Praktikum: Entwicklung interaktiver eingebetteter Systeme

SystemC Tutorial

Joachim Falk (falk@cs.fau.de)
Agenda

- Motivation
- Brief SystemC Overview
- The SystemC Simulation Kernel
- Modules and Channels
- Hierarchy
- Predefined Channels and Ports
- Processes
- Data Types
Motivation

SystemC

Best

OK

Good

module

OK

Best

system

architecture

Best

Good

behavior

OK

block

SW

HW

logic

structure
Motivation

➢ Current industry practice

- System specification in C or C++
- Hardware (HDL)
- Software (C, C++)
- Testbench
Motivation

- SystemC design flow
SystemC Model Nomenclature

Agenda

- Motivation
- Brief SystemC Overview
- The SystemC Simulation Kernel
- Modules and Channels
- Hierarchy
- Predefined Channels and Ports
- Processes
- Data Types
From C++ to SystemC

- **C++ has**
  - sequential program execution
  - Object Oriented design

- **SystemC has**
  - discrete event simulation by the SystemC simulation kernel.
  - modules communicating over channels
  - concurrently executed processes implementing cooperative multitasking
  - additional data types for bitvectors, fix point arithmetic, etc.

- We use OSCI SystemC
  An open source implementation of SystemC.
  http://www.systemc.org
SystemC Language Architecture

- Has a layered approach.
- Lower layers are independent from upper layers.
- Enables application specific library development on top of SystemC.

<table>
<thead>
<tr>
<th>Application Spec. Libraries</th>
<th>Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elementary Channels</td>
<td>Arbitrary prec. integers</td>
</tr>
<tr>
<td>Signal, Buffer</td>
<td>2 valued logic</td>
</tr>
<tr>
<td>FIFO</td>
<td>4 valued logic</td>
</tr>
<tr>
<td>Mutex, Semaphore</td>
<td>Fixed-point numbers</td>
</tr>
<tr>
<td>Core Language</td>
<td>Master &amp; Slave Library</td>
</tr>
<tr>
<td>Modules</td>
<td>Verification Library</td>
</tr>
<tr>
<td>Ports</td>
<td>TLM Library</td>
</tr>
<tr>
<td>Interfaces</td>
<td></td>
</tr>
<tr>
<td>Channels</td>
<td></td>
</tr>
<tr>
<td>Processes</td>
<td></td>
</tr>
<tr>
<td>Events</td>
<td></td>
</tr>
<tr>
<td>ANSI C++ Language Features</td>
<td></td>
</tr>
</tbody>
</table>

Friedrich-Alexander-Universität Erlangen-Nürnberg
Joachim Falk
Agenda

- Motivation
- Brief SystemC Overview
- The SystemC Simulation Kernel
- Modules and Channels
- Hierarchy
- Predefined Channels and Ports
- Processes
- Data Types
The SystemC Simulation Kernel

- SystemC is a discrete event (DE) simulator. The simulator performs computation if an event occurs. This can generate new events in the future. When there are no more events in the system the simulation halts.

- An event \( e_i \) can occur before, after or concurrent to other events.

- Each event \( e_i \) is associated with its occurrence time \( t(e_i, t) \).

- The SystemC simulator provides a cooperative multitasking environment.
The SystemC Simulation Kernel

SystemC Events

// Somewhere in your code
void module2::amethod2() {
    // Do something
}

// Somewhere in your code
void module2::amethod1() {
    ev5.notify(SC_ZERO_TIME);
    ev4.notify(10, SC_NS);
}

Friedrich-Alexander-Universität Erlangen-Nürnberg
Joachim Falk
The SystemC Simulation Kernel

```c
int sc_main(
    int argc, char *argv[])
{
    // Elaboration stuff
    sc_start(); // call it
    ...
}
```

Find next event and execute code.

```c
void module2::amethod1() {
    ev5.notify(SC_ZERO_TIME);
    ev4.notify(10, SC_NS);
}
```

```c
void module2::amethod2() {
    wait(ev5);
}
```

```
// Somewhere in your code
void module2::amethod2() {
    wait(ev5);
}
```

SystemC Events

```
sc_event _startup
sc_event
```

Friedrich-Alexander-Universität Erlangen-Nürnberg
Joachim Falk
The SystemC Simulation Kernel

Elaborate → sc_start → Simulation

```c
int sc_main(
    int argc, char *argv[])
{
    // Elaboration stuff
    sc_start(); // call it
    ...
}
```

```
void module2::amethod1()
{
    ev5.notify(SC_ZERO_TIME);
    ev4.notify(10, SC_NS);
}
```

```
void module2::amethod2()
{
    wait(ev5);
}
```

```
void module1::amethod()
{
    // Do something
}
```

```
// Somewhere in your code
```

SystemC Events

```
sc_event ev4
sc_event ev5
```

```
0ns,0 0ns,1 10ns,0
```

Find next event and execute code.
The SystemC Simulation Kernel

Elaborate

int sc_main(
    int argc, char *argv[])
{
    // Elaboration stuff
    sc_start(); // call it
    ...
}

Find next event and execute code.

sc_start

Simulation

// Somewhere in your code
void module2::amethod2() {
    wait(ev5);
}

// Somewhere in your code
void module1::amethod() {
    wait(ev4);
}

Elaborate

// Elaborate

int sc_main(
    int argc, char *argv[])
{
    // Elaboration stuff
    sc_start(); // call it
    ...
}

// Somewhere in your code
void module2::amethod2() {
    wait(ev5);
}

// Somewhere in your code
void module1::amethod() {
    wait(ev4);
}

0ns,0 0ns,1 10ns,0
The SystemC Simulation Kernel

```c
int sc_main(
    int argc, char *argv[])
{
    // Elaboration stuff
    sc_start(); // call it
    ...
}
```

Find next event and execute code.

```c
sc_start
```

Simulation

```c
// Somewhere in your code
void module1::amethod() {
    wait(ev4);
}
```

```c
// Somewhere in your code
void module2::amethod2() {
    wait(ev5);
}
```

```c
int sc_main(
    int argc, char *argv[])
{
    // Elaboration stuff
    sc_start(); // sc_start returns
    // Do cleanup here
    ...
}
```
Time in SystemC

- is represented by the `sc_time` class.

