Selected Slides for EECS 222,
Part 1 (RD, 04/24/17)

The Definitive Guide to SystemC: The SystemC Language

David C Black, Doulos

DESIGN AUTOMATION CONFERENCE

DOULOS THURSDAY IS TRAINING DAY

The SystemC Language

DOULOS THURSDAY IS TRAINING DAY

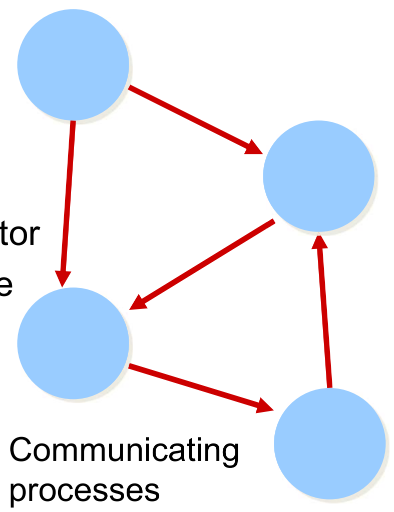
What is SystemC?

System-level modeling language

- Network of communicating processes (c.f. HDL)
- Supports heterogeneous models-of-computation
- Models hardware and software

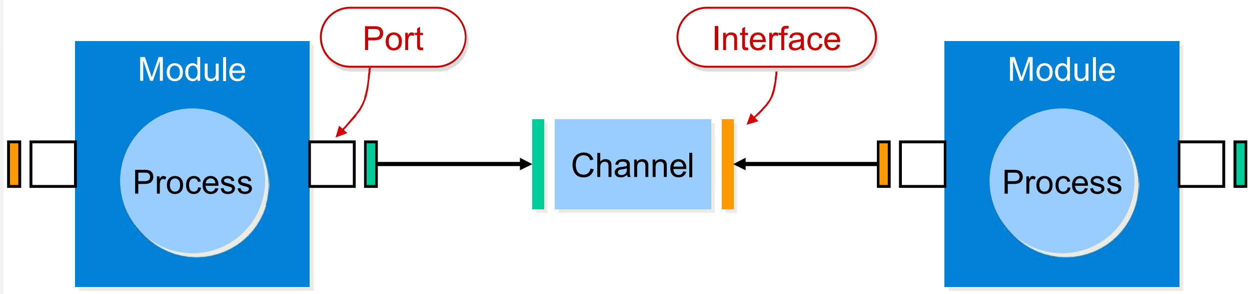
C++ class library

- Open source proof-of-concept simulator
- Owned by Accellera Systems Initiative

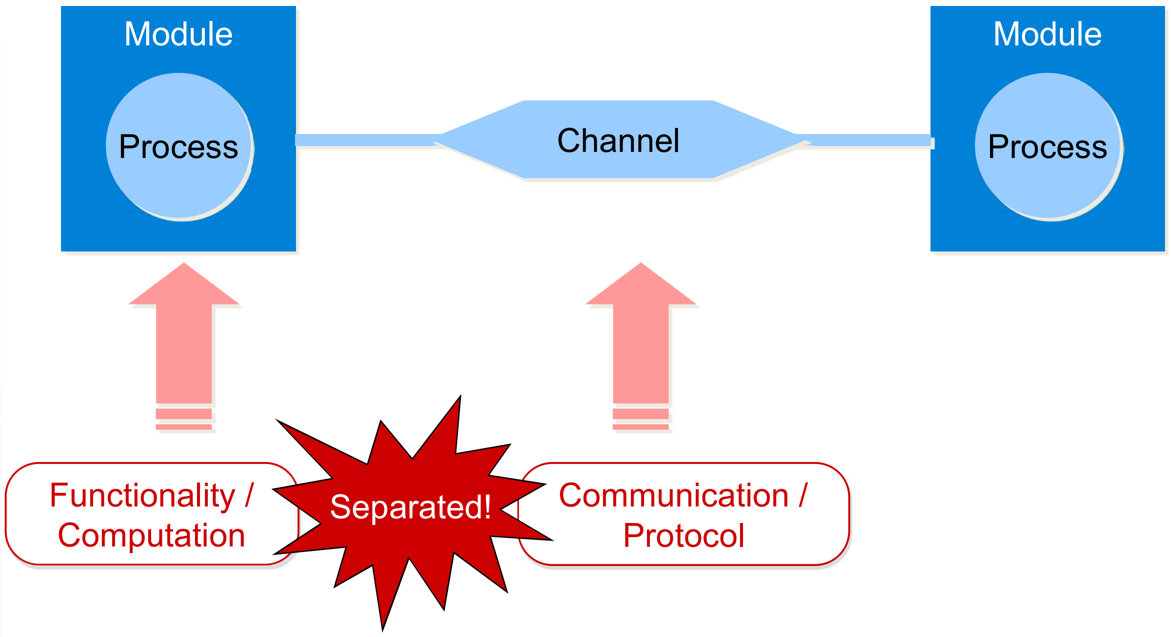


Features of SystemC

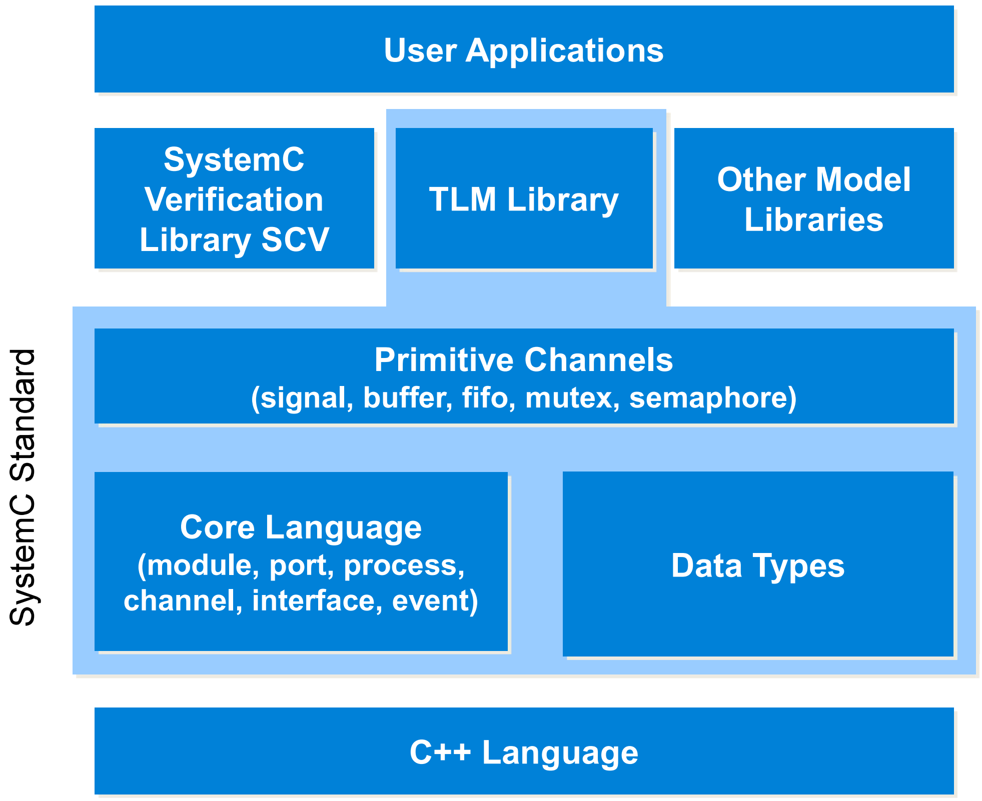
• Modules	(structure)
• Ports	(structure)
• Processes	(computation, concurrency)
• Channels	(communication)
• Interfaces	(communication refinement)
• Events	(time, scheduling, synchronization)
• Data types	(hardware, fixed point)



