lv_base& operator = ( unsigned long a );
    sc_lv_base& operator = ( long a );
    sc_lv_base& operator = ( unsigned int a );
    sc_lv_base& operator = ( int a );
    sc_lv_base& operator = ( uint64 a );
    sc_lv_base& operator = ( int64 a );

    // Methods
    int length() const;
    bool is_01() const;
};
```

Description

sc_lv_base is a vector of four value logic values of arbitary length. Its length is set at construction and can not be changed later.

For `sc_lv_base` description:

A bit means a four value logic bit.

T in { const char*, const bool*, const sc_logic*, const
    sc_unsigned&, const sc_signed&, const sc_uint_base&,


```
const sc_int_base&, unsigned long, long, unsigned int,
int, uint64, int64 }
```

Pointer arguments are arrays. In the case of 'const bool*' and 'const sc_logic*' the size has to be at least as large as the length of the bit vector.

Public Constructors

```
explicit
sc_lv_base( int = sc_length_param().len() ) ;
```

Create an sc_lv_base of specified length. All bits are set to Log_X.

```
explicit
sc_lv_base( const sc_logic& a, int =
    sc_length_param().len() ) ;
```

Create an sc_lv_base of specified length. All bits are set to a.

```
sc_lv_base( const char* a ) ;
```

Create an sc_lv_base with the contents of a. The character string a must be convertible into a bit string. The length of the newly created sc_lv_base is identical to the length of the bit value representation of a.

```
sc_lv_base( const char *a, int i ) ;
```

Create an sc_lv_base with the contents of a. The character string a must be convertible into a bit string. The length of the bit vector is determined by i. If the length of a is less than i, the MSBs are set to Log_X. If the length of a is greater than i, the MSBs are truncated.

Copy Constructor

```
sc_lv_base( const sc_lv_base& ) ;
```

Methods

```
bool
is_01() const ;
```

Return true, if all bits are Log_0 or Log_1.

```
int
length() const
```

Return the length of the bit vector.

```
void
print( ostream& os = cout ) const ;
```

Print the sc_bv_base instance to an output stream.

```
void
scan( istream& is = cin ) ;
```

Read an sc_bv_base value from an input stream.

Assignment Operators

sc_lv_base& **operator =** (const sc_lv_base&)
sc_lv_base& **operator =** (T)

The value of the right-hand side is assigned to the left-hand side. If the lengths of the two operands are different, the right-hand side gets either truncated or sign extended.

Bitwise Operators

sc_lv_base& **operator &=** (T) ;
Calculate the bitwise AND operation and assign the result to the left-hand side. Both operands have to be of equal length.

const sc_lv_base **operator &** (T) const ;
Return the result of the bitwise AND operation. Both operands have to be of equal length.

sc_lv_base& **operator |=** (T) ;
Calculate the bitwise OR operation and assign the result to the left-hand side. Both operands have to be of equal length.

const sc_lv_base **operator |** (T) const ;
Return the result of the bitwise OR operation. Both operands have to be of equal length.

sc_lv_base& **operator ^=** (T) ;
Calculate the bitwise XOR operation and assign the result to the left-hand side. Both operands have to be of equal length.

const sc_lv_base **operator ^** (T) const ;
Return the result of the bitwise XOR operation. Both operands have to be of equal length.

sc_lv_base& **operator <<=** (int i) ;
Shift the contents of the left hand side operand i bits to the left and assign the result to the left hand side operand. i must not be negative. Log_0 values are inserted at the LSB side.

const sc_lv_base **operator <<** (int i) const ;
Shift the contents of the left-hand side operand i bits to the left and return the result. i must not be negative. Log_0 bits are inserted at the LSB side.

sc_lv_base& **operator >>=** (int i) ;
Shift the contents of the left-hand side operand i bits to the right and assign the result to the left-hand side operand. i must not be negative. Log_0 values are inserted at the MSB side.

const sc_lv_base **operator >>** (int i) const ;

Shift the contents of the left-hand side operand i bits to the right and return the result. i must not be negative. Log_0 bits are inserted at the MSB side.

Bitwise Rotation & Reverse Methods

```
sc_lv_base&
lrotate( int i ) ;
```

Rotate the contents of the bit vector `i` bits to the left.

```
sc_lv_base&
rrotate( int i ) ;
```

Rotate the contents of the bit vector `i` bits to the right.

```
sc_lv_base&
reverse() ;
```

Reverse the contents of the bit vector. LSB becomes MSB and vice versa.

Bit Selection

```
sc_bitref†<sc_lv_base>   operator [] ( int i ) ;
sc_bitref_r†<sc_lv_base> operator [] ( int i ) const ;
sc_bitref†<sc_lv_base>   bit( int i ) ;
sc_bitref_r†<sc_lv_base> bit( int i ) const ;
```

Return a reference to the i-th bit. Return an r-value if the logic vector is constant.

Part Selection

```
sc_subref<sc_lv_base>   operator () ( int, int ) ;
sc_subref_r<sc_lv_base> operator () ( int, int ) const ;
sc_subref<sc_lv_base>   range( int, int ) ;
sc_subref_r<sc_lv_base> range( int, int ) const ;
```

Return a reference to a range of bits. Return an r-value if the logic vector is constant.

Reduction Methods

```
sc_logic_value_t and_reduce() const ;
sc_logic_value_t nand_reduce() const ;
sc_logic_value_t or_reduce() const ;
sc_logic_value_t nor_reduce() const ;
sc_logic_value_t xor_reduce() const ;
sc_logic_value_t xnor_reduce() const ;
```


Return the result of function F with all bits of the logic vector as input arguments.

```
F in { and nand or nor xor xnor }
```

Relational Operators

```
bool operator == ( T ) const ;
```

Return true if the two logic vectors are equal.

Explicit Conversion

```
int      to_int() const ;
long     to_long() const ;
unsigned int    to_uint() const ;
unsigned long   to_ulong() const ;
```

Convert the logic vector into an int, unsigned int, long or unsigned long respectively. The LSB of the logic vector is put into the LSB of the returned value, etc.

Explicit Conversion to Character String

```
const sc_string to_string() const ;
```

Convert the logic vector into a string representing its contents. Every character represents a logic value. MSBs are on the left.

```
const sc_string to_string( sc_numrep nr ) const ;
```

Convert the logic vector into a string representing its contents. The nr argument specifies the base of the number string. A prefix ensures that the string can be read back without changing the value.

```
const sc_string to_string( sc_numrep, bool prefix ) const ;
```

Convert the logic vector into a string representing its contents. The nr argument specifies the base of the number string. A prefix ensures that the string can be read back without changing the value. If prefix is false, no prefix is pre-pended to the value string.

11.46 sc_module

Synopsis

```
class sc_module
: public sc_object
{
protected:
    virtual void end_of_elaboration();

    // constructors
    sc_module( const char* nm );
    sc_module( const sc_string& nm );
    sc_module( const sc_module_name& nm );  sc_module();
public:
    // destructor
    virtual ~sc_module();

   const sc_pvector<sc_object*>& get_child_objects() const;

protected:
    void dont_initialize();
    void wait();

    // dynamic sensitivity for SC_THREADs and SC_CTHREADs
    void wait( const sc_event& e );
    void wait( sc_event_or_list& el );
    void wait( sc_event_and_list& el );
    void wait( const sc_time& t );
    void wait( double v, sc_time_unit tu );
    void wait( const sc_time& t, const sc_event& e );
    void wait( double v, sc_time_unit tu, const sc_event&
    e );
    void wait( const sc_time& t, sc_event_or_list& el );
    void wait( double v, sc_time_unit tu, sc_event_or_list&
    el );
    void wait( const sc_time& t, sc_event_and_list& el );
    void wait( double v, sc_time_unit tu,
    sc_event_and_list& el );

    // static sensitivity for SC_METHODs
    void next_trigger();

    // dynamic sensitivity for SC_METHODs
    void next_trigger( const sc_event& e );
    void next_trigger( sc_event_or_list& el );
    void next_trigger( sc_event_and_list& el );
    void next_trigger( const sc_time& t );
    void next_trigger( double v, sc_time_unit tu );
    void next_trigger( const sc_time& t, const sc_event&
    e );
    void next_trigger( double v, sc_time_unit tu, const
    sc_event& e );
```



```
    void next_trigger( const sc_time& t,    void
    next_trigger( double v, sc_time_unit tu,
    sc_event_or_list& el );
    void next_trigger( const sc_time& t, const sc_event&
    e );
    void next_trigger( double v, sc_time_unit tu,
    sc_event_and_list& el );

    sc_sensitive  sensitive;
    void set_stack_size( size_t );
public:
    // positional binding methods (cont'd)
    void operator () ( const sc_bind_proxy& p001,
      const sc_bind_proxy& p002 = SC_BIND_PROXY_NIL,
      . . .
      const sc_bind_proxy& p064 = SC_BIND_PROXY_NIL );
};
```

Description

An `sc_module` is the base class for modules. Users inherit from this class to create their own modules.

The `wait()` and `next_trigger()` methods provide for static and dynamic sensitivity for processes. Refer to Chapters 9.3 , 9.4.1 and 9.5.1.

In the description of `sc_module`, *module* refers to the sc_module instance. Chapter 2.4.1 describes the scheduler steps referred to in the description of `sc_module`.

Protected Constructors

`sc_module``( const char* name_) ;`
Create a `sc_module` instance with the instance name initialized to `name_`.

`sc_module``( const sc_string& name_) ;`
Create a `sc_module` instance with the instance name initialized to `name_`.

`sc_module``( const sc_module_name& name_) ;`
Create an `sc_module` instance with the instance name initialized to `name_`.

`sc_module``( ) ;`
Create an `sc_module` instance. The instance name will be obtained from the module name stack via the construction of the sc_module_name object passed to the derived module class's constructor.

Protected Member functions

`void`
`dont_initialize``();`
Prevents initialization of SC_METHODs and SC_THREADs. This method is typically invoked in the module constructor immediately after SC_METHOD

or SC_THREAD statements, and indicates that the specified thread or method should not be triggered by default at the beginning of simulation.

`virtual void`
`end_of_elaboration``();`

This virtual method is automatically invoked at the end of elaboration phase at the point where all modules and channels have been instantiated and before simulation is started. By default this method does nothing, but users can override the default implementation to perform user-defined actions at the end of elaboration.

Protected Data Members

`sc_sensitive sensitive;`

Provides the object through which process sensitivities are specified, using its << and () operators. The calls to these operators must occur before the start of simulation, thus these operators are typically used in module constructors. When event sensitivity is specified using this form, the process that was most recently declared is made statically sensitive to the specified events.

Protected Member Functions for Process Sensitivity

`bool`
`timed_out``();`

Returns true if the triggering of a process was based on the time out value of a wait() or next_trigger() method else returns false.

`void`
`next_trigger``();`

Sets the calling process to be triggered based upon its static sensitivity list.

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

Sets the calling_process to be triggered based upon `type_` (dynamic sensitivity).

`void`

next_trigger(double t_out_val , sc_time_unit t_out_tu,
    type_);

type_ in { const sc_event&, *sc_event_or_list*†&,
    sc_event_and_list†& }

Sets the calling_process to be triggered based upon either the time out (t_out_val, t_out_tu) or type_.

void
next_trigger(const sc_time& t_out, type_);

type_ in { const sc_event&, *sc_event_or_list*†&,
    sc_event_and_list†& }

Sets the calling_process to be triggered based upon either the time out (t_out) or type_.

void
wait() ;

Suspends the calling process. Calling process is triggered based upon its static sensitivity list.

void
wait(type_);

type_ in { const sc_event&, *sc_event_or_list*†&,
    sc_event_and_list†&, (double, sc_time_unit), const
    sc_time& }

Sets the calling_process to be triggered based upon type_ (dynamic sensitivity).

void
wait(double, sc_time_unit, type_);

type_ in { const sc_event&, *sc_event_or_list*†&,
    sc_event_and_list†& }

Sets the calling_process to be triggered based upon either the time out sc_time(double, sc_time_unit) or type_.

void
wait(const sc_time&, type_);

type_ in { const sc_event&, *sc_event_or_list*†&,
    sc_event_and_list†& }

Sets the calling_process to be triggered based upon either the time out sc_time(double, sc_time_unit) or type_.

Public Operators

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

Positionally bind one or more ports or interfaces to the ports of the specified module instance. No more than 64 ports or interfaces can be specified using this form. If you need to bind more than 64 ports or interfaces, use named port binding instead.

11.47 sc_module_name

Synopsis

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

    ~sc_module_name();

    operator const char*() const;
private:
    // disabled
    sc_module_name();
    sc_module_name& operator = ( const sc_module_name& );
};
```

Description

The **sc_module_name** class serves two purposes. Firstly, instances of sc_module_name are passed to module constructors to provide instance names for all modules within the design hierarchy. Secondly, sc_module_name instances help SystemC determine when classes derived from sc_module have started and completed construction. SystemC needs to know when sc_module classes have started and completed construction in order to properly associate child objects such as ports with their containing module instance.

Constructors for classes derived from sc_module should have one constructor argument of this type. Furthermore, when such classes from sc_module are instantiated, a normal C string should be passed to the derived class constructor, which will then be converted to sc_module_name via an implicit conversion. The execution of this implicit conversion informs SystemC that a new module has started construction, and later the destruction of the same sc_module_name object informs SystemC that the construction of a module has completed.

It should be emphasized that while sc_module_name must be used within the declaration of constructor arguments for classes derived from sc_module, users should never explicitly instantiate any sc_module_name objects.

Example

```
class my_module : public sc_module {
public:
    int some_parameter;
    SC_HAS_PROCESS(my_module);

    my_module (sc_module_name name, int some_value):
      sc_module(name),
```



```
        some_parameter(some_value)
    {
        // constructor body not shown
    }
    // rest of module body not shown
};
```

Public Constructors

sc_module_name(const char *name);

Constructs an sc_module_name object from a C string.

sc_module_name(const sc_module_name& orig_);

Copy constructor.

Disabled Constructors

sc_module_name( );

The default constructor is disabled.

Public Operators

operator **const char ***() const;

Provides an implicit type conversion to a constant character string.

11.48 sc_mutex

Synopsis

```
class sc_mutex
: public sc_mutex_if,
  public sc_prim_channel
{
public:
    // constructors
    sc_mutex();
    explicit sc_mutex( const char* name_ );

    // interface methods
    virtual int lock();
    virtual int trylock();
    virtual int unlock();
    static const char* const kind_string;
    virtual const char* kind() const
protected:
    // support methods
    bool in_use() const
private:
    // disabled
    sc_mutex( const sc_mutex& );
    sc_mutex& operator = ( const sc_mutex& );
};
```

Description

An **`sc_mutex`** channel (mutex) is used for a mutual-exclusion lock for access to a shared resource. It implements the `sc_mutex_if` interface.

A process may lock the mutex. Only the process that locked the mutex may unlock it.

If multiple processes attempt to lock an unlocked mutex during the same delta-cycle, only one will be successful. Since the order of execution of processes in a delta-cycle is indeterminate it is indeterminate as to which process is successful. The unsuccessful processes will be suspended as described in the next paragraph.

If a process attempts to lock the mutex, when it is already locked, then the process is suspended. When the mutex is unlocked then the suspended process is triggered and continues the attempt to lock the mutex. The unsuspended process is not guaranteed to be successful in locking the mutex if there are other processes also attempting to lock the mutex.

Public Constructors

`sc_mutex``();`

Create an `sc_mutex` instance.

```
explicit
sc_mutex( const char* name_ );
```

Create an `sc_mutex` instance with the string name initialized to `name_`.

Public Member Functions

```
virtual const char*
kind() const ;
```

Returns string “sc_mutex”.

```
Virtual int
lock() ;
```

Returns 0. If the mutex is not locked then locks mutex else suspends the calling process.

```
virtual int
trylock();
```

If the mutex is not locked then locks mutex and returns 0, else returns -1.

```
virtual int
unlock();
```

If mutex was locked by calling process then unlocks mutex, triggers any processes suspended while attempting to lock the mutex and returns 0, else returns -1.

Disabled Member Functions

```
sc_mutex( const sc_mutex& );

sc_mutex&
operator = ( const sc_mutex& );
```


11.49 sc_mutex_if

Synopsis

```
class sc_mutex_if
: virtual public sc_interface
{
public:
    virtual int lock() = 0;
    virtual int trylock() = 0;
    virtual int unlock() = 0;
protected:
    // constructor
    sc_mutex_if();
private:
    // disabled
    sc_mutex_if( const sc_mutex_if& );
    sc_mutex_if& operator = ( const sc_mutex_if& );
};
```

Description

The **sc_mutex_if** class provides the signatures of the functions for the sc_mutex_if interface. See Chapter 8.1 for a description of interfaces. Implemented by the sc_mutex channel (Chapter 11.12)

Example

```
class sc_mutex
: public sc_mutex_if,
  public sc_prim_channel
{  . . . . };
```

Protected Constructor

sc_mutex_if();

Create a sc_mutex_if instance.

Public Member Functions

virtual int
lock() = 0;

virtual int
trylock() = 0;

virtual int
unlock() = 0;

Disabled Member Functions

sc_mutex_if(const sc_mutex_if&);

sc_mutex_if&
operator = (const sc_mutex_if&);

11.50 sc_object

Synopsis

```
class sc_object
{
public:
    const char* name() const;
    const char* basename() const;
    void print() const;
    virtual void print( ostream& os ) const;
    void dump() const;
    virtual void dump( ostream& os ) const;
    virtual void trace( sc_trace_file* ) const;
    virtual const char* kind() const;
    sc_simcontext* simcontext() const ;
    bool add_attribute( sc_attr_base& );
    sc_attr_base* get_attribute( const sc_string& );
    const sc_attr_base* get_attribute( const sc_string& )
   const;
    sc_attr_base* remove_attribute( const sc_string& );
    void remove_all_attributes();
    int num_attributes() const;
    sc_attr_cltn& attr_cltn();
    const sc_attr_cltn& attr_cltn() const;
protected:
    sc_object();
    sc_object(const char*);
    virtual ~sc_object();
};
```

Description

sc_object is the abstract base class for all channel, module, port and process objects.

Protected Constructors and Destructor

sc_object();

Default constructor. Creates a sc_object instance.

sc_object(const char* name_);

Creates a sc_object instance with the string name initialized to name_.

virtual **~sc_object**();
Virtual destructor.

Public Member Functions

bool
add_attribute(*sc_attr_base*[†]&) ;

Adds an attribute to a collection stored in the object. Returns true if the attribute name is unique, false otherwise. If the name is not unique, the attribute is not added to the collection.


```
sc_attr_cltn&
attr_cltn();
```
Returns a reference to the collection of attributes of this object.

```
const sc_attr_cltn&
attr_cltn() const;
```
Returns a constant reference to the collection of attributes of this object.

```
const char*
basename() const ;
```
Returns the string name of the instance without hierarchical path name.

```
void
print() const ;
```
Prints the string name.

```
virtual void
print(ostream& os) const ;
```
Prints the string name to output stream `os`.

```
void
dump() const ;
```
Prints the string name and the kind.

```
virtual void
dump(ostream& os) const ;
```
Prints the string name and the kind to an output stream `os`.

```
sc_attr_base†*
get_attribute( const sc_string&_ ) ;
```
Returns a constant pointer to the named attribute of the object. If the attribute with this name is not found, returns 0.

```
const sc_attr_base†*
get_attribute( const sc_string& ) const ;
```
Returns a pointer to the named attribute of the object. If the attribute with this name is not found, returns 0.

```
virtual const char*
kind() const ;
```
Returns “sc_object”.

```
const char*
name() const ;
```
Returns the string name of the instance with hierarchical path name.

```
int
num_attributes() const;
```
Returns the number of attributes attached to this object.


```
sc_attr_base†*
remove_attribute( const sc_string& ) ;
```
Removes the named attribute from this object. Returns a pointer to the attribute. If the attribute with this name is not found, returns 0.

```
void
remove_all_attributes();
```
Removes all attributes from this object.

```
sc_simcontext*
simcontext() const ;
```
Returns a pointer to the simulation context of the object.

```
virtual void
trace( sc_trace_file* tf ) const ;
```
Does nothing.

11.51 sc_out

Synopsis

```
template <class T>
class sc_out
: public sc_inout<T>
{
public:
    // typedefs
    typedef T data_type;
    typedef sc_out<data_type>this_type;
    typedef sc_inout<data_type> base_type;
    typedef typename base_type::in_if_type in_if_type;
    typedef typename base_type::in_port_type in_port_type;
   typedef typename base_type::inout_if_type inout_if_type;
    typedef typename base_type::inout_port_type
    inout_port_type;
public:
    // constructors & destructor
    sc_out();
    explicit sc_out( const char* name_ );
    explicit sc_out( inout_if_type& interface_ );
    sc_out( const char* name_, inout_if_type& interface_ );
    explicit sc_out( inout_port_type& parent_ );
    sc_out( const char* name_, inout_port_type& parent_ );
    sc_out( this_type& parent_ );
    sc_out( const char* name_, this_type& parent_ );
    virtual ~sc_out();

    this_type& operator = ( const data_type& value_ );
    this_type& operator = ( const in_if_type& interface_ );
    this_type& operator = ( const in_port_type& port_ );
    this_type& operator = ( const inout_port_type& port_ );
    this_type& operator = ( const this_type& port_ );
    static const char* const kind_string;
    virtual const char* kind() const ;
private:
    // disabled
    sc_out( const this_type& );
};
```

Description

sc_out is a specialized port for use with sc_signal channels (Chapter 11.59). Its behavior is that of a sc_port which has only one interface that is of type sc_signal_inout_if<T>. It has the same functionality as an sc_inout port.

In the description of sc_in, *current_value* refers to the value of the sc_signal instance connected to the port, *new_value* is the value to be written and *old_value* is the previous value. Chapter 2.4.1 describes the scheduler steps referred to in the description of sc_inout.

Public Constructors

sc_out() ;

Create a `sc_out` instance.

explicit
sc_out(const char* name_) ;

Create a `sc_out` instance with the string name initialized to `name_`.

Public Member Functions

virtual const char*
kind() const ;

Returns “sc_out”.

Assignment Operator

operator const T& **()** const ;

Returns current_value.

sc_inout<T>&
operator = (const Type_& val) ;
Type_ in {T, sc_signal_in_if<T>, sc_port<
sc_signal_in_if<T> >, sc_port< sc_signal_inout_if<T> >,
sc_out<T> }

If `val` is not equal to current_value of the left hand side, then an update is scheduled with `val` as the new_value of the left hand side. Returns a reference to the instance.

Disabled Member Functions

sc_out(const sc_out<T>&);

11.52 sc_out_resolved

Synopsis

```
class sc_out_resolved
    : public sc_inout_resolved
{
public:
    // typedefs
    typedef sc_out_resolved            this_type;
    typedef sc_inout_resolved          base_type;
    typedef base_type::data_type       data_type;
    typedef base_type::in_if_type      in_if_type;
    typedef base_type::in_port_type    in_port_type;
    typedef base_type::inout_if_type   inout_if_type;
    typedef base_type::inout_port_type inout_port_type;
public:
    // constructors & destructor
    sc_out_resolved();
    explicit sc_out_resolved( const char* name_ );
    explicit sc_out_resolved( inout_if_type& interface_ );
    sc_out_resolved( const char* name_, inout_if_type&
    interface_ );
    explicit sc_out_resolved( inout_port_type& parent_ );
    sc_out_resolved( const char* name_, inout_port_type&
    parent_ );
    sc_out_resolved( this_type& parent_ );
    sc_out_resolved( const char* name_, this_type&
    parent_ );
    virtual ~sc_out_resolved();

    // Methods
    this_type& operator = ( const data_type& value_ );
    this_type& operator = ( const in_if_type& interface_ );
    this_type& operator = ( const in_port_type& port_ );
    this_type& operator = ( const inout_port_type& port_ );
    this_type& operator = ( const this_type& port_ );
    static const char* const kind_string;
    virtual const char* kind() const;
private:
    // disabled
    sc_out_resolved( const this_type& );
};
```

Description

sc_out_resolved is a specialized port for use with sc_signal_resolved channels (Chapter 11.63). Its behavior is that of a sc_port which has only one interface that is of type sc_signal_inout_if<sc_logic>. It has the same functionality as an sc_inout_resolved port.

Public Constructors

sc_out_resolved() ;

Create a `sc_inout_resolved` instance.

```
explicit
sc_out_resolved( const char* );
```

Create a `sc_inout_resolved` instance with the string name initialized to `name_`.

Public Member Functions

```
virtual const char*
kind() const ;
```

Returns “sc_out_resolved”.

Assignment Operator

```
sc_out_resolved&
operator = ( const Type_& val ) ;
Type_ in {sc_logic, sc_signal_inout_if<sc_logic>, sc_port<
   sc_signal_inout_if <sc_logic> >, sc_out_resolved& }
```

If `val` is not equal to current_value of the left hand side, then an update is scheduled with `val` as the new_value of the left hand side. Returns a reference to the instance.

Disabled Member Functions

```
sc_out_resolved (const sc_out_resolved& );
```


11.53 sc_out_rv

Synopsis

```
template <int W>
class sc_out_rv
    : public sc_inout_rv<W>
{
public:
// typedefs
    typedef sc_out_rv<W>                              this_type;
    typedef sc_inout_rv<W>                            base_type;
    typedef typename base_type::data_type             data_type;
    typedef typename base_type::in_if_type          in_if_type;
    typedef typename base_type::in_port_type in_port_type;
  typedef typename base_type::inout_if_type  inout_if_type;
typedef typename base_type::inout_port_type inout_port_type;
public:
    // constructors, destructor
    sc_out_rv();
    explicit sc_out_rv( const char* name_ );
    explicit sc_out_rv( inout_if_type& interface_ );
    sc_out_rv( const char* name_, inout_if_type&
    interface_ );
    explicit sc_out_rv( inout_port_type& parent_ );
    sc_out_rv( const char* name_, inout_port_type&
    parent_ );
    sc_out_rv( this_type& parent_ );
    sc_out_rv( const char* name_, this_type& parent_ );
    virtual ~sc_out_rv();

    // methods
    this_type& operator = ( const data_type& value_ );
    this_type& operator = ( const in_if_type& interface_ );
    this_type& operator = ( const in_port_type& port_ );
    this_type& operator = ( const inout_port_type& port_ );
    this_type& operator = ( const this_type& port_ );
    static const char* const kind_string;
    virtual const char* kind() const;
private:
    // disabled
    sc_out_rv( const this_type& );
};
```

Description

sc_out_rv is a specialized port for use with sc_signal_rv channels (Chapter 11.63). Its behavior is that of a sc_port which has only one interface that is of type sc_signal_inout_if<sc_lv<W> >. It has the same functionality as an sc_inout_rv port.

In the description of sc_out_rv, *port* refers to the sc_out_rv instance.

Example

```
SC_MODULE (module_name) {
    // ports
    sc_in_rv<8>  a ;
    sc_out_rv<13> b ;
    sc_inout_rv<44> c;

    //  rest of module
} ;
```

Public Constructors

sc_out_rv() ;

Create a `sc_out_rv` instance.

explicit
sc_out_rv(const char*);

Create a `sc_out_rv` instance with the string name initialized to `name_`.

Public Member Functions

virtual const char*
kind() const ;

Returns “sc_out_rv”.

Public Operators

sc_out_rv<W>&
operator = (const Type_& val) ;
Type_ in {sc_lv<W>, sc_signal_inout_if<T>, sc_port<
sc_signal_inout_if<T>,1>, sc_out_rv<W> }

If `val` is not equal to current_value of the left hand side, then an update is scheduled with `val` as the new_value of the left hand side. Returns a reference to the instance.

Disabled Member Function

sc_out_rv(const sc_out_rv<W>&);

11.54 sc_port

Synopsis

```
template <class IF, int N = 1>
class sc_port
: public sc_port_b<IF>
{
    // typdefs
    typedef sc_port_b<IF> base_type;
    typedef sc_port<IF,N> this_type;
public:
    // constructors, destructor
    sc_port();
    explicit sc_port( const char* name_ );
    explicit sc_port( IF& interface_ );
    sc_port( const char* name_, IF& interface_ );
    explicit sc_port( base_type& parent_ );
    sc_port( const char* name_, base_type& parent_ );
    sc_port( this_type& parent_ );
    sc_port( const char* name_, this_type& parent_ );
    virtual ~sc_port();

    static const char* const kind_string;
    virtual const char* kind() const;
private:
    // disabled
    sc_port( const this_type& );
    this_type& operator = ( const this_type& );
};
```

Description

An sc_port instance is associated with an interface of type IF

In the description of `sc_port`, *port* refers to the sc_port object, *interface* refers to the sc_interface type `IF`.

N signifies the maximum number of interfaces that may be attached to the port. If N = 0 then an arbitrary number of interfaces may be connected.

A port may not be bound after elaboration.

Example

```
SC_MODULE(my_module) {
sc_port<sc_fifo_in_if<int> > p1; //“read” fifo port
sc_port<sc_fifo_out_if<int> > p2; // “write” fifo port
sc_port<sc_fifo_in_if<int>,2> in_p;


// body of module
};
```


Public Constructors and Destructor

```
sc_port() ;
```
Default constructor.

```
explicit
sc_port( const char* name_ );
```
Create a `sc_port` instance with string name initialized to `name_`.

```
virtual ~sc_port();
```
Does nothing.

Public Member Functions

```
void
bind( IF& interface_  ) ;
```
Binds `interface_` to the port. For port to channel binding.

```
void
bind( sc_port<IF>& parent_port) ;
```
Binds `parent_` to the port. For port to port binding.

```
virtual sc_interface*
get_interface() ;
```
Returns a pointer to the first interface of the port. No error checking is provided.

```
virtual const sc_interface*
get_interface() const ;
```
Returns a constant pointer to the first interface of the port. No error checking is provided.

```
virtual const char*
kind() const ;
```
Returns “sc_port”.

```
int
size() const ;
```
Returns the number of connected interfaces.

Protected Member Functions

```
virtual void
end_of_elaboration();
```
Does nothing.

Public Operators

```
void
operator () ( IF& interface_ ) ;
```
Binds `interface_` to the `sc_port` instance. For port to channel binding.

```
void
```


operator () (sc_port<IF>& parent_);
Binds `parent_` to the `sc_port` instance. For port to port binding.

IF*
operator -> ();
Returns a pointer to the first interface of the port. Reports an error if the port is not bound. Allows for calling of methods provided by the interface.

const IF*
operator -> () const ;
Returns a pointer to the first interface of the port. Reports an error if the port is not bound. Allows for calling of methods provided by the interface.

IF*
operator [] (int index_);
Returns a pointer to the interface of the port at `index_`. Reports an error if the port is not bound. Allows for calling of methods provided by the interface at index.

const IF*
operator [] (int index_) const;
Returns a pointer to the interface of the port at `index_`. Reports an error if the port is not bound. Allows for calling of methods provided by the interface at index.