  Syntax:

  ```
  sc_time t(v, u);
  ```

  - `v`: Integer or double representing the numerical time value.
  - `u`: Specifies the unit of the numerical time value `v`.

- can be specified in various time units.
  - `SC_FS`: $10^{-15}$ seconds
  - `SC_PS`: $10^{-12}$ seconds
  - `SC_NS`: $10^{-9}$ seconds
  - `SC_US`: $10^{-6}$ seconds
  - `SC_MS`: $10^{-3}$ seconds
  - `SC_SEC`: 1 second

- has arithmetic and comparison operators defined.
- has a stream output (operator `<<`) which prints the time.
- can be queried via `sc_time_stamp()`.
// Example 1:
sc_time tm1(10, SC_NS); // time object representing 10ns
sc_time tm2(10, SC_MS); // time object representing 10ms

// Example 2:
sc_time tm3(2*tm1); // time object representing 20ns
sc_time tm4(42); // time object representing 42 default time units

// Example 3:
sc_time tm5(5, SC_MS);
cout << tm5 << endl; // output: 5 ms

// Example 3:
cout << sc_time_stamp() << endl; // output the current simulation time.
Events

- are implemented by the `sc_event` class.

**Syntax:**
```cpp
sc_event ev;
```

- have no value and no duration.
- may be created.

```cpp
sc_event ev1;  // create an event
cs_event ev2;  // create another event
```

- are neither copyable nor assignable.

```cpp
ev1 = ev2;    // Compile ERROR (events are not assignable)
cs_event ev3(ev1);  // Compile ERROR (events are not copyable)
```
Events – Notification Syntax

- An event has to be notified in order to get triggered.
- Notification may occur anywhere in the code.
- Notification is done by the `notify()` method.

**Syntax:**
```
ev.notify();
ev.notify(SC_ZERO_TIME);
ev.notify(t); ev.notify(v, u);
```

- `v`: Integer or double representing the numerical time value.
- `u`: Specifies the unit of the time value `v`, e.g., `SC_NS`.
- `t`: An object of type `sc_time`.

- A notified event may be cancelled.
- Cancellation may occur anywhere in the code.
- Cancellation is done by `cancel()` method.

**Syntax:**
```
ev.cancel();
```
Events – Notification Semantics

- Event notification can be distinguished into:
  - immediate notification, i.e., in this delta cycle, `ev1.notify()`
  - notification in the next delta cycle, `ev2.notify(SC_ZERO_TIME)`
  - and timed notification.
    `ev3.notify(sc_time(v,u)); ev3.notify(v, u)`

- If notify is called more than once:
  - triggering can be advanced
    `ev3.notify(sc_time(v-10,u))`
  - but not delayed.
    `ev2.notify(sc_time(v,u))`

- Only events at least one delta cycle away can be canceled.
  `ev1.cancel(); ev2.cancel();`
Event Lists

- Event lists can be used instead of events in calls to `wait` and `next_trigger`.
- Event lists can be distinguished into
  - event or lists (`sc_event_or_list`) which are triggered if at least one event in the or list has been triggered.
  - Syntax:
    ```
    sc_event ev1, ev2, ...
    sc_event_or_list &eo = ev1 | ev2 | ...;
    ev1, ev2, ...: Objects of type `sc_event`.
    ```
  - event and lists (`sc_event_and_list`) which are triggered if all events in the and list have been triggered.
  - Syntax:
    ```
    sc_event ev1, ev2, ...
    sc_event_and_list &ea = ev1 & ev2 & ...;
    ev1, ev2, ...: Objects of type `sc_event`.
    ```
Event Lists

Caution: Event lists are dynamically allocated by “operator |” and “operator &” and will be deallocated by the SystemC simulation kernel after they have been triggered.

// Example (assuming the context of a thread process)
wait(ev1 | ev2); // Ok wait till either event ev1 or ev2 is triggered.
wait(ev1 & ev2); // Ok wait till both events have been triggered.
{
    sc_event_and_list &ea = ev1 & ev2;
    wait(ea); // This call works but after it ea will have been freed.
    wait(ea); // SIGSEGV
}
{
    sc_event_or_list &eo = ev1 | ev2 | ev3;
    if (flag)
        wait(eo);
    //else
    //  memory leak
}

// Example (assuming the context of a method process)
// Ok trigger process again if
// either event ev1 or ev2 is triggered
// or if 30 NS have expired.
next_trigger(30, SC_NS, ev1 | ev2);
The SystemC Simulation Kernel

- sc_main
- Simulator
- Process1 (SC_METHOD)
- Process2 (SC_THREAD)

Start the simulation (sc_start())

Startup

Simulation

Simulationsende

return;

wait();

return;

wait();

wait();
Agenda

- Motivation
- Brief SystemC Overview
- The SystemC Simulation Kernel
- Modules and Channels
- Hierarchy
- Predefined Channels and Ports
- Processes
- Data Types
Modules and Channels

- SystemC has modules communicating over channels. (Implements separation of behavior and communication)
  - Behavior is implemented in modules.
  - Communication is implemented in channels.

- SystemC has concurrently executed processes implementing cooperative multitasking.
Module Syntax

- Modules are implemented by C++ classes
  - which must have base class `sc_module`.
  - which can also be defined via the `SC_MODULE` macro.

Syntax:

```cpp
SC_MODULE(module) {
    // the class body
};
```

use at implementation level

```cpp
struct module : public sc_module {
    // the class body
};
```

equivalent

use at higher levels of abstraction

```cpp
// Somewhere in SystemC
#define SC_MODULE(user_module_name) \n    struct user_module_name: ::sc_core::sc_module
```

Friedrich-Alexander-Universität Erlangen-Nürnberg
Joachim Falk
Channel Syntax

- Two types of channels:
  - Primitive channels which
    - are derived from base class `sc_prim_channel`.
    - have no visible internal structure.
    - do not contain processes.
    - cannot directly access other channels.
  - Hierarchical channels which
    - are derived from base class `sc_channel`.
    - are modules that implement interfaces.
    - may contain channels, modules, processes.