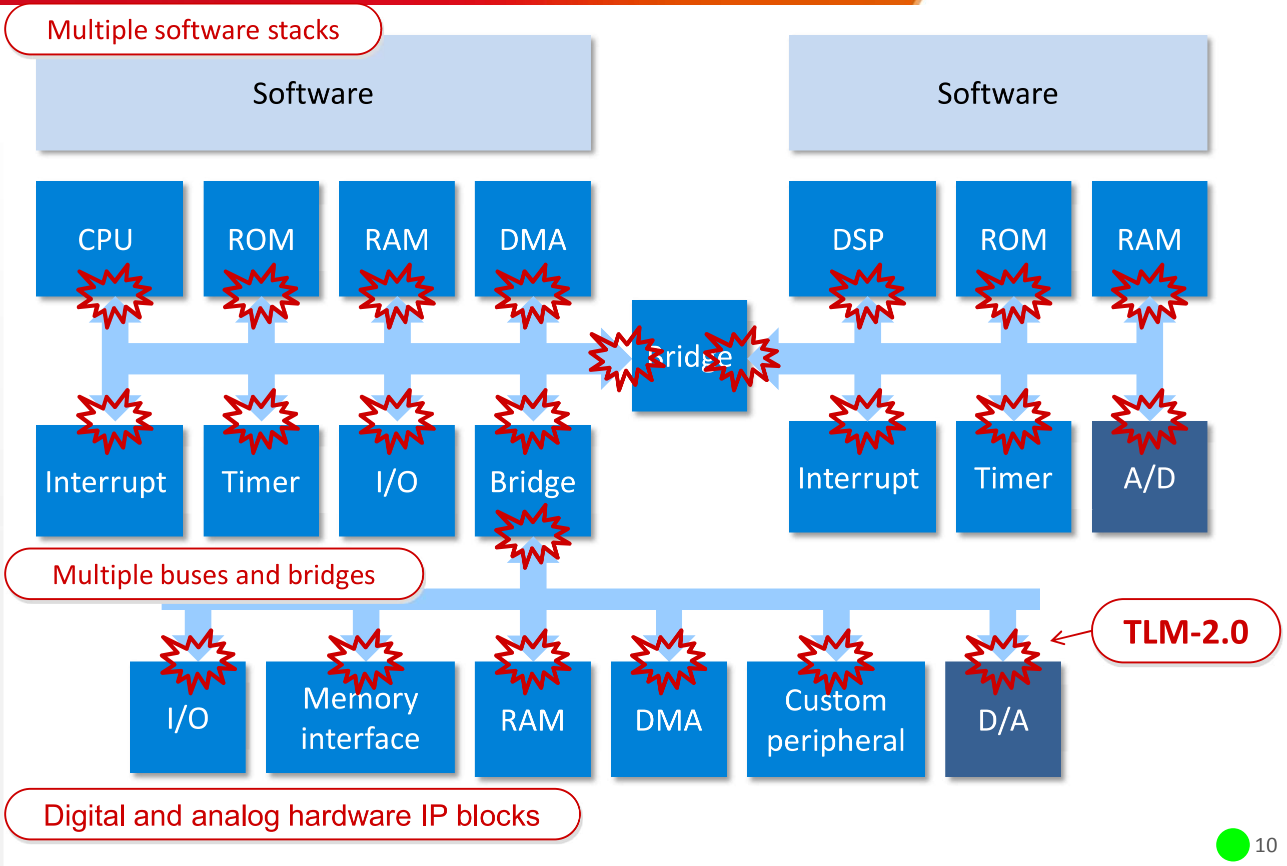
Modules and Channels



Architecture of SystemC


User Applications
SystemC Verification Library SCV
TLM Library
Other Model Libraries
Primitive Channels (signal, buffer, fifo, mutex, semaphore)
Core Language (module, port, process, channel, interface, event)
Data Types
SystemC Standard
C++ Language

Typical Use Case: Virtual Platform



Track 3: The Definitive Guide to SystemC

The SystemC Language

- Introduction to SystemC
- Core Concepts and Syntax
  - Data
  - Modules and connectivity
  - Processes & Events
  - Channels and Interfaces
  - Ports
- Bus Modeling
- Odds and Ends

SystemC Data Types

In namespace sc_dt::

Template	Base class	Description
sc_int<W>	sc_int_base	Signed integer, W < 65
sc_uint<W>	sc_uint_base	Unsigned integer, W < 65
sc_bigint<W>	sc_signed	Arbitrary precision signed integer
sc_biguint<W>	sc_unsigned	Arbitrary precision unsigned integer
		(intermediate results unbounded)
sc_logic		4-valued logic: '0' '1' 'X' 'Z'
sc_bv<W>	sc_bv_base	Bool vector
sc_lv<W>	sc_lv_base	Logic vector
sc_fixed<>	sc_fix	Signed fixed point number
sc_ufixed<>	sc_ufix	Unsigned fixed point number

Limited Precision Integer sc_int

```
int        i;
sc_int<8> j;
i = 0x123;
sc_assert( i == 0x123 );

j = 0x123;
sc_assert( j == 0x23 );

sc_assert( j[0] == 1 );
sc_assert( j.range(7,4) == 0x2 );
sc_assert( concat(j,j) == 0x2323 );
```

Truncated to 8 bits

Bit select

Part select

Concatenation

- Other useful operators: arithmetic, relational, bitwise, reduction, assignment

  length()   to_int()   to_string()   *implicit-conversion-to-64-bit-int*

Logic and Vector Types

sc_logic and sc_lv<W>

- Values SC_LOGIC_0, SC_LOGIC_1, SC_LOGIC_X, SC_LOGIC_Z
- Initial value is SC_LOGIC_X

No arithmetic operators

Can write values as chars and strings, i.e. '0' '1' 'X' 'Z'

```
sc_logic R, S;
R = '1';
S = 'Z';
S = S & R;
```

```
sc_int<4>  n = "0b1010";
bool      boo = n[3];
sc_lv<4>  lv = "01XZ";
sc_assert( lv[0] == 'Z' );
n += lv.to_int();
cout << n.to_string(SC_HEX);
```

