Plan

- Ch1 – Overview of System Design Using SystemC
- Ch2 – Overview of SystemC
- Ch3 – Data Types
- Ch4 – Modules
- Ch5 – Notion of Time
- Ch6 – Concurrency
- Ch7 – Predefined Channels
- Ch8 – Structure
- Ch9 – Communication
- Ch10 – Custom Channels and Data
- Ch11 – Transaction Level Modeling

Custom Channels and Data

- Custom Primitive Channel
- Custom Data Type
- Custom Hierarchical Channel
- Adaptor / Transactor

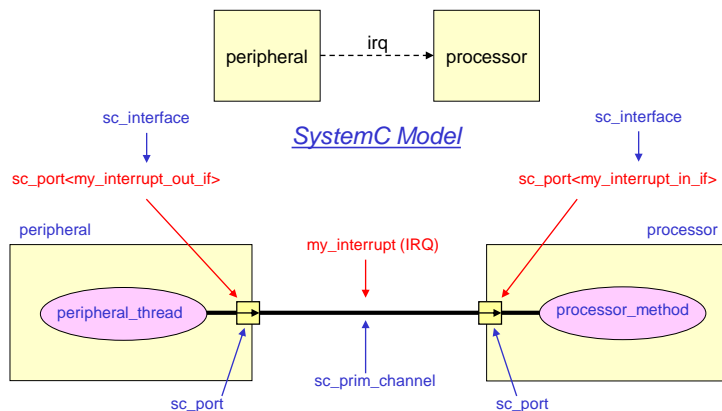| Predefined Primitive Channels (Mutexs, FIFOs, Signals)			
Simulation Kernel	Threads & Methods	**Channels & Interfaces**	Data types Logic, Integers, Fixed point
	Events, Sensitivity & Notification	Modules & Hierarchy	

1

Review of Interface & Channels

- Interface
    - makes a channel usable with ports
    - enables the separation of communication from processing
- Primitive Channel
    - inherits from sc_prim_channel (Chapter 7)
    - no hierarchy, no port
    - no SC_METHODs or SC_THREADs
    - ability to implement the evaluate-update paradigm
    - simple and fast communications
    - built-in channel (sc_fifo, sc_mutex, sc_semaphore, sc_signal)
- Hierarchical Channel
    - inherits from sc_channel
    - accesses ports
    - contains process(es), hierarchy
    - complex communications buses (PCI, AMBA AXI …)
- Channels are important
    - suitable channels enable safe communication between processes
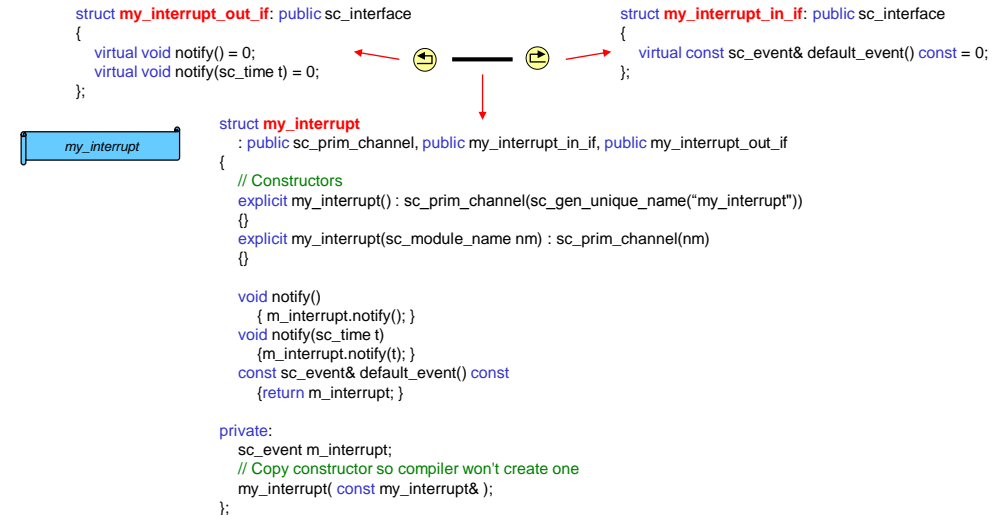   - channels with ports clarify the relationships of communication from processing