Disabled Member Functions

sc_port(const sc_port<IF,N>&);

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

11.55 sc_prim_channel

Synopsis

```
class sc_prim_channel
: public sc_object
{
public:
    static const char* const kind_string;
    virtual const char* kind() const ;
protected:
    // constructors, destructor
    sc_prim_channel();
    explicit sc_prim_channel( const char* );
    virtual ~sc_prim_channel();

    void request_update();
    virtual void update();
    virtual void end_of_elaboration();
protected:
    // static sensitivity for SC_THREADs
    void wait();

    //dynamic sensitivity for SC_THREADs and SC_CTHREADs
    void wait( const sc_event& e );
    void wait( sc_event_or_list& el );
    void wait( sc_event_and_list& el );
    void wait( const sc_time& t );
    void wait( double v, sc_time_unit tu );
    void wait( const sc_time& t, const sc_event& e );
    void wait( double v, sc_time_unit tu, const sc_event&
    e );
    void wait( const sc_time& t, sc_event_or_list& el );
    void wait( double v, sc_time_unit tu, sc_event_or_list&
    el );
    void wait( const sc_time& t, sc_event_and_list& el );
    void wait( double v, sc_time_unit tu,

    // static sensitivity for SC_METHODs
    void next_trigger();

    // dynamic sensitivity for SC_METHODs
    void next_trigger( const sc_event& e );
    void next_trigger( sc_event_or_list& el );
    void next_trigger( sc_event_and_list& el );
    void next_trigger( const sc_time& t );
    void next_trigger( double v, sc_time_unit tu );
    void next_trigger( const sc_time& t, const sc_event&
    e );
    void next_trigger( double v, sc_time_unit tu, const
    sc_event& e );
    void next_trigger( const sc_time& t, sc_event_or_list&
    el );
```



```
    void next_trigger( double v, sc_time_unit tu,
    sc_event_or_list& el );
    void next_trigger( const sc_time& t, sc_event_and_list&
    el );
    void next_trigger( double v, sc_time_unit tu,
    sc_event_and_list& el );

    bool timed_out();
private:
    // disabled
    sc_prim_channel( const sc_prim_channel& );
    sc_prim_channel& operator = ( const sc_prim_channel& );
};
```

Description

An `sc_prim_channel` is the base class for primitive channels. Users inherit from this class to create their own primitive channels.

In the description of `sc_prim_channel`, channel refers to the `sc_prim_channel` instance, *calling_process* refers to the process that calls the method in the channel. Chapter 2.4.1 describes the scheduler steps referred to in the description of `sc_prim_channel`.

The `wait()` and `next_trigger()` methods provide for static and dynamic sensitivity for processes. Refer to Chapters 9.3 , 9.4.1 and 9.5.1.

It provides for support of the request-update method of access.

Example

```
template <class T>
class sc_fifo
: public sc_fifo_in_if<T>,
  public sc_fifo_out_if<T>,
  public sc_prim_channel
{
public:

    // constructors

    explicit sc_fifo( int size_ = 16 )
      : sc_prim_channel( sc_gen_unique_name( "fifo" ) )
      { init( size_ ); }
. . . .
}
```

Protected Constructors

`sc_prim_channel``() ;`

Create an `sc_prim_channel` instance.

explicit
sc_prim_channel(const char* name_);
Create a `sc_prim_channel` instance with the string name initialized to `name`

Public Member Functions

virtual const char*
kind() const ;
Returns “sc_prim_channel”.

Protected Member Functions

virtual void
end_of_elaboration() ;
Does nothing.

void
request_update();
Requests that the update method be executed during the update of the current delta-cycle.

virtual void
update();
Does nothing by default.

Protected Member Functions for Process Sensitivity

bool
timed_out();
Returns true if the triggering of a process was based on the time out value of a wait() or next_trigger() method else returns false.

void
next_trigger();
Sets the calling process to be triggered based upon its static sensitivity list.

void
next_trigger(type_);
type_ in { const sc_event&, *sc_event_or_list*†&, *sc_event_and_list*†&, (double, sc_time_unit), const sc_time& }
Sets the calling_process to be triggered based upon `type_` (dynamic sensitivity).

void
next_trigger(double t_out_val , sc_time_unit t_out_tu, type_);
type_ in { const sc_event&, *sc_event_or_list*†&, *sc_event_and_list*†& }

Sets the calling_process to be triggered based upon either the time out (`t_out_val, t_out_tu`) or `type_`.

```
void
next_trigger( const sc_time& t_out, type_ );
type_ in { const sc_event&, sc_event_or_list†&,
    sc_event_and_list†& }
```

Sets the calling_process to be triggered based upon either the time out (`t_out`) or `type_`.

```
void
wait() ;
```

Suspends the calling process. Calling process is triggered based upon its static sensitivity list.

```
void
wait( type_);
type_ in { const sc_event&, sc_event_or_list†&,
    sc_event_and_list†&, (double, sc_time_unit), const
    sc_time& }
```

Sets the calling_process to be triggered based upon `type_` (dynamic sensitivity).

```
void
wait( double, sc_time_unit, type_ );
type_ in { const sc_event&, sc_event_or_list†&,
    sc_event_and_list†& }

void
wait( const sc_time&, type_ );
type_ in { const sc_event&, sc_event_or_list†&,
    sc_event_and_list†& }
```

Disabled Member Functions

```
sc_prim_channel( const sc_prim_channel& );

sc_prim_channel&
operator = ( const sc_prim_channel& );
```


11.56 sc_pvector

Synopsis

```
template< class T >
class sc_pvector
{
public:
    // typedefs
    typedef T* iterator;
    typedef const T* const_iterator;

    // constructors & destructor
    sc_pvector( int alloc = 10 );
    sc_pvector( const sc_pvector<T>&);
    ~sc_pvector();

    // operators
    sc_pvector<T>& operator = ( const sc_pvector<T>&);
    T& operator [] ( int i );
    const T& operator [] ( int i ) const;

    // other methods
    int size() const;
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    T& fetch( int i );
    const T& fetch( int i ) const;
    T* raw_data();
    const T* raw_data() const;
    void push_back( T item );
    void erase_all();
    void sort( CFT compar );
    void put( T item, int i );
    void decr_count();
    void decr_count( int k );
};
```

Description

sc_pvector is a utility container class that acts like a smart array that maintains size information and can grow dynamically. It provides random access to its data through the C++ subscript operators.

Example

```
sc_pvector<sc_object *> top_objs =
    sc_get_curr_simcontext()->get_child_objects();

for (int i = 0; i < top_objs.size(); i++)
  cout << top_objs[i]->name() << endl;
```


Type Definitions

```
typedef T* iterator;
typedef const T* const_iterator;
```

Public Constructors and Destructor

sc_pvector(int alloc = 10);

Create a new vector. The constructor parameter controls how much memory is pre-allocated. The default value is 10.

sc_pvector(const sc_pvector<T>&);

Copy constructor.

~sc_pvector();

Destructor.

Public Member Functions

iterator
begin() ;

Returns an iterator pointing to the first element in the vector.

const_iterator *
begin() const;

Returns a const-iterator pointing to the first element in the vector.

void
decr_count() ;

Removes the last element from the vector, i.e. ,decreases the size by 1.

void
decr_count(int k) ;

Removes the last k elements from the vector, i.e. ,decreases the size by k.

iterator
end() ;

Returns an iterator pointing one beyond the last element in the vector.

const_iterator
end() const;

Returns a const-iterator pointing one beyond the last element in the vector.

void
erase_all() ;

Removes all elements from the vector, i.e., sets the size to 0.

T &
fetch(int i) ;

Returns a reference to the object at location i. No range checking is performed.


```
const T &
fetch( int i ) const ;
```
Returns a constant reference to the object at location `i`. No range checking is performed.

```
void
push_back( T item ) ;
```
Adds the item to the end of the vector, increasing its size by 1.

```
T &
put( T new_item, int i ) ;
```
Replaces the item at index `i` to `new_item`. No range checking is performed.

```
T *
raw_data() ;
```
Returns a pointer to the first item in the vector.

```
const T *
raw_data() const ;
```
Returns a constant pointer to the first item in the vector.

```
int
size() const ;
```
Returns the number of items in the vector.

```
void
sort( CFT compar ) ;
```
Sorts the elements in the vector according to the compare function `compar`.

The compare function is declared as:

```
extern “C” {
  int compare_func( const void *, const void * );
}
```
This function returns -1 if the first argument is less than the second, 0 if they are equal, and 1 if the first argument is greater than the second.

Public Operators

```
T &
operator []( int i) ;
```
Returns a reference to the item at location i in the vector. If i > size of vector, then the vector is resized to accomodate i.

```
const T &
operator []( int i) const ;
```
Returns a constant reference to the item at location i in the vector. If i > size of vector, then the vector is resized to accomodate i.


```
sc_pvector<T>&
operator = ( const sc_pvector<T>& rhs ) ;
```

Assignment operator.

11.57 sc_semaphore

Synopsis

```
class sc_semaphore
: public sc_semaphore_if,
  public sc_prim_channel
{
public:
    // constructors
    explicit sc_semaphore( int init_value_ );
    sc_semaphore( const char* name_, int init_value_ );

    // methods
    virtual int wait();
    virtual int trywait();
    virtual int post();
    virtual int get_value() const;
    static const char* const kind_string;
    virtual const char* kind() const;
private:
    // disabled
    sc_semaphore( const sc_semaphore& );
    sc_semaphore& operator = ( const sc_semaphore& );
};
```

Description

An **sc_semaphore** channel (semaphore) is similar to an sc_mutex channel (see Chapter 11.47) except for it allows for limited concurrent access. It implements the sc_semaphore_if interface.

An sc_semaphore instance is created with a mandatory integer value which determines the initial number of concurrent accesses to the semaphore.

In the description of sc_semaphore the number of available concurrent accesses is referred to as the *semaphore_value*. The semaphore is considered available if the semaphore_value is greater than 0. Chapter 2.4.1 describes the scheduler steps referred to in the description of sc_semaphore.

When a process successfully locks (takes) the semaphore the semaphore_value is decreased by 1. When a process unlocks (gives) the semaphore the semaphore_value is increased by 1.

No checking is done to ensure that a process unlocking the semaphore is one that locked it.

No checking is done to ensure that the current semaphore_value does not exceed the initial semaphore_value.

If multiple processes attempt to lock a semaphore when the semaphore_value is 1 during the same delta-cycle, only one process will be successful. Since the order of execution of processes in a delta-cycle is indeterminate, it is indeterminate as to which process is successful. The unsuccessful processes will be suspended as described in the next paragraph.

If a process attempts to lock the semaphore, when the semaphore_value is zero or less, then the process is suspended. When the semaphore is unlocked then the suspended process is triggered and continues the attempt to lock the semaphore. The unsuspended process is not guaranteed to be successful in locking the semaphore if there are other processes also attempting to lock the semaphore.

Example

```
SC_MODULE(my_module) {

  sc_semaphore a, b;

  SC_CTOR(my_module):
    a(5),  // init a semaphore_value to 5
    b(3)   // init b semaphore_value to 3
  {
  }
  // rest of module not shown
};
```

Public Constructors

explicit
sc_semaphore(int val) ;
Create an `sc_semaphore` instance with the semaphore_value initialized to `val`.

explicit
sc_mutex(const char*);

sc_semaphore(const char* name_, int val) ;
Create an `sc_semaphore` instance with the semaphore_value initialized to `val` and the string name initialized to `name_`.

Public Member Functions

virtual int
get_value() const ;
Returns the semaphore_value of the semaphore.

virtual const char*
kind() const ;
Returns “sc_semaphore”.

virtual int


```
post() ;
```
Returns 0. Unlocks semaphore and increases by 1 the semaphore_value.

```
virtual int
trywait() ;
```
If the semaphore is available then locks semaphore, decreases by 1 the semaphore_value and returns 0, else returns -1.

```
virtual int
wait() ;
```
Returns 0. If the semaphore is available then locks semaphore decreasing by 1 the number of concurrent accesses available else suspends the calling process.

Disabled Member Functions

```
sc_semaphore( const sc_semaphore& ) ;

sc_semaphore&
operator = ( const sc_semaphore& ) ;
```


11.58 sc_semaphore_if

Synopsis

```
class sc_semaphore_if
: virtual public sc_interface
{
public:
    virtual int wait() = 0;
    virtual int trywait() = 0;
    virtual int post() = 0;
    virtual int get_value() const = 0;
protected:
    // constructor
    sc_semaphore_if();
private:
    // disabled
    sc_semaphore_if( const sc_semaphore_if& );
    sc_semaphore_if& operator = ( const sc_semaphore_if& );
};
```

Description

The **sc_semaphore_if** class provides the signatures of the functions for the sc_semaphore_if interface. See Chapter 8.1 for a description of interfaces. Implemented by the sc_sempahore channel (Chapter 11.56)

Example

```
class sc_semaphore
: public sc_semaphore_if,
  public sc_prim_channel{  . . . . };
```

Protected Constructor

sc_semaphore_if();

Create a sc_semaphore_if instance.

Public Member Functions

virtual int
get_value() = 0;

virtual int
post() = 0;

virtual int
trywait() = 0;

virtual int
wait() = 0;

Disabled Member Functions

sc_semaphore_if(const sc_semaphore_if&);


```
sc_semaphore_if&
operator = ( const sc_semaphore_if& );
```


11.59 sc_sensitive

Synopsis

```
class sc_sensitive
{
private:
    // constructor, destructor
    explicit sc_sensitive( sc_module* );
    ~sc_sensitive();
public:
    //  specify static sensitivity for processes
    sc_sensitive& operator () ( const sc_event& );
    sc_sensitive& operator () ( const sc_interface& );
    sc_sensitive& operator () ( const sc_port_base& );
    sc_sensitive& operator () ( sc_event_finder& );
    sc_sensitive& operator << ( const sc_event& );
    sc_sensitive& operator << ( const sc_interface& );
    sc_sensitive& operator << ( const sc_port_base& );
    sc_sensitive& operator << ( sc_event_finder& );
private:
    // disabled
    sc_sensitive();
    sc_sensitive( const sc_sensitive& );
    sc_sensitive& operator = ( const sc_sensitive& );
};
```

Description

sc_sensitive provides overloaded operators << and (), used in specifying static sensitivity for processes. These operators can only be called before simulation starts, and produce an error message if called after simulation starts.

Public Operators

sc_sensitive&
operator << (const sc_event&);

Adds an event to the list of events that will trigger the last declared process when static sensitivity is used.

sc_sensitive&
operator << (const sc_interface&);

Adds an event (that is returned by the default_event() method of the channel) to the list of events that will trigger the last declared process when static sensitivity is used.

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

Adds an event (that is returned by the default_event() method of the channel bound to the port) to the list of events that will trigger the last declared process when static sensitivity is used.

sc_sensitive&
operator << (*sc_event_finder*† &);
Adds an event (that is returned by the find_event() method of the event finder) to the list of events that will trigger the last process that was declared when static sensitivity is used.

sc_sensitive&
operator () (const sc_event&);
Adds an event to the list of evenst that will trigger the last declared process when static sensitivity is used.

sc_sensitive&
operator () (const sc_interface&);
Adds an event (that is returned by the default_event() method of the channel) to the list of events that will trigger the last declared process when static sensitivity is used.

sc_sensitive&
operator () (const *sc_port_base*† &);
Adds an event (that is returned by the default_event() method of the channel bound to the port) to the list of events that will trigger the last declared process when static sensitivity is used.

sc_sensitive&
operator () (*sc_event_finder*† &);
Adds an event (that is returned by the find_event() method of the event finder) to the list of events that will trigger the last process that was declared when static sensitivity is used.

Disabled Member Functions

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

11.60 sc_signal

Synopsis

```
template <class T>
class sc_signal
: public sc_signal_inout_if<T>,
  public sc_prim_channel
{
public:
    // constructors, destructor
    sc_signal();
    explicit sc_signal( const char* name_ );
    virtual ~sc_signal();

    // methods
    virtual void register_port( sc_port_base&, const
    char* );
    virtual const sc_event& default_event() const;
    virtual const sc_event& value_changed_event() const;
    virtual const T& read() const;
    virtual const T& get_data_ref() const;
    virtual bool event() const;
    virtual void write( const T& );
    operator const T& () const;
    sc_signal<T>& operator = ( const T& a );
    sc_signal<T>& operator = ( const sc_signal<T>& a );
    const T& get_new_value() const;
    void trace( sc_trace_file* tf ) const;
    virtual void print( ostream& ) const;
    virtual void dump( ostream& ) const;
    static const char* const kind_string;
    virtual const char* kind() const;
protected:
    virtual void update();
    void check_writer();
private:
    // disabled
    sc_signal( const sc_signal<T>& );
};
```

Description

`sc_signal` is a primitive channel that implements the `sc_signal_inout_if` interface.

In the description of `sc_signal`, *current_value* refers to the value of the `sc_signal` instance, *new_value* is the value to be written and *old_value* is the previous value. Chapter 2.4.1 describes the scheduler steps referred to in the description of `sc_signal`.

Initialization

The initial current_value of an `sc_signal` instance is dependent upon type `T` and is undefined. The current_value may be explicitly initialized in the sc_main function or in the constructor of the module where it is created.

A `sc_signal` may be written by only one process, but may be read by multiple processes.

`sc_signal` writes and reads follows evaluate-update semantics suitable for describing hardware.

Write
The write method is executed during the evaluate phase of a delta-cycle. If the new_value is different than the current_value, an update is requested. During the update phase the current_value is assigned the new_value and an event occurs.
The evaluate-update is accomplished using the `request_update()` and `update()` methods. `request_update()` is called during the execution of the write method (in the evaluate phase) indicating to the kernel that an update is required. During the update phase the kernel calls the update method provided by the `sc_signal` channel.

Multiple writes in same delta-cycle
If multiple writes by a process to the same `sc_signal` occur during a particular evaluate phase of a delta-cycle, the last write executed determines the new_value the `sc_signal` will receive in the update phase of the same delta-cycle.

Read
A read is executed during the evaluate phase of a delta-cycle and returns the current_value. It does not consume the data.

Simultaneous reads and writes
If during the evaluate phase of a delta-cycle a read and write occur to the same `sc_signal`, the read will return the current_value. The new_value from the write will not be available to read until the next delta-cycle as described above.

Example

```
// GIVEN
sc_signal<int> m;  // channel of type int
      // channel of type sc_uint<12>
sc_signal<sc_uint<12> > n;
sc_signal<bool> clk; // channel of type bool
int i;

//THEN
m.write(i);   //write m with value of i
n.write(8);   //write n with value of 8
if(clk.posedge() ) // was there a posedge?
```



```
i = m.read();   // assign value of m to i
      // wait for posedge of clk
wait(clk.posedge_event() ) ;
```

Public Constructors

```
sc_signal();
```

Create a `sc_signal` instance.

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

Create a `sc_signal` instance with the string name initialized to `name_`.

Public Member Functions

```
virtual const sc_event&
default_event() const ;
```

Returns a reference to an event that occurs when new_value on a write is different from current_value.

```
virtual void
dump( ostream& ) const;
```

Prints the string name, current_value and new_value of the `sc_signal` instance to an output stream.

```
virtual bool
event() const ;
```

Returns true if an event occurred in the previous delta-cycle.

```
virtual const T&
get_data_ref() const ;
```

Returns a reference to current_value.

```
virtual const char*
kind() const ;
```

Returns “sc_signal”.

```
const T&
get_new_value() const ;
```

Returns a reference to new_value.

```
virtual bool
negedge() const ;
```

Type `bool` and `sc_logic` only. Returns true if an event occurred in the previous delta-cycle and current_value is false.

```
virtual const sc_event&
negedge_event() const ;
```

Type `bool` and `sc_logic` only. Returns a reference to an event that occurs when new_value on a write is false and the current_value is not false.

```
virtual bool
```


posedge () const ;
Type bool and sc_logic only. Returns true if an event occurred in the previous delta-cycle and current_value is true.

virtual const sc_event&
posedge_event () const ;
Type bool and sc_logic only. Returns a reference to an event that occurs when new_value on a write is true and the current_value is not true.

virtual const T&
read() const ;
Returns a reference to current_value.

virtual void
register_port(*sc_port_base*† &, const char*);
Checks to ensure at most only one out or inout port is connected to the sc_signal instance.

virtual void
print(ostream&) const;
Prints current_value to an output stream.

void
trace(*sc_trace_file*† * tf) const ;
Adds a trace of current_value to the trace file tf.

virtual void
write(const T& val);
If val is not equal to current_value then schedules an update with val as new_value.

virtual const sc_event&
value_changed_event() const ;
Returns a reference to an event that occurs when new_value on a write is different from current_value.

Public Operators

operator const T& **()** const ;
Returns current_value.

sc_signal<T>&
operator = (const T& val);
If val is not equal to current_value of the left hand side, then an update is scheduled with val as the new_value of the left hand side. Returns a reference to the instance.

sc_signal<T>&

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

If the current_value of `val` is not equal to current_value of the left hand side, then an update is scheduled with the current_value of `val` as the new_value of the left hand side. Returns a reference to the instance.

Protected Member Functions

void
check_writer();

Checks to make sure only one process writes to the `sc_signal` instance. Prints an error message if more than one process attempts to write the `sc_signal` instance.

virtual void
update();

Assigns new_value to current_value and causes an event to occur. Called by the kernel during the update phase in response to the execution of a request_update method.

Disabled Member Function

sc_signal(const sc_signal<T>&);

Specialized ports

The classes `sc_in`, `sc_out` and `sc_inout` are specialized ports for use with `sc_signal` channels.

11.61 sc_signal_in_if

Synopsis

```
template <class T>
class sc_signal_in_if
: virtual public sc_interface
{
public:
  virtual const sc_event& value_changed_event() const = 0;
   virtual const T& read() const = 0;
   virtual const T& get_data_ref() const = 0;
   virtual bool event() const = 0;
protected:
   // constructor
   sc_signal_in_if();
private:
   // disabled
   sc_signal_in_if( const sc_signal_in_if<T>& );
   sc_signal_in_if<T>& operator = ( const
   sc_signal_in_if<T>& );
};
```

Description

The `sc_signal_in_if` class provides the signatures of the functions for the `sc_signal_in_if` interface. See Chapter 8.1 for a description of interfaces.

Example

```
SC_MODULE(my_module) {
sc_port< sc_signal_in_if<int> > p1; //"read" signal port


template <class T>
class sc_in
: public sc_port<sc_signal_in_if<T>,1>
{ . . . . };
```

Protected Constructor

`sc_signal_in_if``();`

Create a `sc_signal_in_if` instance.

Public Member Functions

`virtual bool`
`event``() const = 0;`

`virtual const T&`
`get_data_ref``() const = 0;`

`virtual bool`
`negedge``() const = 0;`

Type `bool` and `sc_logic` only.


```
virtual const sc_event&
negedge_event() const = 0;
```
Type `bool` and `sc_logic` only.

```
virtual const sc_event&
posedge_event() const = 0;
```
Type `bool` and `sc_logic` only.

```
virtual bool
posedge() const = 0;
```
Type `bool` and `sc_logic` only.

```
virtual const T&
read() const = 0;

virtual const sc_event&
value_changed_event() const = 0;
```

Disabled Member Functions

```
sc_signal_in_if( const sc_signal_in_if<T>& );

sc_signal_in_if<T>&
operator = ( const sc_signal_in_if<T>& );
```


11.62 sc_signal_inout_if

Synopsis

```
template <class T>
class sc_signal_inout_if
: public sc_signal_in_if<T>
{
public:
   virtual void write( const T& ) = 0;
protected:
   // constructor
   sc_signal_inout_if();
private:
   // disabled
   sc_signal_inout_if( const sc_signal_inout_if<T>& );
   sc_signal_inout_if<T>& operator = ( const
   sc_signal_inout_if<T>& );
};
```

Description

The `sc_signal_inout_if` class provides the signatures of the functions for the `sc_signal_inout_if` interface. See Chapter 8.1 for a description of interfaces. Implemented by the `sc_signal` channel (Chapter 11.60)

Example

```
SC_MODULE(my_module) {
sc_port<sc_signal_inout_if<int> > p1; //"rw" signal port

template <class T>
class sc_inout
: public sc_port<sc_signal_inout_if<T>,1>
{ . . . . };
```

Protected Constructor

`sc_signal_inout_if``();`

Create a `sc_signal_inout_if` instance.

Public Member Functions

```
virtual void
write( const T& ) = 0;
```

Disabled Member Functions

```
sc_signal_inout_if( const sc_signal_inout_if<T>& );

sc_signal_inout_if<T>&
operator = ( const sc_signal_inout_if<T>& )
```


11.63 sc_signal_resolved

Inheritance

Synopsis

```
class sc_signal_resolved
: public sc_signal<sc_logic>
{
public:
    // typedefs
    typedef sc_signal_resolved  this_type;
    typedef sc_signal<sc_logic> base_type;
    typedef sc_logic         data_type;
public:
    // constructors, destructor
    sc_signal_resolved();
    explicit sc_signal_resolved( const char* name_ );
    virtual ~sc_signal_resolved();

    // methods
    virtual void register_port( sc_port_base&, const
    char* );
    virtual void write( const data_type& );
    this_type& operator = ( const data_type& a );
    this_type& operator = ( const this_type& a );
    static const char* const kind_string;
    virtual const char* kind() const;
protected:
    virtual void update();
private:
    // disabled
    sc_signal_resolved( const this_type& );
};
```

Description

sc_signal_resolved is a primitive channel that implements the sc_signal_inout_if interface. It behaves like a sc_signal< sc_logic > channel except it may be written by multiple processes. Refer to Chapter 11.60 for the behavior of an sc_signal and Chapter 11.43 for the description of the sc_logic data type and its legal values.

In the description of sc_signal_resolved, *current_value* refers to the value of the sc_signal_resolved instance, *new_value* is the value to be written after resolution, and *old_value* is the previous value. For each process that writes there is a separate *pw_value*, which is the value to be written by that particular process. The multiple *pw_values* are resolved to generate *new_value*. Chapter 2.4.1 describes the scheduler steps referred to.

Initialization

The initial current_value of an `sc_signal_resolved` instance is `Log_X`. The current_value may be explicitly initialized in the sc_main function or in the constructor of the module where it is created.

The resultant value for writes by multiple processes during the same delta-cycle is resolved per Table 29 - Resolution of multiple values.

Table 29 - Resolution of multiple values

Value	0	1	Z	X
0	Log_0	Log_X	Log_0	Log_X
1	Log_X	Log_1	Log_1	Log_X
Z	Log_0	Log_1	Log_Z	Log_X
X	Log_X	Log_X	Log_X	Log_X

Example

```
// GIVEN
sc_signal_resolved m;  // channel
sc_signal_resolved n;  // channel
sc_logic i;

// THEN
m.write(n);  //write m with value of n
m = n;  // write m with value of n
i = 'Z';
n.write(i); //write n with value of i
if(m.posedge() ) // was there a posedge?
  i = m.read();   // assign value of m to i
      // wait for posedge of n
wait(n.posedge_event() ) ;
```

Public Constructors

`sc_signal_resolved``() ; ;`

Create a `sc_signal_resolved` instance.

`explicit`
`sc_signal_resolved``( const char* name_ ) ;`

Create a sc_signal_resolved instance with the string name initialized to name_.

Public Member Functions

`virtual const char*`
`kind``() const ;`

Returns “sc_signal_resolved”

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

Does nothing.


```
virtual void
write( const sc_logic& val ) ;
```

If `val` is not equal to current_value then schedules an update with `val` as pw_value for the writing process.

Public Operators

```
sc_signal_resolved&
operator = ( const sc_logic& val ) ;
```

If `val` is not equal to current_value of the left hand side, then an update is scheduled with `val` as the pw_new_value. Returns a reference to the instance.

```
sc_signal_resolved&
operator = ( const sc_signal_resolved& val ) ;
```

If the current_value of `val` is not equal to current_value of the left hand side, then an update is scheduled with the current_value of `val` as the pw_new_value of the left hand side. Returns a reference to the instance.

Protected Member Functions

```
virtual void
update();
```

Resolves pw_values per Table 29 - Resolution of multiple values to new_value. Assigns new_value to current_value and causes an event to occur. Called by the kernel during the update phase in response to the execution of a request_update method.

Disabled Member Functions

```
sc_signal_resolved( const sc_signal_resolved& );
```


11.64 sc_signal_rv

Synopsis

```
template <int W>
class sc_signal_rv
: public sc_signal<sc_lv<W> >
{
public:
    // typedefs
    typedef sc_signal_rv<W>    this_type;
    typedef sc_signal<sc_lv<W> > base_type;
    typedef sc_lv<W>          data_type;
public:
    // constructors, destructor
    sc_signal_rv();
    explicit sc_signal_rv( const char* name_ );
    virtual ~sc_signal_rv();

    // methods
    virtual void register_port( sc_port_base&, const
    char* );
    virtual void write( const data_type& );
    this_type& operator = ( const data_type& a );
    this_type& operator = ( const this_type& a );
    static const char* const kind_string;
    virtual const char* kind() const;
protected:
    virtual void update();
private:
    // disabled
    sc_signal_rv( const this_type& );
};
```

Description

`sc_signal_rv` is a primitive channel that implements the `sc_signal_inout_if` interface. It behaves like an `sc_signal< sc_lv<W> >` channel except it may be written by multiple processes. Refer to Chapter 11.60 for the behavior of an `sc_signal` and Chapter 11.43 for the description of the `sc_logic` data type and its legal values.

In the description of `sc_signal_rv`, *current_value* refers to the value of the `sc_signal_rv` instance, *new_value* is the value to be written after resolution, and *old_value* is the previous value. For each process that writes there is a separate *pw_value*, which is the value to be written by that particular process. The multiple *pw_values* are resolved to generate *new_value*. Chapter 2.4.1 describes the scheduler steps referred to.

Initialization

The initial current_value of an `sc_signal_rv` instance is `Log_X`. The current_value may be explicitly initialized in the sc_main function or in the constructor of the module where it is created.

The resultant value of each bit of an `sc_signal_rv` for writes by multiple processes during the same delta-cycle is resolved per Table 30 - Resolution of multiple values.

Table 30 - Resolution of multiple values

Value	0	1	Z	X
0	Log_0	Log_X	Log_0	Log_X
1	Log_X	Log_1	Log_1	Log_X
Z	Log_0	Log_1	Log_Z	Log_X
X	Log_X	Log_X	Log_X	Log_X

Examples

```
// GIVEN
sc_signal_rv<4> m, p;  // channels
sc_signal_rv<1> n;  // channel
sc_lv<1> i = '0';

// THEN
i = 'Z';
n.write(i); //write n with value of i
m = "01XZ";
p = m;  // write p with value of m
p.write(m.read() ); // write p with value of m
   // wait for a change of value of m
wait(m.default_event() );
```

Public Constructors

`sc_signal_rv(``);`

Create a `sc_signal_rv` instance.

`explicit`
`sc_signal_rv``( const char* name_ ) ;`

Create a `sc_signal_rv` instance with the string name initialized to `name_`.

Public Member Functions

`virtual const char*`
`kind``() const ;`

Returns “sc_signal_rv”.

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

Does nothing.

`virtual void`

write(const sc_lv< W >& val) ;

If `val` is not equal to current_value then schedules an update with `val` as pw_value for the writing process.

