The Definitive Guide to SystemC: The SystemC Language

David C Black, Doulos

The SystemC Language

- Introduction to SystemC
  - Overview and background
  - Central concepts
  - The SystemC World
  - Use cases and benefits
- Core Concepts and Syntax
- Bus Modeling
- Odds and Ends

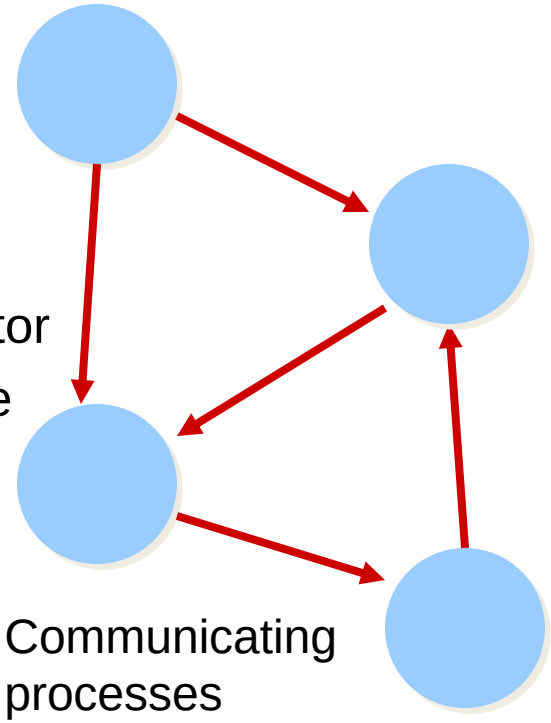
What is SystemC?

System-level modeling language

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

C++ class library

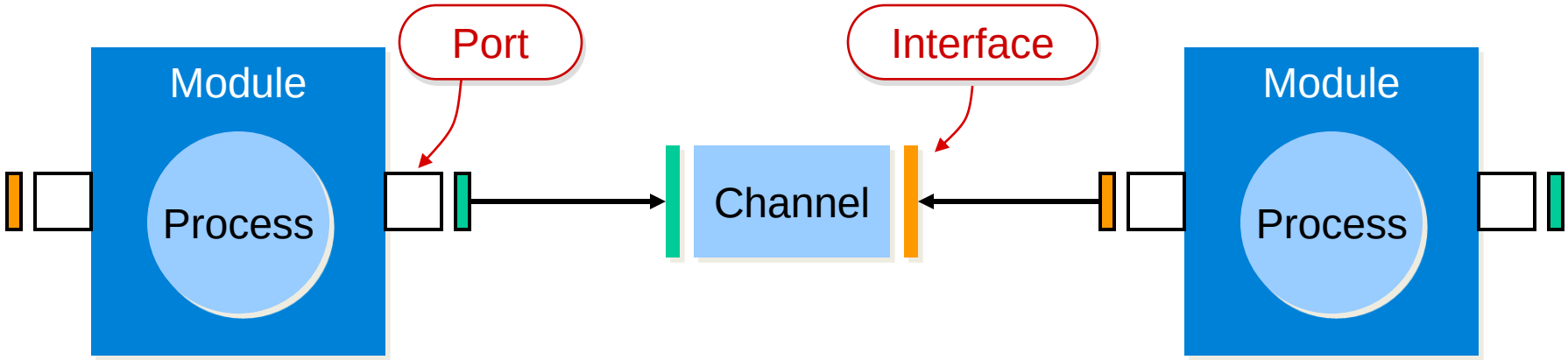
- Open source proof-of-concept simulator
- Owned by Accellera Systems Initiative



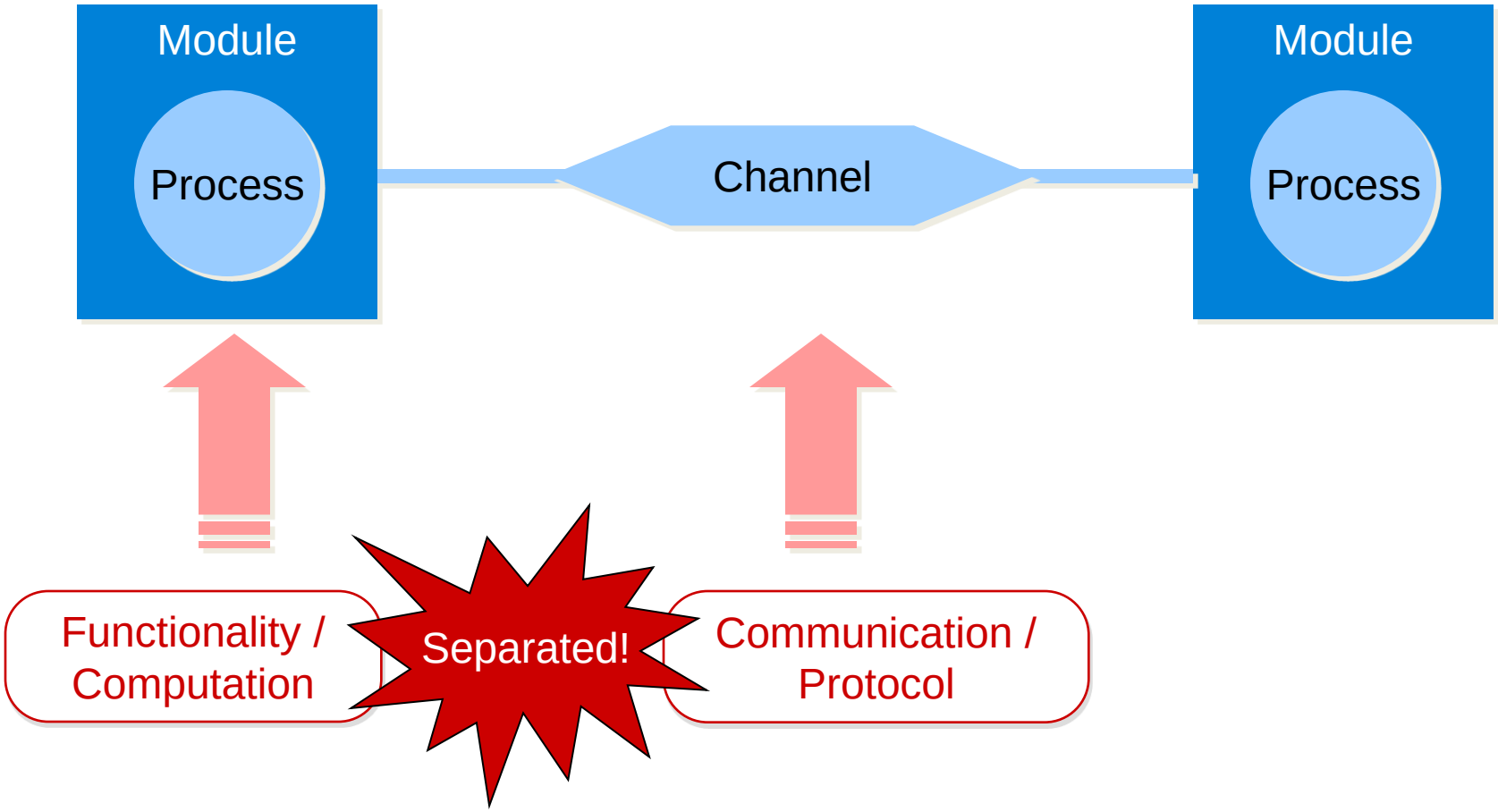
Communicating processes

Features of SystemC

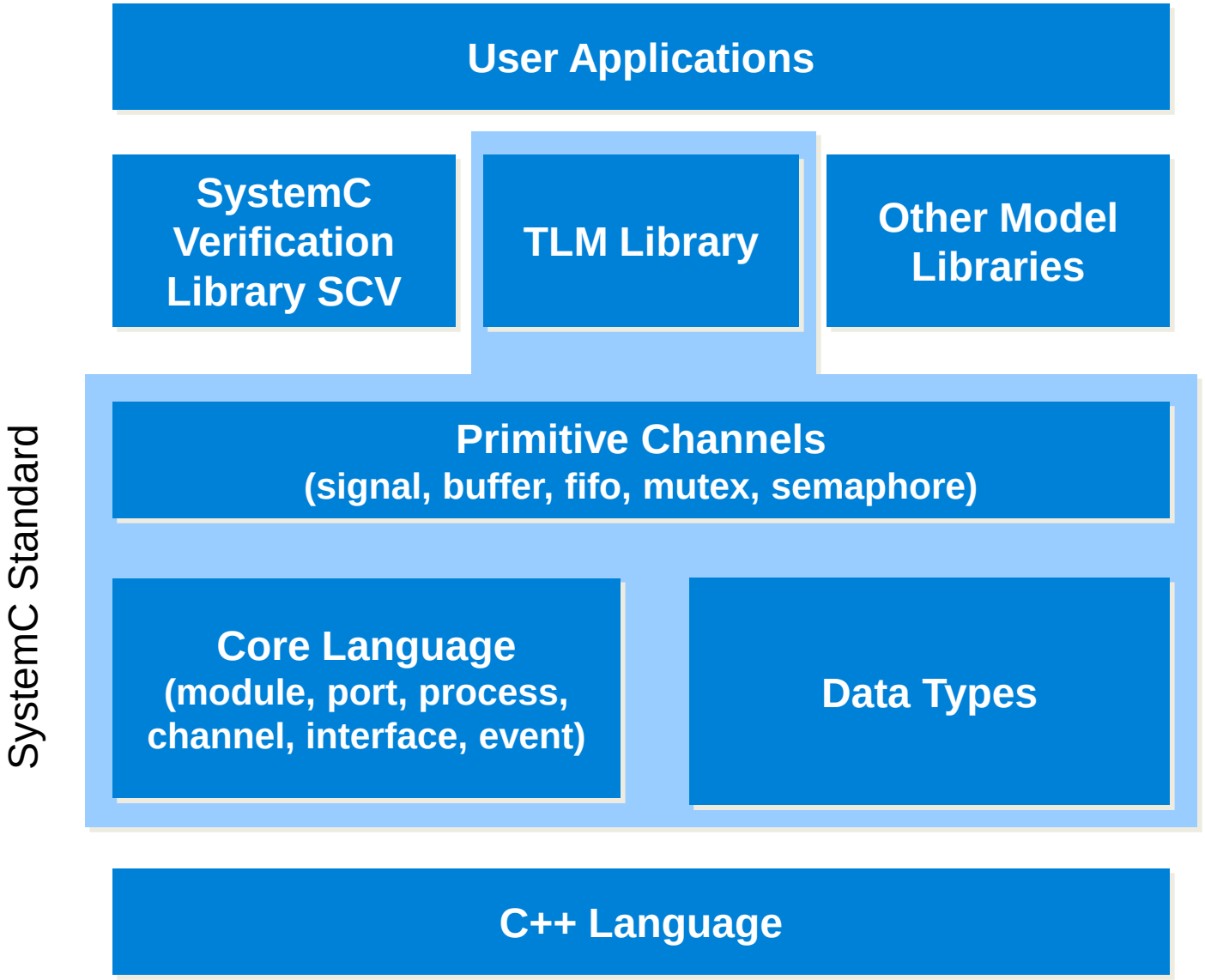
- Modules (structure)
- Ports (structure)
- Processes (computation, concurrency)
- Channels (communication)
- Interfaces (communication refinement)
- Events (time, scheduling, synchronization)
- Data types (hardware, fixed point)



Modules and Channels



Architecture of SystemC



Accellera Systems Initiative

Formed from Open-SystemC Initiative (OSCI) and Accellera

Website http://www.accellera.org/community/systemc

National Users Groups

- http://www-ti.informatik.uni-tuebingen.de/~systemc
- http://www.nascug.org
- http://www.iscug.in
- Others...