Syntax:

```cpp
class channel
  : public sc_prim_channel,
    public channel_if_1,
    public channel_if_2,
...
{ // implement interfaces }
```

```cpp
class hir_channel
  : public sc_channel,
    public channel_if_1,
    public channel_if_2,
...
{ // implement interfaces }
```

As `sc_channel` is a typedef for `sc_module`, there is no clear distinction between a hierarchical channel and a behavior (module), as both derive from the same base class `sc_module`. 

// Somewhere in SystemC
namespace sc_core {
  typedef sc_module sc_channel;
} // namespace sc_core
Encapsulation of Behavior

- A module communicates through its interface
  - which is defined by its ports.
  - which allows direct mapping to routing resources at implementation level.
  - which enables “plug&play” for modules.
  - which allows altered implementations of the module functionality (refinement, alternative).

```cpp
class module_a : public sc_module
{
    void foo() {modb.bar();}
};

class module_b : public sc_module
{
    void bar();
};

// somewhere in your code
...
module_a moda("moda");
module_b modb("modb");
...
Modules and Channels

- SystemC has separation of interface and implementation.
  - A SystemC module has ports.
  - A ports access an interface.
  - A channels implements an interface.

```cpp
class module_a : public sc_module
{
    sc_port<channel_if_1> p;
    ...
    void foo() {
        // use interface
        p->iface_1_method();
    }
    ...
};

class channel : public sc_prim_channel,
                public channel_if_1,
                public channel_if_2
{
    ...
    // implement interfaces
    void iface1_method();
    void iface2_method();
    ...
};

class module_b : public sc_module
{
    sc_port<channel_if_2> p;
    ...
    void foo() {
        // use interface
        p->iface_2_method();
    }
    ...
};
```
Channel Interfaces

- Interfaces are derived virtual from `sc_interface`.
- Methods which are implemented by the channels are declared **pure virtual**.
- Channels derive from all interfaces they want to implement.
- Ports `sc_port<T>` with a given interface `T` may bind to a channel if it provides interface `T`.
  (In fact the channel may provide more interfaces than `T` only.)

```cpp
class channel_if_1 : public virtual sc_interface
{
    // declare pure virtual
    virtual void iface1_method() = 0;
};

class channel_if_2 : public virtual sc_interface
{
    // declare pure virtual
    virtual void iface2_method() = 0;
};

class channel : public sc_prim_channel,
                public channel_if_1,
                public channel_if_2
{
    ... // implement interfaces
    void iface1_method();
    void iface2_method();
    ...}
};
```
Module Syntax - Constructor

- Each module has a constructor
  - which must have a parameter `sc_module_module`
  - which can also be defined via the `SC_CTOR` macro.
  - which is called at the instantiation of the module to
    - initialize all data members of the module to a known state.
    - initialize output ports for HW signals.
    - register processes (more later).
    - instantiate sub-modules (more later).

```c
// Somewhere in SystemC
#define SC_HAS_PROCESS(user_module_name) \
  typedef user_module_name SC_CURRENT_USER_MODULE
#define SC_CTOR(user_module_name) \
  typedef user_module_name SC_CURRENT_USER_MODULE; \
  user_module_name( ::sc_core::sc_module_name )

...  
  SC_CTOR(module) {   
    // constructor body
  }
...  
};
```

```c
SC_MODULE(module) {  
  SC_HAS_PROCESS(module);  
  module(sc_module_name n) : sc_module(n) {  
    // constructor body
  }  
  ...
};
```

**equivalent**

**use at implementation level**

**use at higher levels of abstraction**
Constructor - Syntax Example 1

SC_MODULE(adder) {
    // Input ports
    // Output ports
    SC_CTOR(adder) {
        // body of constructor
    }
};

#include "adder.h"

adder add1("add1");

SC_CTOR automatically generates a constructor taking an arbitrary string (the instance name) as an argument.
Constructor - Syntax Example 2

// Header file adder.h
class adder
 : public sc_module {
public:
    // Input ports
    // Output ports
    SC_HAS_PROCESS(adder);

    // now without SC_CTOR macro
    adder(sc_module_name nm)
    : sc_module(nm)
    {
        /* body of constructor */
    }

    // cannot be done with SC_CTOR!
    adder(sc_module_name nm, int delay)
    : sc_module(nm)
    {
        /* body of constructor */
    }
};

// somewhere in your code
#include "adder.h"
// instantiating some adders
adder add1("add1");
adder add2("add2", 2);

Arbitrary string argument is used for run-time error messages by the simulation kernel. It is good practice to use the instance name as the argument!
Agenda

- Motivation
- Brief SystemC Overview
- The SystemC Simulation Kernel
- Modules and Channels
- Hierarchy
- Predefined Channels and Ports
- Processes
- Data Types
Hierarchy - Overview

- SystemC modules
  - may contain instances of other modules.
  - may connect ports of contained modules to their own ports.
  - may connect ports of contained modules via channels.

```cpp
class hir_module: public sc_module {
  sc_port<channel_if> p1, p2;
  sub_module moda, modb;
  channel chan;
  hir_module(sc_module_name name): sc_module(name)
    , moda("moda"), modb("modb")
  {
    moda.pa(p1); modb.pb(p2);
    modb.pa(chan); modb.pa(chan);
  }
};