Public Operators

sc_signal_rv< W >&
operator = (const sc_lv< W >&) ;

If `val` is not equal to current_value of the left hand side, then an update is scheduled with `val` as the pw_new_value. Returns a reference to the instance.

sc_signal_rv< W >&
operator = (const sc_signal_rv< W >&) ;

If the current_value of `val` is not equal to current_value of the left hand side, then an update is scheduled with the current_value of `val` as the pw_new_value of the left hand side. Returns a reference to the instance.

Protected Member Functions

virtual void
update();

Resolves pw_values per Table 30 - Resolution of multiple values to new_value. Assigns new_value to current_value and causes an event to occur. Called by the kernel during the update phase in response to the execution of a request_update method.

Disabled Member Function

sc_signal_rv(const sc_signal_rv< W >&);

11.65 sc_signed

```
class sc_signed
{
public:
    // constructors & destructors
    explicit sc_signed( int nb = sc_length_param().len() );
    sc_signed( const sc_signed&   v );
    sc_signed( const sc_unsigned& v );
    ~sc_signed()

    // assignment operators
    sc_signed& operator = (const sc_signed&        v);
    sc_signed& operator = (const sc_signed_subref_r& a );
    template <class T1, class T2>
    sc_signed& operator = ( const
    sc_signed_concref_r<T1,T2>& a )
    sc_signed& operator = (const sc_unsigned&    v);
    sc_signed& operator = (const sc_unsigned_subref_r& a );
    template <class T1, class T2>
    sc_signed& operator = ( const
    sc_unsigned_concref_r<T1,T2>& a )
    sc_signed& operator = (const char*             v);
    sc_signed& operator = (int64                      v);
    sc_signed& operator = (uint64      v);
    sc_signed& operator = (long                        v);
    sc_signed& operator = (unsigned long            v);
    sc_signed& operator = (int                          v)
    sc_signed& operator = (unsigned int           v)
    sc_signed& operator = (double      v);
    sc_signed& operator = (const sc_int_base&    v);
    sc_signed& operator = (const sc_uint_base&      v);
    sc_signed& operator = ( const sc_bv_base& );
    sc_signed& operator = ( const sc_lv_base& );
    sc_signed& operator = ( const sc_fxval& );
    sc_signed& operator = ( const sc_fxval_fast& );
    sc_signed& operator = ( const sc_fxnum& );
    sc_signed& operator = ( const sc_fxnum_fast& );

    // Increment operators.
    sc_signed& operator ++ ();
    const sc_signed operator ++ (int);

    // Decrement operators.
    sc_signed& operator -- ();
    const sc_signed operator -- (int);

    // bit selection
    sc_signed_bitref operator [] ( int i )
    sc_signed_bitref_r operator [] ( int i ) const
    sc_signed_bitref bit( int i )
    sc_signed_bitref_r bit( int i ) const
```



```
// part selection
sc_signed_subref range( int i, int j )
sc_signed_subref_r range( int i, int j ) const
sc_signed_subref operator () ( int i, int j )
sc_signed_subref_r operator () ( int i, int j ) const

// explicit conversions
int       to_int() const;
unsigned int  to_uint() const;
long      to_long() const;
unsigned long to_ulong() const;
int64      to_int64() const;
uint64    to_uint64() const;
double    to_double() const;
const sc_string to_string( sc_numrep numrep = SC_DEC )
const;
const sc_string to_string( sc_numrep numrep, bool
w_prefix ) const;

// methods
void print( ostream& os = cout ) const
void scan( istream& is = cin );
void dump( ostream& os = cout ) const;
int length() const { return nbits; } // Bit width.
bool iszero() const;                        // Is the
number zero?
bool sign() const;                          // Sign.

void reverse();

// ADDition operators:
friend sc_signed operator + (const sc_unsigned&   u,
const sc_signed&  v);
friend sc_signed operator + (const sc_signed&u, const
sc_unsigned& v);
friend sc_signed operator + (const sc_unsigned&   u,
int64          v);
friend sc_signed operator + (const sc_unsigned&   u,
long v);
friend sc_signed operator + (const sc_unsigned&   u,
int           v)
friend sc_signed operator + (int64                u,
const sc_unsigned&  v);
friend sc_signed operator + (long                u,
const sc_unsigned&  v);
friend sc_signed operator + (int                  u,
const sc_unsigned&  v)
friend sc_signed operator + (const sc_signed&u, const
sc_signed&v);
friend sc_signed operator + (const sc_signed&u, int64
        v);
friend sc_signed operator + (const sc_signed&u, uint64
       v);
```



```
friend sc_signed operator + (const sc_signed&u, long
   v);
friend sc_signed operator + (const sc_signed&u,
unsigned long     v);
friend sc_signed operator + (const sc_signed&u, int
         v)
friend sc_signed operator + (const sc_signed&u,
unsigned int v)
friend sc_signed operator + (int64  u, const sc_signed&
   v);
friend sc_signed operator + (uint64 u, const sc_signed&
   v);
friend sc_signed operator + (long u, const sc_signed&
   v);
friend sc_signed operator + (unsigned long   u, const
sc_signed&v);
friend sc_signed operator + (int   u, const sc_signed&
   v)
friend sc_signed operator + (unsigned int u, const
sc_signed&v)
sc_signed& operator += (const sc_signed&v);
sc_signed& operator += (const sc_unsigned&   v);
sc_signed& operator += (int64                 v);
sc_signed& operator += (uint64               v);
sc_signed& operator += (long  v);
sc_signed& operator += (unsigned long         v);
sc_signed& operator += (int                   v)
sc_signed& operator += (unsigned int    v)
friend sc_signed operator + (const sc_unsigned&   u,
const sc_int_base&  v);
friend sc_signed operator + (const sc_int_base&   u,
const sc_unsigned&  v);
friend sc_signed operator + (const sc_signed&u, const
sc_int_base& v);
friend sc_signed operator + (const sc_signed&u, const
sc_uint_base& v);
friend sc_signed operator + (const sc_int_base&   u,
const sc_signed&  v);
friend sc_signed operator + (const sc_uint_base& u,
const sc_signed&  v);
sc_signed& operator += (const sc_int_base&   v);
sc_signed& operator += (const sc_uint_base& v);

// SUBtraction operators:
friend sc_signed operator - (const sc_unsigned&   u,
const sc_signed&  v);
friend sc_signed operator - (const sc_signed&u, const
sc_unsigned& v);
friend sc_signed operator - (const sc_unsigned&   u,
const sc_unsigned&  v);
friend sc_signed operator - (const sc_unsigned&   u,
int64          v);
friend sc_signed operator - (const sc_unsigned&   u,
uint64         v);
```



```
friend sc_signed operator - (const sc_unsigned&   u,
long v);
friend sc_signed operator - (const sc_unsigned&   u,
unsigned long     v);
friend sc_signed operator - (const sc_unsigned&   u,
int  v)
friend sc_signed operator - (const sc_unsigned&   u,
unsigned int    v)
friend sc_signed operator - (int64 u, const
sc_unsigned& v);
friend sc_signed operator - (uint64     u, const
sc_unsigned& v);
friend sc_signed operator - (long       u, const
sc_unsigned& v);
friend sc_signed operator - (unsigned long   u, const
sc_unsigned& v);
friend sc_signed operator - (int    u, const
sc_unsigned& v)
friend sc_signed operator - (unsigned int    u, const
sc_unsigned& v)
friend sc_signed operator - (const sc_signed&u, const
sc_signed&v);
friend sc_signed operator - (const sc_signed&u, int64
   v);
friend sc_signed operator - (const sc_signed&u, uint64
        v);
friend sc_signed operator - (const sc_signed&u, long
   v);
friend sc_signed operator - (const sc_signed&u,
unsigned long     v);
friend sc_signed operator - (const sc_signed&u, int
         v)
friend sc_signed operator - (const sc_signed&u,
unsigned int v)
friend sc_signed operator - (int64 u, const sc_signed&
   v);
friend sc_signed operator - (uint64 u, const sc_signed&
   v);
friend sc_signed operator - (long  u, const sc_signed&
   v);
friend sc_signed operator - (unsigned long   u, const
sc_signed&v);
friend sc_signed operator - (int u, const sc_signed&
   v)
friend sc_signed operator - (unsigned int u, const
sc_signed&v)
sc_signed& operator -= (const sc_signed&v);
sc_signed& operator -= (const sc_unsigned&   v);
sc_signed& operator -= (int64                 v);
sc_signed& operator -= (uint64               v);
sc_signed& operator -= (long  v);
sc_signed& operator -= (unsigned long         v);
sc_signed& operator -= (int                   v)
sc_signed& operator -= (unsigned int    v)
```



```
friend sc_signed operator - (const sc_unsigned&   u,
const sc_int_base&  v);
friend sc_signed operator - (const sc_unsigned&   u,
const sc_uint_base& v);
friend sc_signed operator - (const sc_int_base&   u,
const sc_unsigned&  v);
friend sc_signed operator - (const sc_uint_base& u,
const sc_unsigned&  v);
friend sc_signed operator - (const sc_signed&u, const
sc_int_base& v);
friend sc_signed operator - (const sc_signed&u, const
sc_uint_base& v);
friend sc_signed operator - (const sc_int_base&   u,
const sc_signed&  v);
friend sc_signed operator - (const sc_uint_base& u,
const sc_signed&  v);
sc_signed& operator -= (const sc_int_base&   v);
sc_signed& operator -= (const sc_uint_base& v);

// MULtiplication operators:
friend sc_signed operator * (const sc_unsigned&   u,
const sc_signed&  v);
friend sc_signed operator * (const sc_signed&u, const
sc_unsigned& v);
friend sc_signed operator * (const sc_unsigned&   u,
int64           v);
friend sc_signed operator * (const sc_unsigned&   u,
long v);
friend sc_signed operator * (const sc_unsigned&   u,
int           v)
friend sc_signed operator * (int64 u, const
sc_unsigned& v);
friend sc_signed operator * (long  u, const
sc_unsigned& v);
friend sc_signed operator * (int    u, const
sc_unsigned& v)
friend sc_signed operator * (const sc_signed&u, const
sc_signed&v);
friend sc_signed operator * (const sc_signed&u, int64
      v);
friend sc_signed operator * (const sc_signed&u, uint64
  v);
friend sc_signed operator * (const sc_signed&u, long
       v);
friend sc_signed operator * (const sc_signed&u,
unsigned long   v);
friend sc_signed operator * (const sc_signed&u, int
        v)
friend sc_signed operator * (const sc_signed&u,
unsigned int   v)
friend sc_signed operator * (int64  u, const sc_signed&
  v);
friend sc_signed operator * (uint64     u, const
sc_signed&v);
```



```
friend sc_signed operator * (long  u, const sc_signed&
  v);
friend sc_signed operator * (unsigned long    u, const
sc_signed&v);
friend sc_signed operator * (int    u, const sc_signed&
  v)
friend sc_signed operator * (unsigned int u, const
sc_signed&v)
sc_signed& operator *= (const sc_signed&v);
sc_signed& operator *= (const sc_unsigned&   v);
sc_signed& operator *= (int64                v);
sc_signed& operator *= (uint64              v);
sc_signed& operator *= (long  v);
sc_signed& operator *= (unsigned long        v);
sc_signed& operator *= (int                  v)
sc_signed& operator *= (unsigned int    v)
friend sc_signed operator * (const sc_unsigned&   u,
const sc_int_base&  v);
friend sc_signed operator * (const sc_int_base&   u,
const sc_unsigned&  v);
friend sc_signed operator * (const sc_signed&u, const
sc_int_base& v);
friend sc_signed operator * (const sc_signed&u, const
sc_uint_base& v);
friend sc_signed operator * (const sc_int_base&   u,
const sc_signed&  v);
friend sc_signed operator * (const sc_uint_base& u,
const sc_signed&  v);
sc_signed& operator *= (const sc_int_base&   v);
sc_signed& operator *= (const sc_uint_base& v);

// DIVision operators:
friend sc_signed operator / (const sc_unsigned&   u,
const sc_signed&  v);
friend sc_signed operator / (const sc_signed&u, const
sc_unsigned& v);
friend sc_signed operator / (const sc_unsigned&   u,
int64           v);
friend sc_signed operator / (const sc_unsigned&   u,
long v);
friend sc_signed operator / (const sc_unsigned&   u,
int           v)
friend sc_signed operator / (int64  u, const
sc_unsigned& v);
friend sc_signed operator / (long  u, const
sc_unsigned& v);
friend sc_signed operator / (int    u, const
sc_unsigned& v)
friend sc_signed operator / (const sc_signed&u, const
sc_signed&v);
friend sc_signed operator / (const sc_signed&u, int64
        v);
friend sc_signed operator / (const sc_signed&u, uint64
       v);
```



```
friend sc_signed operator / (const sc_signed&u, long
   v);
friend sc_signed operator / (const sc_signed&u,
unsigned long     v);
friend sc_signed operator / (const sc_signed&u, int
        v)
friend sc_signed operator / (const sc_signed&u,
unsigned int v)
friend sc_signed operator / (int64  u, const sc_signed&
   v);
friend sc_signed operator / (uint64 u, const sc_signed&
   v);
friend sc_signed operator / (long  u, const sc_signed&
   v);
friend sc_signed operator / (unsigned long         u,
const sc_signed&  v);
friend sc_signed operator / (int        u, const
sc_signed&v)
friend sc_signed operator / (unsigned int    u, const
sc_signed&v)
sc_signed& operator /= (const sc_signed&v);
sc_signed& operator /= (const sc_unsigned&   v);
sc_signed& operator /= (int64                 v);
sc_signed& operator /= (uint64               v);
sc_signed& operator /= (long  v);
sc_signed& operator /= (unsigned long        v);
sc_signed& operator /= (int                  v)
sc_signed& operator /= (unsigned int    v)
friend sc_signed operator / (const sc_unsigned&   u,
const sc_int_base&  v);
friend sc_signed operator / (const sc_int_base&   u,
const sc_unsigned&  v);
friend sc_signed operator / (const sc_signed&u, const
sc_int_base& v);
friend sc_signed operator / (const sc_signed&u, const
sc_uint_base& v);
friend sc_signed operator / (const sc_int_base&   u,
const sc_signed&  v);
friend sc_signed operator / (const sc_uint_base& u,
const sc_signed&  v);
sc_signed& operator /= (const sc_int_base&   v);
sc_signed& operator /= (const sc_uint_base& v);

// MODulo operators:
friend sc_signed operator % (const sc_unsigned&   u,
const sc_signed&  v);
friend sc_signed operator % (const sc_signed&u, const
sc_unsigned& v);
friend sc_signed operator % (const sc_unsigned&   u,
int64           v);
friend sc_signed operator % (const sc_unsigned&   u,
long v);
friend sc_signed operator % (const sc_unsigned&   u,
int           v)
```



```
friend sc_signed operator % (int64       u, const
sc_unsigned& v);
friend sc_signed operator % (long       u, const
sc_unsigned& v);
friend sc_signed operator % (int        u, const
sc_unsigned& v)
friend sc_signed operator % (const sc_signed&u, const
sc_signed&v);
friend sc_signed operator % (const sc_signed&u, int64
        v);
friend sc_signed operator % (const sc_signed&u, uint64
       v);
friend sc_signed operator % (const sc_signed&u, long
  v);
friend sc_signed operator % (const sc_signed&u,
unsigned long     v);
friend sc_signed operator % (const sc_signed&u, int
        v)
friend sc_signed operator % (const sc_signed&u,
unsigned int v)
friend sc_signed operator % (int64      u, const
sc_signed&v);
friend sc_signed operator % (uint64              u,
const sc_signed&  v);
friend sc_signed operator % (long                u,
const sc_signed&  v);
friend sc_signed operator % (unsigned long        u,
const sc_signed&  v);
friend sc_signed operator % (int                  u,
const sc_signed&  v)
friend sc_signed operator % (unsigned int        u,
const sc_signed&  v)
sc_signed& operator %= (const sc_signed&v);
sc_signed& operator %= (const sc_unsigned&   v);
sc_signed& operator %= (int64                 v);
sc_signed& operator %= (uint64               v);
sc_signed& operator %= (long  v);
sc_signed& operator %= (unsigned long         v);
sc_signed& operator %= (int                   v)
sc_signed& operator %= (unsigned int    v)
friend sc_signed operator % (const sc_unsigned&   u,
const sc_int_base&  v);
friend sc_signed operator % (const sc_int_base&   u,
const sc_unsigned&  v);
friend sc_signed operator % (const sc_signed&u, const
sc_int_base& v);
friend sc_signed operator % (const sc_signed&u, const
sc_uint_base& v);
friend sc_signed operator % (const sc_int_base&   u,
const sc_signed&  v);
friend sc_signed operator % (const sc_uint_base& u,
const sc_signed&  v);
sc_signed& operator %= (const sc_int_base&   v);
sc_signed& operator %= (const sc_uint_base& v);
```



```
// Bitwise AND operators:
friend sc_signed operator & (const sc_unsigned&   u,
const sc_signed&  v);
friend sc_signed operator & (const sc_signed&u, const
sc_unsigned& v);
friend sc_signed operator & (const sc_unsigned&   u,
int64           v);
friend sc_signed operator & (const sc_unsigned&   u,
long v);
friend sc_signed operator & (const sc_unsigned&   u,
int           v)
friend sc_signed operator & (int64                u,
const sc_unsigned&  v);
friend sc_signed operator & (long                u,
const sc_unsigned&  v);
friend sc_signed operator & (int                  u,
const sc_unsigned&  v)
friend sc_signed operator & (const sc_signed&u, const
sc_signed&v);
friend sc_signed operator & (const sc_signed&u, int64
        v);
friend sc_signed operator & (const sc_signed&u, uint64
       v);
friend sc_signed operator & (const sc_signed&u, long
  v);
friend sc_signed operator & (const sc_signed&u,
unsigned long     v);
friend sc_signed operator & (const sc_signed&u, int
        v)
friend sc_signed operator & (const sc_signed&u,
unsigned int v)
friend sc_signed operator & (int64            u, const
sc_signed&v);
friend sc_signed operator & (uint64          u, const
sc_signed&v);
friend sc_signed operator & (long                u,
const sc_signed&  v);
friend sc_signed operator & (unsigned long    u, const
sc_signed&v);
friend sc_signed operator & (int                  u,
const sc_signed&  v)
friend sc_signed operator & (unsigned int         u,
const sc_signed&  v)
sc_signed& operator &= (const sc_signed&v);
sc_signed& operator &= (const sc_unsigned&   v);
sc_signed& operator &= (int64                v);
sc_signed& operator &= (uint64              v);
sc_signed& operator &= (long  v);
sc_signed& operator &= (unsigned long        v);
sc_signed& operator &= (int                  v)
sc_signed& operator &= (unsigned int    v)
friend sc_signed operator & (const sc_unsigned&   u,
const sc_int_base&  v);
```



```
friend sc_signed operator & (const sc_int_base&   u,
const sc_unsigned&  v);
friend sc_signed operator & (const sc_signed&u, const
sc_int_base& v);
friend sc_signed operator & (const sc_signed&u, const
sc_uint_base& v);
friend sc_signed operator & (const sc_int_base&   u,
const sc_signed&  v);
friend sc_signed operator & (const sc_uint_base& u,
const sc_signed&  v);
sc_signed& operator &= (const sc_int_base&   v);
sc_signed& operator &= (const sc_uint_base& v);

// Bitwise OR operators:
friend sc_signed operator | (const sc_unsigned&   u,
const sc_signed&  v);
friend sc_signed operator | (const sc_signed&u, const
sc_unsigned& v);
friend sc_signed operator | (const sc_unsigned&   u,
int64           v);
friend sc_signed operator | (const sc_unsigned&   u,
long v);
friend sc_signed operator | (const sc_unsigned&   u,
int          v)
friend sc_signed operator | (int64                u,
const sc_unsigned&  v);
friend sc_signed operator | (long                u,
const sc_unsigned&  v);
friend sc_signed operator | (int                  u,
const sc_unsigned&  v)
friend sc_signed operator | (const sc_signed&u, const
sc_signed&v);
friend sc_signed operator | (const sc_signed&u, int64
        v);
friend sc_signed operator | (const sc_signed&u, uint64
       v);
friend sc_signed operator | (const sc_signed&u, long
  v);
friend sc_signed operator | (const sc_signed&u,
unsigned long     v);
friend sc_signed operator | (const sc_signed&u, int
        v)
friend sc_signed operator | (const sc_signed&u,
unsigned int v)
friend sc_signed operator | (int64           u, const
sc_signed&v);
friend sc_signed operator | (uint64          u, const
sc_signed&v);
friend sc_signed operator | (long                u,
const sc_signed&  v);
friend sc_signed operator | (unsigned long    u, const
sc_signed&v);
friend sc_signed operator | (int                  u,
const sc_signed&  v)
```



```
friend sc_signed operator | (unsigned int         u,
const sc_signed&  v)
sc_signed& operator |= (const sc_signed&v);
sc_signed& operator |= (const sc_unsigned&   v);
sc_signed& operator |= (int64                 v);
sc_signed& operator |= (uint64               v);
sc_signed& operator |= (long  v);
sc_signed& operator |= (unsigned long         v);
sc_signed& operator |= (int                   v)
sc_signed& operator |= (unsigned int    v)
friend sc_signed operator | (const sc_unsigned&   u,
const sc_int_base&  v);
friend sc_signed operator | (const sc_int_base&   u,
const sc_unsigned&  v);
friend sc_signed operator | (const sc_signed&u, const
sc_int_base& v);
friend sc_signed operator | (const sc_signed&u, const
sc_uint_base& v);
friend sc_signed operator | (const sc_int_base&   u,
const sc_signed&  v);
friend sc_signed operator | (const sc_uint_base& u,
const sc_signed&  v);
sc_signed& operator |= (const sc_int_base&   v);
sc_signed& operator |= (const sc_uint_base& v);

// Bitwise XOR operators:
friend sc_signed operator ^ (const sc_unsigned&   u,
const sc_signed&  v);
friend sc_signed operator ^ (const sc_signed&u, const
sc_unsigned& v);
friend sc_signed operator ^ (const sc_unsigned&   u,
int64           v);
friend sc_signed operator ^ (const sc_unsigned&   u,
long v);
friend sc_signed operator ^ (const sc_unsigned&   u,
int           v)
friend sc_signed operator ^ (int64                 u,
const sc_unsigned&  v);
friend sc_signed operator ^ (long                 u,
const sc_unsigned&  v);
friend sc_signed operator ^ (int                   u,
const sc_unsigned&  v)
friend sc_signed operator ^ (const sc_signed&u, const
sc_signed&v);
friend sc_signed operator ^ (const sc_signed&u, int64
        v);
friend sc_signed operator ^ (const sc_signed&u, uint64
       v);
friend sc_signed operator ^ (const sc_signed&u, long
  v);
friend sc_signed operator ^ (const sc_signed&u,
unsigned long     v);
friend sc_signed operator ^ (const sc_signed&u, int
        v)
```



```
friend sc_signed operator ^ (const sc_signed&u,
unsigned int v)
friend sc_signed operator ^ (int64            u, const
sc_signed&v);
friend sc_signed operator ^ (uint64          u, const
sc_signed&v);
friend sc_signed operator ^ (long                u,
const sc_signed&  v);
friend sc_signed operator ^ (unsigned long    u, const
sc_signed&v);
friend sc_signed operator ^ (int                  u,
const sc_signed&  v)
friend sc_signed operator ^ (unsigned int         u,
const sc_signed&  v)
sc_signed& operator ^= (const sc_signed&v);
sc_signed& operator ^= (const sc_unsigned&   v);
sc_signed& operator ^= (int64                 v);
sc_signed& operator ^= (uint64               v);
sc_signed& operator ^= (long  v);
sc_signed& operator ^= (unsigned long         v);
sc_signed& operator ^= (int                   v)
sc_signed& operator ^= (unsigned int    v)
friend sc_signed operator ^ (const sc_unsigned&   u,
const sc_int_base&  v);
friend sc_signed operator ^ (const sc_int_base&   u,
const sc_unsigned&  v);
friend sc_signed operator ^ (const sc_signed&u, const
sc_int_base& v);
friend sc_signed operator ^ (const sc_signed&u, const
sc_uint_base& v);
friend sc_signed operator ^ (const sc_int_base&   u,
const sc_signed&  v);
friend sc_signed operator ^ (const sc_uint_base& u,
const sc_signed&  v);
sc_signed& operator ^= (const sc_int_base&   v);
sc_signed& operator ^= (const sc_uint_base& v);

// LEFT SHIFT operators:
friend sc_unsigned operator << (const sc_unsigned&u,
const sc_signed&  v);
friend  sc_signed operator << (const sc_signed&   u,
const sc_unsigned&  v);
friend  sc_signed operator << (const sc_signed&   u,
const sc_signed&  v);
friend  sc_signed operator << (const sc_signed&   u,
int64          v);
friend  sc_signed operator << (const sc_signed&   u,
uint64         v);
friend  sc_signed operator << (const sc_signed&   u,
long v);
friend  sc_signed operator << (const sc_signed&   u,
unsigned long     v);
friend  sc_signed operator << (const sc_signed&   u,
int           v)
```



```
friend  sc_signed operator << (const sc_signed&  u,
unsigned int v)
sc_signed& operator <<= (const sc_signed&   v);
sc_signed& operator <<= (const sc_unsigned&  v);
sc_signed& operator <<= (int64             v);
sc_signed& operator <<= (uint64                v);
sc_signed& operator <<= (long v);
sc_signed& operator <<= (unsigned long       v);
sc_signed& operator <<= (int                 v)
sc_signed& operator <<= (unsigned int   v)
friend  sc_signed operator << (const sc_signed&  u,
const sc_int_base&  v);
friend  sc_signed operator << (const sc_signed&  u,
const sc_uint_base& v);
sc_signed& operator <<= (const sc_int_base&  v);
sc_signed& operator <<= (const sc_uint_base& v);

// RIGHT SHIFT operators:
friend sc_unsigned operator >> (const sc_unsigned&u,
const sc_signed&  v);
friend  sc_signed operator >> (const sc_signed&  u,
const sc_unsigned&  v);
friend  sc_signed operator >> (const sc_signed&  u,
const sc_signed&  v);
friend  sc_signed operator >> (const sc_signed&  u,
int64          v);
friend  sc_signed operator >> (const sc_signed&  u,
uint64        v);
friend  sc_signed operator >> (const sc_signed&  u,
long v);
friend  sc_signed operator >> (const sc_signed&  u,
unsigned long     v);
friend  sc_signed operator >> (const sc_signed&  u,
int          v)
friend  sc_signed operator >> (const sc_signed&  u,
unsigned int v)
sc_signed& operator >>= (const sc_signed&   v);
sc_signed& operator >>= (const sc_unsigned&  v);
sc_signed& operator >>= (int64             v);
sc_signed& operator >>= (uint64                v);
sc_signed& operator >>= (long v);
sc_signed& operator >>= (unsigned long       v);
sc_signed& operator >>= (int                 v)
sc_signed& operator >>= (unsigned int   v)
friend sc_signed operator >> (const sc_signed&    u,
const sc_int_base&  v);
friend sc_signed operator >> (const sc_signed&    u,
const sc_uint_base& v);
sc_signed& operator >>= (const sc_int_base&  v);
sc_signed& operator >>= (const sc_uint_base& v);

// Unary arithmetic operators
friend sc_signed operator + (const sc_signed& u);
friend sc_signed operator - (const sc_signed& u);
```



```
friend sc_signed operator - (const sc_unsigned& u);

// Logical EQUAL operators:
friend bool operator == (const sc_unsigned&  u, const
sc_signed&v);
friend bool operator == (const sc_signed&    u, const
sc_unsigned& v);
friend bool operator == (const sc_signed&    u, const
sc_signed&v);
friend bool operator == (const sc_signed&    u, int64
        v);
friend bool operator == (const sc_signed&    u, uint64
       v);
friend bool operator == (const sc_signed&    u, long
  v);
friend bool operator == (const sc_signed&    u,
unsigned long     v);
friend bool operator == (const sc_signed&    u, int
        v)
friend bool operator == (const sc_signed&    u,
unsigned int v)
friend bool operator == (int64                u, const
sc_signed&v);
friend bool operator == (uint64                   u,
const sc_signed&  v);
friend bool operator == (long                u, const
sc_signed&v);
friend bool operator == (unsigned long        u, const
sc_signed&v);
friend bool operator == (int                  u, const
sc_signed&v)
friend bool operator == (unsigned int        u, const
sc_signed&v)
friend bool operator == (const sc_signed&    u, const
sc_int_base& v);
friend bool operator == (const sc_signed&    u, const
sc_uint_base& v);
friend bool operator == (const sc_int_base&  u, const
sc_signed&v);
friend bool operator == (const sc_uint_base& u, const
sc_signed&v);

// Logical NOT_EQUAL operators:
friend bool operator != (const sc_unsigned&  u, const
sc_signed&v);
friend bool operator != (const sc_signed&    u, const
sc_unsigned& v);
friend bool operator != (const sc_signed&    u, const
sc_signed&v);
friend bool operator != (const sc_signed&    u, int64
        v);
friend bool operator != (const sc_signed&    u, uint64
       v);
```



```
friend bool operator != (const sc_signed&    u, long
   v);
friend bool operator != (const sc_signed&    u,
unsigned long     v);
friend bool operator != (const sc_signed&    u, int
        v)
friend bool operator != (const sc_signed&    u,
unsigned int v)
friend bool operator != (int64                u, const
sc_signed&v);
friend bool operator != (uint64                   u,
const sc_signed&  v);
friend bool operator != (long                u, const
sc_signed&v);
friend bool operator != (unsigned long        u, const
sc_signed&v);
friend bool operator != (int                  u, const
sc_signed&v)
friend bool operator != (unsigned int        u, const
sc_signed&v)
friend bool operator != (const sc_signed&    u, const
sc_int_base& v);
friend bool operator != (const sc_signed&    u, const
sc_uint_base& v);
friend bool operator != (const sc_int_base&  u, const
sc_signed&v);
friend bool operator != (const sc_uint_base& u, const
sc_signed&v);

// Logical LESS_THAN operators:
friend bool operator < (const sc_unsigned&   u, const
sc_signed&v);
friend bool operator < (const sc_signed&u, const
sc_unsigned& v);
friend bool operator < (const sc_signed&u, const
sc_signed&v);
friend bool operator < (const sc_signed&u, int64
      v);
friend bool operator < (const sc_signed&u, uint64
     v);
friend bool operator < (const sc_signed&u, long   v);
friend bool operator < (const sc_signed&u, unsigned
long    v);
friend bool operator < (const sc_signed&u, int
      v)
friend bool operator < (const sc_signed&u, unsigned int
   v)
friend bool operator < (int64                u, const
sc_signed&v);
friend bool operator < (uint64               u, const
sc_signed&v);
friend bool operator < (long                 u, const
sc_signed&v);
```



```
friend bool operator < (unsigned long         u, const
sc_signed&v);
friend bool operator < (int                   u, const
sc_signed&v)
friend bool operator < (unsigned int         u, const
sc_signed&v)
friend bool operator < (const sc_signed&u, const
sc_int_base& v);
friend bool operator < (const sc_signed&u, const
sc_uint_base& v);
friend bool operator < (const sc_int_base&   u, const
sc_signed&v);
friend bool operator < (const sc_uint_base& u, const
sc_signed&v);

// Logical LESS_THAN_AND_EQUAL operators:
friend bool operator <= (const sc_unsigned&  u, const
sc_signed&v);
friend bool operator <= (const sc_signed&    u, const
sc_unsigned& v);
friend bool operator <= (const sc_signed&    u, const
sc_signed&v);
friend bool operator <= (const sc_signed&    u, int64
        v);
friend bool operator <= (const sc_signed&    u, uint64
       v);
friend bool operator <= (const sc_signed&    u, long
  v);
friend bool operator <= (const sc_signed&    u,
unsigned long     v);
friend bool operator <= (const sc_signed&    u, int
        v)
friend bool operator <= (const sc_signed&    u,
unsigned int v)
friend bool operator <= (int64                u, const
sc_signed&v);
friend bool operator <= (uint64                   u,
const sc_signed&  v);
friend bool operator <= (long                u, const
sc_signed&v);
friend bool operator <= (unsigned long        u, const
sc_signed&v);
friend bool operator <= (int                  u, const
sc_signed&v)
friend bool operator <= (unsigned int        u, const
sc_signed&v)
friend bool operator <= (const sc_signed&    u, const
sc_int_base& v);
friend bool operator <= (const sc_signed&    u, const
sc_uint_base& v);
friend bool operator <= (const sc_int_base&  u, const
sc_signed&v);
friend bool operator <= (const sc_uint_base& u, const
sc_signed&v);
```



```
// Logical GREATER_THAN operators:

friend bool operator > (const sc_unsigned&   u, const
sc_signed&v);
friend bool operator > (const sc_signed&u, const
sc_unsigned& v);
friend bool operator > (const sc_signed&u, const
sc_signed&v);
friend bool operator > (const sc_signed&u, int64
      v);
friend bool operator > (const sc_signed&u, uint64
     v);
friend bool operator > (const sc_signed&u, long   v);
friend bool operator > (const sc_signed&u, unsigned
long    v);
friend bool operator > (const sc_signed&u, int
      v)
friend bool operator > (const sc_signed&u, unsigned int
  v)
friend bool operator > (int64                 u, const
sc_signed&v);
friend bool operator > (uint64               u, const
sc_signed&v);
friend bool operator > (long                 u, const
sc_signed&v);
friend bool operator > (unsigned long         u, const
sc_signed&v);
friend bool operator > (int                   u, const
sc_signed&v)
friend bool operator > (unsigned int         u, const
sc_signed&v)
friend bool operator > (const sc_signed&u, const
sc_int_base& v);
friend bool operator > (const sc_signed&u, const
sc_uint_base& v);
friend bool operator > (const sc_int_base&   u, const
sc_signed&v);
friend bool operator > (const sc_uint_base& u, const
sc_signed&v);

// Logical GREATER_THAN_AND_EQUAL operators:
friend bool operator >= (const sc_unsigned&  u, const
sc_signed&v);
friend bool operator >= (const sc_signed&    u, const
sc_unsigned& v);

friend bool operator >= (const sc_signed&    u, const
sc_signed&v);
friend bool operator >= (const sc_signed&    u, int64
        v);
friend bool operator >= (const sc_signed&    u, uint64
       v);
```



```
    friend bool operator >= (const sc_signed&    u, long
      v);
    friend bool operator >= (const sc_signed&    u,
    unsigned long     v);
    friend bool operator >= (const sc_signed&    u, int
            v)
    friend bool operator >= (const sc_signed&    u,
    unsigned int v)
    friend bool operator >= (int64                u, const
    sc_signed&v);
    friend bool operator >= (uint64                   u,
    const sc_signed& v);
    friend bool operator >= (long                u, const
    sc_signed&v);
    friend bool operator >= (unsigned long        u, const
    sc_signed&v);
    friend bool operator >= (int                  u, const
    sc_signed&v)
    friend bool operator >= (unsigned int        u, const
    sc_signed&v)
    friend bool operator >= (const sc_signed&    u, const
    sc_int_base& v);
    friend bool operator >= (const sc_signed&    u, const
    sc_uint_base& v);
    friend bool operator >= (const sc_int_base&  u, const
    sc_signed&v);
    friend bool operator >= (const sc_uint_base& u, const
    sc_signed&v);

    // Bitwise NOT operator (unary).
    friend sc_signed operator ~ (const sc_signed& u);
};
```

Description

sc_signed is an integer with an arbitrary word length W. The word length is specified at construction time and can never change..