Language Working Group LWG

- IEEE 1666™ -2005
  - SystemC 2.2 released Mar 2006 (IEEE 1666 compliant)
- IEEE 1666™ -2011
  - SystemC 2.3.1 released Apr 2014 (IEEE 1666 compliant)
  - Includes TLM-2

Other Working Groups

Verification Working Group VWG

- SystemC Verification (SCV) library released 2003

Synthesis Working Group SWG

- Synthesisable subset of SystemC

Transaction-Level Modeling Working Group TLMWG

- TLM-1.0 released May 2005
- TLM-2.0 released June 2008
- TLM-2.0 part of IEEE 1666-2011

Analog and Mixed Signal Working Group AMSWG

Control, Configuration & Inspection Working Group CCIWG

What can you do with SystemC?

Discrete Event Simulation (events, time)

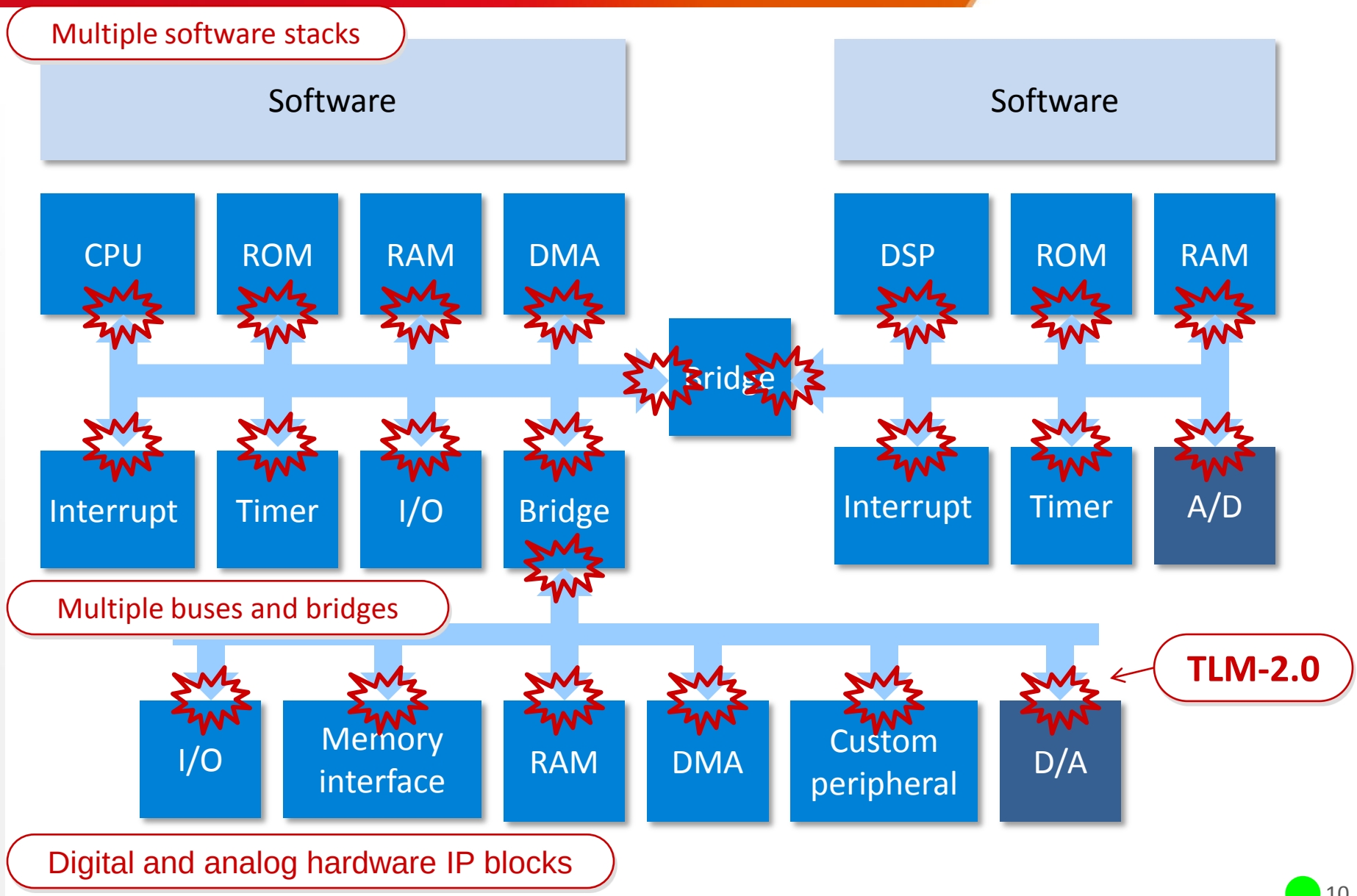
- Register Transfer Level (delta delays, bus resolution)
- Transaction Level (communication using function calls)
- Kahn Process Networks (infinite fifos, reads block when empty)
- Dataflow (input-execution-output stages)
- CSP (rendezvous, blocking reads & writes)

Continuous Time or Frequency Domain (Analog)
SystemC AMS

NOT gate level

NOT abstract RTOS modeling (process scheduling, priorities, pre-emption)
(originally planned as version 3)

Typical Use Case: Virtual Platform

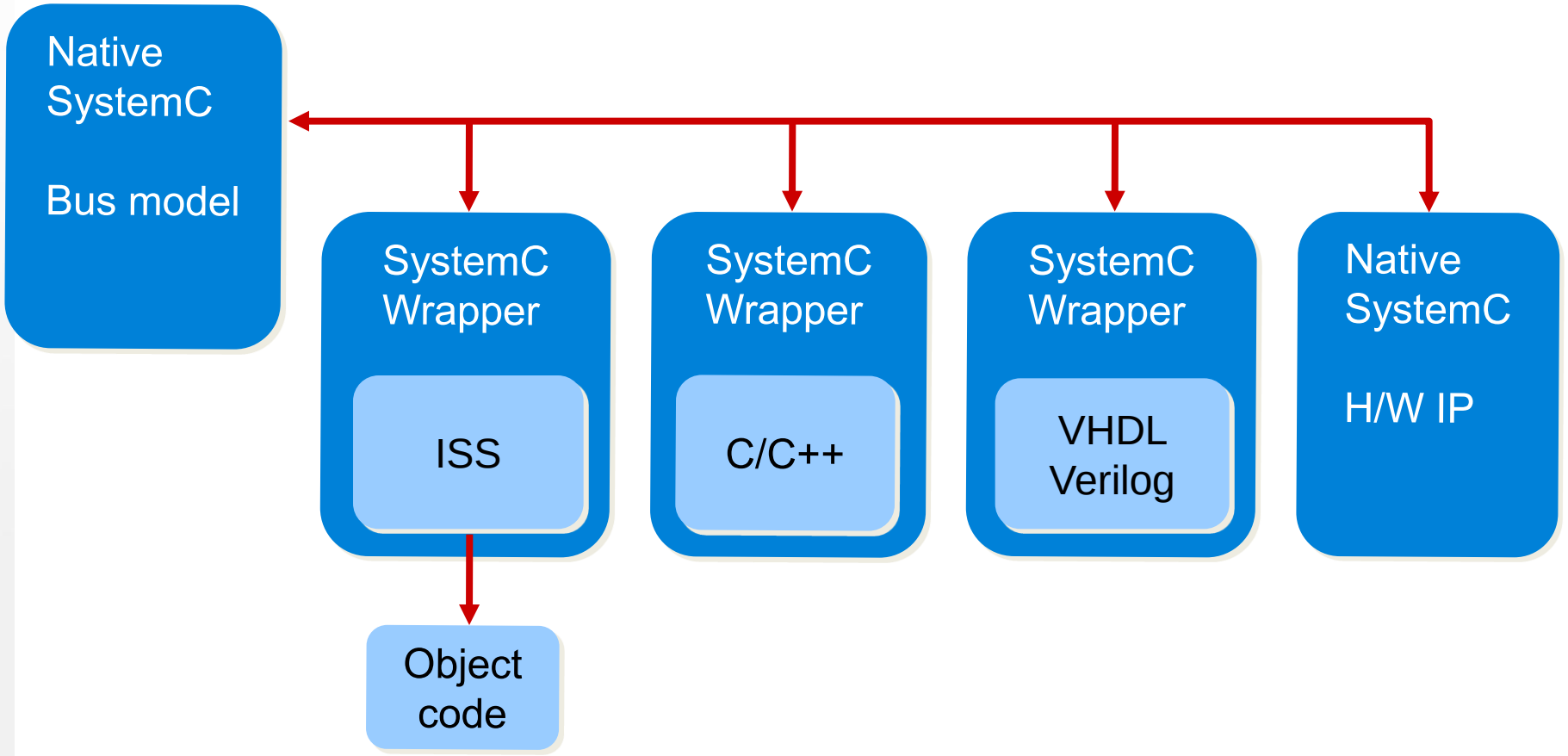


What is SystemC Used For?

- Virtual Platform
  - Architectural exploration, performance modeling
  - Software development
  - Reference model for functional verification
  - Available before RTL - **early!**
  - Simulates much faster than RTL - **fast!**
- High-level synthesis
- Used by
  - Architects, software, hardware, and verification engineers

SystemC is Glue!

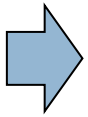
- Transaction-level modeling is communication-centric



Why SystemC in Particular?