class sub_module : public sc_module {
  sc_port<channel_if> pa;
  sc_port<channel_if> pb;
  ...
  SC_CTOR(sub_module) {
    // constructor body
  }
  ...
};
```
Hierarchical Module Name

SystemC constructs a hierarchical module name out of the `sc_module_name` parameter of the various SystemC modules.

Then this is "hir.moda". Assume this is called "hir". And then this is "hir.modb".

Even channels may be named "hir.chan".

class hir_module: public sc_module {
    sc_port<channel_if> p1, p2;
    sub_module moda, modb;
    channel chan;
    hir_module(sc_module_name name): sc_module(name)
        , moda("moda"), modb("modb"), chan("chan")
    {
        moda.pa(p1); modb.pb(p2);
        moda.pb(chan); modb.pa(chan);
    }
};

class sub_module : public sc_module
{
    sc_port<channel_if> pa;
    sc_port<channel_if> pb;
    ...
    SC_CTOR(sub_module) {
        // constructor body
    }
    ...
};
Ports

- have to be defined inside a module with **public** visibility.
- are of type `sc_port<\text{class } T, \text{int } N=1>`.
- have their interface specified via template parameter `T`.
- can have a multiplicity specified via template parameter `N`:
  - which is an integer $\geq 0$.
  - which specifies the number of channels that may maximally be connected with that port. (use `p[n]` to access the ports)
  - where $N=0$ specifies an unlimited number of channels which can be connected to the port.

Note that some special ports derived from `sc_port` exist. These ports are for some channels defined by SystemC (more later).

**Syntax:**

```c++
SC_MODULE(my_mod) {
public:
  ...
  // Port are class vars
  sc_port<T, N> p;
  ...
};
```
Port Mapping

- may be done by position
  - which is the shortest possibility
  - but not recommended, because it is error prone.
- may be done by name
  - which needs some typing effort
  - but is recommended, because it is easy to debug and modify.
- may connect
  - ports with channels.
  - ports with ports.

Mapping by position syntax:

```plaintext
instance_name(channel_name[, channel_name[, ...]]);
instance_name << channel_name [<< channel_name [<< ...]];
```

Mapping by name syntax:

```plaintext
instance_name.port_name(channel_name);
[instance_name.port_name(channel_name);]
```
Agenda

- Motivation
- Brief SystemC Overview
- The SystemC Simulation Kernel
- Modules and Channels
- Hierarchy
- Predefined Channels and Ports
- Processes
- Data Types
Predefined Primitive Channels

- SystemC 2.0 offers several predefined primitive channels
  - `sc_fifo`:
    A simple deterministic fifo channel
  - `sc_mutex`:
    A mutual exclusion lock to prevent multiple processes from simultaneous access to a resource.
  - `sc_semaphore`:
    A semaphore which limits the number of simultaneous accesses to a resource.
  - `sc_signal`:
    A hardware signal, i.e., a wire.
  - `sc_buffer`:
    A hardware signal but generates an event for each write, i.e., even if the value is unchanged.
FIFOs - Definition

- FIFOs are defined inside a module or at top level.
- FIFOs are implemented by class `sc_fifo<T>`.

**Syntax:**

```cpp
sc_fifo<T> fifo_name(depth = 16);
```

- **depth:** optional, default 16.
  - Encodes the number of items which can be stored in the FIFO.
- **T:** The type of data stored in the FIFO.

- The data type is passed as template parameter T and
  - can be any C/C++ built-in data type,
  - any SystemC data type,
  - or a user defined data type.

```cpp
sc_fifo<bool> a; // a FIFO for boolean values (size 16)
sc_fifo<sc_int<8>> b(10); // a FIFO for signed 8bit integers (size 10)
sc_fifo<my_type> c(42); // a FIFO which stores 42 items of type my_type
```

A space is required by the C++ compiler.
Ports for FIFOs - Interfaces

- SystemC provides interfaces for FIFOs
  - The direction is determined by the interface type.
    - Input: `sc_fifo_in_if<T>`
    - Output: `sc_fifo_out_if<T>`
  - The data type is given as template parameter \( T \).

---

**Declaration as data members of the module:**

```cpp
SC_MODULE(my_mod) {
  sc_port<sc_fifo_in_if<T>> port_name_1;
  sc_port<sc_fifo_out_if<T>> port_name_2;
};
```

\( T \): The type of data stored in the connected FIFO.

---

```cpp
sc_port<sc_fifo_in_if<int>> in;  // input port of type int
sc_port<sc_fifo_out_if<bool>> out;  // output port of type bool
int i = in->read();
out->write(i);
```
SystemC provides special port types for FIFOs

- The direction is determined by port type.
  - Input: `sc_fifo_in< T >`
  - Output: `sc_fifo_out< T >`
- The data type is given as template parameter `T`.

**Declaration as data members of the module:**

```c
SC_MODULE (my_mod) {
    sc_fifo_in< T > port_name_1;
    sc_fifo_out< T > port_name_2;
};
```

`T`: The type of data stored in the connected FIFO.

**All interface methods are mirrored on the port.**

```c
sc_fifo_in<int> in; // input port of type int
sc_fifo_out<bool> out; // output port of type bool
int i = in.read();   // equivalent to int i = in->read();
out.write(i);         // equivalent to out->write(i);
```
# FIFOs - Interfaces

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void read(T&amp;)</td>
<td>blocking read. Blocks until at least one sample is stored in the FIFO</td>
</tr>
<tr>
<td>T read()</td>
<td></td>
</tr>
<tr>
<td>bool nb_read(T&amp;)</td>
<td>non-blocking read. Returns true if a sample was available</td>
</tr>
<tr>
<td>int num_available()</td>
<td>number of samples in FIFO</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void write(const T&amp;)</td>
<td>blocking write. Blocks until at least one free space is available</td>
</tr>
<tr>
<td>bool nb_write(const T&amp;)</td>
<td>non-blocking write. Returns true if sample could be written</td>
</tr>
<tr>
<td>int num_free()</td>
<td>number of free spaces in FIFO</td>
</tr>
</tbody>
</table>
HW Signals - Definition

- HW signals are defined inside a module or at top level.
- HW signals are implemented by class `sc_signal<T>`.

**Syntax:**

```cpp
sc_signal<T> signal_name;
```

**T:** The type of data transmitted by the signal.

- The data type is passed as template parameter T and can be:
  - any C/C++ built-in data type,
  - any SystemC data type,
  - or a user defined data type.