Fixed Point Types

```
sc_fixed <wl, iwl, q_mode, o_mode, n_bits> a;
sc_ufixed<wl, iwl, q_mode, o_mode, n_bits> b;
sc_fix   c(wl, iwl, q_mode, o_mode, n_bits);
sc_ufix  d(wl, iwl, q_mode, o_mode, n_bits);
```

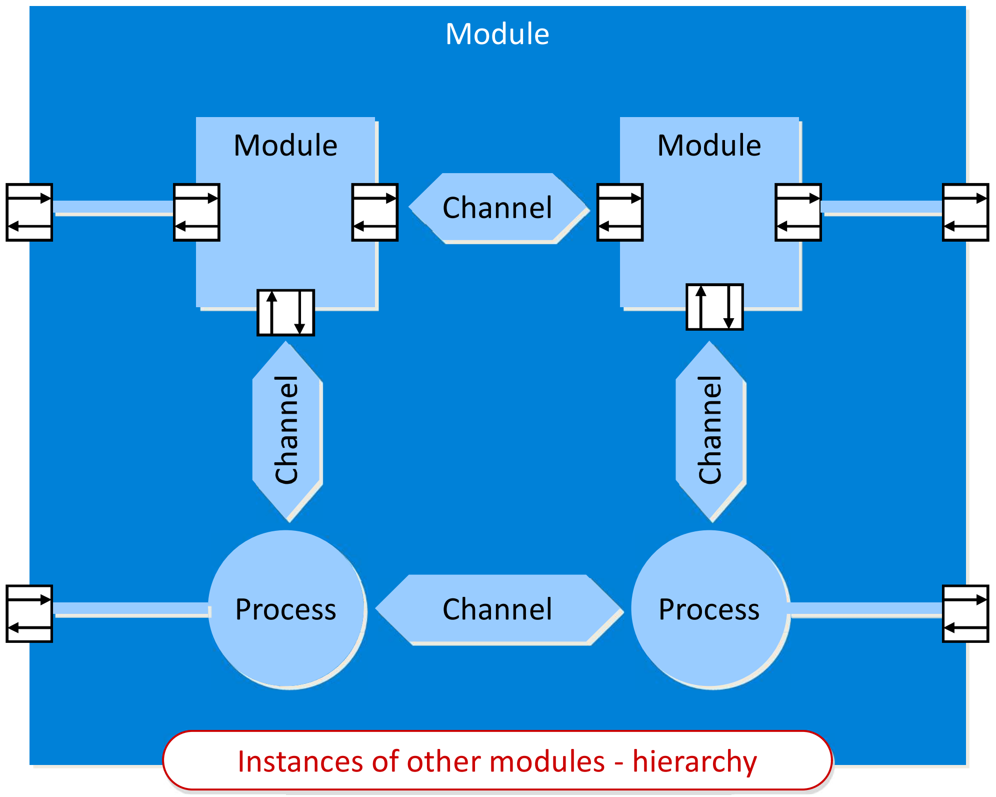
Word length	- number of stored bits - no limit
Integer word length	- number of bits before binary point
Quantization mode	- behavior when insufficient precision
Overflow mode	- behavior when result too big
Number of saturated bits	- used with wrap overflow modes

Compiler flag -DSC_INCLUDE_FX

Data Summary

	C++	SystemC	SystemC	SystemC
Unsigned	**unsigned int**	**sc_bv, sc_lv**	**sc_uint**	**sc_biguint**
Signed	**int**		**sc_int**	**sc_bigint**
Precision	Host-dependent	Limited precision	Limited precision	Unlimited precision
Operators	C++ operators	No arithmetic operators	Full set of operators	Full set of operators
Speed	**Fastest**	Faster	Slower	**Slowest**

Modules



Instances of other modules - hierarchy

SC_MODULE

Class

Ports

Constructor

Process

```
#include "systemc.h"

SC_MODULE(Mult)
{
  sc_in<int>  a;
  sc_in<int>  b;
  sc_out<int> f;

  void action() { f = a * b; }

  SC_CTOR(Mult)
  {
    SC_METHOD(action);
      sensitive << a << b;
  }
};
```

SC_MODULE or sc_module?

- Equivalent
  :

```
SC_MODULE(Name)
{
  ...
};
```

```
struct Name: sc_module
{
  ...
};
```

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

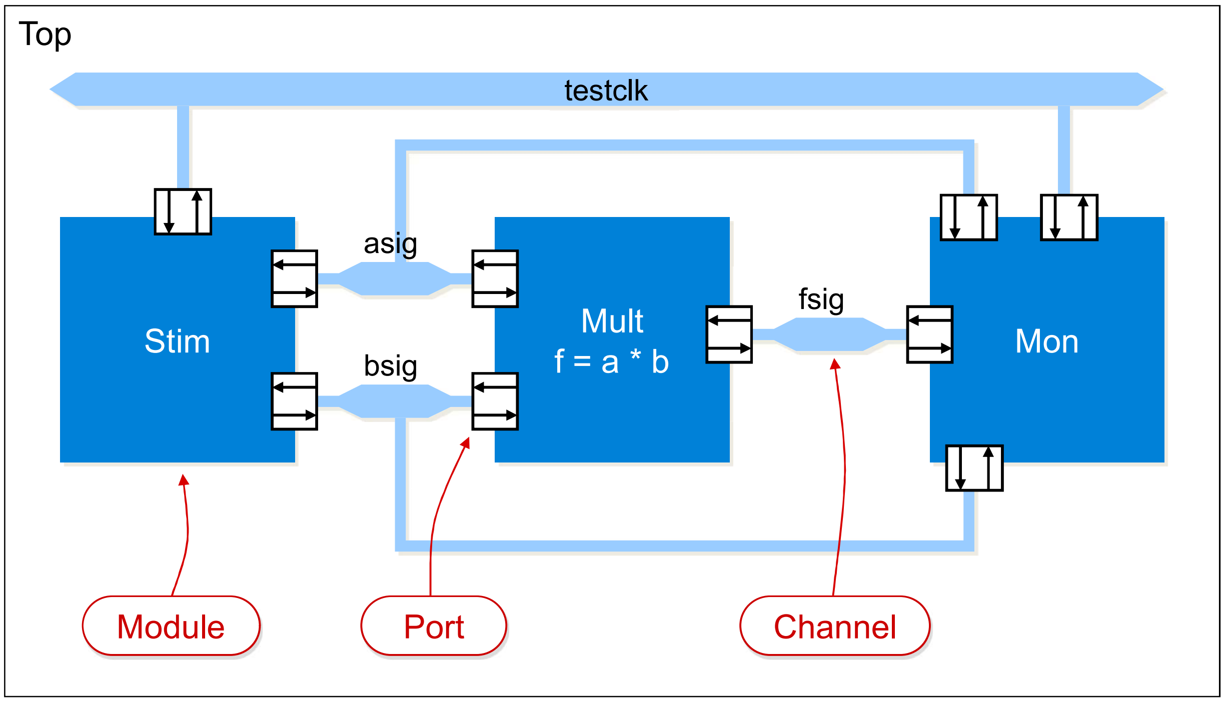
Separate Header File

```
// mult.h
#include "systemc.h"

SC_MODULE(Mult)
{
  sc_in<int>  a;
  sc_in<int>  b;
  sc_out<int> f;

  void action();

  SC_CTOR(Mult)
  {
    SC_METHOD(action);
      sensitive << a << b;
  }
};
```

```
// mult.cpp

#include "mult.h"

void Mult::action()
{
  f = a * b;
}
```