Public Constructors

explicit
sc_signed(int nb);
Create an `sc_signed` instance with an initial value of 0 and word length nb.

sc_signed(const sc_signed& a);
Create an `sc_signed` instance with an initial value of a and word length of a.

Copy Constructor

sc_signed(const sc_signed&);

Methods
bool


```
iszero() const;
```
Return true if the value of the sc_signed instance is zero.

```
int
length() const ;
```
Return the word length.

```
void
print( ostream& os = cout ) const ;
```
Print the `sc_uint_base` instance to an output stream.

```
void
reverse();
```
Reverse the contents of the sc_signed instance. I.e. LSB becomes MSB and vice versa.

```
bool
sign() const;
```
Return false.

```
void
scan( istream& is = cin ) ;
```
Read a `sc_uint_base` value from an input stream.

Assignment Operators

```
sc_signed& operator = ( T ) ;
T in { sc_[un]signed, sc_[un]signed_subref†,
    sc_[un]signed_concref†, char*, [u]int64, [unsigned]
    long, [unsigned] int, double, sc_[u]int_base,
    sc_bv_base, sc_lv_base, sc_fxval, sc_fxval_fast,
    sc_fxnum, sc_fxnum_fast }}
```

Assign the value of the right-hand side to the left-hand side. The value is truncated, if its word length is greater than W. If not, the value is sign extended.

Increment and Decrement Operators

```
sc_signed& operator  ++ () ;
const sc_signed operator ++ ( int ) ;
```
The operation is performed as done for type signed int . The result is sign extended if needed.

```
sc_signed& operator  -- () ;
const sc_signed operator  -- ( int ) ;
```
The operation is performed as done for type signed int . The result is sign extended if needed.

Bit Selection

```
sc_signed_bitref   operator [] ( int );
sc_signed_bitref_r operator [] ( int ) const;
```



```
sc_signed_bitref   bit( int );
sc_signed_bitref_r bit( int ) const;
```

Return a reference to a single bit.

Part Selection

```
sc_signed_subref   range( int high, int low );
sc_signed_subref_r range( int high, int low ) const;
sc_signed_subref   operator () ( int high, int low );
sc_signed_subref_r operator () ( int high, int low ) const;
```

Return a reference to a range of bits. The MSB is set to the bit at position high, the LSB is set to the bit at position low.

Arithmetic Assignment Operators

```
friend sc_signed operator OP ( sc_unsigned , sc_signed );
friend sc_signed operator OP ( sc_signed , sc_unsigned );
friend sc_signed operator OP ( sc_signed , sc_signed );
friend sc_signed operator OP ( sc_signed , T );
friend sc_signed operator OP ( T , sc_signed );
T in { sc_[u]int_base, [u]int64, [unsigned] long,
   [unsigned] int }
OP in { + - * / % & | ^ == != < <= > >= }

friend sc_signed operator OP ( sc_unsigned , T );
friend sc_signed operator OP ( T , sc_unsigned );
T in { sc_int_base, int64, long, int }
OP in { + - * / % & | ^ == != < <= > >= }
```

The operation OP is performed and the result is returned.

```
sc_signed& operator OP (T);
T in { sc_[un]signed, sc_[u]int_base, [u]int64, [unsigned]
   long, [unsigned] int }
OP in { += -= *= /= %= &= |= ^= }
```

The operation OP is performed and the result is assigned to the left-hand side.

Shift Operators

```
friend sc_unsigned
operator OP ( sc_unsigned a , sc_signed b );
friend sc_signed
operator OP ( sc_signed a , sc_unsigned b );
friend sc_signed operator OP ( sc_signed a , T b );
T in { sc_[u]int_base, [u]int64, [unsigned] long,
   [unsigned] int }
OP in { << >> }
```

Shift a to the left/right by b bits and return the result.

```
sc_signed& operator OP ( T );
```



```
T in { sc_[un]signed, sc_[u]int_base, [u]int64, [unsigned]
    long, [unsigned] int }
OP in { <<= >>= }
```

Shift the sc_signed instance to the left/right by i bits and assign the result to the sc_signed instance.

Bitwise not

```
friend sc_signed operator ~ ( sc_signed a );
```

Return the bitwise not of a;

Explicit Conversion

```
sc_string to_string( sc_numrep = SC_DEC ) const
sc_string to_string( sc_numrep, bool ) const
```

Convert the sc_signed instance into its string representation.

```
double   to_double() const ;
int      to_int() const ;
int64    to_int64() const ;
long     to_long() const ;
uint64   to_uint64() const ;
unsigned int    to_uint() const ;
unsigned long   to_ulong() const ;
```

Converts the value of sc_signed instance into the corresponding data type. If the requested type has less word length than the sc_signed instance, the value gets truncated accordingly. If the requested type has greater word length than the sc_signed instance, the value gets sign extended, if necessary.

11.66 sc_simcontext

Synopsis

```
class sc_simcontext
{
public:
    // constructors & destructor
    sc_simcontext();
    ~sc_simcontext();

    // other methods
    bool is_running() const;
    int sim_status() const;
    bool update_phase() const;
    uint64 delta_count() const;
    sc_object* first_object();
    sc_object* next_object();
    sc_object* find_object( const char* name );
    const sc_pvector<sc_object*>& get_child_objects()
      const;
    sc_curr_proc_handle get_curr_proc_info();

private:
    // disabled
    sc_simcontext( const sc_simcontext& );
    sc_simcontext& operator = ( const sc_simcontext& );
};
```

Description

sc_simcontext is a class that is used by the simulation kernel to keep track of the current state of simulation. It can provide information to modelers such as the current delta-cycle count, and provides access to any structural element in the design.

Public Constructors and Destructor

`sc_simcontext();`

Default constructor.

`~sc_simcontext();`

Destructor.

Public Member Functions

`uint64`
`delta_count``();`

Returns the absolute delta-cycle count.

`sc_object*`
`find_object``( const char *pathname );`

Returns a pointer to an object in the design hierarchy, such as a module, port, or channel. The pathname argument is the design hierarchy path to the object.

`sc_object*`
`first_object``();`
Returns a pointer to the first object in a collection of all known design objects, such as modules, ports, and signals. Returns 0 if there are no objects in the collection.

`sc_curr_proc_handle`
`get_curr_proc_info``();`
Returns a handle to a current process info object.

`const sc_pvector<sc_object *> &`
`get_child_objects``();`
Returns a collection of top-level design objects that are instantiated in sc_main.

`bool`
`is_running``();`
Returns true while the simulation is running, false otherwise.

`sc_object *`
`next_object``();`
Returns a pointer to the next object in the collection of all known design objects. Used after calling first_object() to iterate through the collection. Returns 0 if there is no next object in the collection.

`int`
`sim_status``();`
Returns the current status of the simulation. Return value is one of

SC_SIM_OK	The simulation state is normal
SC_SIM_ERROR	The simulation encountered an error
SC_SIM_USER_STOP	The simulation was stopped by sc_stop()

`bool`
`update_phase``();`
Returns true if the simulation is in the update phase, false otherwise.

Disabled Member Functions

`sc_simcontext( const sc_simcontext& );`
Copy constructor.

`sc_simcontext& operator = ( const sc_simcontext& );`
Default assignment operator.

11.67 sc_string

Synopsis

```
class sc_string
{
public:
    //  constructor & destructor
    explicit sc_string( int size = 16 );
    sc_string( const char* s );
    sc_string( const char* s, int n );
    sc_string( const sc_string& s );
    ~sc_string();

    // concatenation and assignment
    sc_string& operator = ( const char* s );
    sc_string& operator = ( const sc_string& s );
    sc_string& operator += ( const char* s );
    sc_string& operator += ( char c );
    sc_string& operator += ( const sc_string& s );
    sc_string operator + ( const char* s ) const;
    sc_string operator + ( char c ) const;
    sc_string operator + ( const sc_string& s ) const;
    friend sc_string operator + ( const char* s, const
    sc_string& t );
    sc_string substr( int first, int last ) const;

    // string comparison operators
    bool operator == ( const char* s ) const;
    bool operator != ( const char* s ) const;
    bool operator <  ( const char* s ) const;
    bool operator <= ( const char* s ) const;
    bool operator >  ( const char* s ) const;
    bool operator >= ( const char* s ) const;
    bool operator == ( const sc_string& s ) const;
    bool operator != ( const sc_string& s ) const;
    bool operator <  ( const sc_string& s ) const;
    bool operator <= ( const sc_string& s ) const;
    bool operator >  ( const sc_string& s ) const;
    bool operator >= ( const sc_string& s ) const;

    int length() const;
    const char* c_str() const;
    operator const char*() const;
    char operator[](int index) const;
    char& operator[](int index);
    static sc_string to_string(const char* format, ...);
    template<class T> sc_string& fmt(const T& t);
    sc_string& fmt(const sc_string& s);
    int pos(const sc_string& sub_string)const;
    sc_string& remove(unsigned index, unsigned length);
    sc_string& insert(const sc_string& sub_string, unsigned
    index);
```



```
    bool is_delimiter(const sc_string& str, unsigned index)
    const;
    bool contains(char c) const;
    sc_string uppercase() const;
    sc_string lowercase() const;
    static sc_string make_str(long n);
    void set( int index, char c );
    int cmp( const char* s ) const;
    int cmp( const sc_string& s ) const;
    void print( ostream& os = cout ) const;
};
```

Description

Public Constructors

`explicit`
`sc_string``( int size = 16 );`

Creates an empty string of the given size. Declared explicit to avoid implicit type conversions (int->sc_string).

`sc_string``( const char* s );`

Constructs a string with the same contents (copy) as the argument s.

`sc_string``( const char* s, int n );`

Get first n chars from the string s.

`sc_string``( const sc_string& s );`

Copy constructor.

Public Member Functions

`const char*`
`c_str()`` const;`

Conversion to C-style string.

`bool`
`contains``(char )const;`

Returns `true` if string contains the character.

`sc_string&`
`insert``(const sc_string& sub_string, unsigned index);`

insert `substring` before `index`. The value of `index` should be <= `length()`.

`bool`
`is_delimiter``(const sc_string& str, unsigned index)const;`

Returns true if the character at byte `index` in this string matches any character in the delimiters string. The value of `index` should be < `length()`.

`int`
`length``() const;`

Returns length of the string (excluding trailing \0).

```
sc_string
lowercase()const;
```
Conversion to lowercase.

```
int
pos(const sc_string& sub_string)const;
```
Find position of `substring` in this string. Returns -1 if not found. If `substring` is empty then this function always returns 0.

```
void
print( ostream& os = cout ) const;
```
Print the `sc_string` object to output stream `os`.

```
sc_string&
remove(unsigned index, unsigned length);
```
Remove `length` characters from string starting at `index`. The value of `index` should be < `length()`.

```
sc_string
substr( int first, int last ) const;
```
Returns substring `[first,last]`. Returns empty string if:
(a) `first < 0` or `first >= length()`
(b) `last < 0` or `last >= length()`
(c) `first > last`.

```
static sc_string
to_string(const char* format, ...)
```
String formatting (see `printf` description).

```
sc_string
uppercase()const;
```
Conversion to uppercase.

Public Operators

```
char
operator[] (int index) const;
```
Returns character at position `index`.

```
char&
operator[] (int index);
```
Returns character at position `index`.

```
// concatenation and assignment operators
sc_string& operator = ( const char* s );
sc_string& operator = ( const sc_string& s );

sc_string& operator += ( const char* s );
sc_string& operator += ( char c );
sc_string& operator += ( const sc_string& s );
```



```
sc_string operator + ( const char* s ) const;
sc_string operator + ( char c ) const;
sc_string operator + ( const sc_string& s ) const;

friend sc_string operator + ( const char* s, const
    sc_string& t );

// string comparison operators
bool operator == ( const char* s ) const;
bool operator != ( const char* s ) const;
bool operator <  ( const char* s ) const;
bool operator <= ( const char* s ) const;
bool operator >  ( const char* s ) const;
bool operator >= ( const char* s ) const;
bool operator == ( const sc_string& s ) const;
bool operator != ( const sc_string& s ) const;
bool operator <  ( const sc_string& s ) const;
bool operator <= ( const sc_string& s ) const;
bool operator >  ( const sc_string& s ) const;
bool operator >= ( const sc_string& s ) const;
```


11.68 sc_time

Synopsis

```
class sc_time
{
public:
    // constructors and default assignment operator
    sc_time();
    sc_time( double, sc_time_unit );
    sc_time( const sc_time& );

    sc_time& operator = ( const sc_time& );

    // conversion functions
    uint64 value() const;
    double to_double() const;
    double to_default_time_units() const;
    double to_seconds() const;
    const sc_string to_string() const;

    // relational operators
    bool operator == ( const sc_time& ) const;
    bool operator != ( const sc_time& ) const;
    bool operator <  ( const sc_time& ) const;
    bool operator <= ( const sc_time& ) const;
    bool operator >  ( const sc_time& ) const;
    bool operator >= ( const sc_time& ) const;

    // arithmetic operators
    sc_time& operator += ( const sc_time& );
    sc_time& operator -= ( const sc_time& );
    friend const sc_time operator + ( const sc_time&, const
    sc_time& );
    friend const sc_time operator - ( const sc_time&, const
    sc_time& );
    sc_time& operator *= ( double );
    sc_time& operator /= ( double );
    friend const sc_time operator * ( const sc_time&,
    double );
    friend const sc_time operator * ( double, const
    sc_time& );
    friend const sc_time operator / ( const sc_time&,
    double );
    friend double    operator / ( const sc_time&, const
    sc_time& );

     // other
    void print( ostream& ) const;
    ostream& operator << ( ostream&, const sc_time& );
};
```

Description

The `sc_time` type is used to represent time values or time intervals, internally stored in an unsigned integer of at least 64 bits. Instances are typically created with a numeric value and a time unit `sc_time_unit` (Chapter 13.1.1). If no value is given at the creation of the instance the default value is `SC_ZERO_TIME`.

Example

```
sc_time t( 123, SC_MS ); // t = 123 milliseconds
```

Public Constructors and Default Assignment Operator

`sc_time``();`

Default constructor. Creates an instance with an initial value of `SC_ZERO_TIME`.

`sc_time``( double val, sc_time_unit tu);`

Creates an instance with an initial value of `val` times `tu` time units.

`sc_time``( const sc_time& );`

Copy constructor.

`sc_time&`
`operator =` `( const sc_time& );`

Default assignment operator.

Conversion Functions

`uint64`
`value``() const;`

Converts to type `uint64` relative to the time resolution

`double`
`to_double``() const;`

Converts to type `double` relative to the time resolution

`double`
`to_default_time_units``() const;`

Converts to type `double` in the default time unit.

`double`
`to_seconds``() const;`

Converts to type `double` in the seconds (SC_SEC) unit.

`const sc_string`
`to_string``() const;`

The value is converted to a character string.

Arithmetic Assignment Operators

`sc_time&`
`operator OP` `( const sc_time& ) ;`
`OP` `in { += -= }`


```
sc_time&
operator OP ( double );
OP in { *= /= }
```

Relational Operators

```
bool
operator op ( const sc_time& ) const;
OP in { == != < <= > >=}
```

Arithmetic Operators

```
friend const sc_time
operator OP ( const sc_time&, const sc_time& );
OP in { + - }

friend const sc_time
operator * ( const sc_time&, double );
OP in { * / }

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

friend double
operator / ( const sc_time&, const sc_time& );
```

Public Member Functions

```
void
print( ostream& ) const;
```

Prints the `sc_time` value to an output stream.

Global Functions

```
ostream&
operator << ( ostream& os, const sc_time& a )
```

Prints the value of a to output stream os.

11.69 sc_ufix

Inheritance

Synopsis

```
class sc_ufix : public sc_fxnum
{
public:
    // constructors
    explicit sc_ufix( sc_fxnum_observer* = 0 );
    sc_ufix( int, int, sc_fxnum_observer* = 0 );
    sc_ufix( sc_q_mode, sc_o_mode,
      sc_fxnum_observer* = 0 );
    sc_ufix( sc_q_mode, sc_o_mode, int,
      sc_fxnum_observer* = 0 );
    sc_ufix( int, int, sc_q_mode, sc_o_mode,
      sc_fxnum_observer* = 0 );
    sc_ufix( int, int, sc_q_mode, sc_o_mode, int,
      sc_fxnum_observer* = 0 );
    explicit sc_ufix( const sc_fxcast_switch&,
      sc_fxnum_observer* = 0 );
    sc_ufix( int, int, const sc_fxcast_switch&,
      sc_fxnum_observer* = 0 );
    sc_ufix( sc_q_mode, sc_o_mode,
      const sc_fxcast_switch&,
      sc_fxnum_observer* = 0 );
    sc_ufix( sc_q_mode, sc_o_mode, int,
      const sc_fxcast_switch&,
      sc_fxnum_observer* = 0 );
    sc_ufix( int, int, sc_q_mode, sc_o_mode,
      const sc_fxcast_switch&,
      sc_fxnum_observer* = 0 );
    sc_ufix( int, int, sc_q_mode, sc_o_mode, int,
      const sc_fxcast_switch&,
      sc_fxnum_observer* = 0 );
    explicit sc_ufix( const sc_fxtype_params&,
      sc_fxnum_observer* = 0 );
    sc_ufix( const sc_fxtype_params&,
      const sc_fxcast_switch&,
    sc_fxnum_observer* = 0 );

#define DECL_CTORS_T(tp)      \
    sc_ufix( tp, int, int, sc_fxnum_observer* = 0 ); \
    sc_ufix( tp, sc_q_mode, sc_o_mode,\
      sc_fxnum_observer* = 0 );   \
    sc_ufix( tp, sc_q_mode, sc_o_mode, int,  \
      sc_fxnum_observer* = 0 );   \
    sc_ufix( tp,  int, int, sc_q_mode, sc_o_mode, \
      sc_fxnum_observer* = 0 );   \
    sc_ufix( tp,  int, int, sc_q_mode, sc_o_mode, int,\
      sc_fxnum_observer* = 0 );   \
    sc_ufix( tp,  const sc_fxcast_switch&,   \
      sc_fxnum_observer* = 0 );   \
```



```
    sc_ufix( tp, int, int, const sc_fxcast_switch&,    \
      sc_fxnum_observer* = 0 );   \
    sc_ufix( tp, sc_q_mode, sc_o_mode,\
      const sc_fxcast_switch&,    \
      sc_fxnum_observer* = 0 );   \
    sc_ufix( tp, sc_q_mode, sc_o_mode, int,  \
      const sc_fxcast_switch&,    \
      sc_fxnum_observer* = 0 );   \
    sc_ufix( tp, int, int, sc_q_mode, sc_o_mode, \
      const sc_fxcast_switch&,    \
      sc_fxnum_observer* = 0 );   \
    sc_ufix( tp, \
      int, int, sc_q_mode, sc_o_mode, int,  \
      const sc_fxcast_switch&,    \
      sc_fxnum_observer* = 0 );   \
    sc_ufix( tp, const sc_fxtype_params&,    \
      sc_fxnum_observer* = 0 );   \
    sc_ufix( tp, const sc_fxtype_params&,    \
      const sc_fxcast_switch&,    \
      sc_fxnum_observer* = 0 );

#define DECL_CTORS_T_A(tp)   \
    sc_ufix( tp,sc_fxnum_observer* = 0 );  \
    DECL_CTORS_T(tp)

#define DECL_CTORS_T_B(tp)   \
    explicit sc_ufix( tp,     \
    sc_fxnum_observer* = 0 ); \
    DECL_CTORS_T(tp)

    DECL_CTORS_T_A(int)
    DECL_CTORS_T_A(unsigned int)
    DECL_CTORS_T_A(long)
    DECL_CTORS_T_A(unsigned long)
    DECL_CTORS_T_A(double)
    DECL_CTORS_T_A(const char*)
    DECL_CTORS_T_A(const sc_fxval&)
    DECL_CTORS_T_A(const sc_fxval_fast&)
    DECL_CTORS_T_A(const sc_fxnum&)
    DECL_CTORS_T_A(const sc_fxnum_fast&)
    DECL_CTORS_T_B(int64)
    DECL_CTORS_T_B(uint64)
    DECL_CTORS_T_B(const sc_int_base&)
    DECL_CTORS_T_B(const sc_uint_base&)
    DECL_CTORS_T_B(const sc_signed&)
    DECL_CTORS_T_B(const sc_unsigned&)

#undef DECL_CTORS_T
#undef DECL_CTORS_T_A
#undef DECL_CTORS_T_B

    // copy constructor
    sc_ufix( const sc_ufix& );
```



```
    // unary bitwise operators
    const sc_ufix operator ~ () const;

    // unary bitwise functions
    friend void b_not( sc_ufix&, const sc_ufix& );

    // binary bitwise operators
    friend const sc_ufix operator & ( const sc_ufix&, const
    sc_ufix& );
    friend const sc_ufix operator & ( const sc_ufix&, const
    sc_ufix_fast& );
    friend const sc_ufix operator & ( const sc_ufix_fast&,
    const sc_ufix& );
    friend const sc_ufix operator | ( const sc_ufix&, const
    sc_ufix& );
    friend const sc_ufix operator | ( const sc_ufix&, const
    sc_ufix_fast& );
    friend const sc_ufix operator | ( const sc_ufix_fast&,
    const sc_ufix& );
    friend const sc_ufix operator ^ ( const sc_ufix&, const
    sc_ufix& );
    friend const sc_ufix operator ^ ( const sc_ufix&, const
    sc_ufix_fast& );
    friend const sc_ufix operator ^ ( const sc_ufix_fast&,
    const sc_ufix& );


    // binary bitwise functions
    friend void b_and( sc_ufix&, const sc_ufix&, const
    sc_ufix& );
    friend void b_and( sc_ufix&, const sc_ufix&, const
    sc_ufix_fast& );
    friend void b_and( sc_ufix&, const sc_ufix_fast&, const
    sc_ufix& );
    friend void b_or ( sc_ufix&, const sc_ufix&, const
    sc_ufix& );
    friend void b_or ( sc_ufix&, const sc_ufix&, const
    sc_ufix_fast& );
    friend void b_or ( sc_ufix&, const sc_ufix_fast&, const
    sc_ufix& );
    friend void b_xor( sc_ufix&, const sc_ufix&, const
    sc_ufix& );
    friend void b_xor( sc_ufix&, const sc_ufix&, const
    sc_ufix_fast& );
    friend void b_xor( sc_ufix&, const sc_ufix_fast&, const
    sc_ufix& );

    // assignment operators
    sc_ufix& operator = ( const sc_ufix& );

#define DECL_ASN_OP_T(op,tp) \
    sc_ufix& operator op ( tp );

#ifndef SC_FX_EXCLUDE_OTHER
```



```
#define DECL_ASN_OP_OTHER(op) \
    DECL_ASN_OP_T(op,int64)   \
    DECL_ASN_OP_T(op,uint64)  \
    DECL_ASN_OP_T(op,const sc_int_base&)      \
    DECL_ASN_OP_T(op,const sc_uint_base&)    \
    DECL_ASN_OP_T(op,const sc_signed&)\
    DECL_ASN_OP_T(op,const sc_unsigned&)
#else
#define DECL_ASN_OP_OTHER(op)
#endif

#define DECL_ASN_OP(op)    \
    DECL_ASN_OP_T(op,int)     \
    DECL_ASN_OP_T(op,unsigned int)\
    DECL_ASN_OP_T(op,long)   \
    DECL_ASN_OP_T(op,unsigned long)   \
    DECL_ASN_OP_T(op,double)  \
    DECL_ASN_OP_T(op,const char*)  \
    DECL_ASN_OP_T(op,const sc_fxval&) \
    DECL_ASN_OP_T(op,const sc_fxval_fast&)  \
    DECL_ASN_OP_T(op,const sc_fxnum&) \
    DECL_ASN_OP_T(op,const sc_fxnum_fast&)  \
    DECL_ASN_OP_OTHER(op)

    DECL_ASN_OP(=)
    DECL_ASN_OP(*=)
    DECL_ASN_OP(/=)
    DECL_ASN_OP(+=)
    DECL_ASN_OP(-=)
    DECL_ASN_OP_T(<<=,int)
    DECL_ASN_OP_T(>>=,int)
    DECL_ASN_OP_T(&=,const sc_ufix&)
    DECL_ASN_OP_T(&=,const sc_ufix_fast&)
    DECL_ASN_OP_T(|=,const sc_ufix&)
    DECL_ASN_OP_T(|=,const sc_ufix_fast&)
    DECL_ASN_OP_T(^=,const sc_ufix&)
    DECL_ASN_OP_T(^=,const sc_ufix_fast&)

#undef DECL_ASN_OP_T
#undef DECL_ASN_OP_OTHER
#undef DECL_ASN_OP

    // auto-increment and auto-decrement
    const sc_fxval operator ++ ( int );
    const sc_fxval operator -- ( int );
    sc_ufix& operator ++ ();
    sc_ufix& operator -- ();
};
```

Description

Unconstrained type **sc_ufix** is an unsigned type. sc_ufix allows specifying the fixed-point type parameters wl, iwl, q_mode, o_mode, and n_bits as variables. See Chapter 6.8.5.