```cpp
sc_signal<bool> a; // a boolean signal
sc_signal<sc_int<8> > b; // a signal for 8bit signed integers
sc_signal<my_type> c; // a signal of user defined type
```

A space is required by the C++ compiler.
Ports for HW Signals - Interfaces

- SystemC provides interfaces for HW signals
  - The direction is determined by the interface type.
    - Input: `sc_signal_in_if<T>`
    - Output: `sc_signal_out_if<T>`
    - Input and output: `sc_signal_inout_if<T>`
  - The data type is given as template parameter `T`.

**Declaration as data members of the module:**

```c++
SC_MODULE(my_mod) {
    sc_port<sc_signal_in_if<T>> p1;
    sc_port<sc_signal_out_if<T>> p2;
    sc_port<sc_signal_inout_if<T>> p3;
};
```

`T`: The type of data transmitted by the connected HW signal

```c++
sc_port<sc_signal_in_if<int>> in;  // input port of type int
sc_port<sc_signal_out_if<bool>> out;  // output port of type bool
int i = in->read();
out->write(i);
```
SystemC provides special port types for HW signals

- The direction is determined by port type.
  - Input: `sc_in<T>`
  - Output: `sc_out<T>`
  - Input and output: `sc_inout<T>`

- The data type is given as template parameter `T`.

### Declaration as data members of the module:

```cpp
SC_MODULE(my_mod) {
  sc_in<T> p1;
  sc_out<T> p2;
  Sc_inout<T> p3;
};
```

`T`: The type of data transmitted by the connected HW signal

All interface methods are mirrored on the port.

```cpp
sc_in<int> in; // input port of type int
sc_out<bool> out; // output port of type bool
int i = in.read(); // equivalent to int i = in->read();
out.write(i); // equivalent to out->write(i);
```
## HW Signals - Interfaces

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>const T&amp; read()</td>
<td>read the current value from the signal</td>
</tr>
<tr>
<td>bool event()</td>
<td>return true if the value of the signal has changed</td>
</tr>
</tbody>
</table>

**sc_signal_out/inout_if<T> additionally has**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void write(const T&amp;)</td>
<td>write a value to the signal</td>
</tr>
</tbody>
</table>

**sc_signal_in/out/inout_if<bool/sc_logic> adds**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool posedge()</td>
<td>returns true on a positive edge (change from 0 to 1)</td>
</tr>
<tr>
<td>bool negedge()</td>
<td>returns true on a negative edge (change from 1 to 0)</td>
</tr>
</tbody>
</table>
HW Signals - Write Semantics

- Writing a port or signal within a process
  - assigns the value to the port/signal
  - value not immediately visible
    - scheduled for update by simulation kernel
    - update occurs if process is finished or suspended (more later)
  - consecutive writes overwrite the old port/signal value
    - even if value has never been read
    - if multiple writes to the same port/signal occur in the same process, only the value of the last write will be visible

```cpp
// Example (assuming the context of a process)

// write the value 10 to signal sig
sig.write(10);
// write 11 to sig. The value 10 will never be visible outside
// the process, because it is overwritten by 11
sig.write(11);
```
HW Signals - Event Semantics

- Checking for events on a port/signal within a process
  - `event()` method to check for an event on a port or signal
    - Returns **true** if value has changed
    - Returns **false** if value has not changed
  - `posedge()` and `negedge()` check for edges
    - Only with ports/signals of type `bool` or `sc_logic`
    - Returns **true** if positive/negative edge detected
    - Returns **false** otherwise

```cpp
sc_signal<bool> reset;
if(reset.event() && reset.read())
  cout << "reset" << endl;
```

Equivalent to:

```cpp
sc_signal<bool> reset;
if(reset.posedge())
  cout << "reset" << endl;
```
HW Signals - Event Semantics

- Getting a `sc_event` for various conditions from a signal
  - `default_event()`/`value_changed_event()` checks for changes
    - returns an event that is notified if the value of the signal changes
  - `posedge_event()` and `negedge_event()` checks for edges
    - only with signals of type `bool` or `sc_logic`
    - returns an event which is triggered on `positive/negative` edge

- Getting a `sc_event` for various conditions from a port
  - `value_changed()` checks for changes
    - returns an event that is notified if the value of the port changes
  - `pos()` and `neg()` checks for edges
    - only with ports of type `bool` or `sc_logic`
    - returns an event which is triggered on `positive/negative` edge
Agenda

- Motivation
- Brief SystemC Overview
- The SystemC Simulation Kernel
- Modules and Channels
- Hierarchy
- Predefined Channels and Ports
- Processes
- Data Types
Processes and Events

- A mechanism to react on events is needed.
- Although an event may be notified from anywhere in the code, only a process may react on an event!
  - waiting for an event in the sensitivity list (called static sensitivity)
  - waiting for an event directly within the process (called dynamic sensitivity)

notify event

react on event

to make use of events a mechanism to react on events is needed!
Processes

- are member functions of the module
  - return type `void`
  - no arguments
- are registered inside the constructor
- can have one of three types
  - `SC_METHOD`
  - `SC_THREAD`
  - `SC_CTHREAD`
- can have a sensitivity list
  - contains channels and events that activate the process

```cpp
SC_MODULE(my_module) {
    sc_in<bool> clk;
    void proc_method();
    void proc_thread();
    void proc_cthread();

    SC_CTOR(my_module) {
        SC_METHOD(proc_method);
        sensitive << clk.pos();
        SC_THREAD(proc_thread);
        sensitive << clk.pos();
        SC_CTHREAD(proc_cthread, clk.pos());
    }
}
```
Processes and Events

- Process describe concurrent execution semantics
- Process are activated by events
  - static sensitivity (sensitivity list)
  - dynamic sensitivity (discussed later)
- Process can be distinguished into two classes
  - method processes
  - thread processes
Proc. Registration – Syntax

➢ To make a member function of a module a process
  ▪ the member function
    - must not take any arguments
    - must have return type void
  ▪ the member function must be registered via the MACROS
    - SC_METHOD
    - SC_THREAD
    - SC_CTHREAD

Syntax:

