Transfer
Goorseweg 5
7475 BB Markelo
Tel +31 547 367 367

SystemC Quickreference Card

For Training: www.Transfer.nl

email: Training@Transfer.nl

Module structure

```
// Header file
SC_MODULE(module_name) {
   Port declarations
   Local channel declarations
   Variable declarations
   Process declarations
   Other method declarations
   Module instantiations
SC_CTOR(module_name){
   Process registration
   Static sensitivity list
   Module variable initialization
   Module instance / channel binding
   }
} ;
```

Port declarations:

```
   sc_port<interface_name> [, number of channels]> port_name,...
;
```

Local channel declaration:

```
   channel_type name [, name, name … ];
```

Variable declaration:

```
   Variable_type name [, name, name … ];
```

Process declaration:

```
   void process_name();
```

Process registration:

```
   SC_METHOD(process_name);
   SC_THREAD(process_name);
   SC_CTHREAD(process_name, clock_edge_reference);
   SC_SLAVE(process_name, slave_port);
```

Static sensitivity list:

```
   Functional syntax:
   sensitive(event1 [, event2, …])
   sensitive_pos(event1 [, event2, …])
   sensitive_neg(event1 [, event2, …])

   Stream syntax:
   sensitive << event1 << event2 …;
   sensitive_pos << event1 << event2 …;
   sensitive_neg << event1 << event2 …;
```

Custom constructor:

```
SC_MODULE( module_name )
{
…
   SC_HAS_PROCESS( module_name );

   module_name( sc_module_name name_string, arg1 [, arg2, …]
   )
   : sc_module( name_string )
{
   Process registration
   Static sensitivity list
   Module variable initialization
   Module instance / channel binding
 }
};
```

Hierarchical module instantation:

Method1: Initializaing using constructor initialization list

```
#include "submodule_name.h"
SC_MODULE(module_name) {
   Module port declarations
   …
   module_name_A instance_name_A;
   module_name_N instance_name_N;
   …
   Local channel declarations

   SC_CTOR(module_name): instance_name_A("name_A"),
                         instance_name_N("name_N")
   {
      instance_name_A.subport_name(modport_name);
      instance_name_A.subport_name(local_channel_name);
      or:
      instance_name_N(modport_name, local_channel_name,...);
      …
   }
} ;
```

Method2: Using pointers and dynamic memory allocation

```
#include "submodule_name.h"
SC_MODULE(module_name) {
   Module port declarations
   …
   module_name_A *instance_name_A;
   module_name_N *instance_name_N;
   …
   Local channel declarations

   SC_CTOR(module_name)
   {
      instance_name_A = new
submodule_name("instance_name");
      instance_name_A->subport_name(modport_name);
      instance_name_A->subport_name(local_channel_name);
      or:
      (*instance_name_N)(modport_name,
local_channel_name,...);
      …
   }
} ;
```

Main Routine Structure

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

int sc_main(int argc, char *agv[ ])
{
   Channel declarations
   Variable declarations
   Module instance declarations
   Module port  binding
   Time unit / resolution setup
   Set up tracing
   Start simulation
   return 0;
}
```

Module instantation:

```
   Module_name instance_name("instance_name");
```

With custom constructor:

```
   Module_name instance_name("instance_name", arg1 [, arg2,
…]);
```

Port binding:

Named method