- Define constructor in .cpp?
  Yes - explained later

The Test Bench



Top Level Module

Header files

Channels

Modules

```
#include "systemc.h"
#include "stim.h"
#include "mult.h"
#include "mon.h"

SC_MODULE(Top)
{

  sc_signal<int> asig, bsig, fsig;

  sc_clock testclk;

  Stim stim1;
  Mult uut;
  Mon  mon1;

  ...
}
```

Module Instantiation

```
SC_MODULE(Top)
{
  sc_signal<int> asig, bsig, fsig;

  sc_clock testclk;

  Stim stim1;
  Mult uut;
  Mon  mon1;

  SC_CTOR(Top)
  : testclk("testclk", 10, SC_NS),
    stim1("stim1"),
    uut  ("uut"),
    mon1 ("mon1")
  {
    ...
  }
}
```

Name of data member

String name of instance (constructor argument)

Port Binding

```
SC_CTOR(Top)
: testclk("testclk", 10, SC_NS),
  stim1("stim1"),
  uut("uut"),
  mon1("mon1")
{
  stim1.a(asig);
  stim1.b(bsig);
  stim1.clk(testclk);

  uut.a(asig);
  uut.b(bsig);
  uut.f(fsig);

  mon1.a.bind(asig);
  mon1.b.bind(bsig);
  mon1.f.bind(fsig);
  mon1.clk.bind(testclk);
}
```

Alternative function

Port name

Channel name

sc_main

- sc_main is the entry point to a SystemC application

```
#include "systemc.h"
#include "top.h"

int sc_main(int argc, char* argv[])
{

  Top top("top");

  sc_start();

  return 0;
}
```

Called from main()

Instantiate one top-level module

End elaboration, run simulation

Namespaces

```
#include "systemc.h"

SC_MODULE(Mod)
{
  sc_in<bool> clk;
  sc_out<int> out;

  ... cout << endl;
```

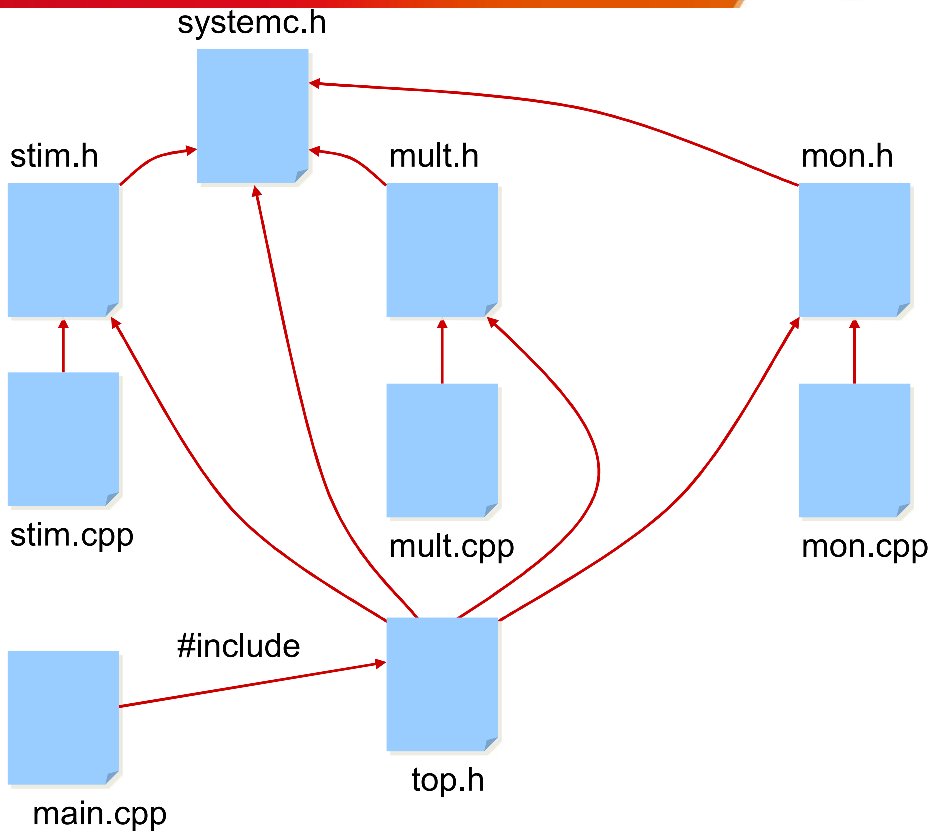
Old header - global namespace

```
#include "systemc"

SC_MODULE(Mod)
{
  sc_core::sc_in<bool> clk;
  sc_core::sc_out<int> out;

  ... std::cout << std::endl;
```

