

## sc\_main

```
#include "systemc.h"
// include module declarations

int sc_main(int argc, char *argv[] )
{
    // Create channels
    sc_signal<type> signal_name, signal_name, ...;
    // Create clock
    sc_clock clock_name ("name", period, duty_cycle, start_time, positive_first);
    // Module instantiations
    module_name instance_name("name");
    // Module port bindings
    // By name binding, do for each port
    instance_name.port_name (signal_name);
    // By order port binding
    instance_name ( signal_name, signal_name, ... );
    // By order using stream
    instance_name << signal_name << signal_name, ...;
    // Clock generation
    sc_start(value);

    return 0;
}
```

## Clock syntax

```
sc_clock clock_name ("name", period, duty_cycle, start_time, positive_first );
name: name type: char *
period: clock period type: variable of type sc_time or constant of type uint64
duty_cycle: clock duty cycle type: double default value: 0.5
start_time: time of first edge type: variable of type sc_time or
constant of type uint64

default value: 0
positive_first: first edge positive type: bool default value: true
```

## Clock object methods:

```
clock_name.name() returns the "name"
clock_name.period() returns the clock period
clock_name.duty_cycle() returns the clock duty cycle
clock_name.pos() Gives a reference to the positive edge of clk
usage: sensitive << clock_name.pos()
clock_name.neg() Gives a reference to the negative edge of clk
usage: sensitive << clock_name.neg()
```

## Clock functions

```
sc_start() Generate the waveforms for all sc_clock objects
sc_stop() Stops simulations
sc_time_stamp() Returns the current simulation time as sc_time
sc_simulation_time() Returns the current simulation time as double
```

## Data Types

### Scalar

```
sc_int<length> variable_name, variable_name, ...;
sc_uint<length> variable_name, variable_name, ...;
sc_bigint<length> variable_name, variable_name, ...;
sc_biguint<length> variable_name, variable_name, ...;
//length: specifies the number of elements in the array.
//Rightmost is LSB(0), Leftmost is MSB (length-1).
sc_bit variable_name, variable_name, ...;
//Values: '0', '1'
sc_bv<length> variable_name, variable_name, ...;
//length: specifies the number of elements in the array.
//Values: '0', '1'. More than one bit represented by "0011".
sc_logic variable_name, variable_name, ...;
//Values: '0', '1', 'X', 'Z'
sc_lv<length> variable_name, variable_name, ...;
//length: specifies the number of elements in the array.
//Values: '0', '1', 'X', 'Z'. More than one bit represented by "0011XXZZ".
```

### Fixedpoint

```
sc_fixed<wl, iwl, q_mode, o_mode, n_bits> object_name, object_name, ...;
sc_ufixed<wl, iwl, q_mode, o_mode, n_bits> object_name, object_name, ...;
sc_fixed_fast<wl, iwl, q_mode, o_mode, n_bits> object_name, object_name, ..;
sc_ufixed_fast<wl, iwl, q_mode, o_mode, n_bits> object_name, object_name, ..;
wl: total word length, number of bits used in the type
iwl: integer word length, number of bits to the left of the binary point (.)
q_mode: quantization mode
o_mode: overflow mode
n_bits: number of saturated bits, used for overflow mode
sc_fix object_name (list of options);
sc_fix_fast object_name (list of options);
sc_ufix object_name (list of options);
sc_ufix_fast object_name (list of options)

q_mode: SC_RND, SC_RND_ZERO, SC_RND_MIN_INF, SC_RND_INF,
SC_RND_CONV, SC_TRN, SC_TRN_ZERO
o_mode: SC_SAT, SC_SAT_ZERO, SC_SAT_SYM, SC_WRAP, SC_WRAP_SM
```