```
   Instance_named.port_name(channel_name);
```

Positional method

```
   Instance_name(channel_name [, channel_name, …]);
   Instance_name << channel_name [ << channel_name << … ];
```

Function Reference

dont_initialize(); Prevent SC_METHOD or SC_THREAD process from automatically running at start of simulation.

gen_unique_name(basename**);** Returns a unique string that can be used to satisfy object initializations that require a unique string.

name(); Returns a string with the current module instance name

next_trigger(); Temporary overrides the static sensitivity list.
next_trigger(event_expression **);**
next_trigger(time**);**
next_trigger(time, event_expression**);**

sc_assert_fail(message_string, file_name, line_num**);**
prints out an error message

sc_copyright(); Returns string with SystemC copyright information

sc_cycle(duration); Advanced simulation time by specified amount of time. Should be called after **sc_initialize()**
sc_cycle(value [, sc_time_unit]**);**

sc_initialize(); Initialize the simulation

sc_simulation_time(); Returns current simulation time as double

sc_start(run_time); Initialize simulation and advances time

sc_start(value [, sc_time_unit]**);**
sc_start(-1);

sc_stop(); Stops simulation

sc_set_default_time_unit(value, sc_time_unit**);**
sc_get_default_time_unit(); Set and get the default time unit

sc_set_default_time_resolution(value, sc_time_unit**);**
sc_get_default_time_resolution(); Set and get minimum time resolution

sc_time_stamp(); Retuns current simulation time as sc_time

sc_version(); Returns string with SystemC library version

timed_out(); Returns bool, true if reactivation after last wait was due to timenout an not to event trigger

wait(); Wait for event as specified in static sensitivity list

wait(event_expression**);** Temporary overrides the static sensitivitiy list
wait(time**);**
wait(time, event_expression**);**

Data Types

sc_time_unit

can be one of:
SC_FS, SC_PS, SC_NS, SC_US, SC_MS, SC_SEC

sc_clock("ID", period, duty_cycle, offset, first_edge_positive);

sc_int<N>, sc_uint<N>

N: number of bits, max=64
Methods:

range(x,y)	**to_int**()	**to_uint**()
length()	**test**(i)	**set**(i)
set(i,b)		

sc_bigint<N>, sc_biguint<N>

N: number of bits, max=512
Methods:

range(x,y)	**to_int**()	**to_uint**()
length()	**test**(i)	**set**(i)
set(i,b)	**to_string**()	

to_string can take an argument:
SC_NOBASE, SC_BIN, SC_OCT, SC_DEC, SC_HEX

sc_logic

single bit value: '0', '1', 'X', 'Z'

sc_lv<N>

Vector of sc_logic values, N is number of bits
Methods:

range(x,y)	**to_int()**	**to_uint()**	**length()**
	set_bit(i, d)	**get_bit**(i)	**to_string()**
and_reduce()			
nand_reduce()	**nor_reduce()**	**or_reduce()**	**xor_reduce()**
xnor_reduce()			

sc_bit

single bit value: '0', '1'

sc_bv

Vector of sc_bit values, N is number of bits
Methods:

range(x,y)	**to_int()**	**to_uint()**	**length()**
	set_bit(i, d)	**get_bit**(i)	**to_string()**
and_reduce()			
nand_reduce()	**nor_reduce()**	**or_reduce()**	**xor_reduce()**
	xnor_reduce()		

sc_fixed<wl, iwl, q_mode, o_mode, n_bits>
sc_ufixed<wl, iwl, q_mode, o_mode, n_bits>

wl: total word_length
iwl: integer word length
q_mode: quantization mode
o_mode: overflow mode
n_bits: number of saturated bits

Quantization modes: **SC_RND, SC_RND_ZERO, SC_RND_INF, SC_RND_MIN_INF, SC_RND_CONV, SC_TRN, SC_TRN_ZERO**

Overflow modes:**SC_SAT, SC_SAT_ZERO, SC_SAT_SYM, SC_WRAP, SC_WRAP_SM**

sc_fixed_fast and sc_ufixed_fast

Faster implementations of sc_fixed and sc_ufixed, max bits is 53

Transfer
Goorseweg 5
7475 BB Markelo
Tel +31 547 367 367

SystemC Quickreference Card

For Training: www.Transfer.nl

email: Training@Transfer.nl

Operators

	Arithmetic					Bitwise					
	+	-	*	/	%	~	&	|	^	>>	<<
sc_int	+	+	+	+	+	+	+	+	+	+	+
sc_uint											
sc_bigint											
sc_biguint											
sc_bv						+	+	+	+	+	+
sc_lv											
sc_fixed	+	+	+	+	+	+	+	+	+	+	+
sc_ufixed											
sc_fix											
sc_ufix											

	Assignment									Equal	
	=	+=	-=	*=	/=	%=	&=	|=	^=	==	!=