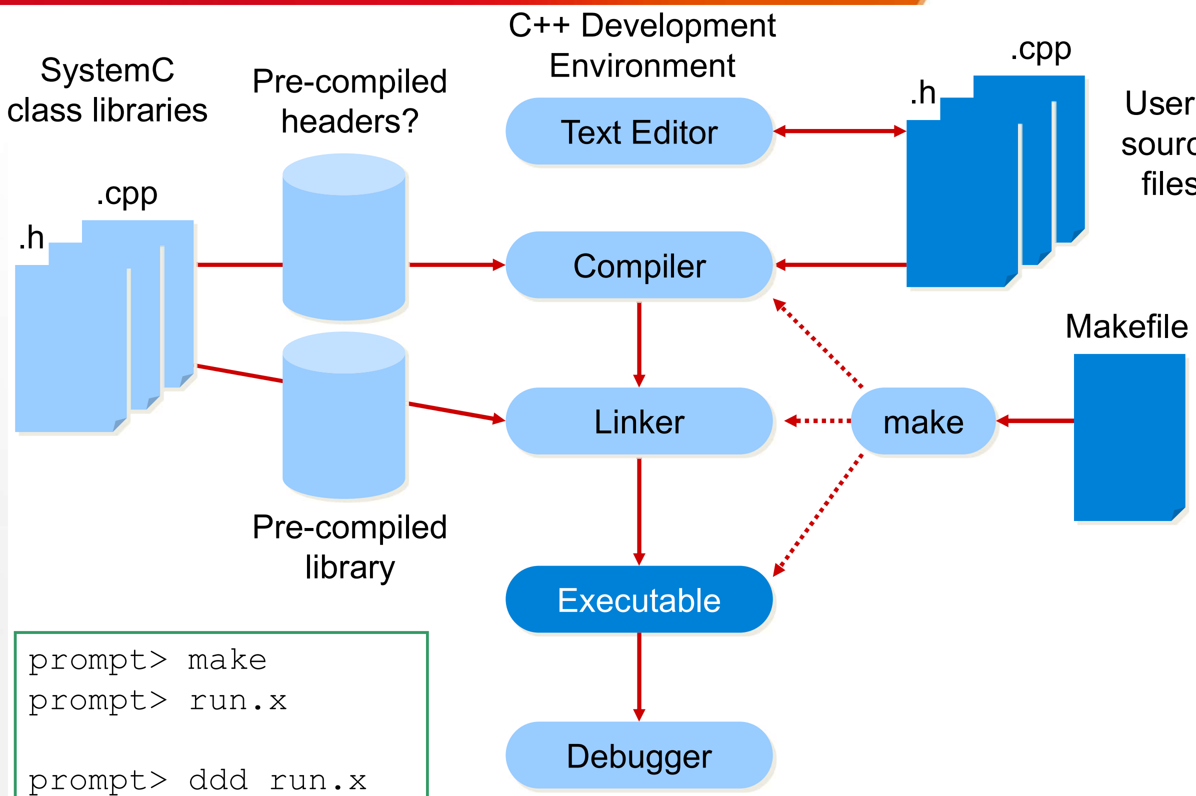
New header

```
#include "systemc"
using namespace sc_core;
using namespace sc_dt;
using std::cout;
using std::endl;

SC_MODULE(Mod) {
  sc_in<bool> clk;
  sc_out<int> out;
  ... cout << endl;
```

Summary of Files



Compilation and Simulation



```
prompt> make
prompt> run.x

prompt> ddd run.x
```

Kinds of Process

- Processes
  - Must be within a module (not in a function)
  - A module may contain many processes
- Three different kinds of process

Methods	SC_METHOD	
Threads	SC_THREAD	
Clocked threads	SC_CTHREAD	(for synthesis)

- Processes can be *static* or *dynamic*

SC_METHOD Example

```
#include <systemc.h>

template<class T>
SC_MODULE(Register)
{
  sc_in<bool> clk, reset;
  sc_in<T>    d;
  sc_out<T>   q;

  void entry();

  SC_CTOR(Register)
  {
    SC_METHOD(entry);
      sensitive << reset;
      sensitive << clk.pos();
  }
};
```

```
template<class T>
void Register<T>::entry()
{
  if (reset)
    q = 0; // promotion
  else if (clk.posedge())
    q = d;
}
```

- SC_METHODs execute in zero time
- SC_METHODs cannot be suspended
- SC_METHODs should not contain infinite loops

SC_THREAD Example

```
#include "systemc.h"

SC_MODULE(Stim)
{
  sc_in<bool> Clk;
  sc_out<int> A;
  sc_out<int> B;

  void stimulus();

  SC_CTOR(Stim)
  {
    SC_THREAD(stimulus);
      sensitive << Clk.pos();
  }
};
```

```
#include "stim.h"

void Stim::stimulus()
{
  wait();
  A = 100;
  B = 200;
  wait();      ← for Clk edge
  A = -10;
  B = 23;
  wait();
  A = 25;
  B = -3;
  wait();
  sc_stop();   ← Stop simulation
}
```

- More general and powerful than an `SC_METHOD`
- Simulation may be slightly slower than an `SC_METHOD`
- Called once only: hence often contains an infinite loop

SC_HAS_PROCESS

THURSDAY IS TRAINING DAY

```
#include "systemc.h"
class Counter: public sc_module
{
public:
  sc_in<bool> clock, reset;
  sc_out<int> q;

  Counter(sc_module_name _nm, int _mod)
  : sc_module(_nm), count(0) , modulus(_mod)
  {
    SC_HAS_PROCESS(Counter);

    SC_METHOD(do_count);
      sensitive << clock.pos();
  }


private:
  void do_count();
  int       count;
  int const modulus;
};
```

Constructor arguments

Needed if there's a process and not using SC_CTOR

Dynamic Sensitivity

THURSDAY IS TRAINING DAY

```
SC_CTOR(Module)
{
  SC_THREAD(thread);
    sensitive << a << b;
}

void thread()
{
  for (;;)
  {
    wait();
    ...
    wait(10, SC_NS);
    ...
    wait(e);
    ...
  }
}
```

Static sensitivity list

Wait for event on *a* or *b*

Wait for 10ns

Wait for event *e*

ignore *a* or *b*

sc_event and Synchronization

```
SC_MODULE(Test)
{
  int data;
  sc_event e;
  SC_CTOR(Test)
  {
    SC_THREAD(producer);
    SC_THREAD(consumer);
  }
  void producer()
  {
    wait(1, SC_NS);
    for (data = 0; data < 10; data++) {
      e.notify();
      wait(1, SC_NS);
    }
  }
  void consumer()
  {
    for (;;) {
      wait(e);
      cout << "Received " << data << endl;
    }
  }
};
```

Shared variable

Primitive synchronization object

Schedule event immediately

Resume when event occurs

sc_time

- Simulation time is a 64-bit unsigned integer
- Time resolution is programmable - must be power of 10 x fs
- Resolution can be set once only, before use and before simulation
- Default time resolution is 1 ps

```
enum sc_time_unit {SC_FS, SC_PS, SC_NS, SC_US, SC_MS, SC_SEC};
```

```
sc_time(double, sc_time_unit);
```

Constructor

```
void    sc_set_time_resolution(double, sc_time_unit);
sc_time sc_get_time_resolution();
```

```
const sc_time& sc_time_stamp();
```