- Industry standard IEEE 1666™
- Open-source C++ simulator
  - Available across multiple platforms (Linux, Windows, Mac)
  - No vendor lock-in, no licensing issues
- Easy integration with the C/C++ world
- Easy to put SystemC wrappers around existing HDL or C models
- Wide availability of tools, models and know-how
- Mature tool vendor support for co-simulation and debug

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);
```

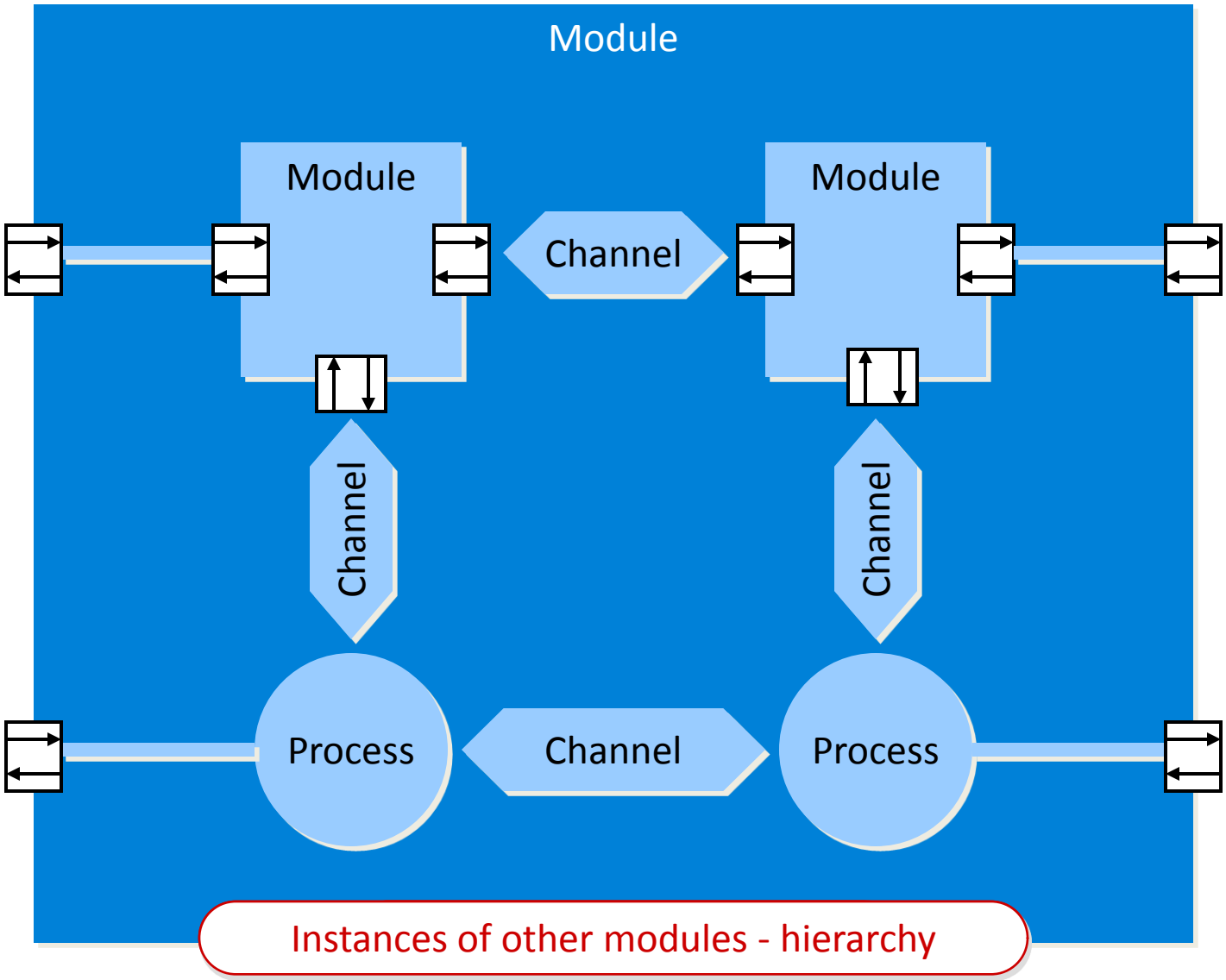
Word length	- number of stored bits - no limit
Integer word length	- number of bits before binary point
Quantization mode	- behavior when insufficient precision
Overflow mode	- behavior when result too big
Number of saturated bits	- used with wrap overflow modes

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



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

Implicit read/write and Delta delays


```
sc_in<int>      a, b;
```

```
SC_METHOD(action);
  sensitive << a << b;
```

- Implicit method calls

For convenience

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

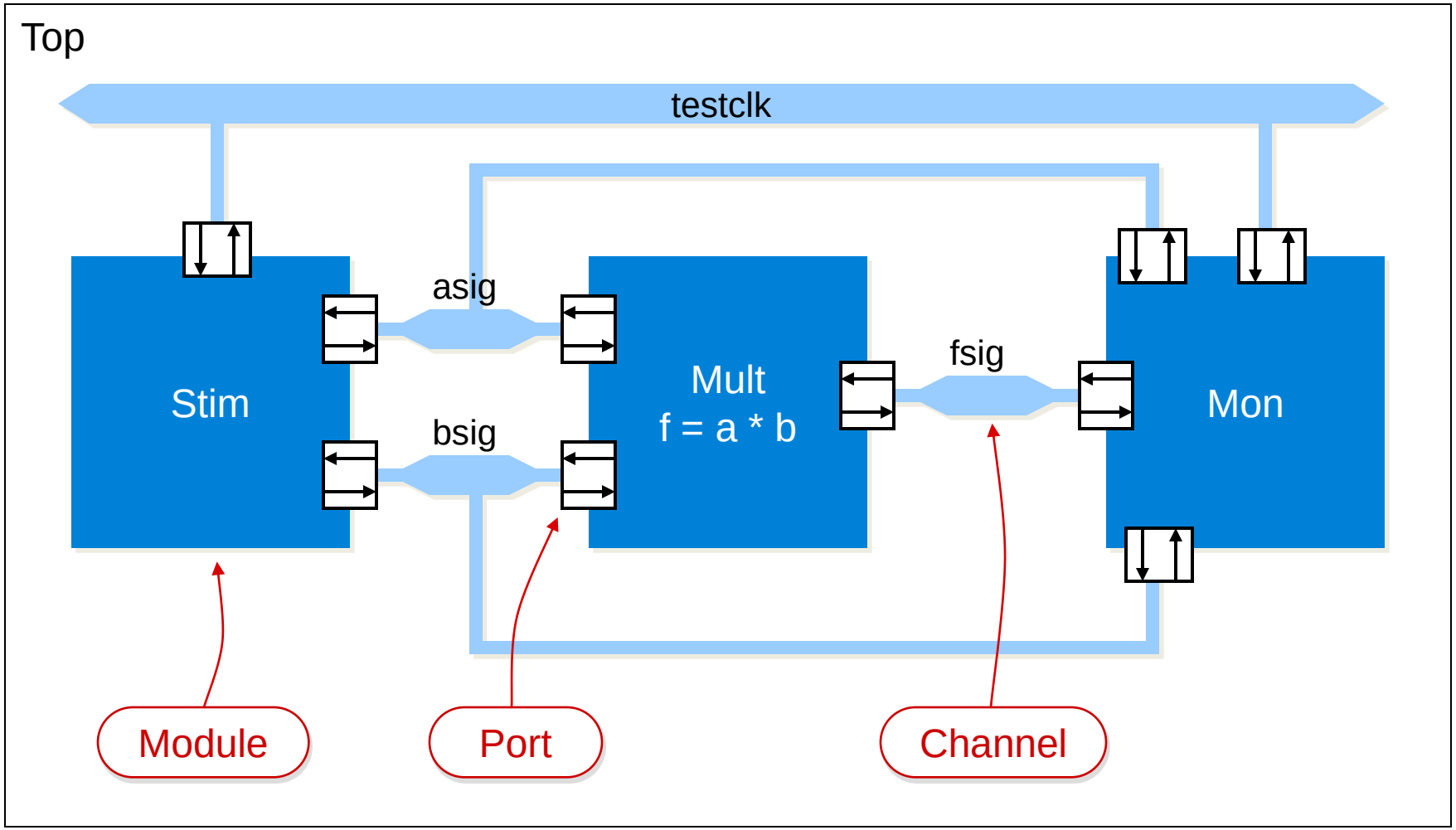
- Explicit method calls

write() schedules event with delta delay

```
f->write( a->read() * b->read() );

cout << f->read() << endl;
```

Old value!

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

An Alternative to sc_main

- A SystemC implementation is not obliged to support sc_main

```
#include "top.h"

SC_MODULE_EXPORT(Top);

// NCSC_MODULE_EXPORT(Top);
```

QuestaSim

Incisive

Tool-specific macro

Namespaces

Old header - global namespace

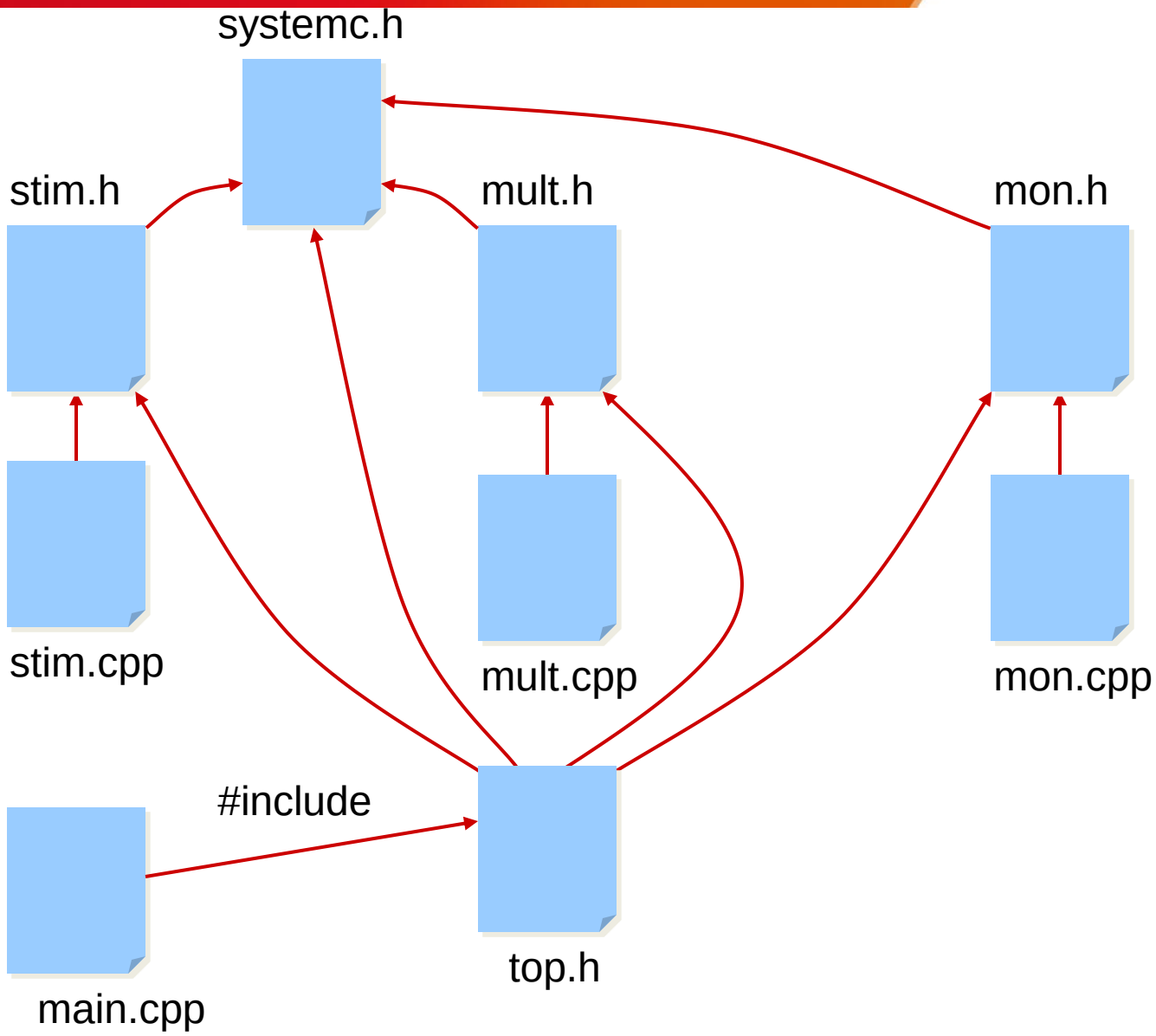
```
#include "systemc.h"