Custom Primitive Channel

- Problem : connect an event or interrupt between 2 modules
    - OK for 2 processes … (Chapter 6 – Concurrency)
    - side effect : we can use sc_signal<bool> !
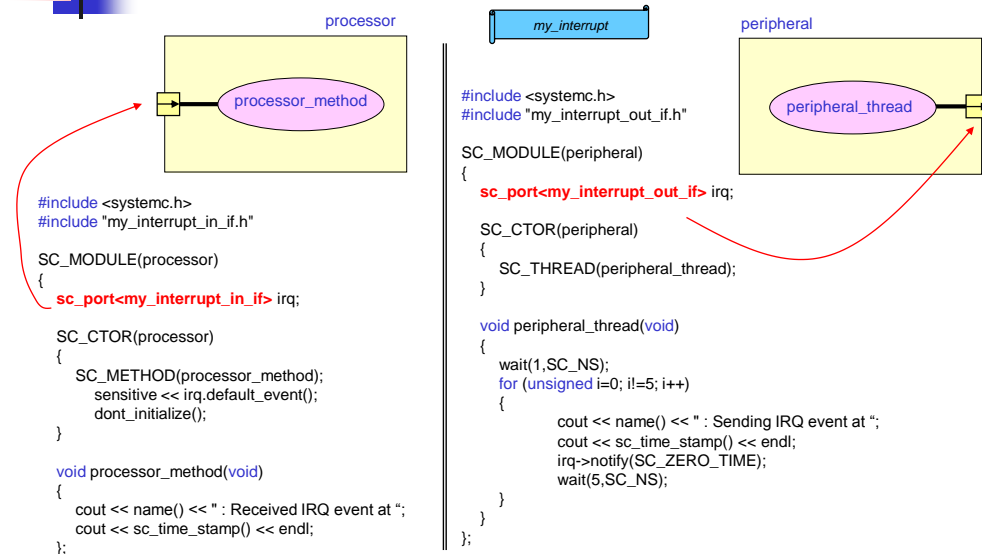
2

Interrupt Example
Define Primitive Channel

```
struct my_interrupt_out_if: public sc_interface          struct my_interrupt_in_if: public sc_interface
{                                                         {
    virtual void notify() = 0;                                virtual const sc_event& default_event() const = 0;
    virtual void notify(sc_time t) = 0;                   };
};
```

my_interrupt

```
struct my_interrupt
    : public sc_prim_channel, public my_interrupt_in_if, public my_interrupt_out_if
{
    // Constructors
    explicit my_interrupt() : sc_prim_channel(sc_gen_unique_name("my_interrupt"))
    {}
    explicit my_interrupt(sc_module_name nm) : sc_prim_channel(nm)
    {}

    void notify()
        { m_interrupt.notify(); }
    void notify(sc_time t)
        {m_interrupt.notify(t); }
    const sc_event& default_event() const
        {return m_interrupt; }

private:
    sc_event m_interrupt;
    // Copy constructor so compiler won't create one
    my_interrupt( const my_interrupt& );
};
```

Interrupt Example
Modules

processor

processor_method

my_interrupt

peripheral

peripheral_thread

```
#include <systemc.h>
#include "my_interrupt_in_if.h"

SC_MODULE(processor)
{
    sc_port<my_interrupt_in_if> irq;

    SC_CTOR(processor)
    {
        SC_METHOD(processor_method);
            sensitive << irq.default_event();
            dont_initialize();
    }

    void processor_method(void)
    {
        cout << name() << " : Received IRQ event at ";
        cout << sc_time_stamp() << endl;
};
```

```
#include <systemc.h>
#include "my_interrupt_out_if.h"

SC_MODULE(peripheral)
{
    sc_port<my_interrupt_out_if> irq;

    SC_CTOR(peripheral)
    {
        SC_THREAD(peripheral_thread);
    }

    void peripheral_thread(void)
    {
        wait(1,SC_NS);
        for (unsigned i=0; i!=5; i++)
        {
            cout << name() << " : Sending IRQ event at ";
            cout << sc_time_stamp() << endl;
            irq->notify(SC_ZERO_TIME);
            wait(5,SC_NS);
        }
    }
};
```