Get current simulation time

sc_clock

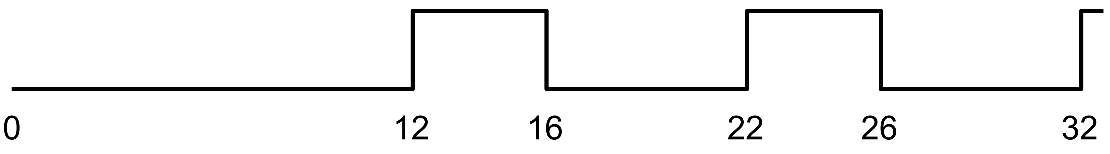
```
sc_clock clk("clk",    period, 0.4, firstedge, true);

sc_clock clk("clk", 10, SC_NS, 0.4, 12, SC_NS, true);
```

sc_time

sc_time

Duty cycle

1st edge rising

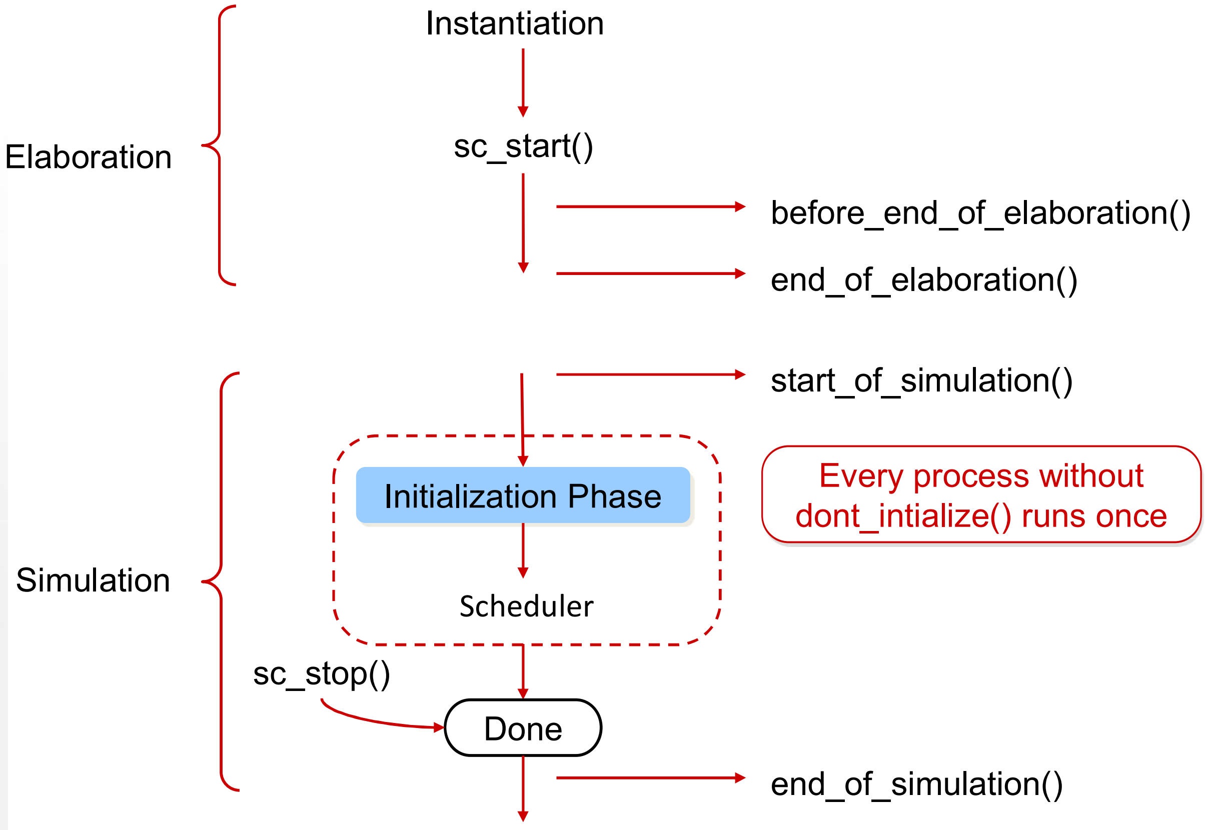


Defaults

```
("clock_N", 1, SC_NS, 0.5,  0, SC_NS, true);
```

Avoid clocks altogether for fast models!

Elaboration / Simulation Callbacks



Overriding the Callbacks

THURSDAY IS TRAINING DAY

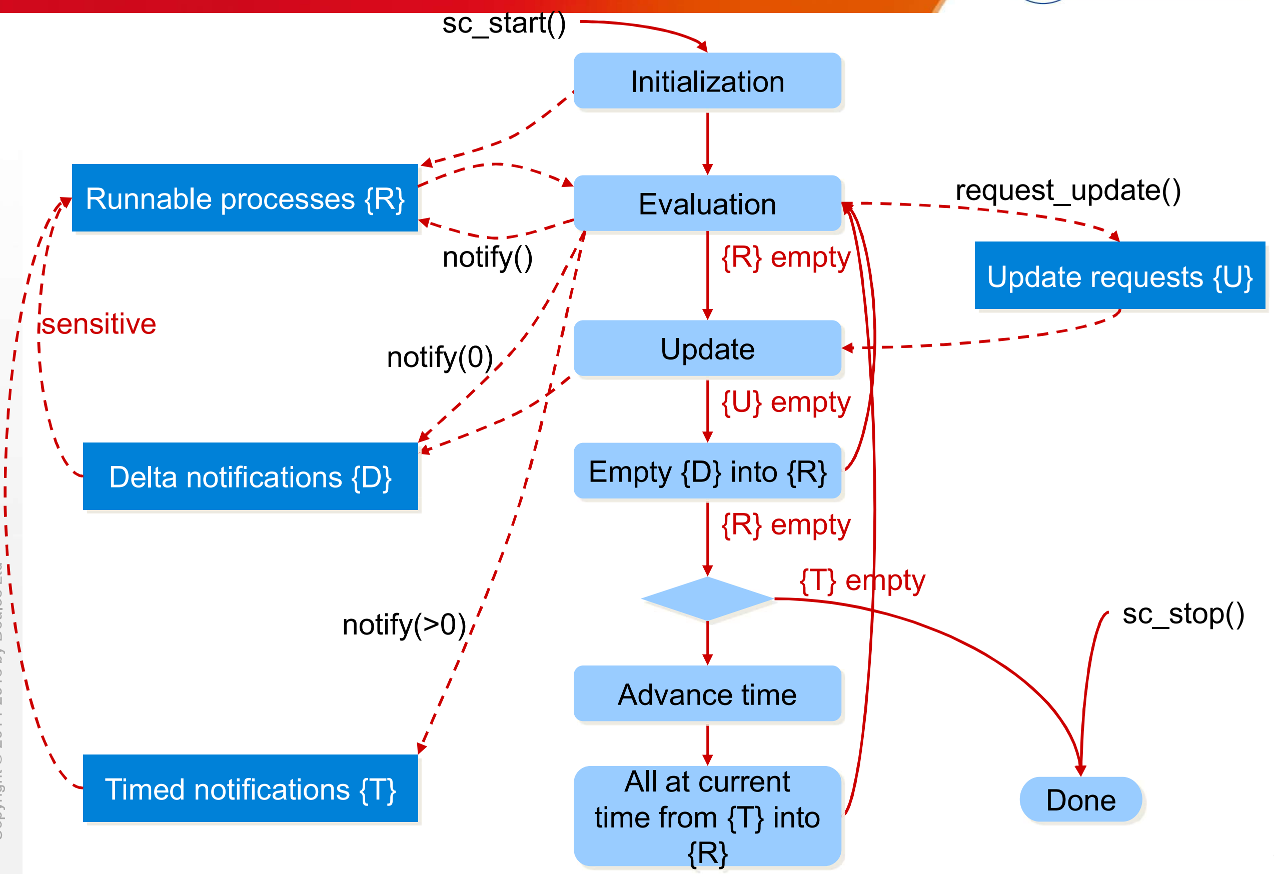
- Called for each
  - Module
  - Primitive channel
  - Port
  - Export

```
SC_MODULE(Test)
{
  SC_CTOR(Test) {}

  void before_end_of_elaboration(){...}
  void end_of_elaboration()       {...}
  void start_of_simulation()      {...}
  void end_of_simulation()        {...}
};
```