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

  ... cout << endl;
```

New header

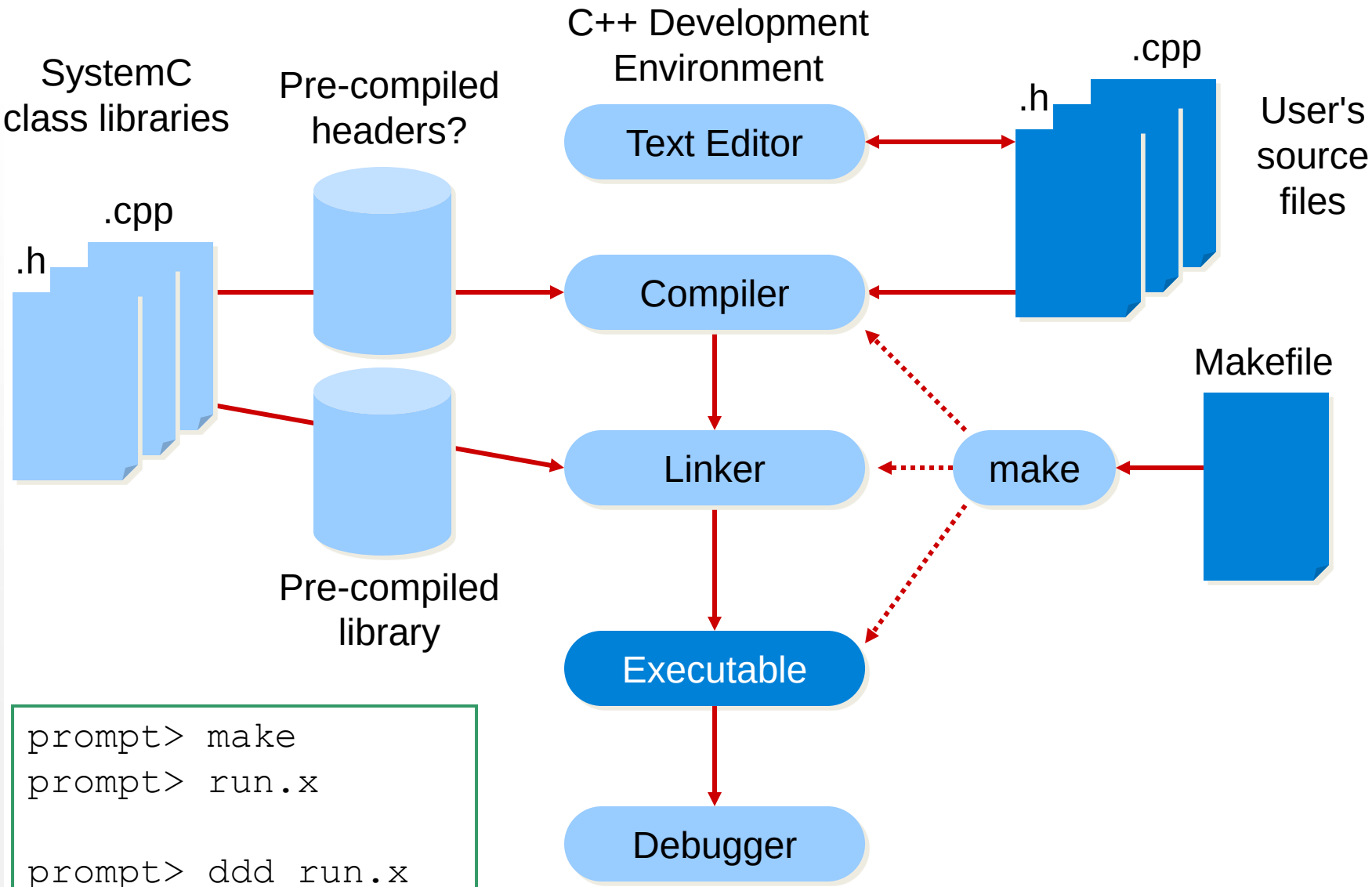
```
#include "systemc"

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

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

```
#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();
  A = -10;
  B = 23;
  wait();
  A = 25;
  B = -3;
  wait();
  sc_stop();
}
```

for Clk edge

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

```
#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

```
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);
```

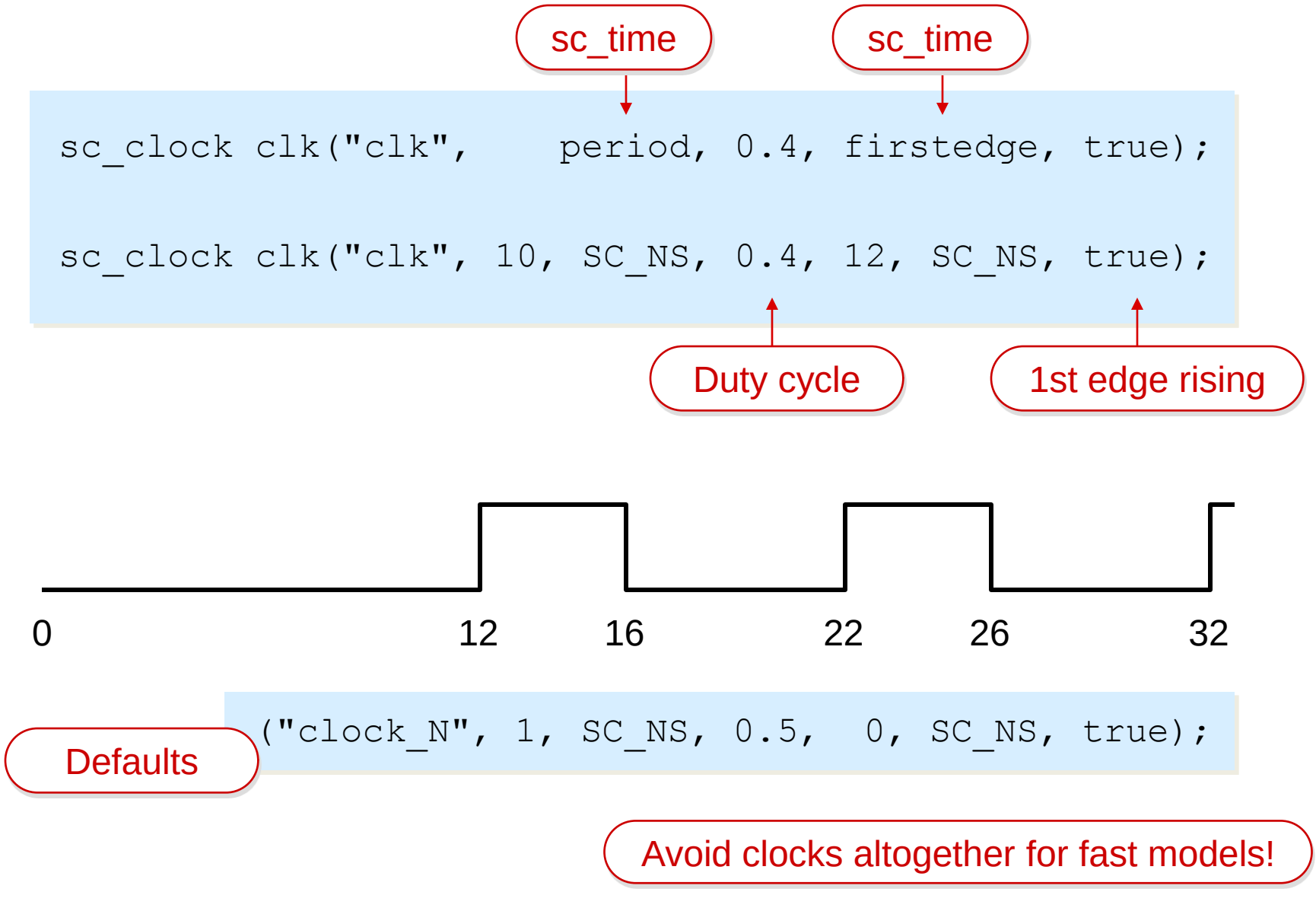
Constructor

```
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



Dynamic Processes


```
#define SC_INCLUDE_DYNAMIC_PROCESSES
```

PoC sim

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

```
void static_thread()
{
  wait(20, SC_NS);
  ...
  sc_spawn( &global_func );
  ...
}
```

```
SC_THREAD(static_thread);
```