3

test_interrupt.h

```
#include <systemc.h>
#include "processor.h"
#include "peripheral.h"
#include "my_interrupt.h"
#include "my_interrupt_in_if.h"
#include "my_interrupt_out_if.h"

SC_MODULE(test_interrupt)
{
  my_interrupt* irq;
  processor* processor_i;
  peripheral* peripheral_i;

  SC_HAS_PROCESS(test_interrupt);
  test_interrupt(sc_module_name nm);
};
```

my_interrupt

test_interrupt.cpp

```
#include "test_interrupt.h"

test_interrupt::test_interrupt(sc_module_name nm)
{
  irq = new my_interrupt();
  processor_i = new processor("processor_i");
  peripheral_i = new peripheral("peripheral_i");
  processor_i->irq(*irq);
  peripheral_i->irq(*irq);
}
```

main.cpp

```
#include <systemc.h>
#include "test_interrupt.h"

int sc_main(int argc, char* argv[])
{
  cout << "INFO: Elaborating " << endl;
  test_interrupt interrupt_i("interrupt_i");
  cout << "INFO: Simulating " << endl;
  sc_start();
  cout << "INFO: Post-processing " << endl;

  return 0;
}
```

```
INFO: Elaborating
INFO: Simulating
interrupt_i.peripheral_i : Sending IRQ event at 1 ns
interrupt_i.processor_i : Received IRQ event at 1 ns
interrupt_i.peripheral_i : Sending IRQ event at 6 ns
interrupt_i.processor_i : Received IRQ event at 6 ns
interrupt_i.peripheral_i : Sending IRQ event at 11 ns
interrupt_i.processor_i : Received IRQ event at 11 ns
interrupt_i.peripheral_i : Sending IRQ event at 16 ns
interrupt_i.processor_i : Received IRQ event at 16 ns
interrupt_i.peripheral_i : Sending IRQ event at 21 ns
interrupt_i.processor_i : Received IRQ event at 21 ns
INFO: Post-processing
```
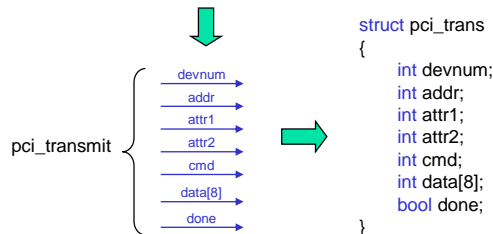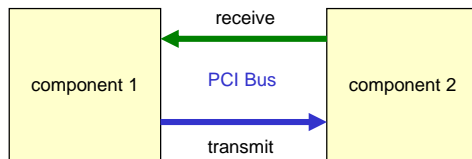

Ecole **polytechnique**
de l'université de Nice-Sophia Antipolis
Département Electronique

Custom Channels and Data

- Custom Primitive Channel
- Custom Data Type
- Custom Hierarchical Channel
- Adaptor / Transactor

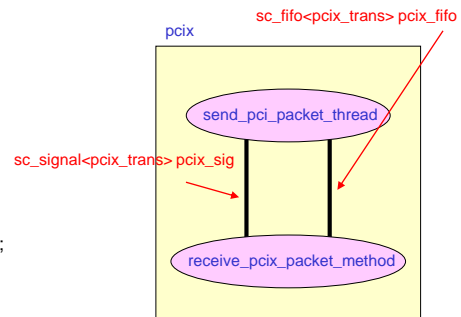| Predefined Primitive Channels (Mutexs, FIFOs, Signals)			
Simulation Kernel	Threads & Methods	**Channels & Interfaces**	Data types Logic, Integers, Fixed point
	Events, Sensitivity & Notification	Modules & Hierarchy	

4

- Transmit a packet on the PCI-X Bus

SystemC Model

receive

component 1

PCI Bus

component 2

transmit

pcix

sc_fifo<pcix_trans> pcix_fifo

send_pci_packet_thread

sc_signal<pcix_trans> pcix_sig

receive_pcix_packet_method

pci_transmit
- devnum
- addr
- attr1
- attr2
- cmd
- data[8]
- done

struct pci_trans
{
    int devnum;
    int addr;
    int attr1;
    int attr2;
    int cmd;
    int data[8];
    bool done;
}

pcix_trans.h

```
struct pcix_trans
{
  int  devnum;
  int  addr;
  int  attr1;
  int  attr2;
  int  cmnd;
  int  data[8];
  bool done;
```

pci_datatype

PCI signals

```
  // Constructors
  pcix_trans()
    : devnum(-1), addr(-1), cmnd(-1),
      attr1(-1), attr2(-1), done(false)
  {
    for (unsigned i=0; i!=8; i++) data[i] = 0;
  }

  pcix_trans(int _devnum, int _addr,
        int _attr1, int _attr2, int _cmnd, int* _data,
        int _done)
    : devnum(_devnum), addr(_addr), cmnd(_cmnd)
      ,attr1(_attr1), attr2(_attr2), done(_done)
  {
    for (unsigned i=0; i!=8; i++) data[i] = _data[i];
  }
```

```
  // Required by sc_signal<> and sc_fifo<>
  pcix_trans& operator= (const pcix_trans& rhs);

  // Required by sc_signal<>
  bool operator== (const pcix_trans& rhs) const;
};

// Stream operator to output a PCIX transaction packet to terminal
ostream& operator<<(ostream& file, const pcix_trans& trans);

// Trace a PCIX transaction packet in case it is used in an sc_signal
void sc_trace(sc_trace_file*& tf, const pcix_trans& trans, sc_string nm);
```