```c++
SC_CTOR(module_name) {
    SC_METHOD(method_name);
    SC_THREAD(thread_name);
    SC_CTHREAD(cthread_name, edge);
}
```

edge: An edge of a bit signal, e.g., signal.pos() or signal.neg()
Proc. Initialization – Overview

➢ Before starting a simulation
  ▪ all `SC_METHOD` processes are executed once
  ▪ all `SC_THREAD` processes are executed until the first wait statement is reached
  ▪ to avoid this, use `dont_initialize()`
    - used directly after the process registration
    - used only with `SC_METHOD` and `SC_THREAD`

`Syntax:`

```cpp
SC_CTOR(module_name) {
    SC_METHOD(method_name);
    dont_initialize();
    SC_THREAD(thread_name);
    dont_initialize();
}
```

➢ `SC_CTHREAD` process are not initialized by the simulation kernel
➢ especially useful with events (discussed later)
➢ not recommended at implementation level
Proc. Sensitivity – Syntax

- Specify a sensitivity list directly behind the process registration.
- This is only possible for
  - SC_METHOD processes.
  - SC_THREAD processes.

**Syntax:**

```plaintext
sensitive (chan|event|edge);
sensitive << chan|event|edge [<< chan|event|edge ...];
```

- **chan:** An object of type `sc_interface` or `sc_port`.
- **event:** An object of type `sc_event`.
- **edge:** An edge of a bit signal, e.g., `signal.pos()` or `signal.neg()`.
SC_METHOD - Characteristics

- Method Process **SC_METHOD**
  - is sensitive to a set of signals (called sensitivity list)
    - May be sensitive to any change on a signal.
    - May be sensitive to the positive or negative edge of a boolean signal.
  - is invoked whenever any of the inputs it is sensitive to changes.
  - Once an **SC_METHOD** process is invoked:
    - Entire body of the block is executed.
    - Instructions are executed infinitely fast (in terms of internal simulation time).
SC_METHOD - Characteristics

- Method Process SC_METHOD
  - **CANNOT** be suspended, that is no **wait** statements allowed (a **wait** statement suspends the process).
  - **May NOT** contain an infinite loop (**Causes a dead lock**)!
  - The complete code is executed at each invocation.
  - Local variables are redefined each time the block is invoked.
  - Need to save the state of the block in member variables.
Thread Process **SC_THREAD**

- is sensitive to a set of signals (called sensitivity list).
  - May be sensitive to any change on a signal.
  - May be sensitive to the positive or negative edge of a boolean signal.
- is reactivated whenever any of the inputs it is sensitive to changes.
- Once a **SC_THREAD** process is reactivated
  - the code is executed until the next occurrence of a `wait` statement. The next time the process is reactivated execution will continue after the `wait statement` which caused the suspension.
  - the instructions are executed infinitely fast. (in terms of internal simulation time)
SC_THREAD - Characteristics

- Thread Process SC_THREAD
  - MUST contain an infinite loop!
  - MUST have a wait statement inside the infinite loop (Otherwise a dead lock is generated)!
  - invoked only once at simulation start-up
  - the state of all local variables of a thread process at suspension time will be restored after the next reactivation
Thread Process **SC_CTHREAD**:  
- is sensitive to a clock edge.  
- is reactivated whenever the clock has the specified edge, e.g., rising edge if `clk.pos()` was specified or falling edge if `clk.neg()` was specified as second parameter of `SC_CTHREAD`.  
- Once a **SC_CTHREAD** process is reactivated  
  - the code is executed until the next occurrence of a `wait` statement. The next time the process is reactivated execution will continue after the `wait` statement which caused the suspension.  
  - the instructions are executed infinitely fast. (in terms of internal simulation time)
SC_CTHREAD - Characteristics

➢ Thread Process SC_CTHREAD:
  ▪ Synthesis MUST have a reset signal specified via
    reset_signal_is(<signal>, <level>);
    - <signal> specifies the signal to use for reset
    - <level> is a boolean; true => active high reset, false => active low
  ▪ MUST contain an infinite loop!
  ▪ MUST have a wait statement inside the infinite loop
    (Otherwise a dead lock is generated)!
  ▪ invoked only once at simulation start-up
  ▪ the state of all local variables of a thread process at suspension time will be restored after the next reactivation
Dynamic Sensitivity

- Dynamic Sensitivity is a means to directly react on events.
  - Is supported within `SC_THREAD` processes by `wait`.
  - Is supported within `SC_METHOD` processes by `next_trigger`.
  - Is not supported within `SC_CTHREAD` processes which are only sensitive to a clock edge.

- Dynamic vs. static sensitivity
  - **Static sensitivity** is declared only once for the complete process in the module constructor using the `sensitivity list`.
  - **Dynamic sensitivity** may be changed during process execution.
  - **Dynamic sensitivity** temporarily overrides static sensitivity.
SystemC Threads – wait

Given an integer \( n \), an sc_event \( ev \), a sc_time \( t \), a time value \( v \) and its unit \( u \):

- **The wait() method is allowed in**
  - SC_THREAD processes, where it suspends execution of the process till at least one event in the sensitivity list is triggered.
  - SC_CTHREAD processes, where it suspends execution of the process for one clock cycle.

- **The wait(n) method is allowed in**
  - SC_CTHREAD processes, where it suspends execution of the process for \( n \) clock cycles.

- **The wait(ev) method is allowed in**
  - SC_THREAD processes, where it suspends execution of the process till event \( ev \) is triggered.

- **The wait(t) (wait(v, u)) method is allowed in**
  - SC_THREAD processes, where it suspends execution of the process for a time period \( t \) (sc_time \( (v, u) \)).

- **The wait(t, ev) (wait(v, u , ev)) method is allowed in**
  - SC_THREAD processes, where it suspends execution of the process till event \( ev \) is triggered or time period \( t \) (sc_time \( (v, u) \)) has expired, whichever comes first.
SystemC Methods – next_trigger

- Given an sc_event \textit{ev}, a sc_time \textit{t}, a time value \textit{v} and its unit \textit{u}:
  - The \textit{next_trigger()} method is allowed in 
    - SC_METHOD processes, where it overrides the previous \textit{next_trigger} call in the method to \textit{switch back to static sensitivity}.
  - The \textit{next_trigger(\textit{ev})} method is allowed in 
    - SC_METHOD processes, where it ensures that the process will be triggered again exactly when the event \textit{ev} is triggered.
  - The \textit{next_trigger(\textit{t})(\textit{next_trigger(v, u))}} method is allowed in 
    - SC_METHOD processes, where it ensures that the process will be triggered again exactly after a delay of time period \textit{t} (\textit{sc_time(v, u)}).
  - The \textit{next_trigger(\textit{t, ev})(\textit{next_trigger(v, u , ev))}} method is allowed in 
    - SC_METHOD processes, where it ensures that the process will be triggered again exactly when the event \textit{ev} is triggered or when a delay of time period \textit{t} (\textit{sc_time(v, u)}) has expired, whichever comes first.

- If \textit{next_trigger} is not called in an invocation of a 
  - SC_METHOD process the process will \textit{switch back to static sensitivity}. 
Agenda

- Motivation
- Brief SystemC Overview
- The SystemC Simulation Kernel
- Modules and Channels
- Hierarchy
- Predefined Channels and Ports
- Processes
- Data Types
Data Types - Overview

- **C++ data types**
  - **Compiler builtin**
    - (signed) char, short, int, long, (long long)
    - unsigned char, unsigned short, unsigned int, unsigned long long
    - float, double, long double
    - bool
  - **From <stdint.h>**
    - int8_t, int16_t, int32_t, int64_t
    - uint8_t, uint16_t, uint32_t, uint64_t

  **May be used, but are not always adequate to model hardware!**

- **SystemC data types**
  - Scalar boolean type: *sc_logic*
  - Vector boolean types: *sc_bv<>*, *sc_lv<>*
  - Integer types: *sc_int<>*, *sc_uint<>*, *sc_bigint<>*, *sc_biguint<>*
  - Fixed point types: *sc_fixed<>*, *sc_ufixed<>*, *sc_fix*, *sc_ufix*
Data Types - Usage

➢ To get fast simulations, choose the right data type
  ▪ use built-in C/C++ data types as much as possible
  ▪ use to model stuff not for synthesis, e.g., testbenches
  ▪ use `sc_int<>/sc_uint<>`
    model arithmetic operations on integers of up to 64 bits
  ▪ use `sc_bv<>` (bool for a single bit)
    model boolean operations on arbitrary length bit vectors
  ▪ use `sc_lv<>` (sc_logic for a single bit)
    model 4 valued arbitrary length logic vectors (tri-state)
  ▪ use `sc_bigint<>/sc_biguint<>`
    model arithmetic operations on arbitrary length integers
  ▪ use `sc_(u)fixed_fast<>`, `sc_(u)fix_fast`
    model operations on fixed-point numbers with up to 53 bits
  ▪ use `sc_(u)fixed<>`, `sc_(u)fix`
    model operations on arbitrary length fixed-point numbers

Fastest

Slowest
Integer Types | Overview

- SystemC integer data types
  - `sc_int<nr. bits>, sc_uint<nr. bits>`
    precision up to 64 bits (on a 32 bit machine)
  - `sc_bigint<nr. bits>, sc_biguint<nr. bits>`
    arbitrary precision, but slower than `sc_int<>/sc_uint<>`

- Common properties
  - array of bits
    - bit and part select supported
    - interpreted as integer for arithmetic operations
  - supported operators
    - boolean (slow for `sc_bigint<>`, `sc_biguint<>`, use `sc_bv<>`)
    - arithmetic
  - twos-complement representation for signed numbers
Integer Types | Syntax

Syntax:

\[
\text{sc\_int}<\text{nr. bits}> \quad \text{variable\_name};
\]
\[
\text{sc\_uint}<\text{nr. bits}> \quad \text{variable\_name};
\]
\[
\text{sc\_bigint}<\text{nr. bits}> \quad \text{variable\_name};
\]
\[
\text{sc\_biguint}<\text{nr. bits}> \quad \text{variable\_name};
\]

Nr. bits: Specifies the number of bits
- Must be greater than 0
- Must be a compile time constant