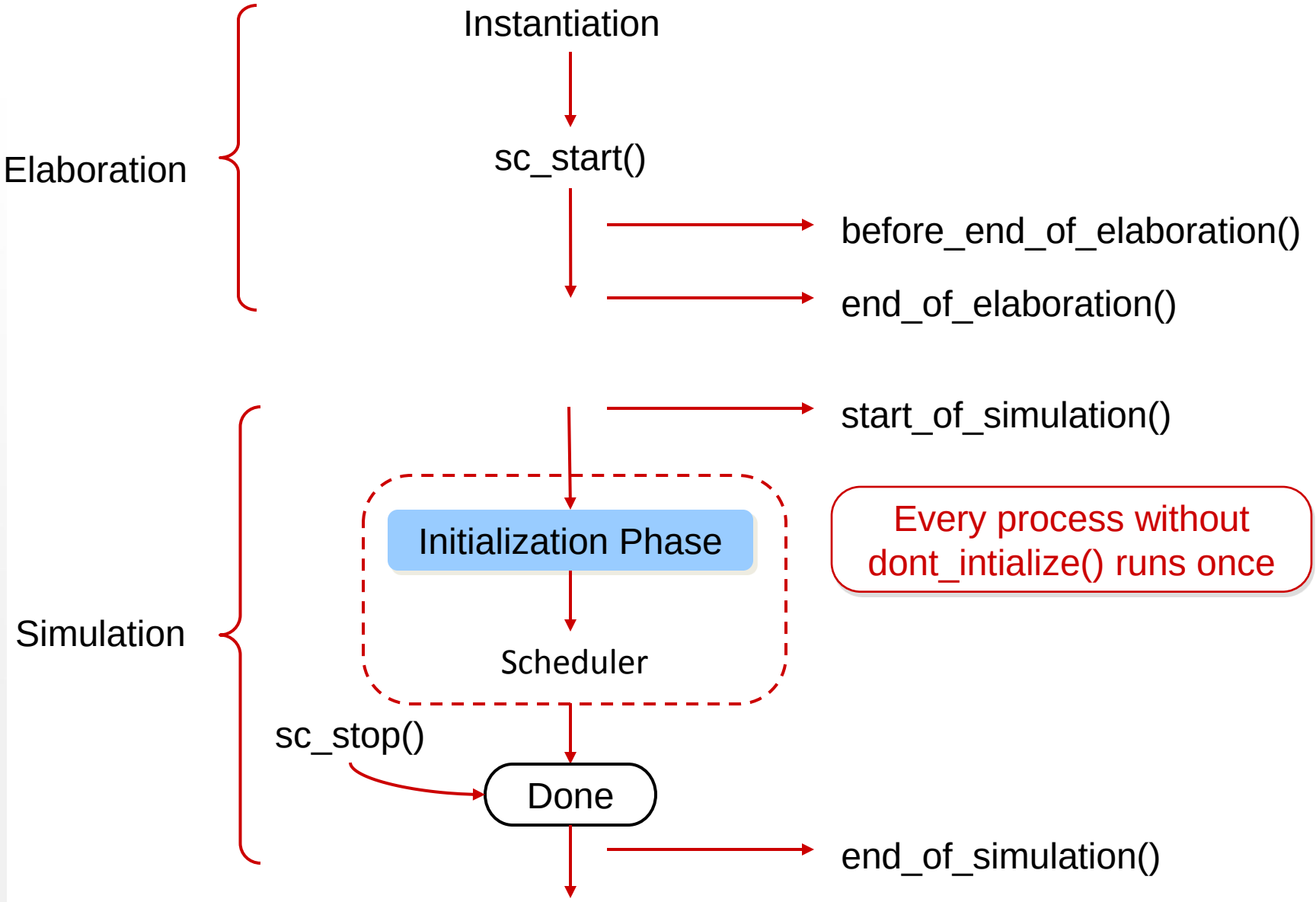
Spawning Member Functions


```
struct Mod: sc_module
{
  SC_CTOR(Mod)
  {
    SC_THREAD(T);

    sc_spawn( sc_bind( &Mod::proc, this) );
  }

  void T()
  {
    sc_spawn( sc_bind( &Mod::proc, this ) );
  }

  void proc ()
  {
    ...
  }
};
```