5

```cpp
//********* Member Functions *********
// Required by sc_signal<> and sc_fifo<>
pcix_trans& pcix_trans::operator= (const pcix_trans& rhs)
{
    devnum = rhs.devnum;
    addr   = rhs.addr;
    attr1  = rhs.attr1;
    attr2  = rhs.attr2;
    cmnd   = rhs.cmnd;
    for (unsigned i=0; i!=8; i++)
        data[i] = rhs.data[i];
    done   = rhs.done;
    return *this;
}

// Required by sc_signal<>
bool pcix_trans::operator== (const pcix_trans& rhs) const
{
    return (
        devnum == rhs.devnum && addr   == rhs.addr  &&
        attr1  == rhs.attr1  && attr2  == rhs.attr2 &&
        cmnd   == rhs.cmnd   && data[0] == rhs.data[0] &&
        data[1] == rhs.data[1] && data[2] == rhs.data[2] &&
        data[3] == rhs.data[3] && data[4] == rhs.data[4] &&
        data[5] == rhs.data[5] && data[6] == rhs.data[6] &&
        data[7] == rhs.data[7] && done   == rhs.done
        );
}
```

pci_datatype

```cpp
//********* Non-Member Functions *********
// Print a PCIX transaction packet out to a stream (usually just the terminal
// window), in a nice-looking format
ostream& operator<<(ostream& os, const pcix_trans& trans)
{
    os << "{" << endl
        << " cmnd: " << trans.cmnd  << ", "
        << "attr1:" << trans.attr1  << ", "
        …
        <<    "." << trans.data[4] << "."
        << "," << endl
        << " done:" << (trans.done?"true":"false") << endl
        << "}";
    return os;
}

// trace function, only required if actually used
void sc_trace(sc_trace_file*& tf, const pcix_trans& trans, sc_string nm)
{
    sc_trace(tf, trans.devnum,  nm + ".devnum");
    sc_trace(tf, trans.addr,    nm + ".addr");
    sc_trace(tf, trans.attr1,   nm + ".attr1");
    sc_trace(tf, trans.attr2,   nm + ".attr2");
    sc_trace(tf, trans.cmnd,    nm + ".cmnd");
    sc_trace(tf, trans.data[0], nm + ".data[0]");
    …
    sc_trace(tf, trans.data[7], nm + ".data[7]");
    sc_trace(tf, trans.done,    nm + ".done");
}
```

```cpp
SC_MODULE(pcix)
{
    sc_signal<pcix_trans> pcix_sig;
    sc_fifo<pcix_trans> pcix_fifo;

    SC_CTOR(pcix)
    {
        SC_THREAD(send_pcix_packet_thread);
        SC_METHOD(receive_pcix_packet_method);
            sensitive << pcix_sig;
            dont_initialize();
    }

    void send_pcix_packet_thread(void);
    void receive_pcix_packet_method(void);
};
```
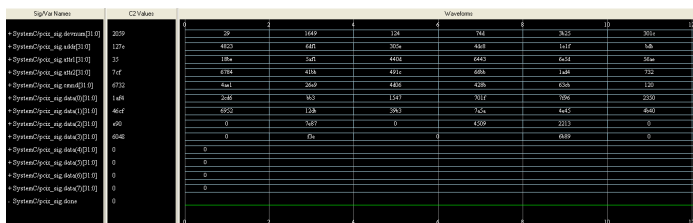
pci_datatype

```
Elaborating ...
Simulating ...
0 s New PCIX transaction {
  cmnd: 19169, attr1:6334, attr2:26500,
  devnum:41, addr:18467,
  data: 11478.26962.0.0.0.0.0.0,
  done:false
}
2 ns New PCIX transaction {
  cmnd: 9961, attr1:23281, attr2:16827,
  devnum:5705, addr:28145,
  data: 2995.4827.32391.3902.0.0.0.0,
  done:false
}
4 ns New PCIX transaction {
  cmnd: 19718, attr1:17421, attr2:18716,
  devnum:292, addr:12382,
  data: 5447.14771.0.0.0.0.0.0,
  done:false
}
6 ns New PCIX transaction {
  cmnd: 17035, attr1:25667, attr2:26299,
  devnum:1869, addr:19912,
  data: 28703.31322.17673.0.0.0.0.0,
  done:false
}
8 ns New PCIX transaction {
  cmnd: 25547, attr1:28253, attr2:6868,
  devnum:15141, addr:7711,
  data: 32662.20037.8723.27529.0.0.0.0,
  done:false
}
…
```