sc_int	+	+	+	+	+	+	+	+	+	+	+
sc_uint											
sc_bigint											
sc_biguint											
sc_bv	+						+	+	+	+	+
sc_lv											
sc_fixed	+	+	+	+	+	+	+	+	+	+	+
sc_ufixed											
sc_fix											
sc_ufix											

	Relational				Auto		Bit	
	<	<=	>	>=	++	--	[ ]	(,)
sc_int	+	+	+	+	+	+✓	+	+
sc_uint								
sc_bigint								
sc_biguint								
sc_bv							+	+
sc_lv								
sc_fixed	+	+	+	+	+	+	+	+
sc_ufixed								
sc_fix								
sc_ufix								

Channel Reference

sc_buffer

Multipoint communications, one writer, many readers
Implements interface: **sc_signal_inout_if**

Events:
default_event() **value_changed_event()**

Methods:
delayed() **event()** **get_data_ref()** **operator=()**
get_new_value() **get_old_value** **kind()**
negedge() **posedge()**
read() **read(**port_or_signal**)** **write(val)**

Custom ports: **sc_in<**T**>, sc_out<**T**>, sc_inout<**T**>**

sc_fifo

Point-to-point communication, one reader, one writer. Can not be used in **SC_METHOD** process
Implements interface: **sc_fifo_in_if, sc_fifo_out_if**

Methods: **nb_read()** **read()** **read(**port/signal**)**
operator=() **write(**val**)** **nb_write(**val**)**
num_free() **num_available()** **kind()**

Custom ports: **sc_fifo_in<**T**>, sc_fifo_out<**T**>**

Specify fifo depth in **sc_main: sc_fifo<**T**>** f(10); Depth is 10
Specify fifo depth in a module: **SC_CTOR(**module_name): f(10)

sc_mutex

Multipoint communication, used to access a shared resource. Can not be used in **SC_METHOD** process
Implements interface: **sc_mutex_if**

Methods: **kind()** **lock()** **trylock()** **unlock()**

sc_semaphore

Multipoint communication, limited concurrent access. Parameter to assign the maximum number of users. Can not be used in **SC_METHOD** process.
Implements interface: **sc_semaphore_if**

Methods: **kind()** **post()** **trywait()** **wait()** **get_value()**

Number of concurrent users:
In **sc_main:** **sc_semaphore s(4)**
In a module: **SC_CTOR(**module_name**): s(4) { … };**

sc_signal<T>

Multipoint communications, one writer, many readers
Implements interface: **sc_signal_inout_if**

Events:
default_event() **value_changed_event()**

Methods: **delayed()** **read(port/signal)** **event()**
operator=() **get_data_ref()** **kind()**
negedge() **get_new_value()** **write(val)**
read() **get_old_value()** **posedge()**

Custom ports: **sc_in<**T**>, sc_out<**T**>, sc_inout<**T**>**

sc_signal_resolved

Multiple writers
Implements interface: **sc_signal_inout_if**

Events: **default_event()value_changed_event()**

Methods: **delayed()** **read(port/signal)** **event()**
operator=() **get_data_ref()** **kind()**
negedge() **get_new_value()** **write(val)**
read() **get_old_value()** **posedge()**
Resolvetable:

	'0'	'1'	'X'	'Z'
'0'	'0'	'X'	'X'	'0'
'1'	'X'	'X'	'X'	'X'
'X'	'X'	'X'	'X'	'X'
'Z'	'0'	'1'	'X'	'Z'

sc_signal_rv<N>

Similar to sc_signal_resolved, N is the number of bits

Methods used by datatypes and channels

and_reduce( )	Reduction operation "and"-ing each bit
delayed( )	Used in delay-evaluated expressions
event( )	Returns bool, **true** if default_event has triggered in the current timestep
get_bit(i)	Returns long, representing bit value of i^{th} bit (0 = '0', 1 = '1', 2 = 'Z', 3 = 'X')
get_data_ref( )	Get a reference to the current value (for tracing)
get_new_value( )	Returns value after update
get_old_value( )	Returns value before update
get_value( )	Returns int, value of the semaphore
kind( )	Returns channel type (sc_buffer or sc_fifo etc)
length( )	Returns the bit width
lock( )	Blocks until mutex can be locked
nand_reduce( )	Reduction operation "nand"-ing each bit
nb_read()	Non-blocking read, returns bool, **false** if fifo is empty, **true** if read successful
nb_write(val)	Non-blocking write, returns bool, **false** if fifo is full, **true** if write successful
negedge( )	<bool> only, returns true if a **true**-to-**false** transition occurred in the current timestep
nor_reduce( )	Reduction operation "nor"-ing each bit
num_available()	Returns int, number of elements that are in the fifo