Spawn a member function as a static process

Spawn a member function as a dynamic process

Elaboration / Simulation Callbacks



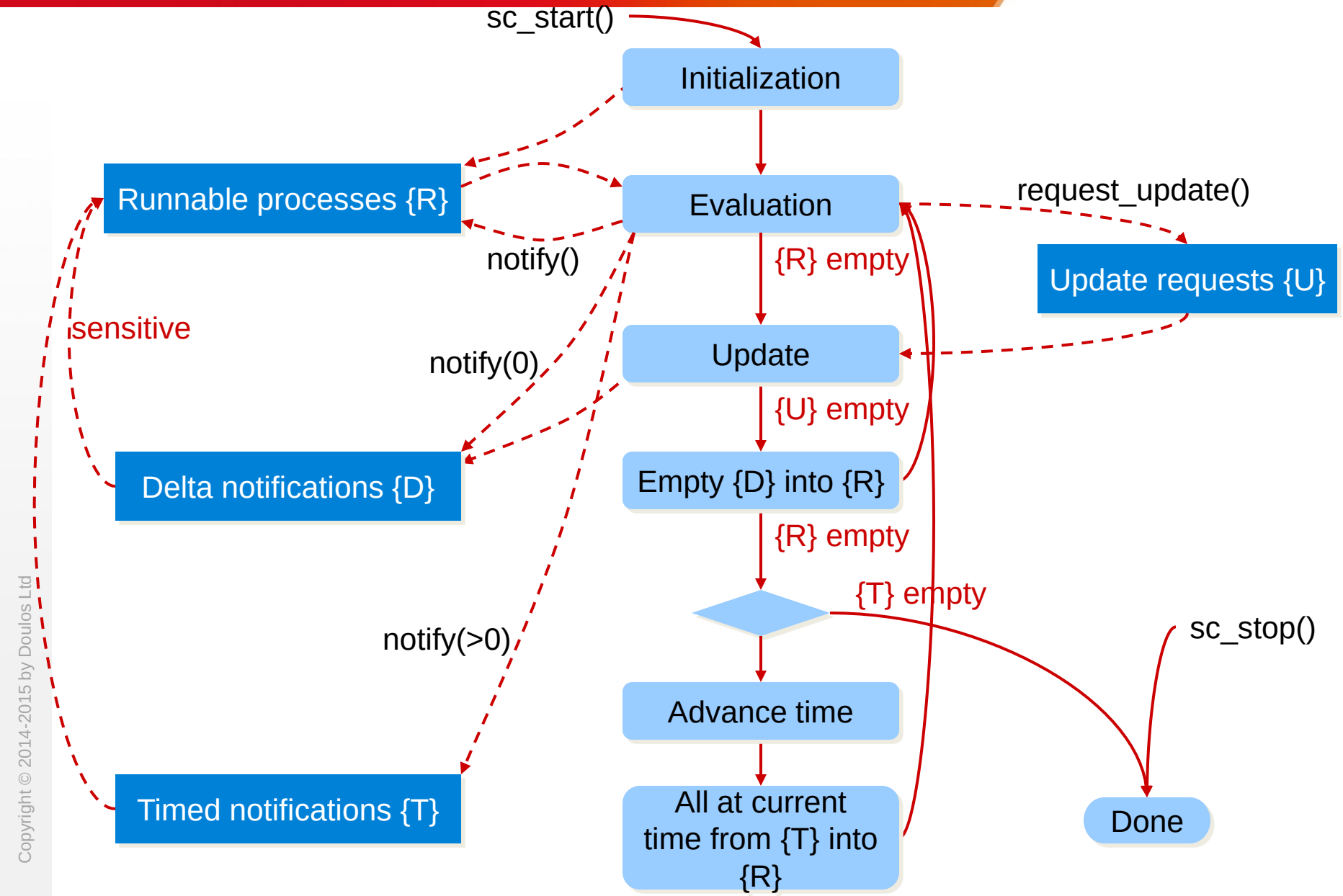
Overriding the Callbacks

- 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



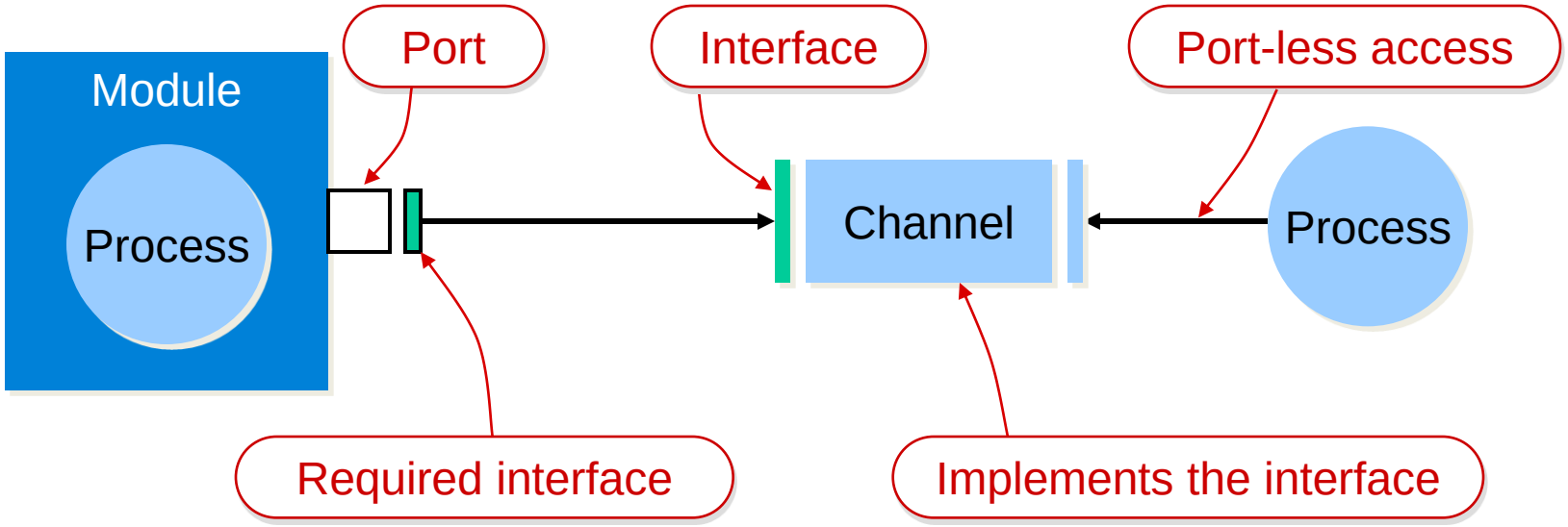
Kinds of Channel

- 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

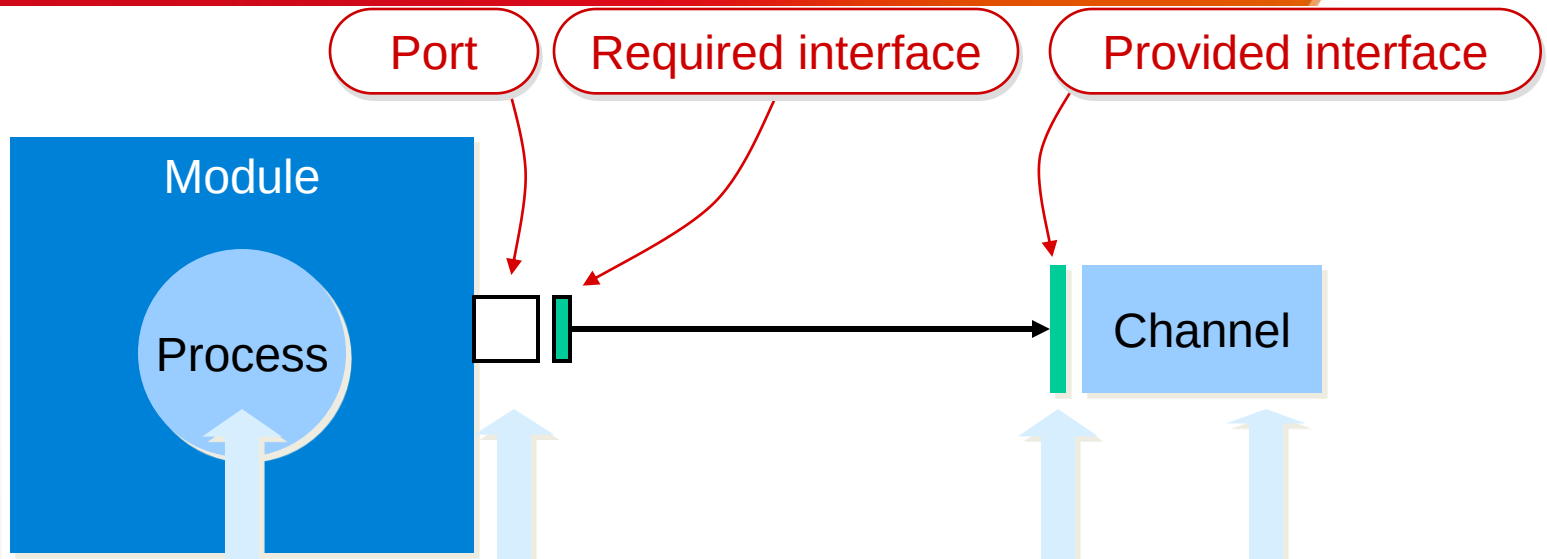
```
#include "systemc"
class queue_if : virtual public sc_core::sc_interface
{
public:
  virtual void write(char c) = 0;
  virtual char read() = 0;
};
```

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;
```

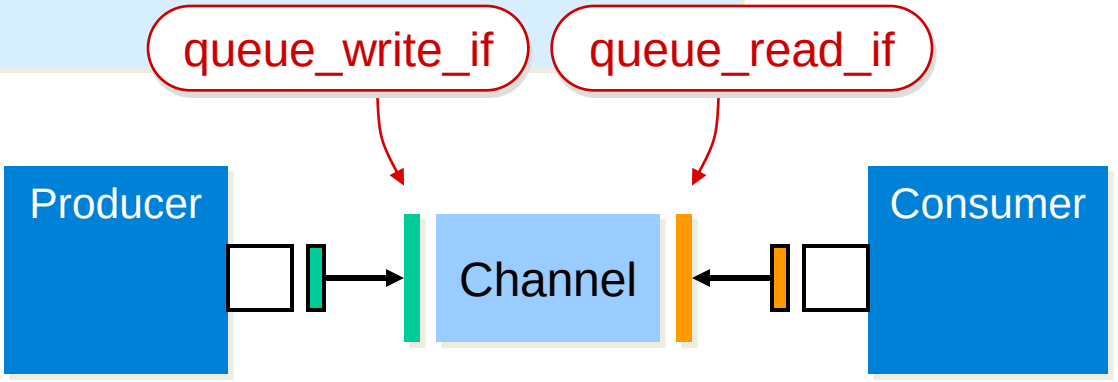
```
p->method();
```

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

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

Queue Ports