Declaration Syntax
```
sc_ufix var_name([init_val]
            [,wl,iwl]
            [,q_mode,o_mode[,n_bits]]
            [,cast_switch]
            [,observer]);

sc_ufix var_name([init_val]
            ,type_params
            [,cast_switch]
            [,observer]);
```

Examples
```
sc_ufix b(0,32,32);
sc_ufix d(a+b);
sc_ufix c = 0.1;
```

Public Constructors
```
sc_ufix (
    [type_ init_val]
    [,int wl,int iwl]
    [,sc_q_mode q_mode,sc_o_mode o_mode[,int n_bits]]
    [,const sc_fxcast_switch& cast_switch]
    , sc_fxnum_observer* observer) ;
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }

sc_ufix (
    [type_ init_val]
    ,const sc_fxtype_param& type_params
    [,sc_fxcast_switch cast_switch]
    , sc_fxnum_observer* observer) ;
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base&, const sc_uint_base†&, const
    sc_signed&, const sc_unsigned, const sc_fxval&, const
    sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

Notes on `type_`
For all types in `type_` , except `sc_[u]fix` and `sc_[u]fix_fast`, only the value of the argument is taken, that is, any type information is discarded. This ensures that initialization during declaration and initialization after declaration behave identical.

A fixed-point variable can be initialized with a C/C++ character string (type const char*) either when the number will be expressed in binary form or when the number is too large to be written as a C/C++ built-in type literal

init_val
The initial value of the variable. If the initial value is not specified, the instance is uninitialized.
wl
The total number of bits in the fixed-point format. `wl` must be greater than zero, otherwise, a runtime error is produced. The default value for `wl` is obtained from the fixed-point context type `sc_fxtype_context`. See Chapter 11.26. The total word length parameter cannot change after declaration.
iwl
The number of integer bits in the fixed-point format. `iwl` can be positive or negative. The default value for `iwl` is obtained from the fixed-point context type `sc_fxtype_context`. See See Chapter 11.26. The number of integer bits parameter cannot change after declaration.
q_mode
The quantization mode to use. Valid values for `q_mode` are given in Chapter 6.8.12.7. The default value for `q_mode` is obtained from the fixed-point context type `sc_fxtype_context`. See See Chapter 11.26. The quantization mode parameter cannot change after declaration.
o_mode
The overflow mode to use. Valid values for `o_mode` are given in Chapter 6.8.12.1. The default value for `o_mode` is obtained from the fixed-point context type `sc_fxtype_context`. See Chapter 11.26. The overflow mode parameter cannot change after declaration.
n_bits
The number of saturated bits parameter for the selected overflow mode. `n_bits` must be greater than or equal to zero, otherwise a runtime error is produced. If the overflow mode is specified, the default value is zero. If the overflow mode is not specified, the default value is obtained from the fixed-point context type `sc_fxtype_context`. See Chapter 11.26. The number of saturated bits parameter cannot change after declaration.
type_params
A fixed-point type parameters object.
cast_switch
The cast switch, which allows to switch fixed-point type casting on or off. Valid values for `cast_switch` are:
`SC_OFF` for casting off
`SC_ON` for casting on
The default value for `cast_switch` is obtained from the fixed-point context type `sc_fxcast_context`. The `cast_switch` parameter cannot change after declaration.
observer

A pointer to an observer object. The `observer` argument is of type `sc_fxnum_observer*`. See Chapter 11.25. The default value for `observer` is 0 (null pointer). The `observer` parameter cannot change after declaration.

Copy Constructor

sc_ufix(const sc_ufix&);

Operators

The operators defined for the `sc_ufix` are given in Table 31.

Table 31. Operators for sc_ufix

Operator class	Operators in class
Bitwise	~ & ^ |
Arithmetic	* / + - << >> ++ --
Equality	== !=
Relational	<<= >>=
Assignment	= *= /= += -= <<= >>= &= ^= |=

Note:
Operator << and operator >> define arithmetic shifts, not bitwise shifts. The difference is that no bits are lost and proper sign extension is done.

In expressions with the non-bitwise operators from Table 31, fixed-point types can be mixed with all types given:

```
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

The return type of any arithmetic operation is the fixed-point value type, which guarantees that the operation is performed without overflow or quantization.

A floating-point variable or a fixed-point value variable can contain one of the special values +Inf (plus infinity), -Inf (minus infinity), or Nan (not a number). Assignment of one of these special values to a fixed-point variable will produce a runtime error.

For the fixed-point types, a minimal set of bitwise operators is defined. These bitwise operators are only defined on either the signed fixed-point types or the unsigned fixed- point types. Mixing between signed and unsigned fixed-point types is not allowed. Mixing with any other type is also not allowed.

The semantics of the bitwise operators is as follows. For the unary ~ operator, the type of the result is the type of the operand. The bits in the two's complement mantissa of the operand are inverted to get the mantissa of the result. For the binary operators, the type of the result is the maximum aligned type of the two operands, that is, the two operands are aligned by the binary point and the maximum integer word length and the maximum fractional word length is taken. The operands are temporarily extended to this type before performing a bitwise and, bitwise exclusive-or, or bitwise or.

Member Functions

The functions defined for `sc_ufix` are given in Table 32.

Table 32. Functions for sc_ufix

Function class	**Functions in class**
Bitwise	b_not, b_and, b_xor, b_or
Arithmetic	neg, mult, div, add, sub, lshift, rshift

The functions in Table 32 have return type void. The first argument of these functions is a reference to the result object. The remaining arguments of these functions are the operands.

For the bitwise functions, the result object and the operands are of the same type, which is either `sc_fix` or `sc_ufix`.

The `neg` arithmetic function takes one operand, the other arithmetic functions take two operands. At least one of the operands of the arithmetic functions should have a fixed- point type, the other operand can have any of the types given:

```
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

The arithmetic functions are defined twice: once with the result object of type sc_fxval, and once with the result object of type sc_ufix.

Bit Selection

```
const sc_fxnum_bitref† operator [] ( int i) const;
sc_fxnum_bitref†    operator [] ( int i);

const sc_fxnum_bitref† bit( int i) const;
sc_fxnum_bitref†    bit( int i);
```


These functions take one argument of type int, which is the index into the fixed-point mantissa. The index argument must be between `wl-1` (MSB) and 0 (LSB). Otherwise, a runtime error is produced. The return type of the bit selection functions is (const or non- const) *sc_fxnum_bitref*†, which is a proxy class. The proxy class allows bit selection to be used both as `rvalue` (for reading) and `lvalue` (for writing). For bit selection, the fixed-point binary point is ignored.

Part Selection

```
const sc_fxnum_subref† operator () ( int, int ) const;
sc_fxnum_subref†    operator () ( int, int );

const sc_fxnum_subref† range( int, int ) const;
sc_fxnum_subref†    range( int, int );
```

These functions take two arguments of type `int`, which are the begin and end indices into the fixed-point mantissa. The index arguments must be between `wl-1` (MSB) and 0 (LSB). Otherwise, a runtime error is produced. The return type of the part selection functions is (const or non-const) *sc_fxnum_subref*†, which is a proxy class that behaves like type `sc_bv_base`. The proxy class allows part selection to be used both as `rvalue` (for reading) and `lvalue` (for writing). All operators and methods that are available for type `sc_bv_base` are also available for part selection. For part selection, the fixed-point binary point is ignored.

```
const sc_fxnum_subref† operator () () const;
sc_fxnum_subref†    operator () ();

const sc_fxnum_subref† range() const;
sc_fxnum_subref†    range();
```

As a shortcut for part selection of the complete mantissa, operator () and the range() method can be called without any arguments.

Query Parameters

```
const sc_fxcast_switch&
cast_switch() const;
```

Returns the cast switch parameter.

```
int
iwl() const;
```

Returns the integer word length parameter.

```
int
```



```
n_bits() const;
```
Returns the number of saturated bits parameter.

```
sc_o_mode
o_mode() const;
```
Returns the overflow mode parameter.

```
sc_q_mode
q_mode() const;
```
Return the quantization mode parameter.

```
const sc_fxtype_params&
type_params() const;
```
Returns the type parameters.

```
int
wl() const;
```
Returns the total word length parameter.

Query Value

```
bool
is_neg() const;
```
Always returns false.

```
bool
is_zero() const;
```
Returns true if the variable holds a zero value. Returns false otherwise.

```
bool
overflow_flag() const;
```
Returns true if the last write action on this variable caused overflow. Returns false otherwise.

```
bool
quantization_flag() const;
```
Returns true if the last write action on this variable caused quantization. Returns false otherwise.

```
const sc_fxval
value() const;
```
Returns the value.

Implicit Conversion

```
operator double() const;
```
Implicit conversion to the implementation type `double`. The value does not change.

Explicit Conversion


```
short          to_short() const;
unsigned short  to_ushort() const;
int            to_int() const;
unsigned int    to_uint() const;
long            to_long() const;
unsigned long   to_ulong() const;
float           to_float() const;
double          to_double() const
```

```
const sc_string to_string() const;
const sc_string to_string( sc_numrep ) const;
const sc_string to_string( sc_numrep, bool ) const;
const sc_string to_string( sc_fmt ) const;
const sc_string to_string( sc_numrep, sc_fmt ) const;
const sc_string to_string( sc_numrep, bool, sc_fmt ) const;
```

The value of a fixed-point variable can be converted to a character string with the to_string() method. This method takes different arguments for formatting purposes. See Chapter 6.8.8 for more information on converting fixed-point variables to/from character strings. Furthermore, writing to C++ output streams with operator << is supported, e.g. cout << a;, where a is a fixed-point variable. The decimal number representation is used in this case.

```
const sc_string to_dec() const;
const sc_string to_bin() const;
const sc_string to_oct() const;
const sc_string to_hex() const;
```

Shortcut methods for conversion to a character string. See Chapter 6.8.9.2.

Print or dump content

```
void
print( ostream& = cout ) const;
```

Print the `sc_ufix` instance value to an output stream.

```
void
scan( istream& = cin );
```

Read an `sc_ufix` value from an input stream.

```
void
dump( ostream& = cout )
const;
```

Prints the `sc_ufix` instance value, parameters and flags to an output stream.

```
ostream&
operator << ( ostream& os, const sc_ufix& a )
```

Print the instance value of `a` to an output stream `os`.

11.70 sc_ufix_fast

Synopsis

```
class sc_ufix_fast : public sc_fxnum_fast
{
public:
    // constructors
    explicit sc_ufix_fast( sc_fxnum_fast_observer* = 0 );
    sc_ufix_fast( int, int,
      sc_fxnum_fast_observer* = 0 );
    sc_ufix_fast( sc_q_mode, sc_o_mode,
      sc_fxnum_fast_observer* = 0 );
    sc_ufix_fast( sc_q_mode, sc_o_mode, int,
      sc_fxnum_fast_observer* = 0 );
    sc_ufix_fast( int, int, sc_q_mode, sc_o_mode,
      sc_fxnum_fast_observer* = 0 );
    sc_ufix_fast( int, int, sc_q_mode, sc_o_mode, int,
      sc_fxnum_fast_observer* = 0 );
    explicit sc_ufix_fast( const sc_fxcast_switch&,
      sc_fxnum_fast_observer* = 0 );
    sc_ufix_fast( int, int, const sc_fxcast_switch&,
      sc_fxnum_fast_observer* = 0 );
    sc_ufix_fast( sc_q_mode, sc_o_mode,
      const sc_fxcast_switch&,
      sc_fxnum_fast_observer* = 0 );
    sc_ufix_fast( sc_q_mode, sc_o_mode, int,
      const sc_fxcast_switch&,
      sc_fxnum_fast_observer* = 0 );
    sc_ufix_fast( int, int, sc_q_mode, sc_o_mode,
      const sc_fxcast_switch&,
      sc_fxnum_fast_observer* = 0 );
    sc_ufix_fast( int, int, sc_q_mode, sc_o_mode, int,
      const sc_fxcast_switch&,
      sc_fxnum_fast_observer* = 0 );
    explicit sc_ufix_fast( const sc_fxtype_params&,
      sc_fxnum_fast_observer* = 0 );
    sc_ufix_fast( const sc_fxtype_params&,
      const sc_fxcast_switch&,
      sc_fxnum_fast_observer* = 0 );

#define DECL_CTORS_T(tp)     \
   sc_ufix_fast( tp, int, int,  \
      sc_fxnum_fast_observer* = 0 ); \
   sc_ufix_fast( tp, sc_q_mode, sc_o_mode,\
      sc_fxnum_fast_observer* = 0 ); \
   sc_ufix_fast( tp, sc_q_mode, sc_o_mode, int,  \
      sc_fxnum_fast_observer* = 0 ); \
   sc_ufix_fast( tp, int, int, sc_q_mode, sc_o_mode,\
      sc_fxnum_fast_observer* = 0 ); \
   sc_ufix_fast( tp,    \
      int, int, sc_q_mode, sc_o_mode, int,  \
      sc_fxnum_fast_observer* = 0 ); \
   sc_ufix_fast( tp, const sc_fxcast_switch&,  \
```



```
        sc_fxnum_fast_observer* = 0 ); \
    sc_ufix_fast( tp, int, int,  \
        const sc_fxcast_switch&,       \
        sc_fxnum_fast_observer* = 0 ); \
    sc_ufix_fast( tp, sc_q_mode, sc_o_mode,\
        const sc_fxcast_switch&,       \
        sc_fxnum_fast_observer* = 0 ); \
    sc_ufix_fast( tp, sc_q_mode, sc_o_mode, int,  \
        const sc_fxcast_switch&,       \
        sc_fxnum_fast_observer* = 0 ); \
    sc_ufix_fast( tp,    \
        int, int, sc_q_mode, sc_o_mode,\
        const sc_fxcast_switch&,       \
        sc_fxnum_fast_observer* = 0 ); \
    sc_ufix_fast( tp,    \
        int, int, sc_q_mode, sc_o_mode, int, \
        const sc_fxcast_switch&,       \
        sc_fxnum_fast_observer* = 0 ); \
    sc_ufix_fast( tp, const sc_fxtype_params&,      \
        sc_fxnum_fast_observer* = 0 ); \
    sc_ufix_fast( tp,  const sc_fxtype_params&,   \
        const sc_fxcast_switch&,       \
        sc_fxnum_fast_observer* = 0 );

#define DECL_CTORS_T_A(tp)  \
    sc_ufix_fast( tp,    \
         sc_fxnum_fast_observer* = 0 ); \
     DECL_CTORS_T(tp)

#define DECL_CTORS_T_B(tp)  \
     explicit sc_ufix_fast( tp,         \
         sc_fxnum_fast_observer* = 0 ); \
     DECL_CTORS_T(tp)

     DECL_CTORS_T_A(int)
     DECL_CTORS_T_A(unsigned int)
     DECL_CTORS_T_A(long)
     DECL_CTORS_T_A(unsigned long)
     DECL_CTORS_T_A(double)
     DECL_CTORS_T_A(const char*)
     DECL_CTORS_T_A(const sc_fxval&)
     DECL_CTORS_T_A(const sc_fxval_fast&)
     DECL_CTORS_T_A(const sc_fxnum&)
     DECL_CTORS_T_A(const sc_fxnum_fast&)
     DECL_CTORS_T_B(int64)
     DECL_CTORS_T_B(uint64)
     DECL_CTORS_T_B(const sc_int_base&)
     DECL_CTORS_T_B(const sc_uint_base&)
     DECL_CTORS_T_B(const sc_signed&)
     DECL_CTORS_T_B(const sc_unsigned&)

#undef DECL_CTORS_T
#undef DECL_CTORS_T_A
#undef DECL_CTORS_T_B
```



```
    // copy constructor
    sc_ufix_fast( const sc_ufix_fast& );

    // unary bitwise operators
    const sc_ufix_fast operator ~ () const;

    // unary bitwise functions
   friend void b_not( sc_ufix_fast&, const sc_ufix_fast& );

    // binary bitwise operators
    friend const sc_ufix_fast operator & ( const
    sc_ufix_fast&, const sc_ufix_fast& );
    friend const sc_ufix_fast operator ^ ( const
    sc_ufix_fast&, const sc_ufix_fast& );
    friend const sc_ufix_fast operator | ( const
    sc_ufix_fast&, const sc_ufix_fast& );

    // binary bitwise functions
    friend void b_and( sc_ufix_fast&, const sc_ufix_fast&,
    const sc_ufix_fast& );
    friend void b_or ( sc_ufix_fast&, const sc_ufix_fast&,
    const sc_ufix_fast& );
    friend void b_xor( sc_ufix_fast&, const sc_ufix_fast&,
    const sc_ufix_fast& );

    // assignment operators
    sc_ufix_fast& operator = ( const sc_ufix_fast& );
#define DECL_ASN_OP_T(op,tp) \
    sc_ufix_fast& operator op ( tp );

#ifndef SC_FX_EXCLUDE_OTHER
#define DECL_ASN_OP_OTHER(op)          \
    DECL_ASN_OP_T(op,int64) \
    DECL_ASN_OP_T(op,uint64)\
    DECL_ASN_OP_T(op,const sc_int_base&)         \
    DECL_ASN_OP_T(op,const sc_uint_base&)      \
    DECL_ASN_OP_T(op,const sc_signed&)\
    DECL_ASN_OP_T(op,const sc_unsigned&)
#else
#define DECL_ASN_OP_OTHER(op)
#endif

#define DECL_ASN_OP(op) \
    DECL_ASN_OP_T(op,int)   \
    DECL_ASN_OP_T(op,unsigned int)\
    DECL_ASN_OP_T(op,long)  \
    DECL_ASN_OP_T(op,unsigned long)   \
    DECL_ASN_OP_T(op,double)\
    DECL_ASN_OP_T(op,const char*)  \
    DECL_ASN_OP_T(op,const sc_fxval&) \
    DECL_ASN_OP_T(op,const sc_fxval_fast&)    \
    DECL_ASN_OP_T(op,const sc_fxnum&) \
    DECL_ASN_OP_T(op,const sc_fxnum_fast&)    \
```



```
    DECL_ASN_OP_OTHER(op)

    DECL_ASN_OP(=)
    DECL_ASN_OP(*=)
    DECL_ASN_OP(/=)
    DECL_ASN_OP(+=)
    DECL_ASN_OP(-=)
    DECL_ASN_OP_T(<<=,int)
    DECL_ASN_OP_T(>>=,int)
    DECL_ASN_OP_T(&=,const sc_ufix&)
    DECL_ASN_OP_T(&=,const sc_ufix_fast&)
    DECL_ASN_OP_T(|=,const sc_ufix&)
    DECL_ASN_OP_T(|=,const sc_ufix_fast&)
    DECL_ASN_OP_T(^=,const sc_ufix&)
    DECL_ASN_OP_T(^=,const sc_ufix_fast&)

#undef DECL_ASN_OP_T
#undef DECL_ASN_OP_OTHER
#undef DECL_ASN_OP

    // auto-increment and auto-decrement
    const sc_fxval_fast operator ++ ( int );
    const sc_fxval_fast operator -- ( int );
    sc_ufix_fast& operator ++ ();
    sc_ufix_fast& operator -- ();
};
```

Description

`sc_ufix_fast` is an unsigned limited precision type. `sc_ufix_fast` allows specifying the fixed-point type parameters wl, iwl, q_mode, o_mode, and n_bits as variables. See Chapter 6.8.5.

`sc_ufix_fast` provides the same API as `sc_ufix`.

`sc_ufix_fast` uses double precision (floating-point) values. The mantissa of a double precision value is limited to 53 bits. This means that bit-true behavior cannot be guaranteed with the limited precision types. For bit-true behavior with the limited precision types, the following guidelines should be followed:
Make sure that the word length of the result of any operation or expression does not exceed 53 bits.
The result of an addition or subtraction requires a word length that is one bit more than the maximum *aligned* word length of the two operands.
The result of a multiplication requires a word length that is the sum of the word lengths of the two operands.

Declaration Syntax

```
sc_ufix_fast var_name([init_val]
             [,wl,iwl]
             [,q_mode,o_mode[,n_bits]]
             [,cast_switch]
```



```
            [,observer]);

sc_ufix_fast var_name([init_val]
            ,type_params
            [,cast_switch]
            [,observer]);
```

Examples

```
sc_ufix_fast b(0,32,32);
sc_ufix_fast d(a+b);
```

Public Constructors

```
sc_ufix_fast (
    [type_ init_val]
    [,int wl,int iwl]
    [,sc_q_mode q_mode,sc_o_mode o_mode[,int n_bits]]
    [,const sc_fxcast_switch& cast_switch]
    , sc_fxnum_fast_observer* observer) ;
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }

sc_ufix_fast (
    [type_ init_val]
    ,const sc_fxtype_param& type_params
    [,sc_fxcast_switch cast_switch]
    , sc_fxnum_fast_observer* observer) ;
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

Notes on `type_`

For all types in `type_` , except `sc_[u]fix` and `sc_[u]fix_fast`, only the value of the argument is taken, that is, any type information is discarded. This ensures that initialization during declaration and initialization after declaration behave identical.

A fixed-point variable can be initialized with a C/C++ character string (type const char*) either when the number will be expressed in binary form or when the number is too large to be written as a C/C++ built-in type literal

init_val

The initial value of the variable. If the initial value is not specified, the instance is uninitialized.

wl

The total number of bits in the fixed-point format. `wl` must be greater than zero, otherwise, a runtime error is produced. The default value for `wl` is obtained from the fixed-point context type `sc_fxtype_context`. See Chapter 11.26. The total word length parameter cannot change after declaration.

iwl

The number of integer bits in the fixed-point format. `iwl` can be positive or negative. The default value for `iwl` is obtained from the fixed-point context type `sc_fxtype_context`. See See Chapter 11.26. The number of integer bits parameter cannot change after declaration.

q_mode

The quantization mode to use. Valid values for `q_mode` are given in Chapter 6.8.12.7. The default value for `q_mode` is obtained from the fixed-point context type `sc_fxtype_context`. See See Chapter 11.26. The quantization mode parameter cannot change after declaration.

o_mode

The overflow mode to use. Valid values for `o_mode` are given in Chapter 6.8.12.1. The default value for `o_mode` is obtained from the fixed-point context type `sc_fxtype_context`. See Chapter 11.26. The overflow mode parameter cannot change after declaration.

n_bits

The number of saturated bits parameter for the selected overflow mode. `n_bits` must be greater than or equal to zero, otherwise a runtime error is produced. If the overflow mode is specified, the default value is zero. If the overflow mode is not specified, the default value is obtained from the fixed-point context type `sc_fxtype_context`. See Chapter 11.26. The number of saturated bits parameter cannot change after declaration.

type_params

A fixed-point type parameters object.

cast_switch

The cast switch, which allows to switch fixed-point type casting on or off. Valid values for `cast_switch` are:

`SC_OFF` for casting off

`SC_ON` for casting on

The default value for `cast_switch` is obtained from the fixed-point context type `sc_fxcast_context`.. The `cast_switch` parameter cannot change after declaration.

observer

A pointer to an observer object. The `observer` argument is of type `sc_fxnum_fast_observer*`. See Chapter 11.24. The default value for `observer` is 0 (null pointer). The `observer` parameter cannot change after declaration.

Copy Constructor

sc_ufix_fast(const sc_ufix_fast&);

Operators

The operators defined for the `sc_ufix_fast` are given in Table 33.

Table 33. Operators for sc_ufix_fast

Operator class	**Operators in class**
Bitwise	~ & ^ |
Arithmetic	* / + - << >> ++ --
Equality	== !=
Relational	<<= >>=
Assignment	= *= /= += -= <<= >>= &= ^= |=

Note:
Operator << and operator >> define arithmetic shifts, not bitwise shifts. The difference is that no bits are lost and proper sign extension is done.

In expressions with the non-bitwise operators from Table 33, fixed-point types can be mixed with all types given:

`type_ in` `{short, unsigned short, int, unsigned int, long, unsigned long, float, double, const char*, int64, uint64, const` *`sc_int_base`*† `&, const` *`sc_uint_base`*† `&, const sc_signed&, const sc_unsigned, const sc_fxval&, const sc_fxval_fast&, const sc_[u]fix&, const sc_[u]fix_fast& }`

The return type of any arithmetic operation is the fixed-point value type, which guarantees that the operation is performed without overflow or quantization.

A floating-point variable or a fixed-point value variable can contain one of the special values +Inf (plus infinity), -Inf (minus infinity), or Nan (not a number). Assignment of one of these special values to a fixed-point variable will produce a runtime error.

For the fixed-point types, a minimal set of bitwise operators is defined. These bitwise operators are only defined on either the signed fixed-point types or the unsigned fixed- point types. Mixing between signed and unsigned fixed-point types is not allowed. Mixing with any other type is also not allowed.

The semantics of the bitwise operators is as follows. For the unary ~ operator, the type of the result is the type of the operand. The bits in the two's complement mantissa of the operand are inverted to get the mantissa of the result. For the binary operators, the type of the result is the maximum aligned type of the two operands, that is, the two operands are aligned by the binary point and the maximum integer word length and the maximum fractional word length is taken. The operands are temporarily extended to this type before performing a bitwise and, bitwise exclusive-or, or bitwise or.

Member Functions

The functions defined for `sc_ufix_fast` are given in Table 34.

Table 34. Functions for sc_ufix_fast

Function class	**Functions in class**
Bitwise	b_not, b_and, b_xor, b_or
Arithmetic	neg, mult, div, add, sub, lshift, rshift

The functions in Table 34 have return type void. The first argument of these functions is a reference to the result object. The remaining arguments of these functions are the operands.

For the bitwise functions, the result object and the operands are of the same type, which is either `sc_fix` or `sc_ufix`.

The `neg` arithmetic function takes one operand, the other arithmetic functions take two operands. At least one of the operands of the arithmetic functions should have a fixed- point type, the other operand can have any of the types given:

```
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

The arithmetic functions are defined twice: once with the result object of type sc_fxval, and once with the result object of type sc_ufix_fast.

Bit Selection

```
const sc_fxnum_bitref† operator [] ( int i) const;
sc_fxnum_bitref†   operator [] ( int i);

const sc_fxnum_bitref† bit( int i) const;
sc_fxnum_bitref†   bit( int i);
```

These functions take one argument of type int, which is the index into the fixed-point mantissa. The index argument must be between `wl-1` (MSB) and 0 (LSB). Otherwise, a runtime error is produced. The return type of the bit selection functions is (const or non- const) *`sc_fxnum_bitref`*†, which is a proxy class. The proxy class allows bit selection to be used both as `rvalue` (for reading) and `lvalue` (for writing). For bit selection, the fixed-point binary point is ignored.

Part Selection

```
const sc_fxnum_subref† operator () ( int, int ) const;
sc_fxnum_subref†   operator () ( int, int );

const sc_fxnum_subref† range( int, int ) const;
sc_fxnum_subref†   range( int, int );
```

These functions take two arguments of type `int`, which are the begin and end indices into the fixed-point mantissa. The index arguments must be between `wl-1` (MSB) and 0 (LSB). Otherwise, a runtime error is produced. The return type of the part selection functions is (const or non-const) *`sc_fxnum_subref`*†, which is a proxy class that behaves like type `sc_bv_base`. The proxy class allows part selection to be used both as `rvalue` (for reading) and `lvalue` (for writing). All operators and methods that are available for type `sc_bv_base` are also available for part selection. For part selection, the fixed-point binary point is ignored.

```
const sc_fxnum_subref† operator () () const;
sc_fxnum_subref†   operator () ();

const sc_fxnum_subref† range() const;
sc_fxnum_subref†   range();
```

As a shortcut for part selection of the complete mantissa, operator () and the range() method can be called without any arguments.

Query Parameters

```
const sc_fxcast_switch&
cast_switch() const;
```
Returns the cast switch parameter.

```
int
iwl() const;
```
Returns the integer word length parameter.

```
int
n_bits() const;
```
Returns the number of saturated bits parameter.

```
sc_o_mode
o_mode() const;
```
Returns the overflow mode parameter.

```
sc_q_mode
```



```
q_mode() const;
```
Return the quantization mode parameter.

```
const sc_fxtype_params&
type_params() const;
```
Returns the type parameters.

```
int
wl() const;
```
Returns the total word length parameter.

Query Value

```
bool
is_neg() const;
```
Always returns false.

```
bool
is_zero() const;
```
Returns true if the variable holds a zero value. Returns false otherwise.

```
bool
overflow_flag() const;
```
Returns true if the last write action on this variable caused overflow. Returns false otherwise.

```
bool
quantization_flag() const;
```
Returns true if the last write action on this variable caused quantization. Returns false otherwise.

```
const sc_fxval
value() const;
```
Returns the value.

Implicit Conversion

```
operator double() const;
```
Implicit conversion to the implementation type `double`. The value does not change.

Explicit Conversion

```
short          to_short() const;
unsigned short  to_ushort() const;
int            to_int() const;
unsigned int    to_uint() const;
long            to_long() const;
unsigned long   to_ulong() const;
float           to_float() const;
double          to_double() const
```



```
const sc_string to_string() const;
const sc_string to_string( sc_numrep ) const;
const sc_string to_string( sc_numrep, bool ) const;
const sc_string to_string( sc_fmt ) const;
const sc_string to_string( sc_numrep, sc_fmt ) const;
const sc_string to_string( sc_numrep, bool, sc_fmt ) const;
```

The value of a fixed-point variable can be converted to a character string with the to_string() method. This method takes different arguments for formatting purposes. See Chapter 6.8.8 for more information on converting fixed-point variables to/from character strings. Furthermore, writing to C++ output streams with operator << is supported, e.g. cout << a;, where a is a fixed-point variable. The decimal number representation is used in this case.

```
const sc_string to_dec() const;
const sc_string to_bin() const;
const sc_string to_oct() const;
const sc_string to_hex() const;
```

Shortcut methods for conversion to a character string. See Chapter 6.8.9.2.

Print or dump content

```
void
print( ostream& = cout ) const;
```

Print the `sc_ufix_fast` instance value to an output stream.

```
void
scan( istream& = cin );
```

Read an `sc_ufix_fast` value from an input stream.

```
void
dump( ostream& = cout )
const;
```

Prints the `sc_ufix_fast` instance value, parameters and flags to an output stream.