- Do nothing by default
- before_end_of_elaboration() may perform instantiation and port binding
- In PoC simulator, end_of_simulation() only called after sc_stop()

The Scheduler in Detail

THURSDAY IS TRAINING DAY



Kinds of Channel

- Primitive channels
  - Implement one or more interfaces
  - Derived from sc_prim_channel
  - Have access to the update phase of the scheduler
  - Examples - sc_signal, sc_signal_resolved, sc_fifo
- Hierarchical channels
  - Implement one or more interfaces
  - Derived from sc_module
  - Can instantiate ports, processes and modules
- Minimal channels - implement one or more interfaces

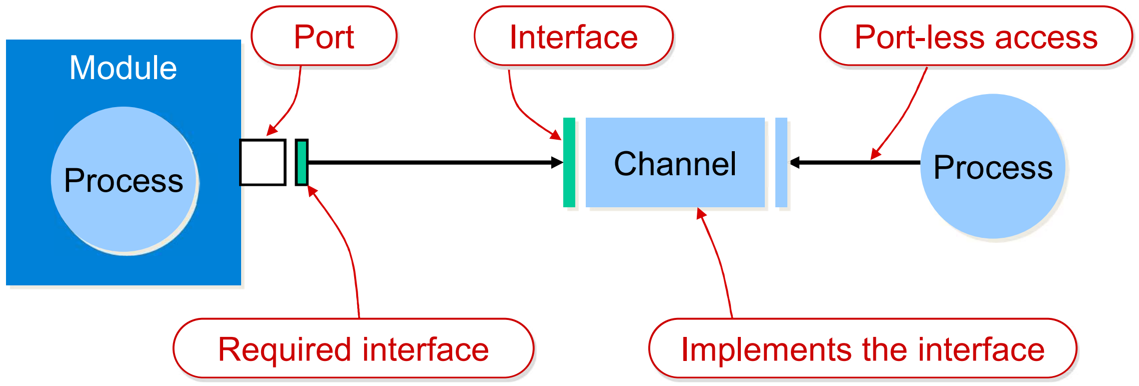
Built-in Primitive Channels

Channel	Interfaces	Events
sc_signal<T>	sc_signal_in_if<T> sc_signal_inout_if<T>	value_changed_event()
sc_buffer<T>	Same as sc_signal	On every write()
sc_signal_resolved sc_signal_rv<W>	Same as sc_signal<sc_logic>	Same a sc_signal
sc_clock	Same as sc_signal<bool>	posedge & negedge

sc_fifo<T>	sc_fifo_in_if<T>
	sc_fifo_out_if<T>
		data_written_event()
		data_read_event()

sc_mutex	sc_mutex_if	n/a
sc_semaphore	sc_semaphore_if	n/a
sc_event_queue	n/a	Every notify() invocation

Interface Method Call



An interface declares of a set of methods (pure virtual functions)

An interface is an abstract base class of the channel

A channel *implements* one or more interfaces (c.f. Java)

A module calls interface methods via a port

Declare the Interface

Important

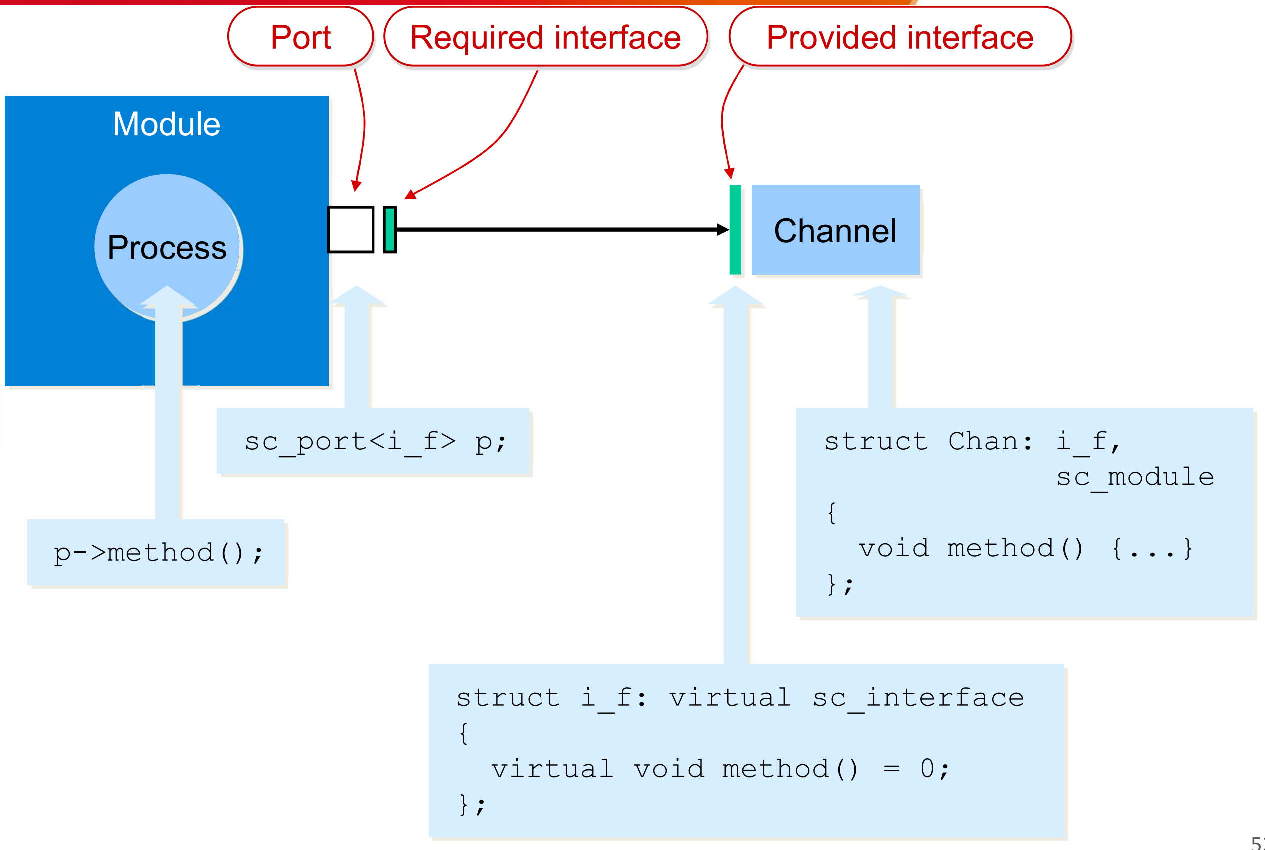
```
#include "systemc"
class queue_if : virtual public sc_core::sc_interface
{
public:
  virtual void write(char c) = 0;
  virtual char read() = 0;
};
```