Wave of Signal

6

Custom Channels and Data

- Custom Primitive Channel
- Custom Data Type
- **Custom Hierarchical Channel**
- Adaptor / Transactor

Predefined Primitive Channels (Mutexs, FIFOs, Signals)			
Simulation Kernel	Threads & Methods	**Channels & Interfaces**	Data types Logic, Integers, Fixed point
	Events, Sensitivity & Notification	Modules & Hierarchy	

Hierarchical Channel

- To model complex buses
- inherit from sc_channel
    - sc_channel is just a sc_module
- Body of sc_channel
    - Ports
    - Member channel instances (sub-channel)
    - Member data instance
    - Constructor
    - Destructor
    - Process member functions (processes)
    - Helper functions
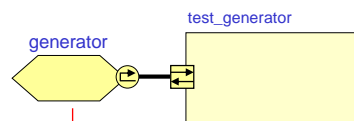
generator   test_generator

```
struct generator : public sc_channel, public generator_if
{
       …
};
```

Custom Channels and Data

7

Example : Generator Channel Declaration

generator

hierarchical_channel

```
struct generator : public sc_channel, public generator_if
{
  SC_HAS_PROCESS(generator);

  // Constructors
  explicit generator() : sc_channel(sc_gen_unique_name("generator"))
    , m_period(sc_get_default_time_unit()), m_tick(0)
  {
    SC_METHOD(generator_method);
      sensitive << m_clk;
  }
  explicit generator(sc_module_name nm, sc_time _period) : sc_channel(nm)
    , m_period(_period), m_tick(0)
  …

  // User methods
  const sc_event& default_event() const { return m_clk }
  const sc_event& posedge_event() const { return m_clk }
  const int getTickNumber() const { return m_tick }

  // Processes
  void generator_method();

private:
  sc_event m_clk;
  sc_time m_period;
  int m_tick;

  // Copy constructor so compiler won't create one
  generator( const generator& );
};
```

```
struct generator_if: public sc_interface
{
  virtual const sc_event& default_event() const = 0;
  virtual const sc_event& posedge_event() const = 0;
  virtual const int getTickNumber() const = 0;
};
```

```
void generator::generator_method(void)
{
  m_tick++;
  m_clk.notify(m_period);
}
```

Example : Generator Channel Simulation

hierarchical_channel

test_generator

```
SC_MODULE(test_generator)
{
  sc_port<generator_if> clock;

  SC_CTOR(test_generator)
  {
    SC_METHOD(test_method);
      sensitive << clock;
      dont_initialize();
  }

  void test_method(void)
  {
    cout << clock->getTickNumber() << " : Received a tick at ";
    cout << sc_time_stamp() << "." << endl;
  }
};
```

```
int sc_main(int argc, char* argv[])
{
  cout << "INFO: Elaborating ..." << endl;
  sc_set_time_resolution(100,SC_PS);

  sc_time period(2,SC_NS);
  generator clock("clock", period);
  test_generator test_generator_i("test_generator_i");
  test_generator_i.clock(clock);

  cout << "INFO: Simulating " << endl;
  sc_start(20, SC_NS);
  cout << "INFO: Post-processing ... " << endl;
  return 0;
}
```

```
INFO: Elaborating ...
INFO: Simulating
1 : Received a tick at 2 ns.
2 : Received a tick at 4 ns.
3 : Received a tick at 6 ns.
4 : Received a tick at 8 ns.
5 : Received a tick at 10 ns.
6 : Received a tick at 12 ns.
7 : Received a tick at 14 ns.
8 : Received a tick at 16 ns.
9 : Received a tick at 18 ns.
INFO: Post-processing ...
```

Custom Channels and Data

- Custom Primitive Channel
- Custom Data Type
- Custom Hierarchical Channel
- Adaptor / Transactor

| Predefined Primitive Channels (Mutexs, FIFOs, Signals)			
Simulation Kernel	Threads & Methods	**Channels & Interfaces**	Data types Logic, Integers, Fixed point
	Events, Sensitivity & Notification	Modules & Hierarchy	