```
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

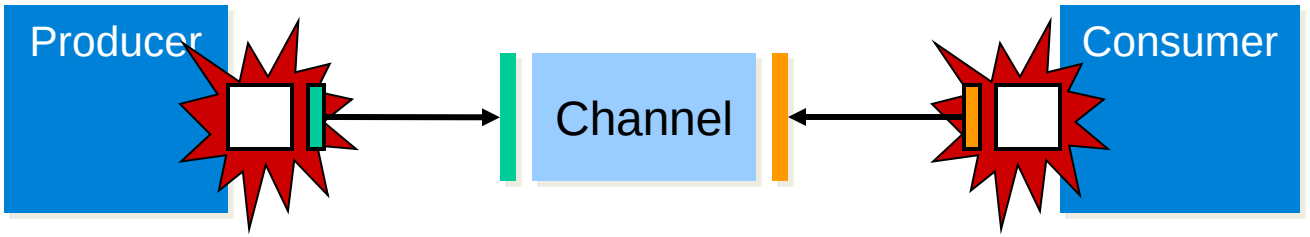
```
#include <systemc>
#include "producer.h"
using namespace sc_core;

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

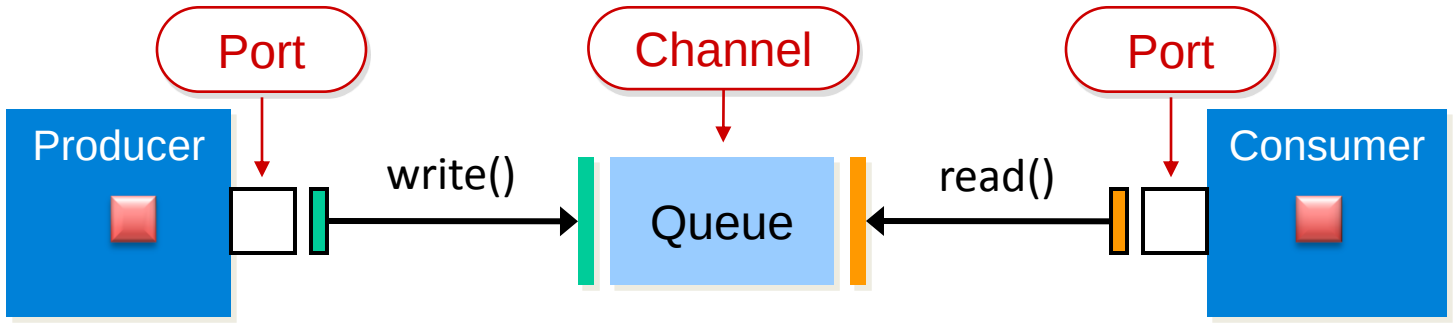
Interface Method Call

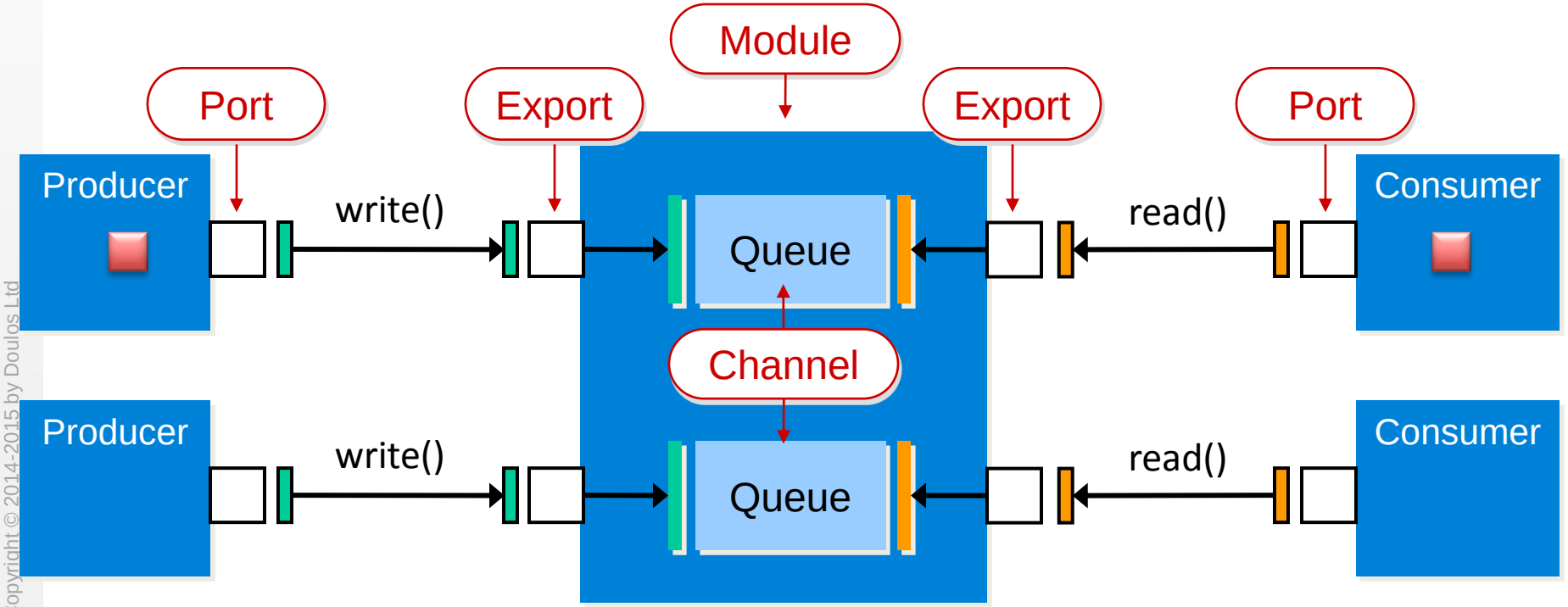
Why Ports?

- 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



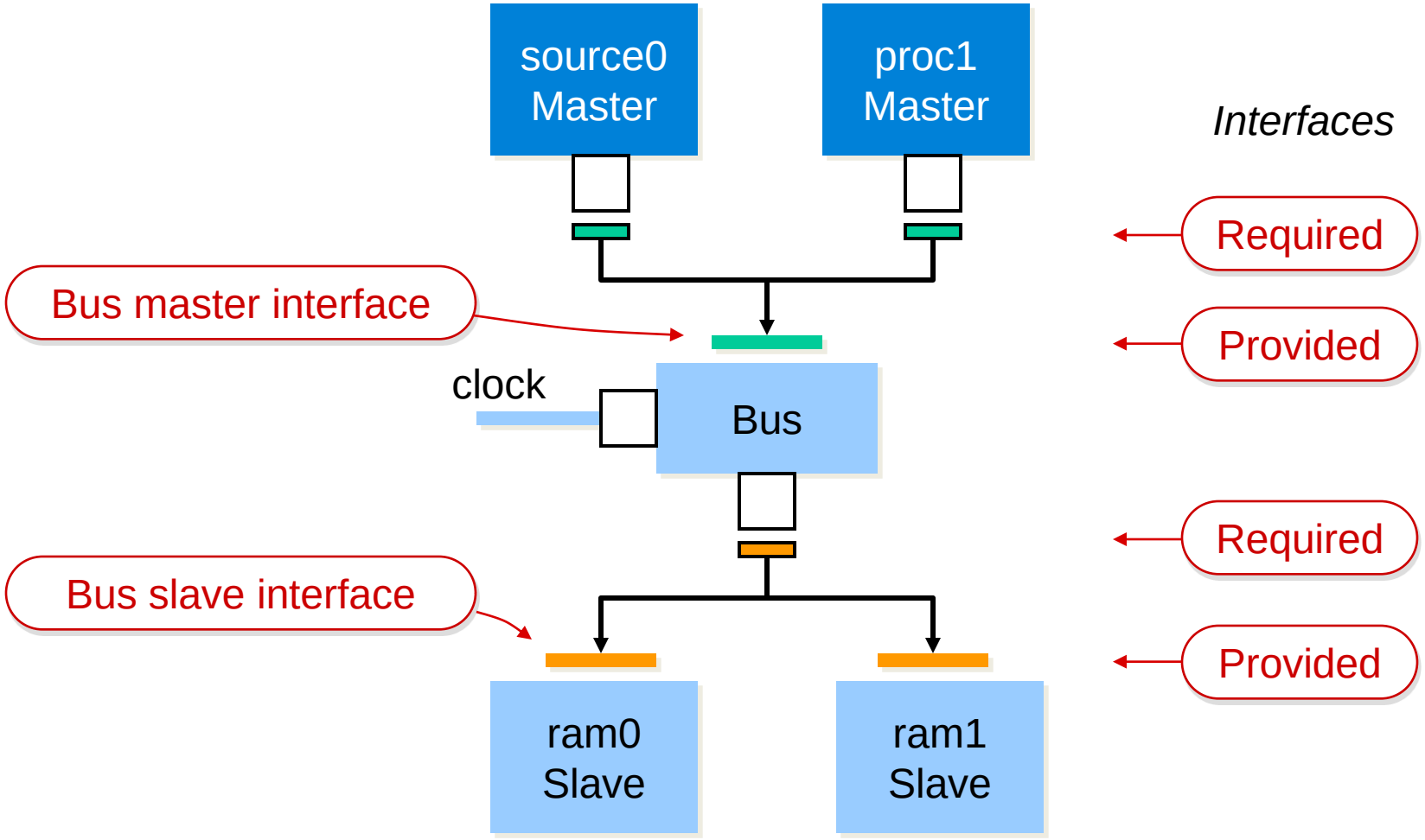


The SystemC Language

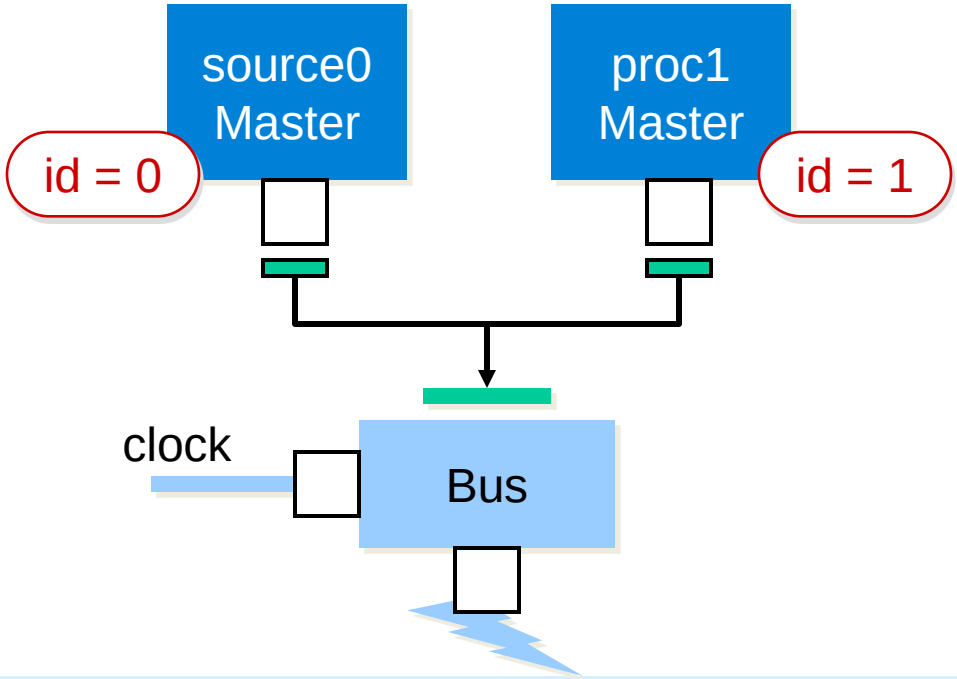
- Introduction to SystemC
- Core Concepts and Syntax
- Bus Modeling
  - Master and slave interfaces
  - Blocking versus non-blocking
  - Multiports
- Odds and Ends

Example Bus Model

Multiple bus masters (modules), shared bus (channel), multiple slaves (channels)
Bus arbitration and memory mapping built into the bus

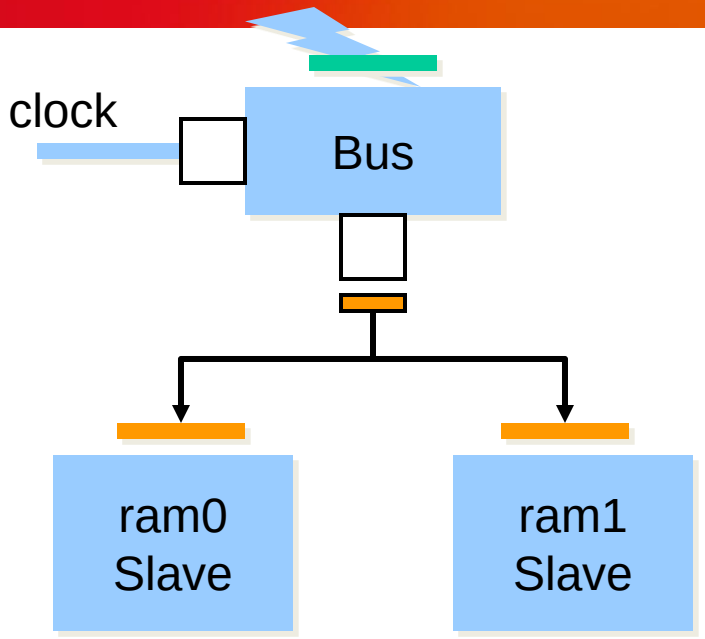


Master Interface Definition




```
class master_if : virtual public sc_interface
{
public:
  virtual void write(sc_uint<8> address, sc_uint<12> data,
                     int id) = 0;
  virtual void read (sc_uint<8> address, sc_uint<12> &data,
                     int id) = 0;
};
```

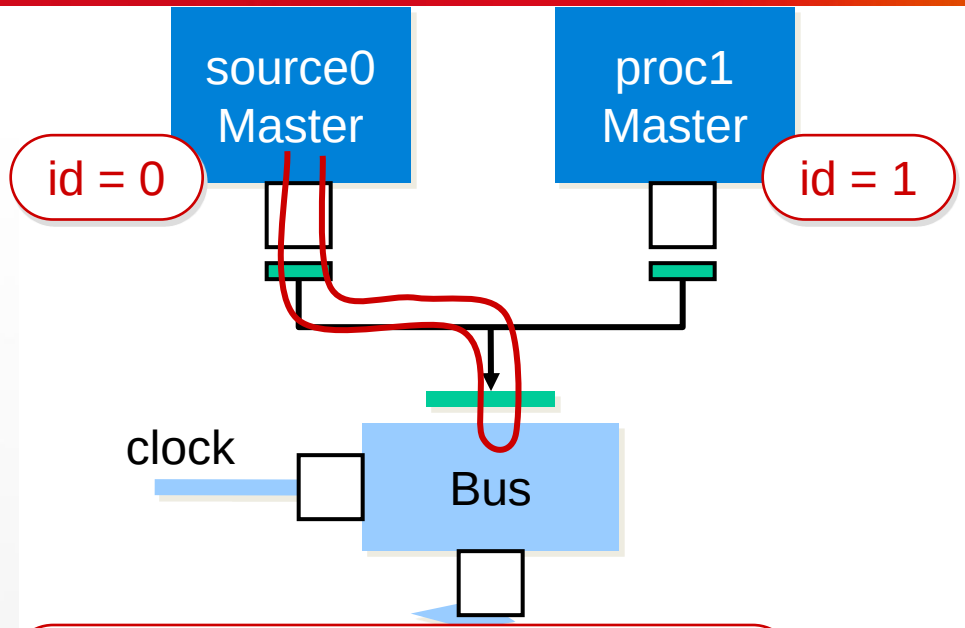
Slave Interface Definition




```
class slave_if : virtual public sc_interface
{
public:
  virtual void slave_write(sc_uint<8> address, sc_uint<12>  data) = 0;
  virtual void slave_read (sc_uint<8> address, sc_uint<12> &data) = 0;
  virtual void get_map(unsigned int &start, unsigned int &size)   = 0;
};
```

Memory map managed within bus channel

Master Write and Read



Runs in the context of the caller

Array of flags

request[0]	bool
request[1]	bool

Array of events

proceed[0]	sc_event
proceed[1]	sc_event

```
void Bus::write(sc_uint<8> address, sc_uint<12> data, int id)
{
  request[id] = true;          // request access to the bus
  wait(proceed[id]);           // wait for permission
  request[id] = false;         // clear the flag
  slave_port[find_port(address)]->slave_write(address, data);
}
```

Blocking and Non-blocking Calls

An Interface Method Call runs in the context of the caller

Important!

ASI terminology:

- A *blocking* method may call wait
- A *blocking* method must be called from a thread process

- A *non-blocking* method must not call wait
- A *non-blocking* method may be called from a thread or method process
- Naming convention nb_*

Bus Controller Process

```
void Bus::control_bus()
{
  int highest;
  for (;;)
  {
    wait(clock->posedge_event());

    // Pick out a master that's made a request
    highest = -1;
    for (int i = 0; i < n_masters; i++)
      if (request[i])
        highest = i;