```
ostream&
operator << ( ostream& os, const sc_ufix_fast& a )
```

Print the instance value of a to an output stream os.

11.71 sc_ufixed

Synopsis

```
template <int W, int I,
      sc_q_mode Q = SC_DEFAULT_Q_MODE_,
      sc_o_mode O = SC_DEFAULT_O_MODE_, int N =
    SC_DEFAULT_N_BITS_>
class sc_ufixed : public sc_ufix
{
public:
    // constructors
    explicit sc_ufixed( sc_fxnum_observer* = 0 );
    explicit sc_ufixed( const sc_fxcast_switch&,
    sc_fxnum_observer* = 0 );

#define DECL_CTORS_T_A(tp)  \
   sc_ufixed( tp, sc_fxnum_observer* = 0 );  \
   sc_ufixed( tp, const sc_fxcast_switch&,
    sc_fxnum_observer* = 0 );

#define DECL_CTORS_T_B(tp)  \
    explicit sc_ufixed( tp, sc_fxnum_observer* = 0 ); \
   sc_ufixed( tp, const sc_fxcast_switch&, \
    sc_fxnum_observer* = 0 );

    DECL_CTORS_T_A(int)
    DECL_CTORS_T_A(unsigned int)
    DECL_CTORS_T_A(long)
    DECL_CTORS_T_A(unsigned long)
    DECL_CTORS_T_A(double)
    DECL_CTORS_T_A(const char*)
    DECL_CTORS_T_A(const sc_fxval&)
    DECL_CTORS_T_A(const sc_fxval_fast&)
    DECL_CTORS_T_A(const sc_fxnum&)
    DECL_CTORS_T_A(const sc_fxnum_fast&)
    DECL_CTORS_T_B(int64)
    DECL_CTORS_T_B(uint64)
    DECL_CTORS_T_B(const sc_int_base&)
    DECL_CTORS_T_B(const sc_uint_base&)
    DECL_CTORS_T_B(const sc_signed&)
    DECL_CTORS_T_B(const sc_unsigned&)

#undef DECL_CTORS_T_A
#undef DECL_CTORS_T_B

    // copy constructor
    sc_ufixed( const sc_ufixed<W,I,Q,O,N>& );

    // assignment operators
    sc_ufixed& operator = ( const sc_ufixed<W,I,Q,O,N>& );
#define DECL_ASN_OP_T(op,tp)\
    sc_ufixed& operator op ( tp );
```



```
#ifndef SC_FX_EXCLUDE_OTHER
#define DECL_ASN_OP_OTHER(op)          \
    DECL_ASN_OP_T(op,int64) \
    DECL_ASN_OP_T(op,uint64)\
    DECL_ASN_OP_T(op,const sc_int_base&)         \
    DECL_ASN_OP_T(op,const sc_uint_base&)      \
    DECL_ASN_OP_T(op,const sc_signed&)\
    DECL_ASN_OP_T(op,const sc_unsigned&)
#else
#define DECL_ASN_OP_OTHER(op)
#endif

#define DECL_ASN_OP(op)  \
    DECL_ASN_OP_T(op,int)   \
    DECL_ASN_OP_T(op,unsigned int)\
    DECL_ASN_OP_T(op,long)  \
    DECL_ASN_OP_T(op,unsigned long)   \
    DECL_ASN_OP_T(op,double)\
    DECL_ASN_OP_T(op,const char*)  \
    DECL_ASN_OP_T(op,const sc_fxval&) \
    DECL_ASN_OP_T(op,const sc_fxval_fast&)    \
    DECL_ASN_OP_T(op,const sc_fxnum&) \
    DECL_ASN_OP_T(op,const sc_fxnum_fast&)    \
    DECL_ASN_OP_OTHER(op)

    DECL_ASN_OP(=)
    DECL_ASN_OP(*=)
    DECL_ASN_OP(/=)
    DECL_ASN_OP(+=)
    DECL_ASN_OP(-=)
    DECL_ASN_OP_T(<<=,int)
    DECL_ASN_OP_T(>>=,int)
    DECL_ASN_OP_T(&=,const sc_ufix&)
    DECL_ASN_OP_T(&=,const sc_ufix_fast&)
    DECL_ASN_OP_T(|=,const sc_ufix&)
    DECL_ASN_OP_T(|=,const sc_ufix_fast&)
    DECL_ASN_OP_T(^=,const sc_ufix&)
    DECL_ASN_OP_T(^=,const sc_ufix_fast&)

#undef DECL_ASN_OP_T
#undef DECL_ASN_OP_OTHER
#undef DECL_ASN_OP


    // auto-increment and auto-decrement
    const sc_fxval operator ++ ( int );
    const sc_fxval operator -- ( int );
    sc_ufixed& operator ++ ();
    sc_ufixed& operator -- ();
};
```

Description

Templatized type **sc_ufixed** is an unsigned (two's complement) type. The fixed-point type parameters wl, iwl, q_mode, o_mode, and n_bits are part of the type in sc_ufixed. It is required that these parameters be constant expressions. See Chapter 6.8.1.

Declaration syntax

```
sc_ufixed <wl,iwl[,q_mode[,o_mode[,n_bits]]]>
   var_name([init_val][,cast_switch])
                        [,observer]);
```

wl
The total number of bits in the fixed-point format. The wl argument is of type int and must be greater than zero. Otherwise, a runtime error is produced. The wl argument must be a constant expression. The total word length parameter cannot change after declaration.
iwl
The number of integer bits in the fixed-point format. The iwl argument is of type int and can be positive or negative. See Chapter 6.8.1. The iwl argument must be a constant expression. The number of integer bits parameter cannot change after declaration.
q_mode
The quantization mode to use. The q_mode argument is of type sc_q_mode. Valid values for q_mode are given in Chapter 6.8.2.2 . The q_mode argument must be a constant expression. The default value for q_mode is obtained from the set of built-in default values. See Chapter 6.8.8. The quantization mode parameter cannot change after declaration.
o_mode
The overflow mode to use. The o_mode argument is of type sc_o_mode. Valid values for o_mode are given in Chapter 6.8.2.1 . The o_mode argument must be a constant expression. The default value for o_mode is obtained from the set of built-in default values. See Chapter 6.8.8. The overflow mode parameter cannot change after declaration.
n_bits
The number of saturated bits parameter for the selected overflow mode. The n_bits argument is of type int and must be greater than or equal to zero. Otherwise, a runtime error is produced. The n_bits argument must be a constant expression. If the overflow mode is specified, the default value is zero. If the overflow mode is not specified, the default value is obtained from the set of built-in default values. See Chapter 6.8.8. The number of saturated bits parameter cannot change after declaration.

Examples

```
sc_ufixed<16,1,SC_RND_CONV,SC_SAT_SYM> b(0.75);
sc_ufixed<16,16> d(SC_OFF);
```

Public Constructor


```
explicit sc_ufixed ([type_ init_val]
    [, const sc_fxcast_switch& cast_switch]
    [, sc_fxnum_observer* observer]);

type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

Notes on `type_`
For all types in `type_` , except `sc_[u]fix` and `sc_[u]fix_fast`, only the value of the argument is taken, that is, any type information is discarded. This ensures that initialization during declaration and initialization after declaration behave identical.
A fixed-point variable can be initialized with a C/C++ character string (type const char*) either when the number will be expressed in binary form or when the number is too large to be written as a C/C++ built-in type literal

init_val
The initial value of the variable. If the initial value is not specified, the instance is uninitialized.
cast_switch
The cast switch, which allows to switch fixed-point type casting on or off. Valid values for `cast_switch` are:
`SC_OFF` for casting off
`SC_ON` for casting on
The default value for `cast_switch` is obtained from the fixed-point context type `sc_fxcast_context`. See Chapter 6.8.7. The `cast_switch` parameter cannot change after declaration.
observer
A pointer to an observer object. The `observer` argument is of type `sc_fxnum_observer*`. See Chapter 11.25. The default value for `observer` is 0 (null pointer). The `observer` parameter cannot change after declaration.

Copy Constructor

```
sc_ufixed( const sc_ufixed<W,I,Q,O,N>& );
```

Operators

The operators defined for the `sc_ufixed` are given in Table 35.

Table 35. Operators for sc_ufixed

Operator class	Operators in class
Bitwise	~ & ^ |

Arithmetic	* / + - << >> ++ --
Equality	== !=
Relational	<<= >>=
Assignment	= *= /= += -= <<= >>= &= ^= |=

Note:
Operator << and operator >> define arithmetic shifts, not bitwise shifts. The difference is that no bits are lost and proper sign extension is done.

In expressions with the non-bitwise operators from Table 35, fixed-point types can be mixed with all types given:

```
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

The return type of any arithmetic operation is the fixed-point value type, which guarantees that the operation is performed without overflow or quantization.

A floating-point variable or a fixed-point value variable can contain one of the special values +Inf (plus infinity), -Inf (minus infinity), or Nan (not a number). Assignment of one of these special values to a fixed-point variable will produce a runtime error.

For the fixed-point types, a minimal set of bitwise operators is defined. These bitwise operators are only defined on either the signed fixed-point types or the unsigned fixed- point types. Mixing between signed and unsigned fixed-point types is not allowed. Mixing with any other type is also not allowed.

The semantics of the bitwise operators is as follows. For the unary ~ operator, the type of the result is the type of the operand. The bits in the two's complement mantissa of the operand are inverted to get the mantissa of the result. For the binary operators, the type of the result is the maximum aligned type of the two operands, that is, the two operands are aligned by the binary point and the maximum integer word length and the maximum fractional word length is taken. The operands are temporarily extended to this type before performing a bitwise and, bitwise exclusive-or, or bitwise or.

Member Functions

The functions defined for `sc_ufixed` are given in Table 36.

Table 36. Functions for sc_ufixed

Function class	**Functions in class**

Bitwise	b_not, b_and, b_xor, b_or
Arithmetic	neg, mult, div, add, sub, lshift, rshift

The functions in Table 36 have return type void. The first argument of these functions is a reference to the result object. The remaining arguments of these functions are the operands.

For the bitwise functions, the type of the result is sc_ufixed, and the type of the operands are either both `sc_ufixed` or a mix of `sc_ufixed` and `sc_ufixed_fast`

The `neg` arithmetic function takes one operand, the other arithmetic functions take two operands. At least one of the operands of the arithmetic functions should have a fixed- point type, the other operand can have any of the types given:

```
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

The arithmetic functions are defined twice: once with the result object of type sc_fxval, and once with the result object of type sc_fixed or sc_ufixed.

Bit Selection

```
const sc_fxnum_bitref† operator [] ( int i) const;
sc_fxnum_bitref†   operator [] ( int i);

const sc_fxnum_bitref† bit( int i) const;
sc_fxnum_bitref†   bit( int i);
```

These functions take one argument of type int, which is the index into the fixed-point mantissa. The index argument must be between `wl-1` (MSB) and 0 (LSB). Otherwise, a runtime error is produced. The return type of the bit selection functions is (const or non- const) *sc_fxnum_bitref*†, which is a proxy class. The proxy class allows bit selection to be used both as `rvalue` (for reading) and `lvalue` (for writing). For bit selection, the fixed-point binary point is ignored.

Part Selection

```
const sc_fxnum_subref† operator () ( int, int ) const;
sc_fxnum_subref†   operator () ( int, int );

const sc_fxnum_subref† range( int, int ) const;
```



```
sc_fxnum_subref†    range( int, int );
```

These functions take two arguments of type `int`, which are the begin and end indices into the fixed-point mantissa. The index arguments must be between `wl-1` (MSB) and 0 (LSB). Otherwise, a runtime error is produced. The return type of the part selection functions is (const or non-const) *sc_fxnum_subref*†, which is a proxy class that behaves like type `sc_bv_base`. The proxy class allows part selection to be used both as `rvalue` (for reading) and `lvalue` (for writing). All operators and methods that are available for type `sc_bv_base` are also available for part selection. For part selection, the fixed-point binary point is ignored.

```
const sc_fxnum_subref† operator () () const;
sc_fxnum_subref†    operator () ();

const sc_fxnum_subref† range() const;
sc_fxnum_subref†    range();
```

As a shortcut for part selection of the complete mantissa, operator () and the range() method can be called without any arguments.

Query Parameters

```
const sc_fxcast_switch&
cast_switch() const;
```

Returns the cast switch parameter.

```
int
iwl() const;
```

Returns the integer word length parameter.

```
int
n_bits() const;
```

Returns the number of saturated bits parameter.

```
sc_o_mode
o_mode() const;
```

Returns the overflow mode parameter.

```
sc_q_mode
q_mode() const;
```

Return the quantization mode parameter.

```
const sc_fxtype_params&
type_params() const;
```

Returns the type parameters.


```
int
wl() const;
```
Returns the total word length parameter.

Query Value

```
bool
is_neg() const;
```
Returns true if the variable holds a negative value. Returns false otherwise.

```
bool
is_zero() const;
```
Returns true if the variable holds a zero value. Returns false otherwise.

```
bool
overflow_flag() const;
```
Returns true if the last write action on this variable caused overflow. Returns false otherwise.

```
bool
quantization_flag() const;
```
Returns true if the last write action on this variable caused quantization. Returns false otherwise.

```
const sc_fxval
value() const;
```
Returns the value.

Implicit Conversion

```
operator double() const;
```
Implicit conversion to the implementation type `double`. The value does not change, if the wordlength of the `sc_ufixed` is less than or equal to 53 bits.

Explicit Conversion

```
short          to_short() const;
unsigned short  to_ushort() const;
int            to_int() const;
unsigned int    to_uint() const;
long            to_long() const;
unsigned long   to_ulong() const;
float           to_float() const;
double          to_double() const

const sc_string to_string() const;
const sc_string to_string( sc_numrep ) const;
const sc_string to_string( sc_numrep, bool ) const;
const sc_string to_string( sc_fmt ) const;
const sc_string to_string( sc_numrep, sc_fmt ) const;
const sc_string to_string( sc_numrep, bool, sc_fmt ) const;
```


The value of a fixed-point variable can be converted to a character string with the to_string() method. This method takes different arguments for formatting purposes. See Chapter 6.8.8 for more information on converting fixed-point variables to/from character strings. Furthermore, writing to C++ output streams with operator << is supported, e.g. cout << a;, where a is a fixed-point variable. The decimal number representation is used in this case.

```
const sc_string to_dec() const;
const sc_string to_bin() const;
const sc_string to_oct() const;
const sc_string to_hex() const;
```

Shortcut methods for conversion to a character string. See Chapter 6.8.9.2.

Print or dump content

```
void
print( ostream& = cout ) const;
```

Print the `sc_ufixed` instance value to an output stream.

```
void
scan( istream& = cin );
```

Read an `sc_ufixed` value from an input stream.

```
void
dump( ostream& = cout )
const;
```

Prints the `sc_ufixed` instance value, parameters and flags to an output stream.

```
ostream&
operator << ( ostream& os, const sc_ufixed& a )
```

Print the instance value of `a` to an output stream `os`.

11.72 sc_ufixed_fast

Synopsis

```
template <int W, int I,
      sc_q_mode Q = SC_DEFAULT_Q_MODE_,
      sc_o_mode O = SC_DEFAULT_O_MODE_, int N =
    SC_DEFAULT_N_BITS_>
class sc_ufixed_fast : public sc_ufix_fast
{
public:
    // constructors
    explicit sc_ufixed_fast( sc_fxnum_fast_observer* = 0 );
    explicit sc_ufixed_fast( const sc_fxcast_switch&,
      sc_fxnum_fast_observer* = 0 );

#define DECL_CTORS_T_A(tp)  \
    sc_ufixed_fast( tp, sc_fxnum_fast_observer* = 0 ); \
    sc_ufixed_fast( tp, const sc_fxcast_switch&, \
    sc_fxnum_fast_observer* = 0 );

#define DECL_CTORS_T_B(tp)  \
    explicit sc_ufixed_fast \
    ( tp, sc_fxnum_fast_observer* = 0 );     \
    sc_ufixed_fast( tp, const sc_fxcast_switch&, \
    sc_fxnum_fast_observer* = 0 );

    DECL_CTORS_T_A(int)
    DECL_CTORS_T_A(unsigned int)
    DECL_CTORS_T_A(long)
    DECL_CTORS_T_A(unsigned long)
    DECL_CTORS_T_A(double)
    DECL_CTORS_T_A(const char*)
    DECL_CTORS_T_A(const sc_fxval&)
    DECL_CTORS_T_A(const sc_fxval_fast&)
    DECL_CTORS_T_A(const sc_fxnum&)
    DECL_CTORS_T_A(const sc_fxnum_fast&)
    DECL_CTORS_T_B(int64)
    DECL_CTORS_T_B(uint64)
    DECL_CTORS_T_B(const sc_int_base&)
    DECL_CTORS_T_B(const sc_uint_base&)
    DECL_CTORS_T_B(const sc_signed&)
    DECL_CTORS_T_B(const sc_unsigned&)

#undef DECL_CTORS_T_A
#undef DECL_CTORS_T_B

    // copy constructor
    sc_ufixed_fast( const sc_ufixed_fast<W,I,Q,O,N>& );

    // assignment operators
    sc_ufixed_fast& operator = ( const
    sc_ufixed_fast<W,I,Q,O,N>& );
```



```
#define DECL_ASN_OP_T(op,tp)\
    sc_ufixed_fast& operator op ( tp );

#ifndef SC_FX_EXCLUDE_OTHER
#define DECL_ASN_OP_OTHER(op)           \
    DECL_ASN_OP_T(op,int64) \
    DECL_ASN_OP_T(op,uint64)\
    DECL_ASN_OP_T(op,const sc_int_base&)         \
    DECL_ASN_OP_T(op,const sc_uint_base&)       \
    DECL_ASN_OP_T(op,const sc_signed&)\
    DECL_ASN_OP_T(op,const sc_unsigned&)
#else
#define DECL_ASN_OP_OTHER(op)
#endif

#define DECL_ASN_OP(op)  \
    DECL_ASN_OP_T(op,int)    \
    DECL_ASN_OP_T(op,unsigned int)\
    DECL_ASN_OP_T(op,long)  \
    DECL_ASN_OP_T(op,unsigned long)  \
    DECL_ASN_OP_T(op,double)\
    DECL_ASN_OP_T(op,const char*) \
    DECL_ASN_OP_T(op,const sc_fxval&) \
    DECL_ASN_OP_T(op,const sc_fxval_fast&) \
    DECL_ASN_OP_T(op,const sc_fxnum&) \
    DECL_ASN_OP_T(op,const sc_fxnum_fast& \
    DECL_ASN_OP_OTHER(op)

    DECL_ASN_OP(=)
    DECL_ASN_OP(*=)
    DECL_ASN_OP(/=)
    DECL_ASN_OP(+=)
    DECL_ASN_OP(-=)
    DECL_ASN_OP_T(<<=,int)
    DECL_ASN_OP_T(>>=,int)
    DECL_ASN_OP_T(&=,const sc_ufix&)
    DECL_ASN_OP_T(&=,const sc_ufix_fast&)
    DECL_ASN_OP_T(|=,const sc_ufix&)
    DECL_ASN_OP_T(|=,const sc_ufix_fast&)
    DECL_ASN_OP_T(^=,const sc_ufix&)
    DECL_ASN_OP_T(^=,const sc_ufix_fast&)

#undef DECL_ASN_OP_T
#undef DECL_ASN_OP_OTHER
#undef DECL_ASN_OP

    // auto-increment and auto-decrement
    const sc_fxval_fast operator ++ ( int );
    const sc_fxval_fast operator -- ( int );
    sc_ufixed_fast& operator ++ ();
    sc_ufixed_fast& operator -- ();
};
```


Description

Templatized type **`sc_ufixed_fast`** is an unsigned type. The fixed-point type parameters wl, iwl, q_mode, o_mode, and n_bits are part of the type in `sc_ufixed_fast`. It is required that these parameters be constant expressions. See Chapter 6.8.1.

`sc_ufixed_fast` provides the same API as `sc_ufixed`.

`sc_ufixed_fast` uses double precision (floating-point) values. The mantissa of a double precision value is limited to 53 bits. This means that bit-true behavior cannot be guaranteed with the limited precision types. For bit-true behavior with the limited precision types, the following guidelines should be followed:
Make sure that the word length of the result of any operation or expression does not exceed 53 bits.
The result of an addition or subtraction requires a word length that is one bit more than the maximum *aligned* word length of the two operands.
The result of a multiplication requires a word length that is the sum of the word lengths of the two operands.

Declaration syntax

```
sc_ufixed_fast <wl,iwl[,q_mode[,o_mode[,n_bits]]]>
   var_name([init_val][,cast_switch])
                        [,observer]);
```

wl
The total number of bits in the fixed-point format. The wl argument is of type int and must be greater than zero. Otherwise, a runtime error is produced. The wl argument must be a constant expression. The total word length parameter cannot change after declaration.
iwl
The number of integer bits in the fixed-point format. The iwl argument is of type int and can be positive or negative. See Chapter 6.8.1. The iwl argument must be a constant expression. The number of integer bits parameter cannot change after declaration.
q_mode
The quantization mode to use. The q_mode argument is of type sc_q_mode. Valid values for q_mode are given in Chapter 6.8.2.2 . The q_mode argument must be a constant expression. The default value for q_mode is obtained from the set of built-in default values. See Chapter 6.8.8. The quantization mode parameter cannot change after declaration.
o_mode
The overflow mode to use. The o_mode argument is of type sc_o_mode. Valid values for o_mode are given in Chapter 6.8.2.1 . The o_mode argument must be a constant expression. The default value for o_mode is obtained from the set of built-in default values. See Chapter 6.8.8. The overflow mode parameter cannot change after declaration.

n_bits
The number of saturated bits parameter for the selected overflow mode. The n_bits argument is of type int and must be greater than or equal to zero. Otherwise, a runtime error is produced. The n_bits argument must be a constant expression. If the overflow mode is specified, the default value is zero. If the overflow mode is not specified, the default value is obtained from the set of built-in default values. See Chapter 6.8.8. The number of saturated bits parameter cannot change after declaration.

Examples

```
sc_ufixed_fast<32,32> a;
sc_ufixed_fast<8,1,SC_RND> c(b);
sc_ufixed_fast<8,8> c = “0.1”;
sc_ufixed_fast<8,8> d = 1;
sc_ufixed<16,8> e = 2;
sc_ufixed_fast<16,16> f = d + e;
d *= 2;
```

Public Constructor

```
explicit sc_ufixed_fast ([type_ init_val]
    [, const sc_fxcast_switch& cast_switch]
    [, sc_fxnum_fast_observer* observer]);

type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

Notes on `type_`
For all types in `type_` , except `sc_[u]fix` and `sc_[u]fix_fast`, only the value of the argument is taken, that is, any type information is discarded. This ensures that initialization during declaration and initialization after declaration behave identical.
A fixed-point variable can be initialized with a C/C++ character string (type const char*) either when the number will be expressed in binary form or when the number is too large to be written as a C/C++ built-in type literal

init_val
The initial value of the variable. If the initial value is not specified, the instance is uninitialized.
cast_switch
The cast switch, which allows to switch fixed-point type casting on or off. Valid values for `cast_switch` are:
`SC_OFF` for casting off

`SC_ON` for casting on
The default value for `cast_switch` is obtained from the fixed-point context type `sc_fxcast_context`. See Chapter 6.8.7. The `cast_switch` parameter cannot change after declaration.
observer
A pointer to an observer object. The `observer` argument is of type `sc_fxnum_fast_observer*`. See Chapter 11.24. The default value for `observer` is 0 (null pointer). The `observer` parameter cannot change after declaration.

Copy Constructor

```
sc_ufixed_fast( const sc_ufixed_fast<W,I,Q,O,N>& );
```

Operators

The operators defined for the `sc_ufixed_fast` are given in Table 37.

Table 37. Operators for sc_ufixed_fast

Operator class	Operators in class
Bitwise	~ & ^ |
Arithmetic	* / + - << >> ++ --
Equality	== !=
Relational	<<= >>=
Assignment	= *= /= += -= <<= >>= &= ^= |=

Note:
Operator << and operator >> define arithmetic shifts, not bitwise shifts. The difference is that no bits are lost and proper sign extension is done.

In expressions with the non-bitwise operators from Table 37, fixed-point types can be mixed with all types given:

```
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

The return type of any arithmetic operation is the fixed-point value type, which guarantees that the operation is performed without overflow or quantization.

A floating-point variable or a fixed-point value variable can contain one of the special values +Inf (plus infinity), -Inf (minus infinity), or Nan (not a number). Assignment of one of these special values to a fixed-point variable will produce a runtime error.

For the fixed-point types, a minimal set of bitwise operators is defined. These bitwise operators are only defined on either the signed fixed-point types or the unsigned fixed- point types. Mixing between signed and unsigned fixed-point types is not allowed. Mixing with any other type is also not allowed.

The semantics of the bitwise operators is as follows. For the unary ~ operator, the type of the result is the type of the operand. The bits in the two's complement mantissa of the operand are inverted to get the mantissa of the result. For the binary operators, the type of the result is the maximum aligned type of the two operands, that is, the two operands are aligned by the binary point and the maximum integer word length and the maximum fractional word length is taken. The operands are temporarily extended to this type before performing a bitwise and, bitwise exclusive-or, or bitwise or.

Member Functions

The functions defined for `sc_ufixed_fast` are given in Table 38.

Table 38. Functions for sc_ufixed_fast

Function class	**Functions in class**
Bitwise	b_not, b_and, b_xor, b_or
Arithmetic	neg, mult, div, add, sub, lshift, rshift

The functions in Table 38 have return type void. The first argument of these functions is a reference to the result object. The remaining arguments of these functions are the operands.

For the bitwise functions, the type of the result is sc_ufixed_fast, and the type of the operands are either both `sc_ufixed_fast or a mix of sc_fixed_fast` and `sc_ufixed_fast`.

The `neg` arithmetic function takes one operand, the other arithmetic functions take two operands. At least one of the operands of the arithmetic functions should have a fixed- point type, the other operand can have any of the types given:

```
type_ in {short, unsigned short, int, unsigned int, long,
    unsigned long, float, double, const char*, int64,
    uint64, const sc_int_base†&, const sc_uint_base†&,
    const sc_signed&, const sc_unsigned, const sc_fxval&,
    const sc_fxval_fast&, const sc_[u]fix&, const
    sc_[u]fix_fast& }
```

The arithmetic functions are defined twice: once with the result object of type sc_fxval, and once with the result object of type sc_fixed_fast or sc_ufixed_fast.

Bit Selection


```
const sc_fxnum_bitref† operator [] ( int i) const;
sc_fxnum_bitref†   operator [] ( int i);

const sc_fxnum_bitref† bit( int i) const;
sc_fxnum_bitref†   bit( int i);
```

These functions take one argument of type int, which is the index into the fixed-point mantissa. The index argument must be between `wl-1` (MSB) and 0 (LSB). Otherwise, a runtime error is produced. The return type of the bit selection functions is (const or non- const) *sc_fxnum_bitref*†, which is a proxy class. The proxy class allows bit selection to be used both as `rvalue` (for reading) and `lvalue` (for writing). For bit selection, the fixed-point binary point is ignored.

Part Selection

```
const sc_fxnum_subref† operator () ( int, int ) const;
sc_fxnum_subref†   operator () ( int, int );

const sc_fxnum_subref† range( int, int ) const;
sc_fxnum_subref†   range( int, int );
```

These functions take two arguments of type `int`, which are the begin and end indices into the fixed-point mantissa. The index arguments must be between `wl-1` (MSB) and 0 (LSB). Otherwise, a runtime error is produced. The return type of the part selection functions is (const or non-const) *sc_fxnum_subref*†, which is a proxy class that behaves like type `sc_bv_base`†. The proxy class allows part selection to be used both as `rvalue` (for reading) and `lvalue` (for writing). All operators and methods that are available for type `sc_bv_base`† are also available for part selection. For part selection, the fixed-point binary point is ignored.

```
const sc_fxnum_subref† operator () () const;
sc_fxnum_subref†   operator () ();

const sc_fxnum_subref† range() const;
sc_fxnum_subref†   range();
```

As a shortcut for part selection of the complete mantissa, operator () and the range() method can be called without any arguments.

Query Parameters

```
const sc_fxcast_switch&
```



```
cast_switch() const;
```
Returns the cast switch parameter.

```
int
iwl() const;
```
Returns the integer word length parameter.

```
int
n_bits() const;
```
Returns the number of saturated bits parameter.

```
sc_o_mode
o_mode() const;
```
Returns the overflow mode parameter.

```
sc_q_mode
q_mode() const;
```
Return the quantization mode parameter.

```
const sc_fxtype_params&
type_params() const;
```
Returns the type parameters.

```
int
wl() const;
```
Returns the total word length parameter.

Query Value

```
bool
is_neg() const;
```
Returns true if the variable holds a negative value. Returns false otherwise.

```
bool
is_zero() const;
```
Returns true if the variable holds a zero value. Returns false otherwise.

```
bool
overflow_flag() const;
```
Returns true if the last write action on this variable caused overflow. Returns false otherwise.

```
bool
quantization_flag() const;
```
Returns true if the last write action on this variable caused quantization. Returns false otherwise.

```
const sc_fxval
value() const;
```
Returns the value.

Implicit Conversion

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

Implicit conversion to the implementation type `double`. The value does not change, if the wordlength of the `sc_ufixed_fast` is less than or equal to 53 bits.

Explicit Conversion

```
short          to_short() const;
unsigned short  to_ushort() const;
int            to_int() const;
unsigned int    to_uint() const;
long            to_long() const;
unsigned long   to_ulong() const;
float           to_float() const;
double          to_double() const
```

```
const sc_string to_string() const;
const sc_string to_string( sc_numrep ) const;
const sc_string to_string( sc_numrep, bool ) const;
const sc_string to_string( sc_fmt ) const;
const sc_string to_string( sc_numrep, sc_fmt ) const;
const sc_string to_string( sc_numrep, bool, sc_fmt ) const;
```

The value of a fixed-point variable can be converted to a character string with the to_string() method. This method takes different arguments for formatting purposes. See Chapter 6.8.8 for more information on converting fixed-point variables to/from character strings. Furthermore, writing to C++ output streams with operator << is supported, e.g. cout << a;, where a is a fixed-point variable. The decimal number representation is used in this case.

```
const sc_string to_dec() const;
const sc_string to_bin() const;
const sc_string to_oct() const;
const sc_string to_hex() const;
```

Shortcut methods for conversion to a character string. See Chapter 6.8.9.2.

Print or dump content

```
void
print( ostream& = cout ) const;
```

Print the `sc_ufixed_fast` instance value to an output stream.

```
void
scan( istream& = cin );
```

Read an `sc_ufixed_fast` value from an input stream.

```
void
dump( ostream& = cout )
const;
```

Prints the `sc_ufixed_fast` instance value, parameters and flags to an output stream.


```
ostream&
operator << ( ostream& os, const sc_ufixed_fast& a )
```

Print the instance value of a to an output stream os.

11.73 sc_uint

Synopsis

```
template <int W>
class sc_uint
    : public sc_uint_base
{
public:
    // constructors
    sc_uint();
    sc_uint( uint64 v );
    sc_uint( const sc_uint<W>& a );
    sc_uint( const sc_uint_base& a );
    sc_uint( const sc_uint_subref_r& a );
    template <class T1, class T2>
    sc_uint( const sc_uint_concref_r<T1,T2>& a );
    sc_uint( const sc_signed& a );
    sc_uint( const sc_unsigned& a );
    explicit sc_uint( const sc_fxval& a );
    explicit sc_uint( const sc_fxval_fast& a );
    explicit sc_uint( const sc_fxnum& a );
    explicit sc_uint( const sc_fxnum_fast& a );
    sc_uint( const sc_bv_base& a );
    sc_uint( const sc_lv_base& a );
    sc_uint( const char* a );
    sc_uint( unsigned long a );
    sc_uint( long a );
    sc_uint( unsigned int a );
    sc_uint( int a );
    sc_uint( int64 a );
    sc_uint( double a );

    // assignment operators
    sc_uint<W>& operator = ( uint64 v );
    sc_uint<W>& operator = ( const sc_uint_base& a );
    sc_uint<W>& operator = ( const sc_uint_subref_r& a );
    sc_uint<W>& operator = ( const sc_uint<W>& a );
    template <class T1, class T2>
    sc_uint<W>& operator = ( const
    sc_uint_concref_r<T1,T2>& a );
    sc_uint<W>& operator = ( const sc_signed& a );
    sc_uint<W>& operator = ( const sc_unsigned& a );
    sc_uint<W>& operator = ( const sc_fxval& a );
    sc_uint<W>& operator = ( const sc_fxval_fast& a );
    sc_uint<W>& operator = ( const sc_fxnum& a );
    sc_uint<W>& operator = ( const sc_fxnum_fast& a );
    sc_uint<W>& operator = ( const sc_bv_base& a );
    sc_uint<W>& operator = ( const sc_lv_base& a );
    sc_uint<W>& operator = ( const char* a );
    sc_uint<W>& operator = ( unsigned long a );
    sc_uint<W>& operator = ( long a );
    sc_uint<W>& operator = ( unsigned int a );
    sc_uint<W>& operator = ( int a );
```



```
    sc_uint<W>& operator = ( int64 a );
    sc_uint<W>& operator = ( double a );

    // arithmetic assignment operators
    sc_uint<W>& operator += ( uint64 v );
    sc_uint<W>& operator -= ( uint64 v );
    sc_uint<W>& operator *= ( uint64 v );
    sc_uint<W>& operator /= ( uint64 v );
    sc_uint<W>& operator %= ( uint64 v );

    // bitwise assignment operators
    sc_uint<W>& operator &= ( uint64 v );
    sc_uint<W>& operator |= ( uint64 v );
    sc_uint<W>& operator ^= ( uint64 v );
    sc_uint<W>& operator <<= ( uint64 v );
    sc_uint<W>& operator >>= ( uint64 v );

    // prefix and postfix increment and decrement operators
    sc_uint<W>& operator ++ (); // prefix
    const sc_uint<W> operator ++ ( int ); // postfix
    sc_uint<W>& operator -- (); // prefix
    const sc_uint<W> operator -- ( int ); // postfix
};
```

Description

sc_uint<W> is an unsigned integer with a fixed word length W between 1 and 64 bits. The word length is built into the type and can never change. If the chosen word length exceeds 64 bits, an error is reported and simulation ends. All operations are performed with 64 bits of precision with the result converted to appropriate size through truncation.
Methods allow for addressing an individual bit or a sub range of bits.