## Data Operations/Functions

Type	sc_bit sc_bc sc_lv	sc_bc sc_lv	sc_int, sc_uint sc_bigint, sc_biguint	sc_fixed, sc_ufixed, sc_fix, sc_ufix
Operation	~ & ^	~ & ^   << >>	~ & ^   << >>	~ & ^
Arithmetic			+ - * / %	+ - * / % >> <<
Logical				
Equality	== !=	== !=	== !=	== !=
Relational			> < <= >=	
Assignment	= &=  = ^=	= &=  = ^=	= += -= *= /= %= &=  = ^=	= += -= *= /= %= &=  = ^=
Increment Decrement			++ --	++ --
Arithmetic if				
Concatenation	.	.	.	.
Bitselct		[x]	[x]	
Partselct		range()	range()	
Reduction		and_reduce or_reduce xor_reduce		

## Channels

Name	Methods
sc_signal	read(), write(), event()
sc_signal_rv	read(), event(), write()
For vectors, allows multiple writers	
sc_signal_resolved	read(), event(), write()
For non vectors, allows multiple writers	
sc_fifo	read(), nb_read(), num_available(), write(), nb_write(), num_free()
Point to point communication, one reader, one writer per fifo	
sc_mutex	kind(), lock(), trylock(), unlock()
Multipoint communication, only one writer/reader at the time	
sc_semaphore	kind(), wait(), trywait(), get_value(), post()
Limited concurrent access, specify number of concurrent users	
sc_buffer	kind()
Like sc_signal, value_change_event() and default_event() are triggered on each write	

## Resolved ports/signals

```
Syntax:
SC_MODULE ( module_name ) {
// ports
sc_in_rv<N> port_name, port_name,...;
sc_out_rv<N> port_name, port_name,...;
sc_inout_rv<N> port_name, port_name,...;
sc_signal_rv<N> signal_name, signal_name, ...;
// rest of module
}; // N is the number of bits
// Every bit can have either a 0, 1, X or Z value
```

## sc\_signal channel methods

read()	returns value of signal or port
write()	assigns value to signal or port
event()	returns true or false if event on signal or port
default_event()	any change of value
value_changed_event()	any change of value
posedge()	returns true if 0 -> 1 transition
negedge()	returns true if 1 -> 0 transition

## Modules

```
// Header file
SC_MODULE(module_name) {
// module port declarations
// signal variable declarations
// data variable declarations
// process declarations
// other method declarations
// module instantiations
SC_CTOR(module_name){
// process registration & declarations of sensitivity lists
// module instantiations & port connection declarations
// global watching registration
}
};

// Implementation file
void module_name::process_or_method_name() {
// process implementation
// SC_THREAD and SC_CTHREAD has
// while(true) loop
}
```

## Scalar Syntax:

```
SC_MODULE(module_name) {
// ports
  sc_in<port_type> port_name, port_name,... ;
  sc_out<port_type> port_name, port_name,... ;
  sc_inout<port_type> port_name, port_name,... ;
  sc_port<channel_type<port_type>, connections > port_name, port_name,... ;
  sc_port<channel_type<port_type>, connections > port_name, port_name,... ;
  sc_port<channel_type<port_type>, connections > port_name, port_name,... ;
// clock input (for SystemC 2.0 it is recommended to use sc_in<bool>)
  sc_in_clk clock_name;
// clock output (for SystemC 2.0 it is recommended to use sc_out<bool>)
  sc_out_clk clock_name;
// signals
  sc_signal<signal_type> signal_name, signal_name, ...;
// variables
  type variable_name, variable_name...;
// rest of module);
```

## Array Syntax:

```
SC_MODULE ( module_name) {
// ports
  sc_in<port_type> port_name[size], port_name[size], ... ;
  sc_out<port_type> port_name[size], port_name[size], ... ;
  sc_inout<port_type> port_name[size], port_name[size], ... ;
  sc_port<channel_type <port_type>, connections >port_name[size], port_name[size], ... ;
  sc_port<channel_type <port_type>, connections >port_name[size], port_name[size], ... ;
  sc_port<channel_type <port_type>, connections >port_name[size], port_name[size], ... ;
// signals
  sc_signal<signal_type> signal_name [size], signal_name [size], ...
// variables
  type variable_name[size], variable_name[size],...;
// rest of module
};
```