Adaptator

- Custom Primitive Channel (sc_prim_channel)
- Translates between modules with different interfaces
- Moving between different abstractions
    - communication at the TLM Level
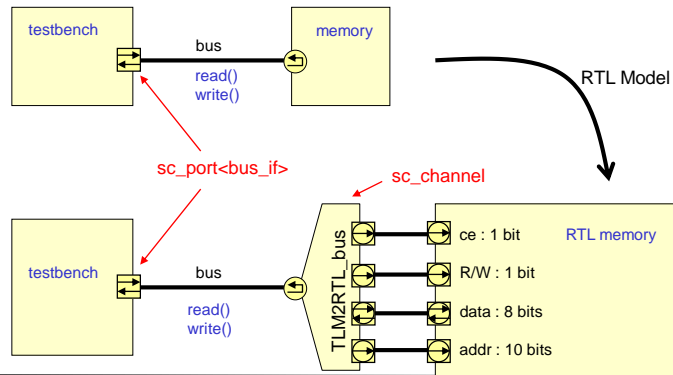   - communication at the RTL Level (Pin accurate level)

9
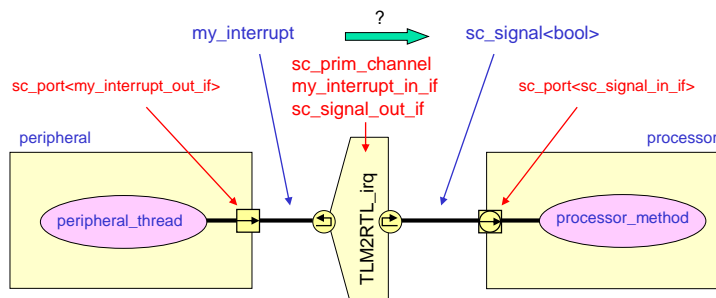
Transactor

- Custom Hierarchical Channel (sc_channel)
- Translates between modules with different interfaces
- Moving between different abstractions
    - testbench at the TLM Level
    - memory IP at th RTL Level (Pin accurate level)

Exercise

- PCIX Bus
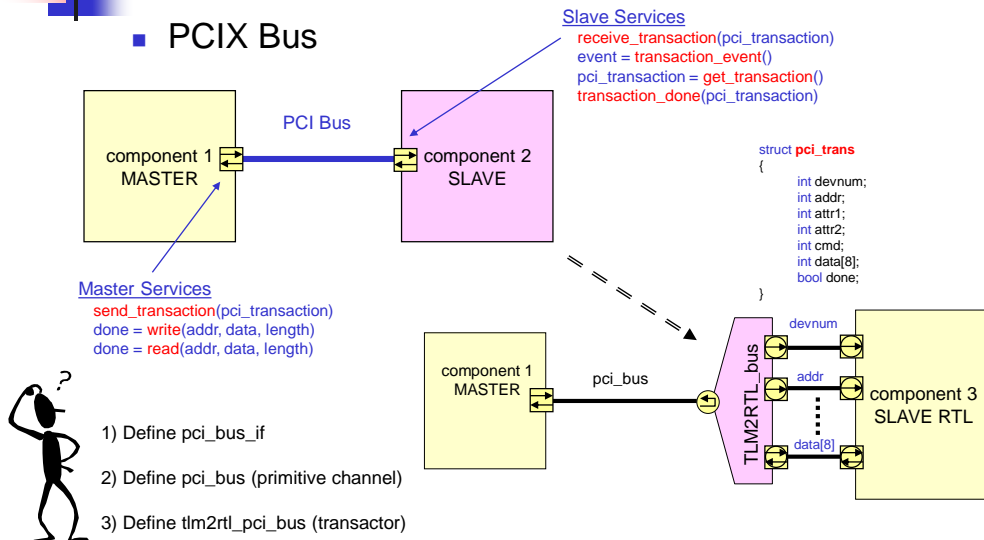
Slave Services
receive_transaction(pci_transaction)
event = transaction_event()
pci_transaction = get_transaction()
transaction_done(pci_transaction)

struct **pci_trans**
{
    int devnum;
    int addr;
    int attr1;
    int attr2;
    int cmd;
    int data[8];
    bool done;
}

Master Services
send_transaction(pci_transaction)
done = write(addr, data, length)
done = read(addr, data, length)

1) Define pci_bus_if

2) Define pci_bus (primitive channel)

3) Define tlm2rtl_pci_bus (transactor)

10