num_free()	Returns int, number of remaining elements that can be written before fifo is full
Operator=( )	Assignment operator For convenience, does a write()
or_reduce( )	Reduction operation "or"-ing each bit
posedge( )	<bool> onlv. returns true if a **false**-to-**true**
post( )	Unlock (give) the semaphore
range(x, y)	Refer to a bit range within the object
read( )	Returns current value of signal Blocking for sc_fifo
read(*port_or_signal*)	Returns void, current value of signal is stored in *port_or_signal.* Blocking for sc_fifo
set (i)	Set i^{th} bit to 1
set(i, b)	Set i^{th} bit to b (b is bool, true = 1, false = 0)
set_bit(i, d)	Set i^{th} bit to d (d is long, can be 0, 1, 2, 3, '0', '1', 'X', 'Z')
test(i)	Return true if i^{th} bit is 1, false if 0
to_int( )	Converts a sc_uint to an int
to_string( )	Returns a string representation of vector
to_uint( )	Converts a sc_int to an unsigned int
trylock( )	Nonblocking, returns bool, **true** if lock successful, **false** otherwise
trywait( )	Nonblocking, attempt to lock, returns int, -1 if semaphore is not available
unlock( )	Gives up mutex ownership, returns int, -1 if mutex was not locked by caller
wait( )	Lock (take) the semaphore, block until it is
write(*val*)	Schedules *val* to be written to the signal at the next update phase. Blocking for sc_fifo
xnor_reduce( )	Reduction operation "xnor"-ing each bit
xor_reduce( )	Reduction operation "xor"-ing each bit

Interface Reference:

When creating a custom channel to replace a standard channel and can be bound to standard port you must implement these methods.

Interface	Required Methods
sc_fifo_in_if<T**>**	T **read(), void read(**T&**), bool nb_read(**T&**), int num_available() const**
sc_fifo_out_if<T**>**	**void write(const** T&**), bool nb_write(const** T&**), int num_free() const**
sc_mutex_if	**void lock(), int trylock(), int unlock()**
sc_semaphore_if	**void wait(), int trywait(), void post(), int get_value()**
sc_signal_in_if<T**>**	**void read(**T&**),** T **read(), bool event()**
sc_signal_inout_if<T**>**	**void read(**T&**),** T **read(), bool event(), void write(const** T&**)**

Master-Slave Library:

Master Ports:
sc_master<> port_name [, port_name, …];
sc_inmaster<T**>** port_name [, port_name,…];
sc_outmaster<T**>** port_named [, port_name,…];
sc_inoutmaster<T**>** port_name [, port_name,…]**];**

Indexed Form
sc_master<sc_indexed<N**> >** port_name [, port_name, …];
sc_inmaster<T**, sc_indexed<**N**> >** port_name [, port_name,…];
sc_outmaster<T**, sc_indexed<**N**> >** port_named [, port_name,…];
sc_inoutmaster<T**, sc_indexed<**N**> >** port_name [, port_name,…]**];**

Methods**: read(), write(), *master_port_name*()**

Slave Ports:
sc_slave<> port_name [, port_name, …];
sc_inslave<T**>** port_name [, port_name,…];
sc_outslave<T**>** port_named [, port_name,…];
sc_inoutslave<T**>** port_name [, port_name,…]**];**

Indexed Form
sc_slave<sc_indexed<N**> >** port_name [, port_name, …];
sc_inslave<T**, sc_indexed<**N**> >** port_name [, port_name,…];
sc_outslave<T**, sc_indexed<**N**> >** port_named [, port_name,…];
sc_inoutslave<T**, sc_indexed<**N**> >** port_name [, port_name,…]**];**

Methods: **read(), write(), input(), get_address()** (for indexed slaves)

Refined ports:
sc_master<protocol**>** port_name [, port_name, …];
sc_inmaster<T, protocol **>** port_name [, port_name,…];
sc_outmaster<T, protocol **>** port_named [, port_name,…];
sc_inoutmaster<T, protocol **>** port_name [, port_name,…]**];**
sc_slave< protocol **>** port_name [, port_name, …];
sc_inslave<T, protocol **>** port_name [, port_name,…];
sc_outslave<T, protocol **>** port_named [, port_name,…];
sc_inoutslave<T, protocol **>** port_name [, port_name,…]**];**

Where protocol is one of**:**
sc_noHandshake<T> **sc_enable_Handshake<T>**
sc_fullHandshake<T> **sc_memenHandshake<T>**
sc_memfullHandshake<T>

T must match the type T specified in the port

sc_link_mp<T>

Channel to connect master en slave ports. Type must be the same as the master and slave ports.