```c
sc\_int<3> a; // a is a 3bit signed integer
sc\_uint<3> b; // b is a 3bit unsigned integer
sc\_int<2> c; // c is a 2bit signed integer
a = 3;        // a = 3 a[2] = 0, a[1] = 1, a[0] = 1
a = "0b011";  // a = 3 a[2] = 0, a[1] = 1, a[0] = 1
b = a;        // b = 3 b[2] = 0, b[1] = 1, b[0] = 1
c = a;        // c = -1 c[1] = 1, c[0] = 1
a = -1;       // a = -1 a[2] = 1, a[1] = 1, a[0] = 1
b = a;        // b = 7 b[2] = 1, b[1] = 1, b[0] = 1
c = a;        // c = -1 c[1] = 1, c[0] = 1
```
Single Bit Types | Overview

- **Two valued logic**
  - 0, false: logic false
  - 1, true: logic true
  - **bool** (built-in C++ data type)

- **Four valued logic**
  - use for tri-state busses and drivers
  - use to model start-up behavior at RT-Level
  - 0, '0', SC_LOGIC_0: logic false
  - 1, '1', SC_LOGIC_1: logic true
  - 'X', SC_LOGIC_X: unknown
  - 'Z', SC_LOGIC_Z: high impedance
  - **sc_logic**
    (slower than bool)
Single Bit Types | Syntax

Syntax:

```cpp
bool variable_name;
sc_logic variable_name;
```

// This is a model of a Tri-state driver.
sc_logic triState(bool control, bool data) {
    sc_logic ts_out;

    if (control == false) {
        ts_out = 'Z'; // Set the drive to Z
    } else {
        ts_out = data; // Set the drive to data
    }
    return ts_out;
}
```
Bit Vector Types | Overview

- **Two valued logic**
  - `sc_bv<nr. bits>`
    - arbitrary length bit vector

- **Four valued logic**
  - `sc_lv<nr. bits>`
    - arbitrary length logic vector