    // Notify the master with the highest id
    if (highest > -1)
      proceed[highest].notify();
  }
}
```

The SystemC Language

- Introduction to SystemC
- Modules, Processes and Ports
- Bus Modeling
- Odds and Ends

Assignments in Multiple Processes


```
SC_CTOR(Test)
{
  SC_THREAD(p1);
  SC_THREAD(p2);
}

sc_logic v;

void p1()
{
  v = '0';
}
void p2()
{
  v = '1';
}
```

```
SC_CTOR(Test)
{
  SC_THREAD(p1);
  SC_THREAD(p2);
}

sc_signal<sc_logic> S;

void p1()
{
  S.write(sc_logic('0'));
}
void p2()
{
  S.write(SC_LOGIC_1);
}
```

Cast necessary

Built-in const

- In either case the value is indeterminate ('0' or '1')
- Indeed, writing a signal from > 1 process gives a run-time error

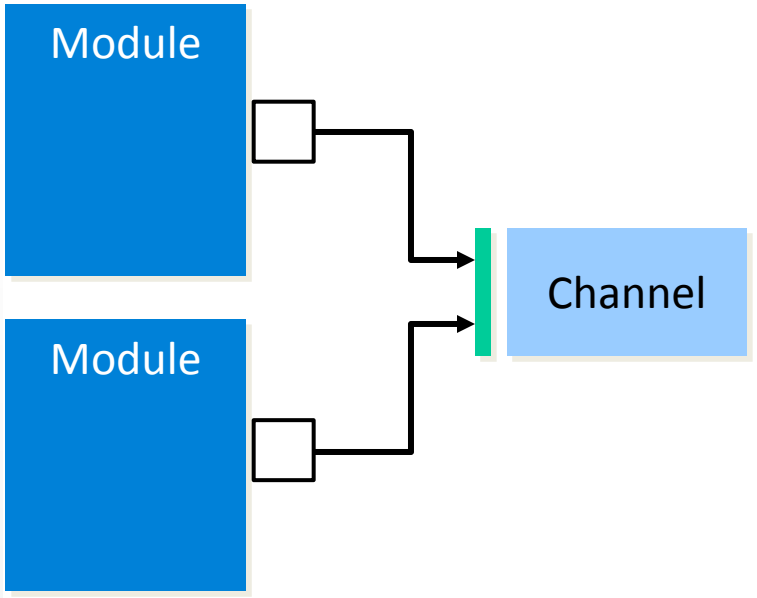
Bus Resolution


```
SC_CTOR(Test)
{
  SC_THREAD(p1);
  SC_THREAD(p2);
}

sc_signal_resolved S;

void p1()
{
  wait(10, SC_NS);
  S = sc_logic('0');
}
void p2()
{
  wait(20, SC_NS);
  S = SC_LOGIC_1;
  wait(10, SC_NS);
  S = SC_LOGIC_Z;
}
```

Resolved	X	0	1	Z
X	X	X	X	X
0	X	0	X	0
1	X	X	1	1
Z	X	0	1	Z

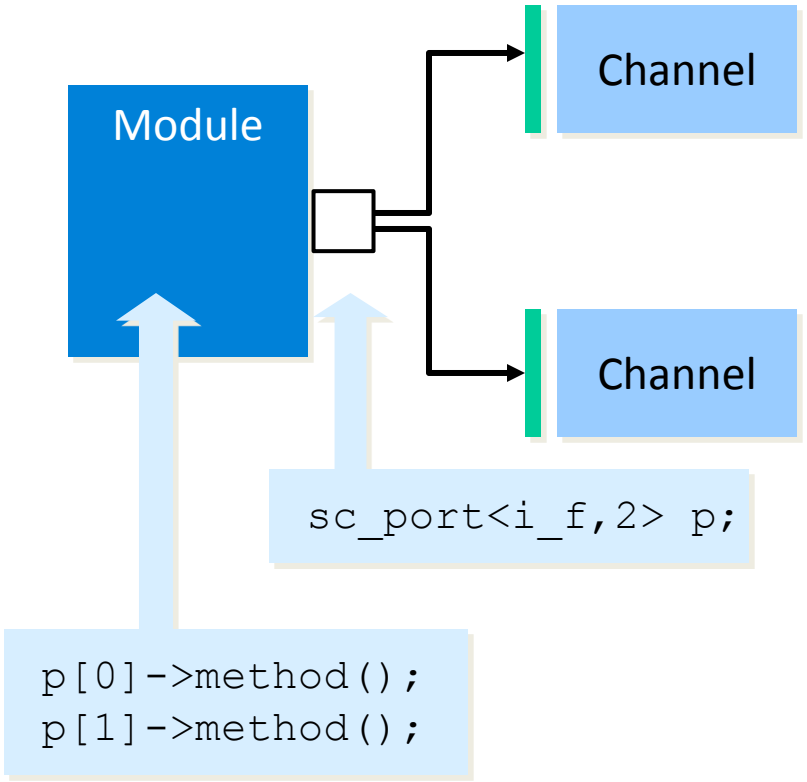
Time	p1	p2	S
0 ns	-	-	'X'
10 ns	'0'	-	'0'
20 ns	'0'	'1'	'X'
30 ns	'0'	'Z'	'0'

Multiple Bindings



Unsynchronised access to shared memory

sc_signal_resolved



Array ports

Resolved Channels and Ports

Resolved Channels

```
class sc_signal_resolved
: public sc_signal<sc_logic>
{...}
```

Ports

```
sc_in<>
sc_inout<>
sc_out<>
```

```
sc_in_resolved
sc_inout_resolved
sc_out_resolved
```

```
template <int W>
class sc_signal_rv
: public sc_signal<sc_lv<W> >
{...}
```

```
sc_in<>
sc_inout<>
sc_out<>
```

```
sc_in_rv<>
sc_inout_rv<>
sc_out_rv<>
```

Can only bind to resolved signals

Event Methods


```
sc_in<bool> clock;
```

Event finder needed - port not yet bound

```
sensitive << clock.pos();
```

sc_event_finder& pos() const;

```
if (clock.posedge())...
```

bool posedge() const;

```
wait(clock.posedge_event());
```

const sc_event& posedge_event() const;

bool and sc_logic

Return type	Method		
sc_event_finder&	value_changed()	pos()	neg()
bool	event()	posedge()	negedge()
sc_event&	value_changed_event()	posedge_event()	negedge_event()

Accessing the command line

- Not always able to access command-line from sc_main
- sc_argc() and sc_argv()[] provide universal access

```
for (int i = 1; i < sc_argc(); ++i) {
  string arg(sc_argv()[i]);
  if ( arg == "-flag" ) {
    flag = true;
  } else if ( arg == "-num"
           && i+1 < sc_argc()){
    number = atoi(sc_argv()[++i]);
}
```

- Alternatives include
  - read from file
  - `cstr = getenv("ENVIRONMENT_VARIABLE");//may be nullptr`

Debugging

- Stream I/O: operator<< overloaded for built-in types

```
cout << "data=" << data << endl;
```

- Error reporting

```
SC_REPORT_INFO("msg_type", "message");
```

- Writing trace files for waveform viewers

```
sc_trace(trace_file, data, "data");
```

- Software debuggers and EDA tools

```
ddd run.x
```

Error Reporting


```
SC_REPORT_INFO    ("msg_type", "msg");
SC_REPORT_WARNING("msg_type", "msg");
SC_REPORT_ERROR   ("msg_type", "msg");
SC_REPORT_FATAL   ("msg_type", "msg");
```

Prints msg_type + msg

also file, line, process, time

also throws exception

calls abort()

Handling Reports

- Use the macros

```
SC_REPORT_WARNING("", "Just a friendly warning");
SC_REPORT_ERROR  ("Me", "Big trouble");
SC_REPORT_FATAL  ("Me", "You're dead");
```

Message itself

Message type = origin of message

- Customize the report handler

```
sc_report_handler::set_actions(SC_WARNING, SC_DO_NOTHING);
sc_report_handler::set_actions("Me", SC_DISPLAY);
sc_report_handler::set_actions("Me", SC_WARNING, SC_LOG);
sc_report_handler::set_log_file_name("log.txt");
sc_report_handler::stop_after(SC_WARNING, 5);
```

Message and Message Type


```
#include <sstream>

...

ostringstream oss;

oss << "Big trouble, address = "

    << addr << " data = " << data << ends;

SC_REPORT_ERROR("/DOULOS/SUPERBUS", oss.str().c_str());
```

Message type = hierarchical vendor string

- e.g. report from Proof-of-Concept simulator

```
Info: (I804) /IEEE_Std_1666/deprecated:
sc_start(double) deprecated, use sc_start(sc_time) or sc_start()
```

- Suppress all such reports

```
sc_report_handler::set_actions("/IEEE_Std_1666/deprecated",

                               SC_DO_NOTHING);
```

Actions and Severity

Actions	Behaviour of the default handler
SC_UNSPECIFIED	No action
SC_DO_NOTHING	No action but inhibit lower priority actions
SC_THROW	Throw exception
SC_LOG	Write message to log file
SC_DISPLAY	Write message to standard output
SC_CACHE_REPORT	Cache the report
SC_INTERRUPT	Call sc_interrupt_here()
SC_STOP	Call sc_stop()
SC_ABORT	Call abort()

Default actions

Each action is a bit mask – can be OR'd

```
SC_INFO    : SC_LOG | SC_DISPLAY
SC_WARNING: SC_LOG | SC_DISPLAY
SC_ERROR   : SC_LOG | SC_CACHE_REPORT | SC_THROW
SC_FATAL   : SC_LOG | SC_DISPLAY | SC_CACHE_REPORT | SC_ABORT
```

Accellera Systems Initiative

Current Accellera Board Members

- AMD
- ARM
- Cadence Design Systems Inc
- Freescale Semiconductor
- Intel Corporation
- Mentor Graphics
- NXP Semiconductors
- Qualcomm Inc
- Renesas Mobile
- ST Microelectronics
- Synopsys
- Texas Instruments

Website http://www.accellera.org

European and North American SystemC Users Groups (and others)

- http://www-ti.informatik.uni-tuebingen.de/~systemc
- http://www.nascug.org

For More FREE Information

- IEEE 1666

standards.ieee.org/getieee/1666/download/1666-2011.pdf

- ASI SystemC 2.3.1

www.accellera.org

- On-line tutorials

www.doulos.com/knowhow/systemc

Hardware Design

» VHDL » Verilog » SystemVerilog

Embedded Systems and ARM

» C » C++ » UML » RTOS » Linux » Yocto
» ARM Cortex » Python » Android » OpenCL » OpenGL

ESL & Verification

» SystemC » TLM-2.0 » SystemVerilog
» OVM/VMM/UVM » Perl » Tcl/Tk