Example

```
SC_MODULE(my_module) {
    // data types
  sc_uint<3> a;
  sc_uint<44> b;
    // process
  void my_proc();

  SC_CTOR(my_module) :
    a(0) // init
  {
    b = 33;  // set value
    SC_THREAD(my_proc);
  }
};

void my_module::my_proc() {
```



```
    a = 1;
    b[30] = a[0];
    cout << b.range(7,0) << endl;
    wait(300, SC_NS);
    sc_stop();
}
```

Public Constructors

sc_uint();

Create an sc_uint instance with an initial value of 0.

sc_uint(uint64 a) ;

Create an sc_uint with value a. If the word length of a is greater then W, a gets truncated to W bits.

sc_uint(T a) ;

T in { sc_uint, *sc_uint_base*†, *sc_uint_subref*†, *sc_uint_concref*†, *sc_[un]signed*†, sc_fxval, sc_fxval_fast, sc_[u]fix[ed][_fast], sc_bv_base†, *sc_lv_base*†, const char*, [unsigned] long, [unsigned] int, int64, double }

Create an sc_uint with value a. If the word length of a is greater then W, a gets truncated to W bits.

Copy Constructor

sc_uint(const sc_uint&)

Methods

int
length() const ;

Return the word length.

void
print(ostream& os = cout) const ;

Print the sc_uint instance to an output stream.

void
scan(istream& is = cin) ;

Read a sc_uint value from an input stream.

Reduction Methods

bool **and_reduce**() const;
bool **nand_reduce**() const ;
bool **nor_reduce**() const ;
bool **or_reduce**() const ;
bool **xnor_reduce**() const ;
bool **xor_reduce**() const;
F in { and nand or nor xor xnor }

Return the result of function F with all bits of the sc_uint instance as input arguments.

Assignment Operators

```
sc_uint<W>&
operator = ( uint64 ) ;

sc_uint<W>&
operator = ( T ) ;
T in { sc_uint, sc_uint_base†, sc_uint_subref†,
    sc_uint_concref†, sc_[un]signed†, sc_fxval,
    sc_fxval_fast, sc_[u]fix[ed][_fast],
      sc_lv_base†, sc_lv_base†, char*, [unsigned] long,
    [unsigned] int, int64, double }
```

Assign the value of the right-hand side to the left-hand side. The value is truncated, if its word length is greater than W.

Arithmetic Assignment Operators

```
sc_uint<W>&
operator OP ( uint64 ) ;
OP in { += -= *= /= %= }
```

The operation of OP is performed and the result is assigned to the lefthand side. If necessary, the result gets truncated.

Bitwise Assignment Operators

```
sc_uint<W>&
operator  OP ( uint64 ) ;
OP in { &= |= ^= <<= >>= }
```

The operation of OP is performed and the result is assigned to the left hand side. The result gets truncated.

Prefix and Postfix Increment and Decrement Operators

```
sc_uint<W>& operator  ++ () ;
const sc_uint<W> operator ++ ( int ) ;
```

The operation of OP is performed as done for type unsigned int.

```
sc_uint<W>& operator  -- () ;
const sc_uint<W> operator  -- ( int ) ;
```

The operation is performed as done for type unsigned int.

Relational Operators

```
friend bool operator OP (sc_uint, sc_uint ) ;
OP in { == != < <= > >= }
```

These functions return the boolean result of the corresponding equality/ inequality check.

Bit Selection

```
sc_uint_bitref   operator [] ( int i ) ;
sc_uint_bitref_r operator [] ( int i ) const ;
sc_uint_bitref   bit( int i) ;
sc_uint_bitref_r bit( int i) const ;
```

Return a reference to a single bit at index i.

Implicit Conversion

```
operator uint64() const ;
```

Implicit conversion to the implementation type uint64. The value does not change.

Explicit Conversion

```
double   to_double() const ;
int      to_int() const ;
int64    to_int64() const ;
long     to_long() const ;
uint64   to_uint64() const ;
unsigned int    to_uint() const ;
unsigned long   to_ulong() const ;
```

Converts the value of sc_uint instance into the corresponding data type. If the requested type has less word length than the sc_uint instance, the value gets truncated accordingly. If the requested type has greater word length than the sc_uint instance, the value gets sign extended, if necessary.

11.74 sc_uint_base

Synopsis

```
class sc_uint_base
{
public:
// constructors & destructors
    explicit sc_uint_base( int w =
    sc_length_param().len() ) ;
    sc_uint_base( uint64 v, int w ) ;
    sc_uint_base( const sc_uint_base& a );
    explicit sc_uint_base( const sc_uint_subref_r& a );
    template <class T1, class T2>
    explicit sc_uint_base( const sc_uint_concref_r<T1,T2>&
    a );
    explicit sc_uint_base( const sc_signed& a );
    explicit sc_uint_base( const sc_unsigned& a );
    ~sc_uint_base();

// assignment operators
    sc_uint_base& operator = ( uint64 v );
    sc_uint_base& operator = ( const sc_uint_base& a );
    sc_uint_base& operator = ( const sc_uint_subref_r& a );
    template <class T1, class T2>
    sc_uint_base& operator = ( const
    sc_uint_concref_r<T1,T2>& a );
    sc_uint_base& operator = ( const sc_signed& a );
    sc_uint_base& operator = ( const sc_unsigned& a );
    sc_uint_base& operator = ( const sc_fxval& a );
    sc_uint_base& operator = ( const sc_fxval_fast& a );
    sc_uint_base& operator = ( const sc_fxnum& a );
    sc_uint_base& operator = ( const sc_fxnum_fast& a );
    sc_uint_base& operator = ( const sc_bv_base& a );
    sc_uint_base& operator = ( const sc_lv_base& a );
    sc_uint_base& operator = ( const char* a );
    sc_uint_base& operator = ( unsigned long a );
    sc_uint_base& operator = ( long a );
    sc_uint_base& operator = ( unsigned int a );
    sc_uint_base& operator = ( int a );
    sc_uint_base& operator = ( int64 a );
    sc_uint_base& operator = ( double a );

// arithmetic assignment operators
    sc_uint_base& operator += ( uint64 v );
    sc_uint_base& operator -= ( uint64 v );
    sc_uint_base& operator *= ( uint64 v );
    sc_uint_base& operator /= ( uint64 v );
    sc_uint_base& operator %= ( uint64 v );

// bitwise assignment operators
    sc_uint_base& operator &= ( uint64 v );
    sc_uint_base& operator |= ( uint64 v );
    sc_uint_base& operator ^= ( uint64 v );
```



```
    sc_uint_base& operator <<= ( uint64 v );
    sc_uint_base& operator >>= ( uint64 v );

 // prefix and postfix increment and decrement operators
    sc_uint_base& operator ++ ();
    const sc_uint_base operator ++ ( int );
    sc_uint_base& operator -- ();
    const sc_uint_base operator -- ( int );
    extend_sign(); return tmp; };

 // relational operators
    friend bool operator == ( const sc_uint_base& a, const
    sc_uint_base& b );
    friend bool operator != ( const sc_uint_base& a, const
    sc_uint_base& b );
    friend bool operator <  ( const sc_uint_base& a, const
    sc_uint_base& b );
    friend bool operator <= ( const sc_uint_base& a, const
    sc_uint_base& b );
    friend bool operator >  ( const sc_uint_base& a, const
    sc_uint_base& b );
    friend bool operator >= ( const sc_uint_base& a, const
    sc_uint_base& b );

 // bit selection
    sc_uint_bitref   operator [] ( int i );
    sc_uint_bitref_r operator [] ( int i ) const;
    sc_uint_bitref   bit( int i );
    sc_uint_bitref_r bit( int i ) const;

 // part selection
    sc_uint_subref   operator () ( int left, int right );
    sc_uint_subref_r operator () ( int left, int right )
    const;
    sc_uint_subref   range( int left, int right );
    sc_uint_subref_r range( int left, int right ) const;

 // Methods
    int length() const;
    bool and_reduce() const;
    bool nand_reduce() const;
    bool or_reduce() const;
    bool nor_reduce() const
    bool xor_reduce() const;
    bool xnor_reduce() const;
    operator uint64() const;
    uint64 value() const;
    int to_int() const;
    unsigned int to_uint() const;
    long to_long() const;
    unsigned long to_ulong() const;
    int64 to_int64() const;
    uint64 to_uint64() const;
    double to_double() const;
```



```
    const sc_string to_string( sc_numrep numrep = SC_DEC )
    const;
    const sc_string to_string( sc_numrep numrep, bool
    w_prefix ) const;
    void print( ostream& os = cout ) const;
    void scan( istream& is = cin );
};
```

Description

sc_uint_base is an unsigned integer with a fixed word length between 1 and 64 bits. The word length is set when construction takes place and cannot be changed later.

Public Constructors

```
explicit
sc_uint_base( int = sc_length_param().len() );
```

Create an `sc_uint_base` instance with specified word length. Its initial value is 0.

```
sc_uint_base( uint64 a, int b );
```

Create an `sc_uint_base` instance with value a and word length b.

```
sc_uint_base( T a ) ;
T in { sc_uint_subref†, sc_uint_concref†, sc_[un]signed }
```

Create an `sc_uint_base` with value a. The word length of a must not exceed 64 bits. If it does, an error is reported and simulation ends.

Copy Constructor

```
sc_uint_base( const sc_uint_base& ) ;
```

Methods

```
int
length() const ;
```

Return the word length.

```
void
print( ostream& os = cout ) const ;
```

Print the `sc_uint_base` instance to an output stream.

```
void
scan( istream& is = cin ) ;
```

Read a `sc_uint_base` value from an input stream.

Reduction Methods

```
bool and_reduce() const;
bool nand_reduce() const ;
bool nor_reduce() const ;
bool or_reduce() const ;
bool xnor_reduce() const ;
bool xor_reduce() const;
```



```
F in { and nand or nor xor xnor }
```
Return the result of function F with all bits of the `sc_uint_base` instance as input arguments.

Assignment Operators

```
sc_uint_base& operator = ( uint64 ) ;
sc_uint_base& operator = ( T ) ;
T in { sc_uint_base, sc_uint_subref†, sc_uint_concref†,
    sc_[un]signed, sc_fxval, sc_fxval_fast, sc_fxnum,
    sc_fxnum_fast, sc_bv_base, sc_lv_base, char*, [unsigned]
    long, [unsigned] int, int64, double }
```

Assign the value of the right-hand side to the left-hand side. The value is truncated, if its word length does not fit into the sc_uint_base instance on the left hand side.

Arithmetic Assignment Operators

```
sc_uint_base&
operator OP ( uint64 ) ;
OP in { += -= *= /= %= }
```
The operation of OP is performed and the result is assigned to the lefthand side. If necessary, the result gets truncated.

Bitwise Assignment Operators

```
sc_uint_base&
operator  OP ( uint64 ) ;
OP in { &= |= ^= <<= >>= }
```
The operation of `OP` is performed and the result is assigned to the left hand side.

Prefix and Postfix Increment and Decrement Operators

```
sc_uint_base<W>& operator  ++ () ;
const sc_uint_base<W> operator ++ ( int ) ;
```
The operation is performed as done for type unsigned int.

```
sc_uint_base<W>& operator  -- () ;
const sc_uint<W> operator  -- ( int ) ;
```
The operation is performed as done for type unsigned int.

Relational Operators

```
friend bool operator OP (sc_uint_base, sc_uint_base ) ;
OP in { == != < <= > >= }
```
These functions return the boolean result of the corresponding equality/ inequality check.

Bit Selection

```
sc_uint_bitref   operator [] ( int i ) ;
sc_uint_bitref_r operator [] ( int i ) const ;
```



```
sc_uint_bitref   bit( int i) ;
sc_uint_bitref_r bit( int i) const ;
```

Return a reference to a single bit at index i.

Implicit Conversion

```
operator uint64() const ;
```

Implicit conversion to the implementation type uint64. The value does not change.

Explicit Conversion

```
double   to_double() const ;
int      to_int() const ;
int64    to_int64() const ;
long     to_long() const ;
uint64   to_uint64() const ;
unsigned int    to_uint() const ;
unsigned long   to_ulong() const ;
```

Converts the value of sc_uint_base instance into the corresponding data type. If the requested type has less word length than the sc_uint_base instance, the value gets truncated accordingly.

11.75 sc_unsigned

Synopsis

```
class sc_unsigned
{
public:
    // constructors & destructors
    explicit sc_unsigned( int nb =
    sc_length_param().len() );
    sc_unsigned( const sc_unsigned& v );
    sc_unsigned( const sc_signed&   v );
    ~sc_unsigned()

    // assignment operators
    sc_unsigned& operator =(const sc_unsigned& v);
   sc_unsigned& operator =(const sc_unsigned_subref_r& a );
    template <class T1, class T2>
    sc_unsigned& operator = ( const
    sc_unsigned_concref_r<T1,T2>& a )
    sc_unsigned& operator =(const sc_signed& v);
    sc_unsigned& operator = (const sc_signed_subref_r& a );
    template <class T1, class T2>
    sc_unsigned& operator = ( const
    sc_signed_concref_r<T1,T2>& a )
    sc_unsigned& operator = ( const char* v);
    sc_unsigned& operator = ( int64  v);
    sc_unsigned& operator = ( uint64 v);
    sc_unsigned& operator = ( long  v);
    sc_unsigned& operator = ( unsigned long v);
    sc_unsigned& operator = ( int v)
    sc_unsigned& operator = ( unsigned int v)
    sc_unsigned& operator = ( double  v);
    sc_unsigned& operator = ( const sc_int_base&  v);
    sc_unsigned& operator = ( const sc_uint_base& v);
    sc_unsigned& operator = ( const sc_bv_base& );
    sc_unsigned& operator = ( const sc_lv_base& );
    sc_unsigned& operator = ( const sc_fxval& );
    sc_unsigned& operator = ( const sc_fxval_fast& );
    sc_unsigned& operator = ( const sc_fxnum& );
    sc_unsigned& operator = ( const sc_fxnum_fast& );

    // Increment operators.
    sc_unsigned& operator ++ ();
    const sc_unsigned operator ++ (int);

    // Decrement operators.
    sc_unsigned& operator -- ();
    const sc_unsigned operator -- (int);

    // bit selection
    sc_unsigned_bitref operator [] ( int i )
    sc_unsigned_bitref_r operator [] ( int i ) const
    sc_unsigned_bitref bit( int i )
```



```
sc_unsigned_bitref_r bit( int i ) const

// part selection
sc_unsigned_subref range( int i, int j )
sc_unsigned_subref_r range( int i, int j ) const
sc_unsigned_subref operator () ( int i, int j )
sc_unsigned_subref_r operator () ( int i, int j ) const

// explicit conversions

int          to_int() const;
unsigned int  to_uint() const;
long         to_long() const;
unsigned long to_ulong() const;
int64         to_int64() const;
uint64       to_uint64() const;
double       to_double() const;
const sc_string to_string( sc_numrep numrep = SC_DEC )
const;
const sc_string to_string( sc_numrep numrep, bool
w_prefix ) const;

// methods
void print( ostream& os = cout ) const
void scan( istream& is = cin );
void dump( ostream& os = cout ) const;
int  length() const { return nbits - 1; }
bool iszero() const;
bool sign() const { return 0; }
void reverse();

// ADDition operators:

friend  sc_signed operator + (const sc_unsigned&  u,
const sc_signed&  v);
friend  sc_signed operator + (const sc_signed&    u,
const sc_unsigned&  v);
friend sc_unsigned operator + (const sc_unsigned& u,
const sc_unsigned&  v);
friend  sc_signed operator + (const sc_unsigned&  u,
int64          v);
friend sc_unsigned operator + (const sc_unsigned& u,
uint64         v);
friend  sc_signed operator + (const sc_unsigned&  u,
long          v);
friend sc_unsigned operator + (const sc_unsigned& u,
unsigned long     v);
friend  sc_signed operator + (const sc_unsigned&  u,
int           v);
friend sc_unsigned operator + (const sc_unsigned& u,
unsigned int   v)
friend  sc_signed operator + (int64
u, const sc_unsigned&    v);
```



```
friend sc_unsigned operator + (uint64
  u, const sc_unsigned&  v);
friend  sc_signed operator + (long            u,
const sc_unsigned&  v);
friend sc_unsigned operator + (unsigned long      u,
const sc_unsigned&  v);
friend  sc_signed operator + (int                u,
const sc_unsigned&  v);
friend sc_unsigned operator + (unsigned int       u,
const sc_unsigned&  v)
 sc_unsigned& operator += (const sc_signed&  v);
sc_unsigned& operator += (const sc_unsigned& v);
sc_unsigned& operator += (int64                  v);
sc_unsigned& operator += (uint64                v);
sc_unsigned& operator += (long            v);
sc_unsigned& operator += (unsigned long      v);
sc_unsigned& operator += (int                v)
sc_unsigned& operator += (unsigned int      v)
friend sc_unsigned operator + (const sc_unsigned& u,
const sc_uint_base& v);
friend  sc_signed operator + (const sc_unsigned&  u,
const sc_int_base&  v);
friend sc_unsigned operator + (const sc_uint_base& u,
const sc_unsigned&  v);
friend  sc_signed operator + (const sc_int_base&  u,
const sc_unsigned&  v);
sc_unsigned& operator += (const sc_int_base& v);
sc_unsigned& operator += (const sc_uint_base& v);

// SUBtraction operators:

friend  sc_signed operator - (const sc_unsigned&  u,
const sc_signed&  v);
friend  sc_signed operator - (const sc_signed&   u,
const sc_unsigned&  v);
friend  sc_signed operator - (const sc_unsigned&  u,
const sc_unsigned&  v);
friend  sc_signed operator - (const sc_unsigned&  u,
int64          v);
friend  sc_signed operator - (const sc_unsigned&  u,
uint64         v);
friend  sc_signed operator - (const sc_unsigned&  u,
long          v);
friend  sc_signed operator - (const sc_unsigned&  u,
unsigned long      v);
friend  sc_signed operator - (const sc_unsigned&  u,
int           v);
friend  sc_signed operator - (const sc_unsigned&  u,
unsigned int   v);
friend  sc_signed operator - (int64
u, const sc_unsigned&    v);
friend  sc_signed operator - (uint64
  u, const sc_unsigned&  v);
```



```
friend  sc_signed operator - (long              u,
const sc_unsigned&  v);
friend  sc_signed operator - (unsigned long      u,
const sc_unsigned&  v);
friend  sc_signed operator - (int                u,
const sc_unsigned&  v);
friend  sc_signed operator - (unsigned int       u,
const sc_unsigned&  v);
sc_unsigned& operator -= (const sc_signed&   v);
sc_unsigned& operator -= (const sc_unsigned& v);
sc_unsigned& operator -= (int64                v);
sc_unsigned& operator -= (uint64              v);
sc_unsigned& operator -= (long             v);
sc_unsigned& operator -= (unsigned long     v);
sc_unsigned& operator -= (int               v)
sc_unsigned& operator -= (unsigned int     v)
friend  sc_signed operator - (const sc_unsigned& u,
const sc_uint_base& v);
friend  sc_signed operator - (const sc_unsigned& u,
const sc_int_base&  v);
friend  sc_signed operator - (const sc_uint_base& u,
const sc_unsigned&  v);
friend  sc_signed operator - (const sc_int_base& u,
const sc_unsigned&  v);
sc_unsigned& operator -= (const sc_int_base& v);
sc_unsigned& operator -= (const sc_uint_base& v);

// MULtiplication operators:

friend  sc_signed operator * (const sc_unsigned& u,
const sc_signed& v);
friend  sc_signed operator * (const sc_signed&   u,
const sc_unsigned&  v);

friend sc_unsigned operator * (const sc_unsigned& u,
const sc_unsigned&  v);
friend  sc_signed operator * (const sc_unsigned& u,
int64          v);
friend sc_unsigned operator * (const sc_unsigned& u,
uint64        v);
friend  sc_signed operator * (const sc_unsigned& u,
long        v);
friend sc_unsigned operator * (const sc_unsigned& u,
unsigned long     v);
friend  sc_signed operator * (const sc_unsigned& u,
int          v);
friend sc_unsigned operator * (const sc_unsigned& u,
unsigned int   v)
friend  sc_signed operator * (int64
u, const sc_unsigned&    v);
friend sc_unsigned operator * (uint64
  u, const sc_unsigned& v);
friend  sc_signed operator * (long              u,
const sc_unsigned&  v);
```



```
friend sc_unsigned operator * (unsigned long       u,
const sc_unsigned&  v);
friend  sc_signed operator * (int                  u,
const sc_unsigned&  v);
friend sc_unsigned operator * (unsigned int       u,
const sc_unsigned&  v)
sc_unsigned& operator *= (const sc_signed&   v);
sc_unsigned& operator *= (const sc_unsigned& v);
sc_unsigned& operator *= (int64                   v);
sc_unsigned& operator *= (uint64                 v);
sc_unsigned& operator *= (long                v);
sc_unsigned& operator *= (unsigned long       v);
sc_unsigned& operator *= (int                 v)
sc_unsigned& operator *= (unsigned int       v)
friend sc_unsigned operator * (const sc_unsigned& u,
const sc_uint_base& v);
friend  sc_signed operator * (const sc_unsigned&  u,
const sc_int_base&  v);
friend sc_unsigned operator * (const sc_uint_base& u,
const sc_unsigned&  v);
friend  sc_signed operator * (const sc_int_base&  u,
const sc_unsigned&  v);
sc_unsigned& operator *= (const sc_int_base& v);
sc_unsigned& operator *= (const sc_uint_base& v);

// DIVision operators:
friend  sc_signed operator / (const sc_unsigned&  u,
const sc_signed&  v);
friend  sc_signed operator / (const sc_signed&    u,
const sc_unsigned&  v);
friend sc_unsigned operator / (const sc_unsigned& u,
const sc_unsigned&  v);
friend  sc_signed operator / (const sc_unsigned&  u,
int64           v);
friend sc_unsigned operator / (const sc_unsigned& u,
uint64         v);
friend  sc_signed operator / (const sc_unsigned&  u,
long         v);
friend sc_unsigned operator / (const sc_unsigned& u,
unsigned long     v);
friend  sc_signed operator / (const sc_unsigned&  u,
int           v);
friend sc_unsigned operator / (const sc_unsigned& u,
unsigned int   v)
friend  sc_signed operator / (int64
u, const sc_unsigned&    v);
friend sc_unsigned operator / (uint64
  u, const sc_unsigned& v);
friend  sc_signed operator / (long                u,
const sc_unsigned&  v);
friend sc_unsigned operator / (unsigned long       u,
const sc_unsigned&  v);
friend  sc_signed operator / (int                  u,
const sc_unsigned&  v);
```



```
friend sc_unsigned operator / (unsigned int       u,
const sc_unsigned&  v)
sc_unsigned& operator /= (const sc_signed&   v);
sc_unsigned& operator /= (const sc_unsigned& v);
sc_unsigned& operator /= (int64                    v);
sc_unsigned& operator /= (uint64                  v);
sc_unsigned& operator /= (long              v);
sc_unsigned& operator /= (unsigned long      v);
sc_unsigned& operator /= (int                v)
sc_unsigned& operator /= (unsigned int      v)
friend sc_unsigned operator / (const sc_unsigned& u,
const sc_uint_base& v);
friend  sc_signed operator / (const sc_unsigned&  u,
const sc_int_base&  v);
friend sc_unsigned operator / (const sc_uint_base& u,
const sc_unsigned&  v);
friend  sc_signed operator / (const sc_int_base&  u,
const sc_unsigned&  v);
sc_unsigned& operator /= (const sc_int_base& v);
sc_unsigned& operator /= (const sc_uint_base& v);

// MODulo operators:
friend  sc_signed operator % (const sc_unsigned&  u,
const sc_signed&  v);
friend  sc_signed operator % (const sc_signed&    u,
const sc_unsigned&  v);
friend sc_unsigned operator % (const sc_unsigned& u,
const sc_unsigned&  v);
friend  sc_signed operator % (const sc_unsigned&  u,
int64          v);
friend sc_unsigned operator % (const sc_unsigned& u,
uint64        v);
friend  sc_signed operator % (const sc_unsigned&  u,
long        v);
friend sc_unsigned operator % (const sc_unsigned& u,
unsigned long     v);
friend  sc_signed operator % (const sc_unsigned&  u,
int          v);
friend sc_unsigned operator % (const sc_unsigned& u,
unsigned int   v)
friend  sc_signed operator % (int64
u, const sc_unsigned&    v);
friend sc_unsigned operator % (uint64
  u, const sc_unsigned&  v);
friend  sc_signed operator % (long              u,
const sc_unsigned&  v);
friend sc_unsigned operator % (unsigned long      u,
const sc_unsigned&  v);
friend  sc_signed operator % (int                 u,
const sc_unsigned&  v);
friend sc_unsigned operator % (unsigned int       u,
const sc_unsigned&  v)
sc_unsigned& operator %= (const sc_signed&   v);
sc_unsigned& operator %= (const sc_unsigned& v);
```



```
sc_unsigned& operator %= (int64                  v);
sc_unsigned& operator %= (uint64                 v);
sc_unsigned& operator %= (long              v);
sc_unsigned& operator %= (unsigned long      v);
sc_unsigned& operator %= (int               v)
sc_unsigned& operator %= (unsigned int      v)
friend sc_unsigned operator % (const sc_unsigned& u,
const sc_uint_base& v);
friend  sc_signed operator % (const sc_unsigned& u,
const sc_int_base&  v);
friend sc_unsigned operator % (const sc_uint_base& u,
const sc_unsigned&  v);
friend  sc_signed operator % (const sc_int_base& u,
const sc_unsigned&  v);
sc_unsigned& operator %= (const sc_int_base& v);
sc_unsigned& operator %= (const sc_uint_base& v);

// Bitwise AND operators:
friend  sc_signed operator & (const sc_unsigned& u,
const sc_signed& v);
friend  sc_signed operator & (const sc_signed&   u,
const sc_unsigned&  v);
friend sc_unsigned operator & (const sc_unsigned& u,
const sc_unsigned&  v);
friend  sc_signed operator & (const sc_unsigned& u,
int64           v);
friend sc_unsigned operator & (const sc_unsigned& u,
uint64         v);
friend  sc_signed operator & (const sc_unsigned& u,
long        v);
friend sc_unsigned operator & (const sc_unsigned& u,
unsigned long     v);
friend  sc_signed operator & (const sc_unsigned& u,
int          v);
friend sc_unsigned operator & (const sc_unsigned& u,
unsigned int   v)
friend  sc_signed operator & (int64
u, const sc_unsigned&    v);
friend sc_unsigned operator & (uint64
  u, const sc_unsigned& v);
friend  sc_signed operator & (long              u,
const sc_unsigned&  v);
friend sc_unsigned operator & (unsigned long      u,
const sc_unsigned&  v);
friend  sc_signed operator & (int                u,
const sc_unsigned&  v);
friend sc_unsigned operator & (unsigned int      u,
const sc_unsigned&  v)
sc_unsigned& operator &= (const sc_signed&   v);
sc_unsigned& operator &= (const sc_unsigned& v);
sc_unsigned& operator &= (int64                  v);
sc_unsigned& operator &= (uint64                 v);
sc_unsigned& operator &= (long              v);
sc_unsigned& operator &= (unsigned long      v);
```



```
sc_unsigned& operator &= (int                 v)
sc_unsigned& operator &= (unsigned int       v)
friend sc_unsigned operator & (const sc_unsigned& u,
const sc_uint_base& v);
friend  sc_signed operator & (const sc_unsigned& u,
const sc_int_base&  v);
friend sc_unsigned operator & (const sc_uint_base& u,
const sc_unsigned&  v);
friend  sc_signed operator & (const sc_int_base& u,
const sc_unsigned&  v);
sc_unsigned& operator &= (const sc_int_base& v);
sc_unsigned& operator &= (const sc_uint_base& v);

// Bitwise OR operators:
friend  sc_signed operator | (const sc_unsigned& u,
const sc_signed& v);
friend  sc_signed operator | (const sc_signed&   u,
const sc_unsigned&  v);
friend sc_unsigned operator | (const sc_unsigned& u,
const sc_unsigned&  v);
friend  sc_signed operator | (const sc_unsigned& u,
int64          v);
friend sc_unsigned operator | (const sc_unsigned& u,
uint64         v);
friend  sc_signed operator | (const sc_unsigned& u,
long        v);
friend sc_unsigned operator | (const sc_unsigned& u,
unsigned long     v);
friend  sc_signed operator | (const sc_unsigned& u,
int          v);
friend sc_unsigned operator | (const sc_unsigned& u,
unsigned int   v)
friend  sc_signed operator | (int64
u, const sc_unsigned&    v);
friend sc_unsigned operator | (uint64
  u, const sc_unsigned&  v);
friend  sc_signed operator | (long               u,
const sc_unsigned&  v);
friend sc_unsigned operator | (unsigned long      u,
const sc_unsigned&  v);
friend  sc_signed operator | (int                 u,
const sc_unsigned&  v);
friend sc_unsigned operator | (unsigned int      u,
const sc_unsigned&  v)
sc_unsigned& operator |= (const sc_signed&   v);
sc_unsigned& operator |= (const sc_unsigned& v);
sc_unsigned& operator |= (int64                 v);
sc_unsigned& operator |= (uint64               v);
sc_unsigned& operator |= (long              v);
sc_unsigned& operator |= (unsigned long      v);
sc_unsigned& operator |= (int                v)
sc_unsigned& operator |= (unsigned int      v)
friend sc_unsigned operator | (const sc_unsigned& u,
const sc_uint_base& v);
```



```
friend  sc_signed operator | (const sc_unsigned& u,
const sc_int_base&  v);
friend sc_unsigned operator | (const sc_uint_base& u,
const sc_unsigned&  v);
friend  sc_signed operator | (const sc_int_base& u,
const sc_unsigned&  v);
sc_unsigned& operator |= (const sc_int_base& v);
sc_unsigned& operator |= (const sc_uint_base& v);

// Bitwise XOR operators:
friend  sc_signed operator ^ (const sc_unsigned& u,
const sc_signed& v);
friend  sc_signed operator ^ (const sc_signed&   u,
const sc_unsigned&  v);
friend sc_unsigned operator ^ (const sc_unsigned& u,
const sc_unsigned&  v);
friend  sc_signed operator ^ (const sc_unsigned& u,
int64          v);
friend sc_unsigned operator ^ (const sc_unsigned& u,
uint64        v);
friend  sc_signed operator ^ (const sc_unsigned& u,
long        v);
friend sc_unsigned operator ^ (const sc_unsigned& u,
unsigned long     v);
friend  sc_signed operator ^ (const sc_unsigned& u,
int          v);
friend sc_unsigned operator ^ (const sc_unsigned& u,
unsigned int   v)
friend  sc_signed operator ^ (int64
u, const sc_unsigned&    v);
friend sc_unsigned operator ^ (uint64
  u, const sc_unsigned& v);
friend  sc_signed operator ^ (long               u,
const sc_unsigned&  v);
friend sc_unsigned operator ^ (unsigned long      u,
const sc_unsigned&  v);
friend  sc_signed operator ^ (int                u,
const sc_unsigned&  v);
friend sc_unsigned operator ^ (unsigned int      u,
const sc_unsigned&  v)
sc_unsigned& operator ^= (const sc_signed&   v);
sc_unsigned& operator ^= (const sc_unsigned& v);
sc_unsigned& operator ^= (int64                v);
sc_unsigned& operator ^= (uint64              v);
sc_unsigned& operator ^= (long             v);
sc_unsigned& operator ^= (unsigned long      v);
sc_unsigned& operator ^= (int                v)
sc_unsigned& operator ^= (unsigned int      v)
friend sc_unsigned operator ^ (const sc_unsigned& u,
const sc_uint_base& v);
friend  sc_signed operator ^ (const sc_unsigned& u,
const sc_int_base&  v);
friend sc_unsigned operator ^ (const sc_uint_base& u,
const sc_unsigned&  v);
```



```
friend  sc_signed operator ^ (const sc_int_base& u,
const sc_unsigned& v);
sc_unsigned& operator ^= (const sc_int_base& v);
sc_unsigned& operator ^= (const sc_uint_base& v);

// LEFT SHIFT operators:
friend sc_unsigned operator << (const sc_unsigned&u,
const sc_signed& v);
friend  sc_signed operator << (const sc_signed&  u,
const sc_unsigned& v);
friend sc_unsigned operator << (const sc_unsigned&u,
const sc_unsigned& v);
friend sc_unsigned operator << (const sc_unsigned&u,
int64          v);
friend sc_unsigned operator << (const sc_unsigned&u,
uint64         v);
friend sc_unsigned operator << (const sc_unsigned&u,
long          v);
friend sc_unsigned operator << (const sc_unsigned&u,
unsigned long     v);
friend sc_unsigned operator << (const sc_unsigned&u,
int          v)
friend sc_unsigned operator << (const sc_unsigned&u,
unsigned int   v)
sc_unsigned& operator <<= (const sc_signed&  v);
sc_unsigned& operator <<= (const sc_unsigned&v);
sc_unsigned& operator <<= (int64                  v);
sc_unsigned& operator <<= (uint64                v);
sc_unsigned& operator <<= (long                  v);
sc_unsigned& operator <<= (unsigned long      v);
sc_unsigned& operator <<= (int                v)
sc_unsigned& operator <<= (unsigned int      v)
friend sc_unsigned operator << (const sc_unsigned&u,
const sc_uint_base& v);
friend sc_unsigned operator << (const sc_unsigned&u,
const sc_int_base& v);
sc_unsigned& operator <<= (const sc_int_base&v);
sc_unsigned& operator <<= (const sc_uint_base& v);

// RIGHT SHIFT operators:
friend sc_unsigned operator >> (const sc_unsigned&u,
const sc_signed& v);
friend  sc_signed operator >> (const sc_signed&  u,
const sc_unsigned& v);
friend sc_unsigned operator >> (const sc_unsigned&u,
const sc_unsigned& v);
friend sc_unsigned operator >> (const sc_unsigned&u,
int64          v);
friend sc_unsigned operator >> (const sc_unsigned&u,
uint64         v);
friend sc_unsigned operator >> (const sc_unsigned&u,
long          v);
friend sc_unsigned operator >> (const sc_unsigned&u,
unsigned long     v);
```



```
friend sc_unsigned operator >> (const sc_unsigned&u,
int           v)
friend sc_unsigned operator >> (const sc_unsigned&u,
unsigned int   v)
sc_unsigned& operator >>= (const sc_signed&  v);
sc_unsigned& operator >>= (const sc_unsigned&v);
sc_unsigned& operator >>= (int64                  v);
sc_unsigned& operator >>= (uint64                v);
sc_unsigned& operator >>= (long                  v);
sc_unsigned& operator >>= (unsigned long      v);
sc_unsigned& operator >>= (int                v)
sc_unsigned& operator >>= (unsigned int      v)
friend sc_unsigned operator >> ( const sc_unsigned& ,
const sc_uint_base& );
friend sc_unsigned operator >> ( const sc_unsigned&,
const sc_int_base& );
sc_unsigned& operator >>= (const sc_int_base&v);
sc_unsigned& operator >>= (const sc_uint_base& v);

// Unary arithmetic operators
friend sc_unsigned operator + (const sc_unsigned& u);
friend   sc_signed operator - (const sc_unsigned& u);

// Logical EQUAL operators:
friend bool operator == (const sc_unsigned&  u, const
sc_signed&v);
friend bool operator == (const sc_signed&    u, const
sc_unsigned& v);
friend bool operator == (const sc_unsigned&  u, const
sc_unsigned& v);
friend bool operator == (const sc_unsigned&  u, int64
        v);
friend bool operator == (const sc_unsigned&  u, uint64
       v);
friend bool operator == (const sc_unsigned&  u, long
       v);
friend bool operator == (const sc_unsigned&  u,
unsigned long     v);
friend bool operator == (const sc_unsigned&  u, int
        v)
friend bool operator == (const sc_unsigned&  u,
unsigned int   v)
friend bool operator == (int64                u, const
sc_unsigned& v);
friend bool operator == (uint64                   u,
const sc_unsigned&  v);
friend bool operator == (long                u, const
sc_unsigned& v);
friend bool operator == (unsigned long        u, const
sc_unsigned& v);
friend bool operator == (int                  u, const
sc_unsigned& v)
friend bool operator == (unsigned int        u, const
sc_unsigned& v)
```



```
friend bool operator == (const sc_unsigned&  u, const
sc_uint_base& v);
friend bool operator == (const sc_unsigned&  u, const
sc_int_base& v);
friend bool operator == (const sc_uint_base& u, const
sc_unsigned& v);
friend bool operator == (const sc_int_base&  u, const
sc_unsigned& v);

// Logical NOT_EQUAL operators:
friend bool operator != (const sc_unsigned&  u, const
sc_signed&v);
friend bool operator != (const sc_signed&    u, const
sc_unsigned& v);
friend bool operator != (const sc_unsigned&  u, const
sc_unsigned& v);
friend bool operator != (const sc_unsigned&  u, int64
        v);
friend bool operator != (const sc_unsigned&  u, uint64
       v);
friend bool operator != (const sc_unsigned&  u, long
       v);
friend bool operator != (const sc_unsigned&  u,
unsigned long      v);
friend bool operator != (const sc_unsigned&  u, int
        v)
friend bool operator != (const sc_unsigned&  u,
unsigned int   v)
friend bool operator != (int64                u, const
sc_unsigned& v);
friend bool operator != (uint64                   u,
const sc_unsigned&  v);
friend bool operator != (long                u, const
sc_unsigned& v);
friend bool operator != (unsigned long        u, const
sc_unsigned& v);
friend bool operator != (int                  u, const
sc_unsigned& v)
friend bool operator != (unsigned int        u, const
sc_unsigned& v)
friend bool operator != (const sc_unsigned&  u, const
sc_uint_base& v);
friend bool operator != (const sc_unsigned&  u, const
sc_int_base& v);
friend bool operator != (const sc_uint_base& u, const
sc_unsigned& v);
friend bool operator != (const sc_int_base&  u, const
sc_unsigned& v);

// Logical LESS_THAN operators:
friend bool operator < (const sc_unsigned&   u, const
sc_signed&v);
friend bool operator < (const sc_signed&u, const
sc_unsigned& v);
```



```
friend bool operator < (const sc_unsigned&   u, const
sc_unsigned& v);
friend bool operator < (const sc_unsigned&   u, int64
        v);
friend bool operator < (const sc_unsigned&   u, uint64
       v);
friend bool operator < (const sc_unsigned&   u, long
       v);
friend bool operator < (const sc_unsigned&   u,
unsigned long     v);
friend bool operator < (const sc_unsigned&   u, int
        v)
friend bool operator < (const sc_unsigned&   u,
unsigned int   v)
friend bool operator < (int64                 u, const
sc_unsigned& v);
friend bool operator < (uint64               u, const
sc_unsigned& v);
friend bool operator < (long                 u, const
sc_unsigned& v);
friend bool operator < (unsigned long         u, const
sc_unsigned& v);
friend bool operator < (int                   u, const
sc_unsigned& v)
{ return operator<((long) u,  v); }
friend bool operator < (unsigned int         u, const
sc_unsigned& v)
{ return operator<((unsigned long) u,   v); }
friend bool operator < (const sc_unsigned&   u, const
sc_uint_base& v);
friend bool operator < (const sc_unsigned&   u, const
sc_int_base& v);
friend bool operator < (const sc_uint_base& u, const
sc_unsigned& v);
friend bool operator < (const sc_int_base&   u, const
sc_unsigned& v);

// Logical LESS_THAN_AND_EQUAL operators:
friend bool operator <= (const sc_unsigned&  u, const
sc_signed&v);
friend bool operator <= (const sc_signed&    u, const
sc_unsigned& v);
friend bool operator <= (const sc_unsigned&  u, const
sc_unsigned& v);
friend bool operator <= (const sc_unsigned&  u, int64
        v);
friend bool operator <= (const sc_unsigned&  u, uint64
       v);
friend bool operator <= (const sc_unsigned&  u, long
       v);
friend bool operator <= (const sc_unsigned&  u,
unsigned long     v);
friend bool operator <= (const sc_unsigned&  u, int
        v)
```



```
friend bool operator <= (const sc_unsigned&  u,
unsigned int   v)
friend bool operator <= (int64                u, const
sc_unsigned& v);
friend bool operator <= (uint64                   u,
const sc_unsigned&  v);
friend bool operator <= (long                 u, const
sc_unsigned& v);
friend bool operator <= (unsigned long        u, const
sc_unsigned& v);
friend bool operator <= (int                  u, const
sc_unsigned& v)
friend bool operator <= (unsigned int        u, const
sc_unsigned& v)
friend bool operator <= (const sc_unsigned&  u, const
sc_uint_base& v);
friend bool operator <= (const sc_unsigned&  u, const
sc_int_base& v);
friend bool operator <= (const sc_uint_base& u, const
sc_unsigned& v);
friend bool operator <= (const sc_int_base&  u, const
sc_unsigned& v);

// Logical GREATER_THAN operators:
friend bool operator > (const sc_unsigned&   u, const
sc_signed&v);
friend bool operator > (const sc_signed&u, const
sc_unsigned& v);
friend bool operator > (const sc_unsigned&   u, const
sc_unsigned& v);
friend bool operator > (const sc_unsigned&   u, int64
        v);
friend bool operator > (const sc_unsigned&   u, uint64
       v);
friend bool operator > (const sc_unsigned&   u, long
       v);
friend bool operator > (const sc_unsigned&   u,
unsigned long     v);
friend bool operator > (const sc_unsigned&   u, int
        v)
friend bool operator > (const sc_unsigned&   u,
unsigned int   v)
friend bool operator > (int64                 u, const
sc_unsigned& v);
friend bool operator > (uint64               u, const
sc_unsigned& v);
friend bool operator > (long                 u, const
sc_unsigned& v);
friend bool operator > (unsigned long         u, const
sc_unsigned& v);
friend bool operator > (int                   u, const
sc_unsigned& v)
friend bool operator > (unsigned int         u, const
sc_unsigned& v)
```



```
    friend bool operator > (const sc_unsigned&   u, const
    sc_uint_base& v);
    friend bool operator > (const sc_unsigned&   u, const
    sc_int_base& v);
    friend bool operator > (const sc_uint_base& u, const
    sc_unsigned& v);
    friend bool operator > (const sc_int_base&   u, const
    sc_unsigned& v);

    // Logical GREATER_THAN_AND_EQUAL operators:

    friend bool operator >= (const sc_unsigned&  u, const
    sc_signed&v);
    friend bool operator >= (const sc_signed&    u, const
    sc_unsigned& v);
    friend bool operator >= (const sc_unsigned&  u, const
    sc_unsigned& v);
    friend bool operator >= (const sc_unsigned&  u, int64
            v);
    friend bool operator >= (const sc_unsigned&  u, uint64
           v);
    friend bool operator >= (const sc_unsigned&  u, long
           v);
    friend bool operator >= (const sc_unsigned&  u,
    unsigned long     v);
    friend bool operator >= (const sc_unsigned&  u, int
            v)
    friend bool operator >= (const sc_unsigned&  u,
    unsigned int   v)
    friend bool operator >= (int64                u, const
    sc_unsigned& v);
    friend bool operator >= (uint64                   u,
    const sc_unsigned&  v);
    friend bool operator >= (long                u, const
    sc_unsigned& v);
    friend bool operator >= (unsigned long        u, const
    sc_unsigned& v);
    friend bool operator >= (int                  u, const
    sc_unsigned& v)
    friend bool operator >= (unsigned int        u, const
    sc_unsigned& v)
    friend bool operator >= (const sc_unsigned&  u, const
    sc_uint_base& v);
    friend bool operator >= (const sc_unsigned&  u, const
    sc_int_base& v);
    friend bool operator >= (const sc_uint_base& u, const
    sc_unsigned& v);
    friend bool operator >= (const sc_int_base&  u, const
    sc_unsigned& v);

    // Bitwise NOT operator (unary).
    friend sc_unsigned operator ~ (const sc_unsigned& u);
};
```