- **Common properties**
  - array of bits
    - bit and part select supported
  - supported operators
    - boolean only (convert to integer types for arithmetic operations)
  - values are represented as
    - integers
    - strings, e.g., "00110" or "001X0"
**Bit Vector Types | Syntax**

**Syntax:**

```plaintext
sc_bv<nr. bits>   variable_name;
sc_lv<nr. bits>   variable_name;
```

**Nr. bits:** Specifies the number of bits
- Must be greater than 0
- Must be a compile time constant

```plaintext
sc_bv<3> a;       // a 3bit bit vector
sc_lv<3> b;       // a 3bit logic vector
a = 3;            // a[2] = 0, a[1] = 1, a[0] = 1
a = "010";        // a[2] = 0, a[1] = 1, a[0] = 0
b = a;            // b[2] = 0, b[1] = 1, b[0] = 0
b = "01X";        // b[2] = 0, b[1] = 1, b[0] = X
b = "00X1";       // b[2] = 0, b[1] = X, b[0] = 1
                   // NOTE: b="0X1" is a hex. 1
a = b;            // A run-time warning is issued that 'X'
                   // cannot be converted!
```
Fixed Point Types | Overview

- **Compile time (templates) parameterized fixed point types**
  - sc_fixed<> , sc_fixed_fast<> are signed fixed point types
  - sc_ufixed<> , sc_ufixed_fast<> are unsigned fixed point types

- **Run time (constructor) parameterized fixed point types**
  - sc_fix , sc_fix_fast are signed fixed point types
  - sc_ufix , sc_ufix_fast are unsigned fixed point types

- **The sc_(u)fixed<> , sc_(u)fix**
  - Are implemented in software
  - Are arbitrary precision

- **The sc_(u)fixed_fast<> , sc_(u)fix_fast**
  - Are implemented via usage of the mantis of a IEEE 754 double precision floating point number
  - Are limited to 53 bits
  - Are faster than their corresponding non-fast types
  - Syntax and semantics are the same as for arbitrary precision types
Fixed Point Types | Syntax

**Syntax:**

```plaintext
sc_fixed<wl, iwl, q_mode, o_mode, n_bits> variable_name;
sc_ufixed<wl, iwl, q_mode, o_mode, n_bits> variable_name;
```

- **wl:** Specifies the total word length used for number representation
  - Must be greater than 0
  - Must be a compile time constant
- **iwl:** Specifies the number of bits to the left of the decimal point
  - Must be a compile time constant
  - May be an arbitrary integer
- **q_mode:** Specifies the quantization mode
- **o_mode:** Specifies the overflow mode
- **n_bits:** Specifies the number of saturated bits (for wrapping overflow modes)

**Syntax:**

```plaintext
sc_fix variable_name;
sc_fix variable_name(wl, iwl, q_mode, o_mode, n_bits);
sc_fix variable_name(param_obj);
```

- **wl, iwl, q_mode, o_mode, n_bits:** Same as for `sc_(u)fixed<>`
- **param_obj:** An object of class `sc_fxttype_params`
// To enable fixed point support you have to define the symbol
// SC_INCLUDE_FX before including the systemc header.
#define SC_INCLUDE_FX
#include <systemc.h>

// define a fixed point number parameter set p
sc_fxttype_params p(9,5,SC_RND,SC_SAT);

// define a fixed point number using the parameter set p
sc_fix x(p);

// do an equivalent definition without a predefined parameter set
sc_fix y(9,5,SC_RND,SC_SAT);

// If the parameters are known at compile time then the
// sc_fixed* types should be used!
sc_fixed<9, 5, SC_RND, SC_SAT> a;
sc_fixed_fast<9, 5, SC_RND, SC_SAT> b;
Fixed Point Types | Word Length

wl = 9, iw = 5

wl = 5, iw = 9

wl = 4, iw = -5

's' stands for sign bits
### Overview of Quantization Modes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC_RND</td>
<td>Rounding to plus infinity</td>
</tr>
<tr>
<td>SC_RND_ZERO</td>
<td>Rounding to zero</td>
</tr>
<tr>
<td>SC_RND_MIN_INF</td>
<td>Rounding to minus infinity</td>
</tr>
<tr>
<td>SC_RND_INF</td>
<td>Rounding to infinity</td>
</tr>
<tr>
<td>SC_RND_CONV</td>
<td>Convergent Rounding</td>
</tr>
<tr>
<td>SC_TRN</td>
<td>Truncation</td>
</tr>
<tr>
<td>SC_TRN_ZERO</td>
<td>Truncation to zero</td>
</tr>
</tbody>
</table>
Fixed Point Quantization Modes

$$\text{sc\_fixed}<3,2,\text{rndmode,SC\_SAT}>$$
Fixed Point Quantization Modes

$sc\_fixed<3,2,\text{rndmode,SC\_SAT}>$

Fixed Point Number
Floating Point Number

Friedrich-Alexander-Universität Erlangen-Nürnberg
Joachim Falk
Fixed Point Overflow Modes

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC_SAT</td>
<td>Saturation</td>
</tr>
<tr>
<td>SC_SAT_ZERO</td>
<td>Saturation to zero</td>
</tr>
<tr>
<td>SC_SAT_SYM</td>
<td>Symmetrical saturation</td>
</tr>
<tr>
<td>SC_WRAP</td>
<td>Wrap around</td>
</tr>
<tr>
<td>SC_WRAP_SM</td>
<td>Sign magnitude wrap around</td>
</tr>
</tbody>
</table>
Fixed Point Overflow Modes

```
sc_fixed<3,2,SC_RND,ovflmode,0>
```

- Fixed Point Overflow Modes
  - SC_SAT
  - SC_SAT_ZERO
  - SC_SAT_SYM
  - SC_WRAP
  - SC_WRAP_SM

Friedrich-Alexander-Universität Erlangen-Nürnberg
Joachim Falk
Fixed Point Overflow Modes

$\text{sc\_fixed}<3,2,\text{SC\_RND},\text{ovflmode},1>$
Fixed Point Overflow Modes

\[\text{sc\_fixed<3,2,SC\_RND,ovflmode,2>}\]