Queue Channel Implementation

```
#include "queue_if.h"
class Queue : public queue_if, public sc_core::sc_object
{
public:
  Queue(char* nm, int _sz)
  : sc_core::sc_object(nm), sz(_sz)
  { data = new char[sz]; w = r = n = 0; }

  void write(char c);
  char read();

private:
  char* data;
  int sz, w, r, n;
};
```

Implements interface methods

Understanding Ports



```
sc_port<i_f> p;
```

```
p->method();
```

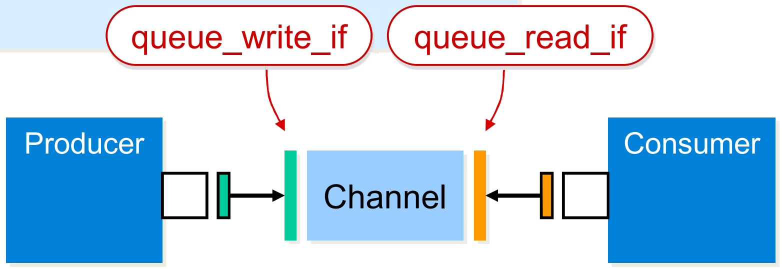
```
struct Chan: i_f,
             sc_module
{
  void method() {...}
};
```

```
struct i_f: virtual sc_interface
{
  virtual void method() = 0;
};
```

Queue Ports

THURSDAY IS TRAINING DAY

```
class Producer : public sc_core::sc_module
{
public:
  sc_core::sc_port<queue_write_if> out;

  void do_writes();

  SC_CTOR(Producer)
  {
    SC_THREAD(do_writes);
  }
};
```



Calling Methods via Ports

THURSDAY IS TRAINING DAY

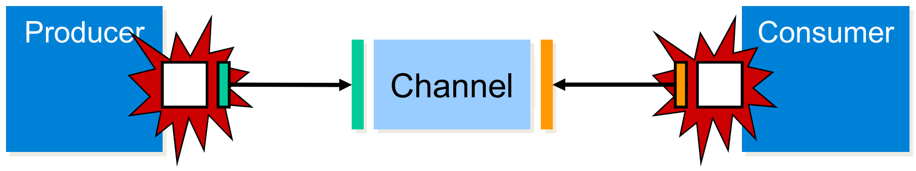
```
#include <systemc>
#include "producer.h"
using namespace sc_core;

void Producer::do_writes()
{
  std::string txt = "Hallo World.";
  for (int i = 0; i < txt.size(); i++)
  {
    wait(SC_ZERO_TIME);

    out->write(txt[i]);
  }
}
```

Note: -> overloaded

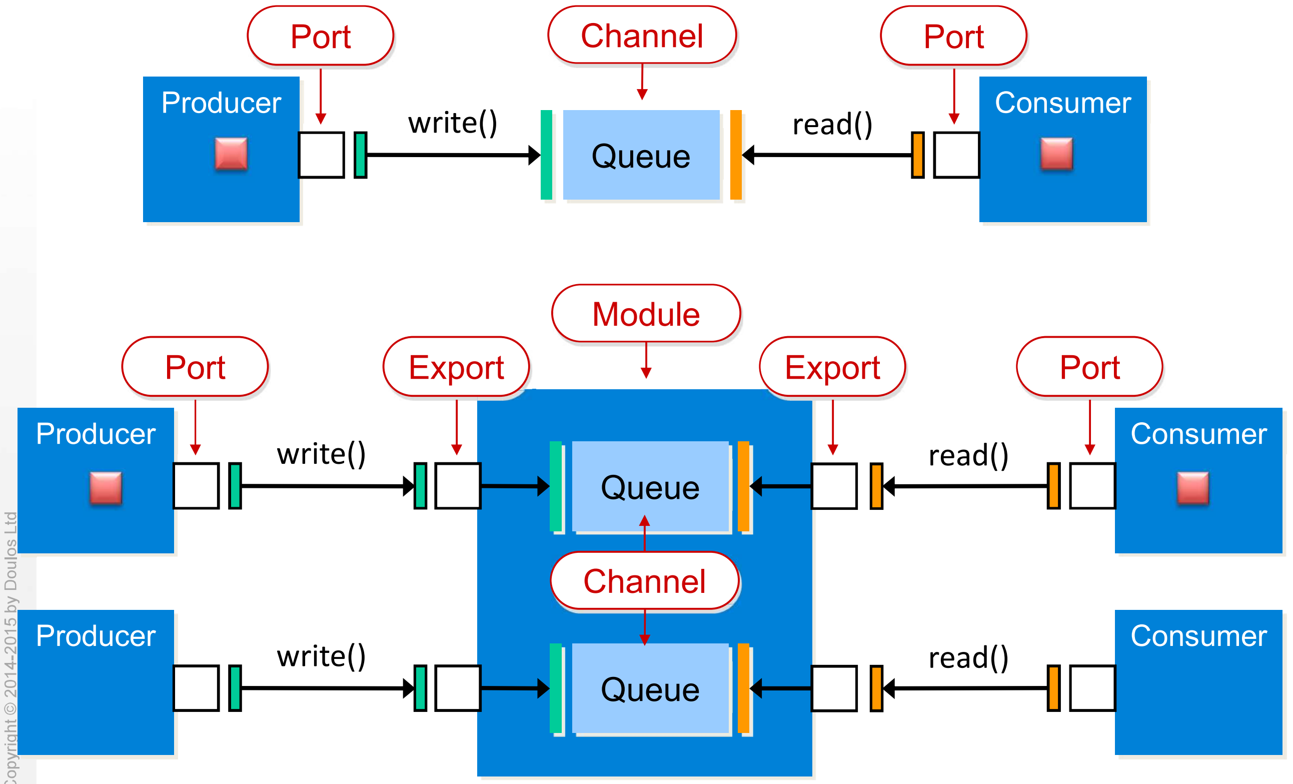
Interface Method Call

Why Ports?

- Ports allow modules to be independent of their environment
- Ports support elaboration-time checks (register_port, end_of_elaboration)
- Ports can have data members and member functions



Exports


Port
Channel
Port
Producer
write()
Queue
read()
Consumer
Module
Port
Export
Export
Port
Producer
write()
Queue
read()
Consumer
Channel
Producer
write()
Queue
read()
Consumer