Description

sc_unsigned is an integer with an arbitrary word length W. The word length is specified at construction and can never change.

Public Constructors

```
explicit
sc_unsigned( int nb );
```

Create an sc_unsigned instance with an initial value of 0 and word length nb.

```
sc_unsigned( const sc_unsigned& a );
```

Create an sc_unsigned instance with an initial value of a and word length of a.

Copy Constructor

```
sc_unsigned( const sc_unsigned& );
```

Methods

```
bool
iszero() const;
```

Return true if the value of the sc_unsigned instance is zero.

```
int
length() const ;
```

Return the word length.

```
void
print( ostream& os = cout ) const ;
```

Print the sc_uint_base instance to an output stream.

```
void
reverse();
```

Reverse the contents of the sc_unsigned instance. I.e. LSB becomes MSB and vice versa.

```
bool
sign() const;
```

Return false.

```
void
scan( istream& is = cin ) ;
```

Read a sc_uint_base value from an input stream.

Assignment Operators

```
sc_ unsigned& operator = ( T ) ;
T in { sc_[un]signed, sc_[un]signed_subref†,
    sc_[un]signed_concref†, char*, [u]int64, [unsigned]
    long, [unsigned] int, double, sc_[u]int_base,
    sc_bv_base, sc_lv_base, sc_fxval, sc_fxval_fast,
    sc_fxnum, sc_fxnum_fast }}
```


Assign the value of the right-hand side to the left-hand side. The value is truncated, if its word length is greater than W. If not, the value is sign extended.

Increment and Decrement Operators

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

The operation is performed as done for type unsigned int.

```
sc_unsigned& operator  -- () ;
const sc_unsigned operator  -- ( int ) ;
```

The operation is performed as done for type unsigned int.

Bit Selection

```
sc_unsigned_bitref   operator [] ( int );
sc_unsigned_bitref_r operator [] ( int ) const;
sc_unsigned_bitref   bit( int );
sc_unsigned_bitref_r bit( int ) const;
```

Return a reference to a single bit.

Part Selection

```
sc_unsigned_subref   range( int high, int low );
sc_unsigned_subref_r range( int high, int low ) const;
sc_unsigned_subref   operator () ( int high, int low );
sc_unsigned_subref_r operator () ( int high, int low )
    const;
```

Return a reference to a range of bits. The MSB is set to the bit at position high, the LSB is set to the bit at position low.

Arithmetic Assignment Operators

```
friend sc_unsigned operator OP ( sc_unsigned , sc_signed );
friend sc_unsigned operator OP ( sc_signed , sc_unsigned );
friend sc_unsigned operator OP ( sc_signed , sc_signed );
friend sc_unsigned operator OP ( sc_signed , T );
friend sc_unsigned operator OP ( T , sc_signed );
T in { sc_[u]int_base, [u]int64, [unsigned] long,
    [unsigned] int }
OP in { + - * / % & | ^ == != < <= > >= }

friend sc_unsigned operator OP ( sc_unsigned , T );
friend sc_unsigned operator OP ( T , sc_unsigned );
T in { sc_int_base, int64, long, int }
OP in { + - * / % & | ^ == != < <= > >= }
```

The operation OP is performed and the result is returned.


```
sc_unsigned& operator OP (T);
T in { sc_[un]signed, sc_[u]int_base, [u]int64, [unsigned]
    long, [unsigned] int }
OP in { += -= *= /= %= &= |= ^= }
```

The operation OP is performed and the result is assigned to the left-hand side.

Shift Operators

```
friend sc_unsigned operator OP ( sc_unsigned a , sc_signed
    b );
friend sc_unsigned operator OP ( sc_signed a , sc_unsigned
    b );
friend sc_unsigned operator OP ( sc_signed a , T b );
T in { sc_[u]int_base, [u]int64, [unsigned] long,
    [unsigned] int }
OP in { << >> }
```

Shift a to the left/right by b bits and return the result.

```
sc_unsigned& operator OP ( T );
T in { sc_[un]signed, sc_[u]int_base, [u]int64, [unsigned]
    long, [unsigned] int }
OP in { <<= >>= }
```

Shift the sc_unsigned instance to the left/right by i bits and assign the result to the sc_unsigned instance.

Bitwise not

```
friend sc_unsigned operator ~ ( sc_unsigned a );
```

Return the bitwise not of a;

Explicit Conversion

```
sc_string to_string( sc_numrep = SC_DEC ) const
sc_string to_string( sc_numrep, bool ) const
```

Convert the sc_unsigned instance into its string representation.

```
double   to_double() const ;
int      to_int() const ;
int64    to_int64() const ;
long     to_long() const ;
uint64   to_uint64() const ;
unsigned int    to_uint() const ;
unsigned long   to_ulong() const ;
```

Converts the value of sc_unsigned instance into the corresponding data type. If the requested type has less word length than the sc_unsigned instance, the value gets truncated accordingly. If the requested type has greater word length than the sc_unsigned instance, the value gets sign extended, if necessary.

12 Global Function Reference

This section contains a summary of the SystemC global functions. The functions are presented in alphabetical order. The function prototype consists of the return type, the function name, and the argument type or types. A brief description and summary of each function follows its prototype. Several of the function descriptions include examples.

12.1 notify

Prototype

```
void
notify( sc_event& e );
```

Description

Causes immediate notification of event e.

Prototype

```
void
notify( const sc_time& t, sc_event& e );
```

Description

If `t` = `SC_ZERO_TIME` then causes notification of event e in the next delta-cycle else schedules notification at current time + `t`.

Prototype

```
void
notify( double v, sc_time_unit tu, sc_event& e );
```

Description

If `sc_time(v, tu) = SC_ZERO_TIME` then causes notification of event e in the next delta-cycle else schedules notification at current time + `sc_time(v, tu)`.

12.2 sc_abs

Prototype

```
template <class T>
T
sc_abs( const T& val_ );
```

Description

Returns the absolute value of `val_`.

12.3 sc_close_vcd_trace_file

Prototype

```
void
sc_close_vcd_trace_file( sc_trace_file† * tf );
```

Description

Closes the trace file `tf`, which was opened with **`sc_create_vcd_trace_file()`**.

12.4 sc_close_wif_trace_file

Prototype

```
void
sc_close_wif_trace_file( sc_trace_file† * tf );
```

Description

Closes the trace file `tf`, which was opened with **`sc_create_wif_trace_file()`**.

12.5 sc_copyright

Prototype

```
const char*
sc_copyright()
```

Description

Returns a character string that contains the copyright notice e. g.:

Copyright (c) 1996-2002 by all Contributors
ALL RIGHTS RESERVED

12.6 sc_create_vcd_trace_file

Prototype

```
sc_trace_file† *
sc_create_vcd_trace_file( const char* file_name );
```

Description

Creates a new `sc_vcd_trace_file` object and opens a VCD trace file named `file_name`. Returns a pointer to the `sc_vcd_trace_file` object. Used for tracing.

12.7 sc_create_wif_trace_file

Prototype

```
sc_trace_file† *
sc_create_wif_trace_file( const char* file_name );
```


Description

Creates a new `sc_wif_trace_file` object and opens a VCD trace file named `file_name`. Returns a pointer to the `sc_wif_trace_file` object. Used for tracing.

12.8 sc_gen_unique_name

Prototype

```
const char*
sc_gen_unique_name( const char* basename_ );
```

Description

Using `basename_` as a base, returns a character string that is unique within the current module (instance) or simulation context.

12.9 sc_get_curr_simcontext

Prototype

```
sc_simcontext*
sc_get_curr_simcontext();
```

Description

Returns a pointer to the `sc_simcontext` object that the simulation kernel maintains.

12.10 sc_get_default_time_unit

Prototype

```
sc_time
sc_get_default_time_unit();
```

Description

Returns the default time unit.

12.11 sc_get_time_resolution

Prototype

```
sc_time
sc_get_time_resolution();
```

Description

Returns the time resolution.

12.12 sc_max

Prototype

```
template <class T>
```



```
T
sc_max( const T& a_val, const T& b_val );
```

Description

Returns the value of which is greater, `a_val` or `b_val`. If `a_val` equals `b_val` then `a_val` is returned.

12.13 sc_min

Prototype

```
template <class T>
T
sc_min( const T& a_val, const T& b_val );
```

Description

Returns the value of which is lesser, `a_val` or `b_val`. If `a_val` equals `b_val` then `a_val` is returned.

12.14 sc_set_default_time_unit

Prototype

```
void
sc_set_default_time_unit( double val, sc_time_unit tu );
```

Description

Sets the default time unit with a value of `sc_time(val, tu)`. Value `val` must be positive and a power of ten. The default time unit value specified must be greater than or equal to the current time resolution. This function may only be called once and only during elaboration (Chapter 2.2), and only before any `sc_time` objects unequal `SC_ZERO_TIME` are created.

12.15 sc_set_time_resolution

Prototype

```
void
sc_set_time_resolution( double val, sc_time_unit tu );
```

Description

Sets the time resolution with a value of `sc_time(val, tu)`. Value must be positive and a power of ten. The time resolution value specified must be greater than or equal to 1 femtosecond. This function may only be called once and only during elaboration (Chapter 2.2) , and only before any `sc_time` objects unequal `SC_ZERO_TIME` are created.

12.16 sc_simulation_time

Prototype

```
double
```



```
sc_simulation_time();
```

Description

Returns a value of type `double`. The value is the current simulation time in default time units.

12.17 sc_start

Prototype

```
void
sc_start( const sc_time& duration )
```

Description

Causes simulation to start and run for the specified amount of time, `duration`. If this is the first call to `sc_start()` the simulation is first initialized, which includes running one delta-cycle sequence before time 0. If `duration` is equal to `SC_ZERO_TIME`, and this is not the first call to `sc_start()` then the simulation runs one delta-cycle sequence at the current time.

Prototype

```
void
sc_start( double d_val, sc_time_unit d_tu );
```

Description

Causes simulation to start and run for the specified amount of time, `sc_time(d_val, d_tu)`. If this is the first call to `sc_start()` the simulation is first initialized, which includes running one delta-cycle sequence before time 0. If the specified amount of time is equal to `SC_ZERO_TIME` and this is not the first call to `sc_start()` then the simulation runs one delta-cycle sequence at the current time.

Prototype

```
void
sc_start( double d_val = -1 );
```

Description

Causes simulation to start and run for the specified amount of time, `sc_time(d_val, sc_get_default_time_unit())`, i.e., `d_val` is specified in terms of the current default time unit. If this is the first call to `sc_start()` the simulation is first initialized, which includes running one delta-cycle sequence before time 0. If the value of `d_val` is not specified or the value of `d_val` is –1 then the simulation runs "forever". If the specified amount of time is `SC_ZERO_TIME` and this is not the first call to `sc_start()`, then the simulation runs one delta-cycle sequence at the current time.

Examples

```
//Given
```



```
sc_time r_time( 1000, SC_NS);

// Then
sc_start(r_time); // run 1000 nSec
sc_start(1000, SC_NS);  // run 1000 nSec
sc_start( 1000 );  // run 1000 default time units
sc_start(); // run forever
sc_start(-1); // run forever
```

12.18 sc_stop

Prototype

```
void
sc_stop();
```

Description

Halts simulation at the end of the current delta-cycle. Causes `sc_start()` to return control to `sc_main()`.

12.19 sc_stop_here

Prototype

```
void
sc_stop_here( const char* id, sc_severity severity );
```

Description

Called by the simulator after an error or warning situation occurs. The id and severity of the error or warning are passed to `sc_stop_here()`. This function is provided as a debugging aid.

12.20 sc_time_stamp

Prototype

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

Description

Returns the current simulation time.

12.21 sc_trace

Prototype

```
// for SystemC types
void sc_trace( sc_trace_file* tf, const tp& object_, const
    sc_string& name_ )
void sc_trace( sc_trace_file* tf, const tp* object_, const
    sc_string& name_ );
tp in {sc_logic, sc_[u]int_base, sc_[un]signed, sc_bv_base,
    sc_lv_base}

// for C++ types
```



```
void sc_trace( sc_trace_file* tf, const tp& object_, const
   sc_string& name_, int width = 8 * sizeof( tp ) )
void sc_trace( sc_trace_file* tf, const tp* object_, const
   sc_string& name_, int width = 8 * sizeof( tp ) )
tp in {bool, float, double, unsigned char, unsigned short,
   unsigned int, unsigned long, char, short, int, long}

// for sc_signal channels
template <class T>
void sc_trace( sc_trace_file* tf, const
   sc_signal_in_if<T>& object_, const sc_string& name_ )
template <class T>
void sc_trace( sc_trace_file* tf, const
   sc_signal_in_if<T>& object_, const char* name_ )
void sc_trace( sc_trace_file* tf, const
   sc_signal_in_if<char>& object_, const sc_string& name_,
   int width );
void sc_trace( sc_trace_file* tf, const
   sc_signal_in_if<short>& object_, const sc_string& name_,
   int width );
void sc_trace( sc_trace_file* tf, const
   sc_signal_in_if<int>& object_, const sc_string& name_,
   int width );
void sc_trace( sc_trace_file* tf, const
   sc_signal_in_if<long>& object_, const sc_string& name_,
   int width );

// for enumerated object
void sc_trace( sc_trace_file* tf, const unsigned int&
   object_, const sc_string& name_, const char**
   enum_literals );

// for sc_signal specialized ports
template <class T>
void sc_trace( sc_trace_file* tf, const sc_in<T>& object_,
   const sc_string& name_ )
template <class T>
void sc_trace( sc_trace_file* tf, const sc_inout<T>&
   object_, const sc_string& name_ )
template <>
void sc_trace<sc_logic>( sc_trace_file* tf, const
   sc_in<sc_logic>& object_, const sc_string& name_ )
template <>
void sc_trace<sc_logic>( sc_trace_file* tf, const
   sc_inout<sc_logic>& object_, const sc_string& name_ )
template <>
void sc_trace<bool>( sc_trace_file* tf, const sc_in<bool>&
   object_, const sc_string& name_ )
template <>
void sc_trace<bool>( sc_trace_file* tf, const
   sc_inout<bool>& object_, const sc_string& name_ )
```

Description

Adds trace of `object_` along with the string `name_` to the trace file `tf`.

12.22 sc_version

Prototype

```
const char*
sc_version();
```

Description

Returns a character string with the version of the SystemC class library, e g:

SystemC 2.0.1 --- Jan 8 2003 16:42:30

13 Global Enumerations, Typedefs and Constants

13.1 Enumerations

13.1.1 sc_time_unit

```
enum sc_time_unit
{
   SC_FS = 0,  // femtosecond
   SC_PS,       // picosecond
   SC_NS,       // nanosecond
   SC_US,       // microsecond
   SC_MS,       // millisecond
   SC_SEC       // second
};
```

13.1.2 sc_logic_value_t

```
enum sc_logic_value_t
{
   Log_0 = 0,
   Log_1,
   Log_Z,
   Log_X
};
```

13.2 Typedefs

13.2.1 sc_behavior

```
typedef sc_module sc_behavior ;
```

13.2.2 sc_channel

```
typedef sc_module sc_channel ;
```

13.2.3 clk ports

```
typedef sc_in<bool>  sc_in_clk ;
```



```
typedef sc_inout<bool> sc_inout_clk ;
typedef sc_out<bool>   sc_out_clk ;
```

13.2.4 Data Types

int64

A signed 64 bit integer type

uint64

An unsigned 64 bit integer type

13.3 Constants

13.3.1 SC_DEFAULT_STACK_SIZE

```
const int SC_DEFAULT_STACK_SIZE;  // value = 0x10000
```

Sets maximum stack size for thread processes.

13.3.2 SC_LOGIC_

```
const sc_logic SC_LOGIC_0( Log_0 );
const sc_logic SC_LOGIC_1( Log_1 );
const sc_logic SC_LOGIC_Z( Log_Z );
const sc_logic SC_LOGIC_X( Log_X );
```

13.3.3 SC_MAX_NUM_DELTA_CYCLES

```
const int SC_MAX_NUM_DELTA_CYCLES; // value = 10000
```

Sets maximum number of delta-cycles per time step before issuing an error.

13.3.4 SC_ZERO_TIME

```
const sc_time SC_ZERO_TIME ;  // value = 0
```

13.3.5 SYSTEMC_DEBUG

Preprocessor macro, not defined by default. If defined when building the SystemC library, it will activate more internal checks and diagnostic messages.

13.3.6 SYSTEMC_VERSION

Preprocessor macro specifying the version of the SystemC library. For version 2.0.1, the value is 20020405

14 Deprecated items

The following list of items in the reference implementation are deprecated and are not included in this document:

- sensitive_pos()
- sensitive_neg()
- sensitive_pos
- sensitive_neg
- sc_create_isdb_file()
- sc_close_isdb_file()
- sc_cycle()
- sc_initialize()
- notify_delayed()
- end_module()

The following list of items in the reference implementation are under consideration to be deprecated and are not included in this document:

- SC_CTHREAD
- Watching
- Local watching
- wait_until()
- delayed() and associated forms