## Module inheritance

```
SC_MODULE( base_module )
{
...
// constructor
SC_CTOR( base_module )
{ ... }
};
class derived_module : public base_module
{
// process(es)
void proc_a();
SC_HAS_PROCESS( derived_module );
// parameter(s)
int some_parameter;
// constructor
derived_module( sc_module_name name_, int some_value )
: base_module( name_ ), some_parameter( some_value )
{
SC_THREAD( proc_a );
}
};
```

## Processes

```
// Header file
SC_MODULE(module_name) {
// module port declarations
// signal variable declarations
// data variable declarations
// process declarations
  void process_name_A();
  void process_name_B();
  void process_name_C();
// other method declarations
// module instantiations
  SC_CTOR(module_name){
// process registration
  SC_METHOD(process_name_A);
// Sensitivity list
  SC_THREAD(process_name_B);
// Sensitivity list
  SC_CTHREAD(process_name_C, clock_edge_reference);
  //clock_name.pos() or clock_name.neg()
// global watching registration
// no sensitivity list
// module instantiations & port connection declarations
}
};
```

## Sensitivity list

Sensitive to any change on port(s) or signal(s)  
**sensitive**(port\_or\_signal)  
**sensitive <<** port\_or\_signal << port\_or\_signal ...;  
 Sensitive to the positive edge of boolean port(s) or signal(s)  
**sensitive\_pos**(port\_or\_signal)  
**sensitive\_pos <<** port\_or\_signal << port\_or\_signal ...;  
 Sensitive to the negative edge of boolean port(s) or signal(s)  
**sensitive\_neg**(port\_or\_signal)  
**sensitive\_neg <<** port\_or\_signal << port\_or\_signal ...;

## Module instantiation

### Style 1

```
// Header file
SC_MODULE(module_name) {
// module port declarations
// signal variable declarations
// data variable declarations
// process declarations
// other method declarations
module_name_A instance_name_A; // module instantiation..
module_name_N instance_name_N; // module instantiation
```

```
SC_CTOR(module_name):
instance_name_A("name_A"),
instance_name_N("name_N")
{
// by name port binding
instance_name_A.port_1(signal_or_port);
// by order port binding
instance_name_N(signal_or_port, signal_or_port,...);
// process registration & declarations of sensitivity lists
// global watching registration
}
};
```

## Style 2

```
// Header file
SC_MODULE(module_name) {
// module port declarations
// signal variable declarations
// data variable declarations
// process declarations
// other method declarations
module_name_A *instance_name_A; // module instantiation..
module_name_N *instance_name_N; // module instantiation
SC_CTOR(module_name)
{
instance_name_A = new module_name_A("name_A"),
instance_name_N = new module_name_N("name_N")
instance_name_A->port_1(signal_or_port);
instance_name_A->port_2(signal_or_port);
(*instance_name_N)(signal_or_port, signal_or_port,...);
// process registration & declarations of sensitivity lists
// global watching registration
}
};
```

## Watching

```
// Header file
SC_MODULE(module_name) {
// module port declarations
// signal variable declarations
// data variable declarations
// process declarations
void process_name(); // other method declarations
// module instantiations
SC_CTOR(module_name){
SC_CTHREAD(process_name, clock_edge_reference // global watching registration
watching (reset.delayed() == 1); // delayed() method required
}
};
```

## Event

```
sc_event my_event; // event
sc_time t_zero (0,sc_ns);
sc_time t(10, sc_ms); // variable t of type sc_time
```

```
Immediate:
  my_event.notify();
  notify(my_event);
Delayed:
  my_event.notify(t_zero); // next delta cycle
  notify(t_zero, my_event); // next delta cycle
Timed:
  my_event.notify(t); // 10 ms delay
  notify(t, my_event); // 10 ms delay
```

## Dynamic sensitivity

```
wait for an event in a list of events:
wait(e1);
wait(e1 | e2 | e3);
wait( e1 & e2 & e3);
wait for specific amount of time:
wait(200, sc_ns);
wait on events with timeout:
wait(200, sc_ns, e1 | e2 | e3);
wait for number of clock cycles:
wait(200); // wait for 200 clock cycles, only for SC_CTHREAD
wait for one delta cycle:
wait( 0, sc_ns ); // wait one delta cycle.
wait( SC_ZERO_TIME ); // wait one delta cycle.
```